ROCA攻击——CVE-2017-15361漏洞学习 by crumbling

2023WMCTF的badprime官方wp提到了这个CVE-2017-15361

badprime

源码：

```
from Crypto.Util.number import *
from secret import flag

M =
0x7cda79f57f60a9b65478052f383ad7dadb714b4f4ac069997c7ff23d34d075fca08fdf20f95fbc
5f0a981d65c3a3ee7ff74d769da52e948d6b0270dd736ef61fa99a54f80fb22091b055885dc22b9f
17562778dfb2aeac87f51de339f71731d207c0af3244d35129feba028a48402247f4ba1d2b6d0755
baff6

def getMyprime(BIT):
    while True:
        p = int(pow(65537, getRandomRange(M>>1, M), M)) + getRandomInteger(BIT-
int(M).bit_length()) * M
        if isPrime(p):
            return p

p = getMyprime(1024)
q = getPrime(1024)
n = p * q
m = bytes_to_long(flag)

print("Try to crack the bad RSA")
print("Public key:", n)
print("The flag(encrypted):", pow(m, 65537, n))
print("Well well, I will give you the hint if you please me ^_^")
leak = int(input("Gift window:"))
if M % leak == 0:
    print("This is the gift for you: ", p % leak)
else:
    print("I don't like this gift!")
```

题解：

这题传M然后copper，很简单。但是官wp提到了个cve，想来看看怎么个事。

exp:

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

```
from Crypto.Util.number import long_to_bytes

M =
0x7cda79f57f60a9b65478052f383ad7dadb714b4f4ac069997c7ff23d34d075fca08fdf20f95fbc
5f0a981d65c3a3ee7ff74d769da52e948d6b0270dd736ef61fa99a54f80fb22091b055885dc22b9f
17562778dfb2aeac87f51de339f71731d207c0af3244d35129feba028a48402247f4ba1d2b6d0755
baff6
gift =
100243133840127462506942085389496864470323552410595691615181166409030097937986046848129899729517944638378982714166567478212461728463180001006429760883512633903435665637068391569525695712959132674812773311992255069664869516615396195970025342581219100699838977896757287282328221621509316953467 89
N =
452781556082389862909064763311530256583708351541597698969099591253314440282815329093879000912661570534592609445720411916560497842793657695746502475237211208823419607310371761863225055045763428783484070144195515229925750331574431433530782106546678593067776927223686158914354496863424769096645715597302901949251584892542237419992222666209508935796958119747245340609273521438894777818144137090566586598978779951677311136623189921290100255739739164102673758181183282735015437609819109693336829571932985535065366482337821729148796897952335205126509556594674308112307318548746769302351541667611381910108555362642151406195 3

maxlen = 1024 -len(bin(M)[2:])
PR.<x> = PolynomialRing(Zmod(N))
f = gift + M*x
root = f.monic().small_roots(X=2**maxlen, beta=0.42)[0]

print(root)
p = int(f(x=root))
q = N // p
assert p*q == N

c =
671423126344208979817020905319387107659521538401974581808466400658740425949121176339811220873187770177211049553654784310187701419630988633907039522390923642257247011797390197264431099444628550241370342966809803434077602089915863876682573081967204200944311396918398626099269491924960069539014293172012589912456689762187833047695244229784895185860115454875875027756981261009120804422582439068086000037380136289751308973297117752176034134418622366871665313790352235666401502989780634803665475527926954123107148643985437557292021664410162756392459081674875458282375811451846338551390975585916239510041697895364036127409
d = inverse_mod(65537, (p-1)*(q-1))
print(long_to_bytes(int(pow(c, d, N))))
```

ROCA攻击:

以下内容主要参考[Analysis of the ROCA vulnerability | BitsDeep]

在 RSALib 的情况下，质数具有特殊的形式：$p=k\cdot M+(65537^a \bmod M)$。 其中，参数 k 和 a 未知，M 是一个素数阶乘（前 n 个连续素数的乘积）。

