Cyber Apocalypse CTF 2022

May 19, 2022 · 3158 words · 15 minute read

On this CTF, I only worked on the crypto challenges. I managed to solve two crypto challenges. Here is my writeup for those challenges.

Crypto

Down the Rabinhole (325 pt)

Initial Analysis

We were given two files,

`source.py`

```
from Crypto.Util.number import getPrime, isPrime, bytes_to_long
from Crypto.Util.Padding import pad
import os


FLAG = b"HTB{--REDACTED--}"


def getPrimes(coefficient):
    while True:
        a = getPrime(512)
        p = 3 * coefficient * a + 2
        if isPrime(p):
```

```python
            break
    while True:
        b = getPrime(512)
        q = 3 * coefficient * b + 2
        if isPrime(q):
            break
    return p, q


def encrypt(message, coefficient):
    p, q = getPrimes(coefficient)
    n = p * q

    padded_message = bytes_to_long(pad(message, 64))
    message = bytes_to_long(message)

    c1 = (message * (message + coefficient)) % n
    c2 = (padded_message * (padded_message + coefficient)) % n
    return (n, c1, c2)


def main():
    coefficient = getPrime(128)
    out = ""

    message = FLAG[0:len(FLAG)//2]
    n1, c1, c2 = encrypt(message, coefficient)
    out += f"{n1}\n{c1}\n{c2}\n"

    message = FLAG[len(FLAG)//2:]
    n2, c3, c4 = encrypt(message, coefficient)
    out += f"{n2}\n{c3}\n{c4}"

    with open("out.txt", "w") as f:
        f.write(out)


if __name__ == '__main__':
    main()
```

and `out.txt`

```
59695566410375916085091065597867624599396247120105936423853186912270957035981683790353782357813780840
20613176923772195500153086395968875668612548541389926119712564174574563635905866439843301335666339421
14350341133918883930676906390648724486852266960811870561648194176794020698141189777337348951219934072
56438641309774959123579452414864548345708278641778632906871133633348990457713200426806112132039095059
42954691200473101288652776725414969457473032295628702816176100727136292765204113836600456089077316725
29903904396126887576044949247400308530425862142675118500848365445245957090320752747039056821346410855
```

Okay, so what this challenge do is basically:

- Split the flag to two parts
- Generate coeff which is a prime ~128 bits
- Use this coeff to generate a new prime which fulfill $\mathrm{prime} = 3.\mathrm{coeff}.a + 2$
- Use those generated primes to encrypt the partial flag with RSA.
    - Each part got encrypted twice, the first one is without padding, and the second one is with padding

- And then, they provide us with $n_1, c_1, c_2, n_2, c_3, c_4$

So, with the given informations, we need to be able decrypt our flag.

Solution

Let's try to create equations from the given source code on generating n_1 and n_2 ($c = \text{coeff}$).

$$\begin{aligned} n_1 &= p_1 q_1 \\ &= (3ca_{11} + 2)(3ca_{12} + 2) \\ &= 9a_{11}a_{12}c^2 + 6c(a_{11} + a_{12}) + 4 \\ (n_1 - 4) &= 9a_{11}a_{12}c^2 + 6c(a_{11} + a_{12}) \\ (n_1 - 4) &= 3c(3a_{11}a_{12}c + 2(a_{11} + a_{12})) \end{aligned} \tag{1}$$

$$\begin{aligned} n_2 &= p_2 q_2 \\ &= (3ca_{21} + 2)(3ca_{22} + 2) \\ &= 9a_{21}a_{22}c^2 + 6c(a_{21} + a_{22}) + 4 \\ (n_2 - 4) &= 9a_{21}a_{22}c^2 + 6c(a_{21} + a_{22}) \\ (n_2 - 4) &= 3c(3a_{21}a_{22}c + 2(a_{21} + a_{22})) \end{aligned} \tag{2}$$

Notice that both of them share common factors (which is c). GCD both equations can help us to retrieve the coeff value, because both of them share common factors c. Just do $GCD(n_1 - 4, n_2 - 4)$, factorize it, and take the prime which has ~128 bits (Because coeff size is ~128 bits). Based on the given output, we get $\text{coeff} = GCD(n_1 - 4, n_2 - 4)//9$.

