Overview

In this challenge, we are given the following Sage code / interpreter output:

```
u = getrandbits(512)
p = next_prime(1337 * u + getrandbits(300))
q = next_prime(2021 * u + getrandbits(300))
n = p * q

sage: n
[...]
sage: e
[...]
sage: round(log(d, 2))
376
sage: Zmod(n)(int.from_bytes(flag, 'big'))^e
[...]
```

We can see that we are supposed to break this custom RSA implementation, where certain aspects of the RSA key (the primes, the decryption exponent d) are constrained in various ways. Such constraints often lead to insecurities in the RSA protocol. In fact, this challenge combines two very common RSA insecurities (at least, common in CTFs):

1. The two primes that comprise n are approximately linearly related (with known linear relation). The simplest / most common form of this vulnerability is when p and q are very close to one another, where it is possible to apply Fermat's factorization method to recover p and q. In this case, we can do something similar: we multiply n by $2021 \cdot 1337$, and then use Fermat's method to recover the two factors $2021p$ and $1337p$, which should be close to each other (in this case, $\approx 2^{300}$ apart from one another).
2. The private key d is quite small. Normally, d should be approximately as large as n (in this case, ~1024 bits), but we are told that d is only 376 bits. When the private key is much smaller than n, it is sometimes possible to apply Wiener's attack to recover d (and hence factor n).

However, the bounds we are given in this problem statement are slightly too weak to apply either method in isolation! Fermat's factorization method works when the distance between the two factors is at most $N^{1/4}$ – in our case, that is $\approx 2^{256}$, whereas we are only guaranteed a distance between the two factors of $\approx 2^{300}$. Similarly, Wiener's attack is only guaranteed to work if $d < N^{1/4}/3$; again, in our case this is approximately 2^{256}, whereas d is of size 2^{376}. These gaps (between the functional regime of the attacks and what we are given in the problem statement) are too large to address by brute force or getting lucky. We need to find a way to use both vulnerabilities at once.

Wiener's attack

To do this, it will be helpful to summarize how Wiener's attack works. For a more in depth explanation, the Wikipedia article linked above explains things quite well.

The main idea behind Wiener's attack is the following. We want to somehow use the fact that the decryption exponent d is small. The decryption exponent d is defined to be the inverse of the (publicly known) encryption exponent modulo $\phi(N)$, i.e. $de \equiv 1 \bmod \phi(N)$. In particular, there exists some $k \leq d$ such that $de = k\phi(N) + 1$.

If d is *extremely* small, you can brute force over k to recover $\phi(N)$. But if d is only moderately small (within the bounds for Wiener's attack) here is one approach for recovering d. Divide through by $d\phi(N)$ and rearrange to get:

$$\frac{e}{\phi(N)} - \frac{k}{d} = \frac{1}{d\phi(N)}$$

Note that this implies that k/d is an extremely good rational approximation to $e/\phi(N)$. In particular, it is extremely good when d is significantly smaller than $\phi(N)$; normally, you'd expect the closest multiple of $1/d$ to a random number to be roughly $1/d$ away from that number, but here we see that it is $1/(d\phi(N))$ away from this number, a factor of roughly $\phi(N)$ smaller than you'd expect. So if we could somehow examine all 'surprisingly close' rational approximations of $e/\phi(N)$, we might be able to find k/d in this list. This is exactly what the theory of continued fractions lets us do. In particular, if we look at the convergents of the continued fraction expansion of $e/\phi(N)$ we'll in fact be guaranteed to find k/d, regardless of the size of d (see Sidenote 1 below).

There is one caveat here: we don't know $e/\phi(N)$ since we don't know $\phi(N)$ (after all, if we did our job would be done). The main observation in Wiener's attack is that e/N is pretty close to $e/\phi(N)$, so k/d could still be a surprisingly close rational approximation of e/N, especially if d is small enough (the same amount of closeness is more surprising the smaller the denominator is). It turns out that whenever $d \leq N^{1/4}/3$, this is close enough to guarantee that k/d will appear as a convergent of e/N. I won't go through the analysis here (see the Wikipedia article for the proof), but one relevant detail is that if we let $\Delta = e/N - e/\phi(N)$ be the difference between $e/\phi(N)$ and our approximation e/N of $e/\phi(N)$, then we can recover d whenever $d = O(\Delta^{-1/2})$. Since $|e/N - e/\phi(N)| = O(N^{-1/2})$, Wiener's attack works whenever $d = O(N^{1/4})$.