文章提供了一个表格：

key size	n	M
512-960	39	$P_{39\#}$
992-1952	71	$P_{71\#}$
1984-3936	126	$P_{126\#}$
3968-4096	225	$P_{225\#}$

这种生成方法存在问题，以512位N，g=65537举例（下面都是）。

此时M=$P_{39\#}$约219bit，p为256bit，可见k约37bit。

而对于a，求满足g^k=1 mod M的最小非0正整数k，可知a约在62bit

文中提供了sage代码：

```
g = 65537
k = Zmod(M)(g).multiplicative_order()
print(log(k, 2).n())
#61.0899035000874
```

即相对于256bit素数的2^{256}种可能性，该方法生成的p只有2^{99}种

然后文中提到了一些在研究中发现的特殊性质，略过。

随后文中提到了两种检测一个密钥是否易受 ROCA 攻击：

- 第一种方法基于前文提到的特殊性质——N 被限制在模小素数的小余数集合
- 第二种更好的方法基于求解$\log_{65537}N$ mod M以及前文提到的一个性质$N=65537^a$ mod M。
  - 因为M是个光滑数，所以 Pohlig-Hellman算法可以快速求解。
  - 以下是文中提供的相关检测代码,实现了给定一个n，自动生成对应的M并求解$\log_{65537}N$ mod M
  - ```
    def isVuln(n):
        params = getParameterSizes(n.bit_length())
        if params is not None:
            M = params[0]
            try:
                a = Zmod(M)(n).log(65537)
                print(f"Vulnerable Key ! Found a={a}")
                return True
            except ValueError:
                print("Key not vulnerable to ROCA :(")
                return False

    # test a vulnerable key
    isVuln(genWeakKey(512))

    # Now test a normal key
    from Crypto.Util.number import getPrime
    notvuln = int(getPrime(300)*getPrime(300))

    isVuln(notvuln)
    ```
    ```

攻击原理：

感觉这一部分有点乱，大概是解释了coppersmith，然后实现了给定a的情况下求解k，但用的是比较复杂（指参数较多）的coppersmith_howgrave_univariate函数，而不是small_roots。

直接看优化部分吧，有提到优化部分会缩小a的可能范围。

优化1：

缩小搜索范围。 （后文：一个不大的优化）

文中有提到两个性质，这里用到其中一个：模数 N 与其素因子具有相同的形式，即$N=k_3\cdot M+(65537^a \bmod M)$;其中a≡a1+a2 mod |G|,G是由 65537(模M) 生成的Z_m的子群，a1、a2是素因子生成时的指数值。

> 原本a_1（a_2）的搜索范围是 [0, a]，但根据a≡a_1+a_2 mod |G|，我们可以缩小为[a/2,（a+|G|）/2]。因为 $a_1=a_2=a/2$ 是可以产生 a 的最小组合，而$a_1=\alpha_2=(\alpha+|G|)/2$是最大的组合。

|G|是乘法阶，应该就是个数，这个"||"符号不知道干什么的；同余式看成a+k|G|=a_1+a_2，然后结合实际的一些隐性条件，大概就出来了上面的结论。

> 对于其他所有组合，只有 α_1 或 α_2 中的一个会在这个区间内，但是只要我们在正确的情况下猜测了α_1 或α_2，就不会有问题。

这里的其他组合大概是$a_1\neq a_2$吧，意思大概是不管什么情况，总有个崽种落在缩小后的区间里，能找到a_1或a_2，也就能求出p或q，然后成功分解N。

但是和后面说的又有些矛盾，在案例中，没落在范围中的a1被求出来了，范围中的a2却没求出来q，为了同时求出，修改参数beta为0.1。但既然有p了,验证一下直接q=N//p就行了吧，这样反倒和上一段的求出a_1或a_2即可分解N比较符合。

coppersmith_howgrave_univariate：