Now we have coeff. Now let re-visit the code on the encryption:

```
def encrypt(message, coefficient):
    p, q = getPrimes(coefficient)
    n = p * q

    padded_message = bytes_to_long(pad(message, 64))
    message = bytes_to_long(message)

    c1 = (message * (message + coefficient)) % n
    c2 = (padded_message * (padded_message + coefficient)) % n
    return (n, c1, c2)
```

n is ~1280 bits. And $c_1 = m * (m + \text{coeff}) \mod n$. Notice that if m bits is less than 640 bits (~80 chars), we actually can ignore the mod operation, because $m * (m + \text{coeff})$ is less than n. I don't think that the partial flag length will be larger than 80 chars, so I think we can actually solve the equations.

So let's try to solve this quadratic equation $n = m * (m + \text{coeff})$. Turn out, we successfully retrieve the flag by solving those quadratic equations on each part.

Full Script

I use sage to solve the solution

```
from Crypto.Util.number import *

n1 = 59695566410375916085091065597867624599396247120105936423853186912270957035981683790353782357813
c1 = 20613176923772195500153086395968875668612548541389926119712564174574563635905866439843301335666
c2 = 14350341133918883930676906390648724486852266960811870561648194176794020698141189777337348951219
n2 = 56438641309774959123579452414864548345708278641778632906871133633348990457713200426806112132039
c3 = 42954691200473101288652776725414969457473032295628702816176100727136292765204113836600456089077
c4 = 29903904396126887576044949247400308530425862142675118500848365445245957090320752747039056821346
coeff = gcd(n1-4, n2-4) // 9

m1 = var('m1')
m2 = var('m2')
m1_roots = solve(m1*(m1+coeff) == c1, m1)
m2_roots = solve(m2*(m2+coeff) == c3, m2)

flag = b''
for root in m1_roots:
    if root.right() > 0:
        flag += long_to_bytes(int(root.right()))
for root in m2_roots:
    if root.right() > 0:
        flag += long_to_bytes(int(root.right()))
print(f'Flag: {flag.decode()}')
```

> Flag: `HTB{gcd_+_2_*_R@6in_.|5_thi5_@_cro55over_epi5ode?}`

Mind in the Clouds (375 pt)

Initial Analysis

We were given a source code:

```
import json
import signal
import subprocess
import socketserver
from hashlib import sha1
from random import randint
from Crypto.Util.number import bytes_to_long, long_to_bytes, inverse
from ecdsa.ecdsa import curve_256, generator_256, Public_key, Private_key, Signature
import os


fnames = [b'subject_kolhen', b'subject_stommb', b'subject_danbeer']
nfnames = []


class ECDSA:
    def __init__(self):
        self.G = generator_256
        self.n = self.G.order()
        self.key = randint(1, self.n - 1)
        self.pubkey = Public_key(self.G, self.key * self.G)
        self.privkey = Private_key(self.pubkey, self.key)

    def sign(self, fname):
```

```python
        h = sha1(fname).digest()
        nonce = randint(1, self.n - 1)
        sig = self.privkey.sign(bytes_to_long(h), nonce)
        return {"r": hex(sig.r)[2:], "s": hex(sig.s)[2:], "nonce": hex(nonce)[2:]}

    def verify(self, fname, r, s):
        h = bytes_to_long(sha1(fname).digest())
        r = int(r, 16)
        s = int(s, 16)
        sig = Signature(r, s)

        if self.pubkey.verifies(h, sig):
            return retrieve_file(fname)
        else:
            return 'Signature is not valid\n'


ecc = ECDSA()

def init_storage():
    i = 0
    for fname in fnames[:-1]:
        data = ecc.sign(fname)
        r, s = data['r'], data['s']
        nonce = data['nonce']
        nfname = fname.decode() + '_' + r + '_' + s + '_' + nonce[(14 + i):-14]
        nfnames.append(nfname)
        i += 2


def retrieve_file(fname):
    try:
        dt = open(fname, 'rb').read()
        return dt.hex()
    except:
        return 'The file does not exist!'


def challenge(req):
    req.sendall(b'This is a cloud storage service.\n' +
                b'You can list the files inside and also see their contents if your signatures are 
    
    while True:
        req.sendall(b'\nOptions:\n1.List files\n2.Access a file\n')
        try:
            payload = json.loads(req.recv(4096))
            if payload['option'] == 'list':
                payload = json.dumps(
                    {'response': 'success', 'files': nfnames})
                req.sendall(payload.encode())
            elif payload['option'] == 'access':
                fname = payload['fname']
                r, s = payload['r'], payload['s']
                dt = ecc.verify(fname.encode(), r, s)
                if ('not exist' in dt) or ('not valid' in dt):
                    payload = json.dumps({'response': 'error', 'message': dt})
                else:
                    payload = json.dumps({'response': 'success', 'data': dt})
                req.sendall(payload.encode())
            else:
                payload = json.dumps(
                    {'response': 'error', 'message': 'Invalid option!'})
```

```
                req.sendall(payload.encode())
        except:
            payload = json.dumps(
                {'response': 'error', 'message': 'An error occured!'})
            req.sendall(payload.encode())


class incoming(socketserver.BaseRequestHandler):
    def handle(self):
        signal.alarm(30)
        req = self.request
        challenge(req)


class ReusableTCPServer(socketserver.ForkingMixIn, socketserver.TCPServer):
    pass


def main():
    init_storage()
    socketserver.TCPServer.allow_reuse_address = True
    server = ReusableTCPServer(("0.0.0.0", 1337), incoming)
    server.serve_forever()


if __name__ == "__main__":
    main()
```