Improving Wiener's attack

How can we improve Wiener's attack our additional information about N? The main idea is simple; instead of using N as an approximation of $\phi(N)$, we'll get a much closer approximation of $\phi(N)$ using what we know about p and q. In particular, note that we can roughly approximate p and q by first computing

$u_0 = \sqrt{N/(1337 \cdot 2021)}$, and approximating $p_0 = 1337u_0$ and $q_0 = 2021u_0$. Both p_0 and q_0 will have error roughly 2^{300}.

Now, since $\phi(N) = N+1-p-q$, we can approximate $\Phi(N)$ via $\phi_0 = N+1-p_0-q_0$. By the logic above, we should expect $|\phi(N) - \phi_0| \lessapprox 2^{300}$. In particular, this means that we should expect $\Delta' = |e/\phi(N) - e/\phi_0| \lessapprox 2^{300}/N \approx 2^{-724}$. And finally, this means that we should expect to be able to recover d that are less than $\sqrt{\Delta'} \approx 2^{362}$.

Our d is a little bit bigger than this (2^{376}), but not *much* bigger than this (and we've been a bit sloppy in the analysis). We can bruteforce the remaining bits of entropy by repeatedly randomly perturbing our approximation and checking that it works (hoping that for some random perturbation, we land close enough to $e/\phi(N)$ that we're in business). If we choose the size of our perturbation appropriately, this should require at most $2^{376-362} = 2^{14}$ tries; in practice we'll see it works much quicker.

Implementation

Below is my implementation of this attack. The Wiener's attack code in `wiener_attack` was taken and adapted from an implementation I found at https://github.com/pwang00/Cryptographic-Attacks. I chose the upper and lower bounds for the perturbation by looking at how far my approximation of $\phi(N)$ was from the true value on some generated input (and then fudging them). In practice, `break_rsa` usually succeeds in 4-5 trials.

```
from Crypto.Util.number import *
import math
import random

def wiener_attack(e,N, phi_approx=None):
    if phi_approx is None:
        phi_approx = N
    cf = continued_fraction(e/phi_approx)
    G.<x> = ZZ['x']
    for index, k in enumerate(cf.convergents()[1:]):
        d0 = k.denominator()
        k = k.numerator()
        if k != 0 and (e * d0 - 1) % k == 0:

            phi = (e*d0 - 1) //k
            s = (N-phi+1)
            f = x^2 - s*x + N
            b = f.discriminant()
            if b > 0 and b.is_square():
                d = d0
                roots = list(zip(*f.roots()))[0]
                if len(roots) == 2 and prod(roots) == N:
                    print("Roots:", roots)
                    print("[x] Recovered! \nd = %0x" %d)
                    return d
            else:
                continue
    return -1

N = 376347864369130929314918003073529176189619811132906053032580291332225522349124770927556
e = 310766221758298615381202293330340201483059349618898159596286032329366528507960364515724
ct = 30395967614926758582504624829894677104552476154094838511931038102273100315217145329535

def break_rsa():
    k1, k2 = 1337, 2021
    u_approx = math.isqrt(N // (k1 * k2))
    p_approx = k1*u_approx
    q_approx = k2*u_approx

    LB = -2*4216596247929605338951293044960913741237438270868018173174825301586125448205080
    UB = 2*172155637752471653045148838634575884674033967258366609155825517864914020462331189

    while True:
        offset = random.randint(LB, UB)
        phi_approx = N + 1 - p_approx - q_approx + offset
        d = wiener_attack(e, N, phi_approx)
        if d is not None and d!=-1:
            break

    return d

def solve():
    d = break_rsa()
    pt = pow(ct, d, N)

    assert pow(pt, e, N) == ct

    print(long_to_bytes(pt))

solve()
```

Sidenotes

1. Why is k/d guaranteed to appear as a convergent of $e/\phi(N)$? You can prove this from the properties given in the linked Wikipedia page, but one nice way to understand this is to notice that computing (convergents of)

continued fractions of a rational number x/y is basically the same thing as running extended Euclid's algorithm to compute $\gcd(x, y)$ – so it should not be surprising that $x^{-1} \bmod y$ appears somewhere.

2. A paper that essentially does the same thing as described in this post is available here.

3. It's also possible to extend Boneh-Durfee in much the same way as we extended Wiener's attack above, i.e. by plugging in a more accurate approximation of $\phi(N)$. This has the added benefit of not requiring any brute-forcing of the remaining randomness (and seems to be how most teams solved it). This approach is also described in the above paper.