ASCS 2020 Qualifikation Schüler
Challenge write-up: IDBased2

Philipp Schweinzer

August 9, 2020

1 Challenge description

The company in which you are working is still using the sane Boneh-Franklin BasicIdent ID-based encryption scheme with the Weil pairing to encrypt the emails. However, since last time, they fixed the previous vulnerability.

They still use the same implementation (the concrete construction of the paper) and the same hash functions.

The message are encrypted with the email address of the receipient as a public key.

In particular, while sniffing the network, you found an interesting email sent to CEO@company.ch that you want to decrypt.

1.1 Goal

Decrypt the message sent to the CEO.

2 Boneh-Franklin Identity-based encryption

The basis of this challenge is an encryption scheme called Boneh-Franklin Identity-based encryption. To try to solve this challenge, the first step is to look and understand this scheme. It is grouped into 4 different sections:

2.1 Setup

The PKG[1] chooses several different parameters:

- two groups G_1 and G_2 with size q
- a corresponding pairing e
- a randomly chosen private master-key $K_m = s \in \mathbb{Z}_q^*$
- a public key $K_{pub} = sP$
- a public hash function $H_1 : \{0, 1\}^* \rightarrow G_1^*$
- a public hash function $H_2 : G_2 \rightarrow \{0, 1\}^n$ for some fixed n
- the message and cipher space $\mathcal{M} = \{0,1\}^n,\ \mathcal{C} = G_1^* \times \{0, 1\}^n$

2.2 Extraction

To create the public key for $ID \in \{0,1\}^*$, the PKG computes

- $Q_{ID} = H_1(ID)$
- the private key $d_{ID} = sQ_{ID}$ which is given to the user

2.3 Encryption

Given $m \in \mathcal{M}$, the ciphertext c is obtained as follows:

1. choose a random $r \in \mathbb{Z}_q^*$
2. compute $g_{ID} = e(Q_{ID}, K_{pub}) \in G_2$
3. set $c = (rP, m \oplus H_2(g_{ID}^r))$

2.4 Decryption

Given $c = (u, v) \in \mathcal{C}$, the plaintext can be retrieved using the private key:

$m = v \oplus H_2(e(d_{ID}, u))$

[1] public key generator

3 Solution

The first thing I did was to look at the parameters of the system:

```
p = 104325713908305334229818410206151915215256669113967974838668
    ↪ 874497680483737320360060218534636592044587098464041933074
    ↪ 100832839644926421234954934504374241718146301656776131015
    ↪ 504309022031926979511064544370288546476496553178800909499
    ↪ 120697939772248835637365286988458947325026984837081671557
    ↪ 1001049082443246885667
q = 727846484219
P = (583082123623774735717580040820510388336821617295141563079
    ↪ 374351496855908218552460443716097429418551584645560639015
    ↪ 033122072727116862494552015023127047361035483699573168523
    ↪ 693056313939984752010697277789981879245300709358159779898
    ↪ 117574193805643735166447053840457059235411165176030821777
    ↪ 8434141122993078494118,
    ↪ 742436192609357890650611915582193999084786793361924515278
    ↪ 767122478477926677040615238394961569966973763575674418014
    ↪ 504283057114961223030540664811435831336481375516077623484
    ↪ 416748662171170421022888776290967397783044619048250976131
    ↪ 957117022521572073442329671345739325029938016561581065041
    ↪ 6342465531729061240965 9)
Ppub = (2242853255389511370672122059085831067325259284927135847
    ↪ 907371175481900631613218781584516832651218077180342161811
    ↪ 748685303644222694989201776669981762933342546728124187138
    ↪ 191319635148784751809518539042184553005843510327016139948
    ↪ 147982802008967651080108783002998483862716177468454633644
    ↪ 3069666650607801761352559,
    ↪ 355745085797555927517738813636244939996379527285931012805
    ↪ 315037110920030853152269604243764461143837329995806896863
    ↪ 007378822900256796439017446583833089545933533862417795212
    ↪ 663532243255611069775746127986577689899035662686057255656
    ↪ 055764471726400841061265278641836077490804820767334832284
    ↪ 3555553302841512572144 6)
```

There you can see that the parameter q is disproportionately small compared to the other values. q is responsible for the size of the finite field $\mathbb{Z}_q$ from which the private master-key is generated. Due to its small size it is possible to calculate the generated master key in a viable time using some optimizations.

This is were the baby-step giant-step algorithm comes into place. It is a meet-in-the-middle algorithm for computing the discrete logarithm of an element in a finite abelian group. Thus it is perfect for this application and we can use it to calculate the s in $K_{pub} = sP$.