So, this is an ECDSA challenge. Let's try to analysis the source code part by part.

```
def main():
    init_storage()
    socketserver.TCPServer.allow_reuse_address = True
    server = ReusableTCPServer(("0.0.0.0", 1337), incoming)
    server.serve_forever()
```

Okay, so this challenge will call `init_storage` and then turn up the server. Let's check what `init_storage` do

```
fnames = [b'subject_kolhen', b'subject_stommb', b'subject_danbeer']
nfnames = []
...
def init_storage():
    i = 0
    for fname in fnames[:-1]:
        data = ecc.sign(fname)
        r, s = data['r'], data['s']
        nonce = data['nonce']
        nfname = fname.decode() + '_' + r + '_' + s + '_' + nonce[(14 + i):-14]
        nfnames.append(nfname)
        i += 2
```

Okay, so from the defined filenames, `init_storage` will only sign **two files** (which is `subject_kolhen` and `subject_stommb`), and then store it with format `filename_r_s_nonce[(14+i):-14]`. (r, s) is the signature that is returned by ECDSA. Usually, in ECDSA scheme, we shouldn't shared our nonce or reused it, because knowning the nonce will allow us to derive the secret key. Let's try to remember how does ECDSA signature works first. We can sign a message with ECDSA by doing this:

$$s \equiv k^{-1}(h + r\alpha) \mod q \tag{1}$$

where, $s = \text{signature}$, $k = \text{nonce}$, $h = \text{hash(msg)}$, $\alpha = \text{secret}$.

Hence, if we know nonce, we can derive the secret by doing this:

$$\alpha \equiv (ks - h)r^{-1} \mod q \tag{2}$$

That's why we shouldn't share the nonce that we use during signing a message. This challenge leaked the middle bits of the nonce (First filename leaks ~144 middle bits, second filename leaks ~136 middle bits). We will keep this in our mind first.

Let's check the challenge code.

```
def retrieve_file(fname):
    try:
        dt = open(fname, 'rb').read()
        return dt.hex()
    except:
        return 'The file does not exist!'


def challenge(req):
    req.sendall(b'This is a cloud storage service.\n' +
                b'You can list the files inside and also see their contents if your signatures are v

    while True:
        req.sendall(b'\nOptions:\n1.List files\n2.Access a file\n')
        try:
            payload = json.loads(req.recv(4096))
            if payload['option'] == 'list':
                payload = json.dumps(
                    {'response': 'success', 'files': nfnames})
                req.sendall(payload.encode())
            elif payload['option'] == 'access':
                fname = payload['fname']
                r, s = payload['r'], payload['s']
                dt = ecc.verify(fname.encode(), r, s)
                if ('not exist' in dt) or ('not valid' in dt):
                    payload = json.dumps({'response': 'error', 'message': dt})
                else:
                    payload = json.dumps({'response': 'success', 'data': dt})
                req.sendall(payload.encode())
            else:
                payload = json.dumps(
                    {'response': 'error', 'message': 'Invalid option!'})
                req.sendall(payload.encode())

        except:
            payload = json.dumps(
                {'response': 'error', 'message': 'An error occured!'})
            req.sendall(payload.encode())
```

Okay, so basically, we can send two commands:

- List files
- Access file

List files command will return two filenames with its signature, and access file will allow us to read the file's content only if we provide the correct signature. Remember that the challenge is actually have three files.