```
def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    """
    Taken from https://github.com/mimoo/RSA-and-LLL-
attacks/blob/master/coppersmith.sage
    Removed unnecessary stuff
    """
    dd = pol.degree()
    nn = dd * mm + tt

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

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

    # 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]

    # LLL
    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
```

solve函数:

```
def solve(M, n, a, m, t, beta):
    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 polynomial f(x)
    F = PolynomialRing(Zmod(n), implementation='NTL', names=('x',))
    (x,) = F._first_ngens(1)
    pol = x + known
    # 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
```

提供的案例：

```
# Gen 2 week primes
# I'm not generating the key directly because I need to know the primes
# in order to calculate the right value for a
M, k_s, a_s = getParameterSizes(512)
#p = genWeakPrime(M, k_s, a_s)
#q = genWeakPrime(M, k_s, a_s)

#Those values were generated using the lines above
#I just kept the ones that produced the desired behaviour
p =
12158149537512377919345241392526593208961072044870310355869029319871753985047 1
q =
11641643487127808177205312424360947619352675363746213863281471932194035820382 9
n = p*q
print(f"n: {n}")
print(f"p: {p}")
print(f"q: {q}")

print(f"p > q: {p > q}")

a3 = Zmod(M)(n).log(65537)
order = Zmod(M)(65537).multiplicative_order()
inf = a3//2
sup = (a3+order)//2
print(f"interval : [{inf}, {sup}]")

# attempt to find p
# Compute the right values for a and k
a1 = int(Zmod(M)(p).log(65537))
print(f"a1: {a1}. In interval: {inf <= a1 <= sup}")
k = (p - int(pow(65537, a1, M))) // M
# Assert correctness of the values
assert(k*M + pow(65537, a1, M) == p)
# solve using the parameters from the paper
factors = solve(M, n, a1, 5, 6, 0.5)
print(f"Known p: {factors}")

# attempt to find q
# Compute the right values for a and k
a2 = int(Zmod(M)(q).log(65537))
print(f"a2: {a2}. In interval: {inf <= a2 <= sup}")
k = (q - int(pow(65537, a2, M))) // M
# Assert correctness of the values
assert(k*M + pow(65537, a2, M) == q)
# solve using the parameters from the paper
factors = solve(M, n, a2, 5, 6, 0.5)
print(f"Known q: {factors}")
```

优化后的案例：

```
# attempt to find p
# Compute the right values for a and k
a1 = int(Zmod(M)(p).log(65537))
print(f"a1: {a1}. In interval: {inf <= a1 <= sup}")
k = (p - int(pow(65537, a1, M))) // M
# Assert correctness of the values
assert(k*M + pow(65537, a1, M) == p)
# solve using a smaller beta
factors = solve(M, n, a1, 5, 6, 0.1)
print(f"Known p: {factors}")

# attempt to find q
# Compute the right values for a and k
a2 = int(Zmod(M)(q).log(65537))
print(f"a2: {a2}. In interval: {inf <= a2 <= sup}")
k = (q - int(pow(65537, a2, M))) // M
# Assert correctness of the values
assert(k*M + pow(65537, a2, M) == q)
# solve using a smaller beta
factors = solve(M, n, a2, 5, 6, 0.1)
print(f"Known q: {factors}")
```

优化2：

调整M的值。（主要的优化）

类比已知p高位的coppersmith，此处M的大小类似于p已知位数的数量。

前面提到的此处利用coppersmith求解的条件是$\log_2(M) > \log_2(N)/4$，实际中的M要比这个需求的值大不少，为减少搜索空间，寻找一个更小的M'，以满足：（这里好像没有证明更小的M和减少搜索空间的关系~~也可能是我学的太少了一眼看不出来~~）

- N的因子与先前相同的形式（M'必须是M的除数）；
- 可以使用CHG（$\log_2(M) > \log_2(N)/4$）；
- 攻击的运行时间是最小的（尝试次数和每次尝试的时间应产生最小的时间）。

对于512位密钥，M中有39个因子，因此有2^{39}个M的除数，无法全部测试。

> 但可以寻找由65537生成的群G'的阶|G'|（应该是对应M'）。事实上，|G'|是|G|的除数，而|G|本身是M的除数，因此其搜索空间较小：

