STEP 1 (SEPARATING THE FIELD)
As we can see, decrypt_flag.sage hints towards the need for SageMath being used, you can either install it or can use it on the web. With the values given to us in chal.txt, we can deduce that all the numbers like the public key or the prime field etc. are 2048 bits. As hinted by the description we see that the field across which the curve exists is $\mathbb{Z}_n$ which is not a prime field but a composite field so we can find the factors of n and then separate the curve across 2 prime fields $\mathbb{F}_p$.

To find the factors of $\mathbb{Z}_n$ you can run the script:

```
n =
21118446916102976402895990864026344698334856017709525356783801889700310530231202571047575428675355506897074071304111326207519843569009536477742566939996640898083990215984354723203893619087649828874178604454087823955729782902135262588990574865142547988290020167374605310213634992713437361708176236419224232050496136541870348165741337699565368324096311422955253644155371580268906186891287667395526597763725715259651349732540800736787781185216559044860903110805075655625886302703053881154656623518488055737197466149004373957011559941273912973110576648516085126301411485385623055625295804141280775201582835450045631873297
start = 0
end = ((2**32) - 1)
for candidate in range(start, end):
        if n % candidate == 0:
        print(f"p1 = {candidate}")
        p1 = candidate
        p2 = n // p1
        break
print(f"p2 = {p2}")
```

To reduce the time required for this we can see the hint that says one of the primes occupies 8 hexadecimal spaces, using this we can limit the range from start = int('10000000', 16) and end = int('FFFFFFFF', 16). Once this script is done we get the output:

```
p1 = 500000003
p2 =
4223689357878459133308443372954608701939318991905991119920813658415180155555159580878557600463725498597061822678451329170795993689025945161392842419642273661763156072617934508933171670218499944463836054107801240144338515714395958231422365584494316090692107489322227
```

```
61261090702418882660210120391212116101191404385124657429948386902985077712826181915665754416512761811666589667812835775698320136863274706271782281672005775049566823278161430764149505136909190788696166519595606288534124578108499568325114064444247911343260445963557206766482705622257063298627872811055733904911707821121361355833382268165377291099
```

Since we have now recovered p1 and p2 where "$\mathbb{Z}_n$ = p1 * p2" we can now split the curve across the 2 prime fields that is "`(y^2 = x^3 + ax + b)modp1`" and "`(y^2 = x^3 + ax + b)modp2`".

STEP 2 (CRACKING THE SMALLER CURVE)

These 2 curves are 32 bits and 2016 bits respectively. A 32 bit curve is vulnerable to many attacks like Pollard rho or Pohlig-Hellman which recovers the private key for that curve, which in our case is d%p1. We can implement these attacks via the discretelogs inbuilt function in SageMath by writing a script like:

```
from sage.all import *

# Curve parameters
a = 5
b = 7
p1 = 500000003
E = EllipticCurve(GF(p1), [a, b])

G_coords =
(17187694640492776212321349425202169393264348204722733953816876501887724992021266156202995744886290247327253891111259175920250026097458283608905694193876153264067388855961584285911871538242808186750676704201714363726449535322394182133216886728540408123668202014595227912277487601232836316914229768776682703020264011048929488558168908821811877351563168417729794914563518957917763970153239245942819595491025548044301364131983827967840862479884200146730266025429245158021030433942555003690275608394454273234888316445543253439382657597139963443092018229046163221390238967830916855415499693055545457623490920824912695098540 % p1,
20782908587899578644252873134286324020860054437775664623465604863200035559590009759067174183945249191559228668705020547825597352888414787807742930861381590674172225620048645459965646338131070593715963479999220185392014002561539969682046692599928210878056312347074311333462000874904690479920252136375143288136340950280966472825516000777934546884838424648573817606565598091811078449736022060280850
```

```
40090087651780866326704509194613935891920288868069929844411393432408250776290038309437888072172110187668434903160231281972876722225054691789241844900779432958139438733651930378636715904688369736392653179112101503933182770 % p1)
Q_coords =
(15191711251116898347014572384264704616385453606951805062507846227848753451241999596467688399640778812846933745617763983440663205713155786108746381519210926605254520133402217038002951461068204445158715469239437552480639305014145924168985145872510912998672323375844290098046099438959780029173863383366153108044694995442247335633356241565421062892724410508008158717978942532713785965103186527082790492023304673181085528365125431031603914223046755900730602063577324162605616741996604654676984463552948035097656871983165382148923213754280620113148447149810287586033208757948046447796156103062359807923452078767280859698514 % p1,
14248250049381203716698393090032003663187995023908848579202369585923258708108114517164874915824651060544184618806974838804673576440612035799614210837287730930827042379127943201423249263836484359268227011196038706870322686171965682959518109397626611965078226610156955862602242075856337863495143075348273030043867449831019981257805178694610664696841459907876154820976883373137580652299115571712414851011163947433361580963643449737029417081420262904357428588947437307849087551993237330659444846528564457459912494672280508967609071172578715558371372542429574100166734604210518826342722682185807869659659403788912883106707 % p1)
# We use %p1 to limit the field to the field of the new curve

G = E(G_coords)
Q = E(Q_coords)

# Get the order and its factorization (not required but can be
used for better accuracy)
order = G.order()
print(f"Curve order: {order} ")

# Estimate Pollard-rho time (also not required but helps us
estimate)
import math
rho_steps = int(math.ceil(order ** 0.5))
print(f"Pollard's rho estimated steps: ~{rho_steps}")

# Show all factors for Pohlig-Hellman/subgroups (this is also not
```

```
required but gives clarity if running Pohlig-Hellman)
print("Prime factors of order:", [factor[0] for factor in
order.factor()])

# Recover the d using the discrete log (this tries all kinds of
algorithms like Pohlig-Hellman and Pollard-rho without need of
specification)
d = discrete_log(Q, G, operation='+')
print("Recovered d :", d)
```