We can deduce that on this challenges, given,

- Two messages (filename)
- Two signatures of the messages
- Two leaked middle bits of the used nonce during signing each messages

we need to be able create a valid signature for the third filename, and then access its content.

Solution

We actually can represent the challenge problems into Hidden Number Problem which was introduced by Boneh and Venkatesan. There are a lot of existing papers those discussed about these problems (paper1, paper2, paper3). I choose to implement the first paper in order to solve this solutions (with some modifications). I will try to explain it.

Basically, given the leaked middle bits of the k, we can say that:

$$k_i = 2^{\ell_i} c_1 + a_1 + b_1 \tag{3}$$

where, $max(k_{bits}) = 256_{bits}$, $\ell_i = max(k_{bits}) - leaked_{i_{bits}}$, $a_i = 2^{56} leaked_i$, and b_i is the 56 lsb of the k_i.

So, in this case, ℓ_1 will be ~200 (because $leaked_{i_{bits}} = 144$) and ℓ_2 will be ~192 (because $leaked_{i_{bits}} = 136$). And c_1 will be $256 - 144 - 56 = 56$ bits, and c_2 will be $256 - 136 - 56 = 64$ bits.

Remember eq1 where,

$$s \equiv k^{-1}(h + r\alpha) \mod q$$

and we have two signatures

$$\begin{aligned} s_1 &\equiv k_1^{-1}(h_1 + r_1\alpha) \mod q \\ s_1 k_1 &\equiv h_1 + r_1\alpha \mod q \end{aligned} \tag{4}$$

$$\begin{aligned} s_2 &\equiv k_2^{-1}(h_2 + r_2\alpha) \mod q \\ s_2 k_2 &\equiv h_2 + r_2\alpha \mod q \end{aligned} \tag{5}$$

If we try to do elimination of α between the equations that we have by multiplying first equation with r_2 and second equation with r_1, we will get

$$
\begin{aligned}
r_2 s_1 k_1 - r_1 s_2 k_2 &\equiv r_2 h_1 + r_2 r_1 \alpha - (r_1 h_2 + r_1 r_2 \alpha) \mod q \\
r_2 s_1 k_1 - r_1 s_2 k_2 &\equiv r_2 h_1 - r_1 h_2 \mod q \\
k_1 - r_2^{-1} s_1^{-1} r_1 s_2 k_2 &\equiv s_1^{-1} h_1 - r_2^{-1} s_1^{-1} r_1 h_2 \mod q \\
k_1 - s_1^{-1} r_1 s_2 r_2^{-1} k_2 + s_1^{-1} r_1 r_2^{-1} h_2 - s_1^{-1} h_1 &\equiv 0 \mod q \\
k_1 + t k_2 + u &\equiv 0 \mod q
\end{aligned}
\tag{6}
$$

where $t = -s_1^{-1} r_1 s_2 r_2^{-1}$ and $u = s_1^{-1} r_1 r_2^{-1} h_2 - s_1^{-1} h_1$

Remember $\mathrm{eq}3$ where we actually can expand the k further. Putting it to $\mathrm{eq}6$.

$$
\begin{aligned}
2^{\ell_1} c_1 + a_1 + b_1 + t 2^{\ell_2} c_2 + t a_2 + t b_2 + u &\equiv 0 \mod q \\
b_1 + 2^{\ell_1} c_1 + t b_2 + t 2^{\ell_2} c_2 + a_1 + t a_2 + u &\equiv 0 \mod q \\
b_1 + 2^{\ell_1} c_1 + t b_2 + t 2^{\ell_2} c_2 + u' &\equiv 0 \mod q \\
b_1 + 2^{\ell_1} c_1 + t b_2 + t 2^{\ell_2} c_2 + u' &\equiv 0 \mod q
\end{aligned}
\tag{7}
$$

where $u' = a_1 + t a_2 + u$.