3.1 Baby-step giant-step

This algorithm uses an efficient lookup table scheme to achieve a time and space complexity of $\mathcal{O}(\sqrt{n})$. This is a usefull improvement over the brute-forcing method, which has a time complexity of $\mathcal{O}(n)$. The algorithm is defined as followed:

Input: A cyclic group G of order n, having a generator α and an element β

Output: A value x satisfying $\alpha^x = \beta$

1. $m = \lceil \sqrt{n} \rceil$
2. For all j where $0 \leq j \leq m$:
   (a) Compute α^j and store that pair (j, α^j) in a table
3. Compute α^{-m}
4. $\gamma \leftarrow \beta$
5. For all i where $0 \leq i \leq m$:
   (a) Check to see if γ is the second component (α^j) in any pair in the table
   (b) If so, return $im + j$
   (c) If not, $\gamma \leftarrow \gamma \bullet \alpha^{-m}$.

3.2 BSGS sage implementation

Now it was time to implement this algorithm into sage.

```
import pickle
import hashlib

#parameters
p=104325...
q=727846484219
P=(583082..., 742436...)
Ppub=(224285..., 3557450...)

```

```
#Create Elliptic curve
Fp = Integers(p)
Pol.<btemp> = PolynomialRing(Fp)
F.<a> = GF(p^2, modulus=btemp^2+1)
E = EllipticCurve(F, [0,0,0,0,1])

#Creating elliptic curve points from point values
P = E(P)
Ppub = E(Ppub)

#setting upper and lower bound on possible values
k2 = 727846484219
k1 = 0

m = floor(sqrt(k2-k1))

#creating lookup table of 1P, 2P, 3P, ..., mP
table = {int(P.xy()[1]): (int(P.xy()[0]), 1)}
for i in range(2, m+1):
  z = (i*P).xy()
  table[int(z[1])] = (int(z[0]), i)
  if i % 10000 == 0:
  print(i, 'of', m)

#saving the table to a file (not necessary)
f = open('baby-step-table.json', 'wb')
pickle.dump(table, f)
f.close()


S = Ppub - k1*P
found = False
step = 0

#iterating through the table
while (not found) and step < (k2-k1):
  try:
    idx = table[int(S.xy()[1])][1]
    b = idx
    found = True
  except Exception:
    S = S - m*P
    step += m

k = k1 + step + b
print(k)
#176182672759
```

This script takes a couple of minutes with the filling of the table taking the longest. When it is finished, it outputs $s = 176182672759 \implies K_{pub} = 176182672759P$.

Now i just had to use this information to decrypt the message. At first i calculated the private key of the user $d_{ID} = sQ_{ID}$ and then just solved for $m = v \oplus H_2(e(d_{ID}, u))$. This is the sage script I used(for simplicity, the given functions are shortened):

```
import hashlib
import base64

def xor(xs, ys):
  ...

def to_bytes(n, length, endianess='big'):
  ...

#Encodes using canonical representation: ax+b is b||a
def canonic(gID):
  ...

#Precomputes some values for the computation of the twisted Weil
    pairing.
def computeTwistedWeilParams(p):
  ...

#Computes the "twisted" Weil pairing between P1 and P2.
#You need to pass as an additionnal argument twistedWeilParams
    that are generated by the method computeTwistedWeilParams(p)
    during key generation
def twistedWeil(P1,P2, twistedWeilParams):
  ...


def H2(input):
  ...

#Hash id to point on E
def HTP(E,p,q,id):
  ...

#ciphertext
#(161713..., 650562...), QodKESJH7Q/ycNrS2qfVfe0hb29AB3n5Sw==

#parameters
p=104325...
q=727846484219
```

```
P=(583082..., 742436...)
Ppub=(224285..., 3557450...)

s = 176182672759

print('Creating elliptic curve')

#Create Elliptic curve
Fp = Integers(p)
Pol.<btemp> = PolynomialRing(Fp)
F.<a> = GF(p^2, modulus=btemp^2+1)
E = EllipticCurve(F, [0,0,0,0,1])

u=(161713..., 650562...)
u = E(u)
v = 'QodKESJH7Q/ycNrS2qfVfe0hb29AB3n5Sw=='

print('Hash ID to point on E')

Q = HTP(E,p,q, 'CEO@company.ch')
dID = s*Q

print('compute Twisted Weil')

e = twistedWeil(dID, u, computeTwistedWeilParams(p))

print('Hash2')

h2 = H2(e)

print('XOR')

m = xor(base64.b64decode('QodKESJH7Q/ycNrS2qfVfe0hb29AB3n5Sw=='),
    h2)
print('Decrypted message: ' + m)
```

3.3 Flag

With the above script, the following flag was evaluated:

`YNOT18{my1D15C0MPR0M153D}`