ECSC Estonia Prequalifier - dlog2

See ECSC Estonia Prequalifier - dlog for the part 1 version and context.

Challenge script:

```
from Crypto.Util.number import isPrime
import random
import signal


p = int(input("give me a safe prime!: "))
assert isPrime(p)
assert isPrime((p-1)//2)

for i in range(5):
    # chop chop buddy, no time to waste here
    signal.alarm(40)
    g = 4
    v = random.randint(2,2**295)
    print(f'{g}^x mod {p} = {pow(g,v,p)}')
    x = input("what was x?: ")
    if int(x) == v:
        print("correct!")
    else:
        print(f"wrong! v = {v}")
        exit()

print(f'well done!')

with open("flag.txt", "r") as f:
    flag = f.read()
    print(flag)
```

This time the prime p must be safe and we must calculate the discrete logarithm within 40 seconds. We also see that the prime must be at least 296 bits so that we don't lose any information about v.

This is a pretty major hint that we need to somehow find a safe prime p such that we are able to calculate the discrete log fast. The bitness is too large for something like Baby step giant step.
After googling around I found 2 candidates - the Index calculus algorithm and the General Number Field Sieve (GNFS). I attempted index calculus but it was too slow. The bitness was still too large.

GNFS, however, is the best method for calculating discrete logarithms and factorizing numbers larger than 10^{100}. There is also a special version of this - the Special Number Field Sieve (SNFS).

> **Note**: It has come to my attention after writing this writeup that you do not need to actually use SNFS and GNFS is good enough. However I will still leave my steps here incase it is needed in the future.

SNFS can be used for calculations with numbers up to double the bitness of GNFS in the same amount of time. Well not exactly, more specifically it can do the first step of the sieving of finding relations among a factor base in the same time. The second step is still the same speed*.

> *This is also why GNFS is fine enough - SNFS only speeds up the first step of the calculations, meaning the first part (that is precomputation and as such does not actually need to be fast for this challenge) is sped up and the second part is the same speed. Asking around then people who used GNFS could do the precalculations in 20 minutes on a very powerful processor or in 2 hours for a cheap rental EC2 instance. On my fairly average laptop I managed to do the first step in about 5 minutes or so, so it does speed it up significantly, but as I said, is not really necessary in the context of this challenge.

The "special form" here means that the number can be represented in the form $r^e + s$ where r and s are small. In general though, for SNFS, we want a number that can be represented as a polynomial with small coefficients.

After hours of research I found a great example I used as reference to find my own polynomial here.

I used a script to try to generate a good polynomial:

```
from Crypto.Util.number import isPrime

m = pow(2, 296 // 4)
for b in range(1, 10_000):
    start_p = m ** 4 + b * m

    c = 1
    while c < 100_000:
        tmp = start_p + c
        if isPrime(tmp) and isPrime((tmp - 1) // 2):
            print(f'p = {tmp}, b = {b}, c = {c}')
        c += 2
```

I took the first output of this script which resulted in the parameters

```
p =
127314748520905380391777855525586135065716774604121015664758778084686610167071501298180963
m = pow(2, 74)
b = 2
c = 9059
```

Which means the resulting safe prime is of the form $p = m^4 + 2 \cdot m + 9059$, where $m = 2^{74}$.

Now, GNFS is no simple algorithm, so I was not about to implement it myself. Luckily a very handy tool called cado-nfs exists, which is an implementation of GNFS written in C++.

Unfortunately, the parameters used were very confusing to me and I could not find good documentation on them, so I just had to guess and try based on the factorization configuration's given inside the project. There was, however a single example of polynomial parameters for using SNFS:

```
# Polynomial for the Fermat number F9 = 2^512+1 = 2424833*n
n:
552937374653949245146945170995522006153799697570611806162468155280044606373863559956577393089210821021077816830539919691531494449801143829139311820 9
skew: 1.0
c5: 4
c0: 1
Y1: -1
Y0: 5070602400912917605986812821504
```

Now, what these numbers mean, I had to guess. But it seems like $c_5, c_4, \ldots, c_0$ are the coefficients of the powers of x (i.e when $c_5 = 12$ then the polynomial is $P(x) = 12 \cdot x^5$). The Y_0 value seemed to be such that $c_5 \cdot Y_0^5 + c_0 = 2^{512} + 1 = F_9$. As to why the $n \neq F_9$. I do not know, maybe it's because the factor 2424833 was already known so it was left out.
I also did not figure out what Y_1 was supposed to be, but later in my testing I found out that if I changed it from -1 then the sieving just took horrendously long to complete (so long that I canceled it). I also read from somewhere that the `skew` value should be 1 when using SNFS, but I am not sure as to what it actually does.

Anyways now that I had figured out what the parameters were, it was time to create my own polynomial parameters out of the prime I chose, i.e the polynomial that generated my prime

p will be used in the sieving process, which means that I want to create polynomial parameters for the polynomial $P(x) = m^4 + 2 \cdot m + 9059$, which looks like this:

```
n:
127314748520905380391777855525586135065716774604121015664758778084686610167071501298180963
skew: 1.0
c4: 1
c1: 2
c0: 9059
Y1: -1
Y0: 18889465931478580854784
```

The n value is the prime p itself.