From this script we get the output:

```
Curve order: 499980621
Pollard's rho estimated steps: ~22361
Prime factors of order: [3, 7, 263, 90527]
Recovered d : 58373
```

STEP 3 (BRUTEFORCING FOR THE PRIVATE KEY)

Here we don't recover the complete private key(d) but a partial private key(d%p1), on solving the second curve we should get d%p2 and can run CRT(Chinese Remainder Theorem) to recover complete d.

Upon trying to solve the curve with p2 as its prime field, using various possibilities like endomorphism or isogeny, you will realize that the curve is secure and it is not practically possible to recover d%p2 and the solution lies elsewhere.

Looking back to the name of the question and the hints provided throughout we know that the d is small and what we have recovered so far is d%p1 which is literally the remainder of d after dividing by p1, So we can also write this in the form of "`d = k.p1 + d%p1`" where k is an integer, now knowing this we can try bruteforcing different values of d which we know might be correct by increasing the value of k in each iteration. For this we can make changes to the decrypt_flag.sage script provided to us:

```
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from binascii import unhexlify

def ecies_decrypt():
    p1 = 500000003
```

```
    dmodp1 = 58373
    for k in range(20):
        d = (k*p1) + dmodp1 # input private key

        R_coords =
(18081875220526357232874641741490118182898665812268205273451792713
426620585296377396054147964582507310800564850996674798904927260338
503255841484867341368367146170940708585092407371864822566468371226
463699042260629461989548670128200524412366893899375491072468173710
372826083166667636157641032787039597593266696305552371915850748086
538020320988134072059027327075454986011577217127325287335301177450
489421904280823362591240305129914661910064046708548921220674036510
885603295854452032608472818669969357566634911060408841930044359140
724848688433857276449888913905750538730556094346667539865904999261
691530595806540686124962,
180390842250375885605935469049194414212305625191280350666979379292
102493941886513969473009622348134637133051999131417771006489136575
130372661373878252565331038340541284369886802595919235371202490149
113014691587416794125753958886014134307812953257812634868986481464
964130588697033928976628093712514575074098061884392871176214970859
029340708471700895133041142484413774824678954014640768304849604127
149620926970466409933606819735868978123398047909104755896378638112
131190370115735427130301704371654259868805287451787675770926935350
705057362120250555335334668770686006011197953254006856171778945679
98606360073764456198722)
        ct_hex =
"6fd6eef34ba7a0dd84706a9c82fda8b44e1b44bd4a5ea3750eee6b0d8ad7b0ba5
a2f0443a2523a870d0be41ad9d34d5c"
        iv_hex = "832005852f5fbb66194300c750e4d21e"
        n =
211184469161029764028959908640263446983348560177095253567838018897
003105302312025710475754286753555068970740713041113262075198435690
095364777425669399966408980839902159843547232038936190876498288741
786044540878239557297829021352625889905748651425479882900201673746
053102136349927134373617081762364192242320504961365418703481657413
376995653683240963114229552536441553715802689061868912876673955265
977637257152596513497325408007367877811852165590448609031108050756
556258863027030538811546566235184880557371974661490043739570115599
412739129731105766485160851263014114853856230556252958041412807752
01582835450045631873297
        Zn = Integers(n)
        a = 5
```

```
        b = 7

        E = EllipticCurve(Zn, [a, b])
        R = E(R_coords[0], R_coords[1])


        S = d * R
        Sx = int(S[0])

        K = hashlib.sha256(str(Sx).encode()).digest()

        iv = unhexlify(iv_hex)
        ciphertext = unhexlify(ct_hex)

        cipher = AES.new(K, AES.MODE_CBC, iv)
        try:
            plaintext = unpad(cipher.decrypt(ciphertext), 16)
            print(f"Flag Found: k={k} d={d}  Flag:
{plaintext.decode()}")
            break
        except ValueError:
            continue  # Incorrect key, try next k
    return 0

ecies_decrypt()
```

Finally we get

```
Flag Found: k=3 d=1500058382  Flag:
DJSISACA{S1z3_D03Sn7_m4t7Er_dO3s_17}
```