```
def primorial(n):
    M = 1
    p = 1
    for i in range(n):
        p = next_prime(p)
        M *= p
    return int(M)
M = primorial(39)
print(M)
print(number_of_divisors(M))
order = Zmod(M)(65537).multiplicative_order()
print(order,M%order)
print(number_of_divisors(order))
```

```
"""
962947420735983927056946215901134429196419130606213075415963491270
549755813888  =2**39
2454106387091158800 1538361133389550470
76800
"""
```

意思大概是M'可能值（549755813888）太多，直接测试很困难;而M'对应的|G'|也是M对应的|G|的因子，又因为|G|的因子的可能值（76800）比较小，所以就比较好爆破。（因为一个|G'|对应多个M'）

然后|G|是M除数这边好像不是很对，上面代码注释里有结果，不是0。 不懂内~~

总之，遍历全部|G|因子，用以下函数寻找|G'|对应的最大M'

```
def recover_M_prime_from_order(M, order):
    M_prime = M
    for p, e in factor(M):
        ordP = Zmod(p)(65537).multiplicative_order()
        if order % ordP != 0:
            M_prime /= p
    return int(M_prime)
```

M所有素因子对应的阶若不为|G|的因子则剔除，这里似乎还是没有解释完原理。

对M'的寻找：

```
import pickle

M = primorial(39)

order = Zmod(M)(65537).multiplicative_order()
di = divisors(order)#order的全部正整数因子
print(f"Number of divisor of the order of M: {len(di)}")

# 使用pickle保存结果，以便在更改代码时无需重新计算
try:
    M_ps = pickle.load(open("save.txt", "rb"))
except FileNotFoundError:
    # 计算足够大的所有M_p
    M_ps = []
    treshold = log(n, 2)/4
    f = factor(M)
    for ordp in di:
        M_p = recover_M_prime_from_order(M, ordp)#这个函数的实现似乎没有
        if log(M_p, 2) >= treshold:
            M_ps.append(M_p)
    M_ps.sort()
    pickle.dump(M_ps, open("save.txt", "wb"))

print(f"Number of possible M': {len(M_ps)}")
"""
Number of divisor of the order of M: 76800
Number of possible M': 13004
"""
```

然后进行了测试，指出512位的密钥下，coppersmith_howgrave_univariate的参数m=5为最佳

后面又是一些测试。

————————————————————————————————分割线
————————————————————————————————

对于这篇二手文章的翻译与学习结束了，觉得有些原理没解释的部分应该可以在原论文找到。

总结：

- 偷了个懒没有直接看论文而是看了二手文章；
- 二手文章确实存在一些问题，不过第一次完整翻译看完一篇类似论文的全英文文章还是值得高兴的；
- > 其做法是通过提交65537的阶在其上较小的M的因子，就可以让a很小
  - 官方wp中这段甚至不通顺的话大概就是优化2了，提交的大概是上文中的最佳M'
- 本意是想要学完后改个题出来的,但是没有什么特别好的想法，而且感觉学的也不是很通透，害怕题改着改着就成了coppersmith而不是ROCA了。

The Return of Coppersmith's Attack:Practical Factorization of Widely Used RSA Moduli (muni.cz)

GKCTF2020_Crypto*复现*roca漏洞_M3ng@L的博客-CSDN博客

ROCA vulnerability - Wikipedia

GKCTF2020]Backdoor - 代码先锋网 (codeleading.com)

那么爱沙尼亚身份证系统和TPM的问题是什么？弱素数生成器（和 RSA！ |作者：比尔·布坎南教授 OBE |A安全网站：当鲍勃遇见爱丽丝 |中等 (medium.com)

FlorianPicca/ROCA: A Sage implementation of the ROCA attack (github.com)

https://github.com/FlorianPicca/ROCA