I also had to create the sieving parameter file itself as well, for which I simply copied the `params.p90` file supplied by `cado-nfs` and edited/added a few values in the polynomial selection so that it would not try to find a polynomial itself and instead would use the one provided (these sieving parameters I took from the example `params.F9` file, but the rest is the `params.p90` file, since F_9 is a lot larger of a number than 2^{296}). After some tweaking of other parameters as well, the final parameters file looked like this (I created the folder `workdir` for the tasks):

```
################################################################################
#
#     Parameter file for Cado-NFS ; DLP version
################################################################################
#
# See params/params.c90 for an example which contains some documentation.

################################################################################
#
# General parameters
################################################################################
#

name = safePrime90

N      =
```

```
127314748520905380391777855525586135065716774604121015664758778084686610167071501298180963

tasks.workdir = /home/user/dlog2/workdir

################################################################################
#
# Polynomial selection with Kleinjung's algorithm
################################################################################
#

descent_hint = /home/user/dlog2/p90.hint

slaves.nrclients=6
tasks.threads=14
slaves.hostnames=localhost

tasks.polyselect.degree = 2

tasks.polyselect.admax = 0
tasks.polyselect.adrange = 0
tasks.polyselect.admin = 0
tasks.polyselect.qmin = 10000

tasks.polyselect.P = 8000
tasks.polyselect.nq = 2000
tasks.polyselect.nrkeep = 100
tasks.polyselect.ropteffort = 0.01

tasks.polyselect.import = ./input2.poly

################################################################################
#
# Sieve
################################################################################
#

tasks.I = 12
tasks.lim0 = 240000
tasks.lim1 = 240000
```

```
tasks.lpb0 = 20
tasks.lpb1 = 20
tasks.sieve.mfb0 = 40
tasks.sieve.mfb1 = 40
tasks.sieve.lambda0 = 2.2
tasks.sieve.lambda1 = 2.2
tasks.qmin = 50000
tasks.sieve.qrange = 10000
tasks.sieve.rels_wanted = 300000

################################################################################
#
# Filtering
################################################################################
#

tasks.filter.target_density = 100

tasks.reconstructlog.partial=false

################################################################################
#
# Individual log
################################################################################
#

tasks.descent.init_I = 12
tasks.descent.init_ncurves = 20
tasks.descent.init_lpb = 38
tasks.descent.init_lim = 240000
tasks.descent.init_mfb = 76
tasks.descent.init_tkewness = 240000
tasks.descent.I = 12
tasks.descent.lim0 = 240000
tasks.descent.lim1 = 240000
tasks.descent.lpb0 = 20
tasks.descent.lpb1 = 20
tasks.descent.mfb0 = 40
tasks.descent.mfb1 = 40
```

Then all that is left to do is run the script once by factorizing some number. I ran the challenge script once to get a random number modulo my prime p.

```
cado-nfs.py --dlp -ell <(p-1) / 2> <NUMBER TO CALCULATE DISCRETE LOG FOR>
<p>
```

The `-ell` flag is the modulus that the discrete log will calculated against, and in the case of a safe prime is just the large factor of $p-1$ of prime p, which in my case is `6365737426045269019588892776279306753285838730206050783237938904234330508353575064909048 1`.
This will calculate the discrete log with some unknown base, so in order to get the discrete log with base 4 that is in the challenge, we also need to know the discrete log of 4, which is just 2, and then convert the base.

Running this once will also leave us with a snapshot file that we can use to skip the sieving process and go straight into calculating the discrete log, which we will use to solve the challenge.

The final solve script looks as such:

```
from Crypto.Util.number import getPrime, isPrime, bytes_to_long
from math import exp, log, sqrt
import math
from sage.all import *
import subprocess
from pwn import *


p =
127314748520905380391777855525586135065716774604121015664758778084686610167071501298180963
ell =
6365737426045269019588892776279306753285838730206050783237938904234330508353575064909048 1

m = pow(2, 74)
assert pow(m, 4) + 2 * m + 9059 == p and isPrime(p) and isPrime((p - 1) //
2)


log4 = 2

def calc_dlog(y):
    outputlines = subprocess.check_output(['/usr/bin/cado-nfs.py',
'/home/user/dlog2/workdir/safePrime90.parameters_snapshot.3', f'target=
{y}'])
```

```
    answer = int(outputlines.split(b'\n')[-2].decode())
    dl = ZZ(Integers(ell)(Integer(answer)/Integer(log4)))
    return int(dl)


HOST = '10.42.11.112'
PORT = 9999
conn = remote(HOST, PORT)
conn.recvuntil(b'give me a safe prime!: ', timeout=3)
conn.sendline(str(p))

for i in range(5):
    line = conn.recvline(timeout=3)
    val = int(line.split(b'=')[1].strip())
    conn.recvuntil(b'what was x?: ', timeout=3)
    conn.sendline(str(calc_dlog(val)))
    conn.recvline()

print(conn.recvall())
```