Now, notice that b_1, c_1, b_2, c_2 are small unknowns. Based on the paper, we can construct a lattice where the result will contains 4 linear equations with 4 unknowns. Now, below is the lattice that I use.

$$
B = \begin{bmatrix}
K & K2^{\ell_1} & Kt & Kt2^{\ell_2} & u' \\
0 & Kq & 0 & 0 & 0 \\
0 & 0 & Kq & 0 & 0 \\
0 & 0 & 0 & Kq & 0 \\
0 & 0 & 0 & 0 & q
\end{bmatrix}
$$

where for this challenge, I choose $K = 2^{56}$ (K is the expected bound for b_1, c_1, b_2, c_2). I know that c_2 value is expected to be larger than K (remember that c_2 bits length should be around 64), but this is enough for some cases (Refer to Extra Notes below).

Notes that the lattice is slightly different from the example in the paper, because on this challenge, the total leaked middle bits that we got on signature 1 and signature 2 is difference ($leaked_{1_{bits}} = 144$ and $leaked_{1_{bits}} = 136$). Hence, the ℓ_i value is difference.

Using this basis and reduce it with LLL, we will got matrix 5x5, where it contains vector

$$
v_i = (x_0 K b_1, x_1 K c_1, x_2 K b_2, x_3 K c_2, y_i)
$$

where $x_i K$ is the coefficients of the linear equation and y_i is the result of the linear equation.

Just like the paper said, the first 4 vector is actually a linear equation of 4 unknown variables. So the vector is basically represents:

$$
x_0 K b_1 + x_1 K c_1 + x_2 K b_2 + x_3 K c_2 - y_i = 0
$$

Solving the equations, we will be able to retrieve the k_1 and k_2 by putting the result into $\mathrm{eq3}$. Refer to $\mathrm{eq2}$, now we can recover the secret (α). After that, we can use the given ECDSA class in the source code to sign the third message `subject_danbeer`, and send it to the server. The server will give the content (which contains the flag) to us.

```
[x] Opening connection to 206.189.126.144 on port 31122
[x] Opening connection to 206.189.126.144 on port 31122: Trying 206.189.126.144
[+] Opening connection to 206.189.126.144 on port 31122: Done
nonce_1: 0xcb3bc4a67de9feed766e69efce7b9ef1e0a2abe5ecbfadf4dc9cd320bbadf9dd
nonce_2: 0x2f5fe486ae3873b7725acec1ed38e00547a82995d4a27b4447d255e0f17569e1
secret: 69249790299545585587231056003577146693561863701265868154554802906288159975090

Response:
Test subject - Danbeer

DEBUG_MSG - Starting Mind...

    What a life this is...
    I lost my only child.
    My home got destroyed by the army.
    They took everything from me and now I'm trapped inside a cloud server.
    I don't even know where my real body is.
    I remember the day that they captured me.
    It was 2 weeks after I lost my precious Klaus.
    I was in the Inn of the city, trying to find more information about the
    android graveyard planet.
    3 men jumped on me!
    I won't give up, I shouted!
    On the day that the suns of the Nim cluster are aligned, I will be there.
    Draeger won't wi...

DEBUG_MSG - Shutting Down...

Notes:
    The subject seems to be rebellious and dangerous for android usage
    HTB{y0u_4r3_th3_m4st3r_0f_LLL}


[*] Closed connection to 206.189.126.144 port 31122
```

> Flag: `HTB{y0u_4r3_th3_m4st3r_0f_LLL}`

Extra notes

The solution is unstable, which after testing in the local, I found out that we can only apply our constructed lattice only if the bits length of the c_2 is < 63 bits. So, during the challenge, what I do is simply restart the docker so many times, until the c_2 value is < 63 bits.

I believe one of the reason of this unstable result is due to the K value that I set is 2^{56}. I'm pretty sure that my lattice can be improved so that it can recover the nonce even though the c_2 bits length is between $63, 64$ bits. Also in the paper, it is stated that

> The determinant bounds guarantee that we will find one short lattice vector, but do not guarantee that we will find four short lattice vectors. For that, we rely on the heuristic that the reduced vectors of a random lattice are close to the same length.

So maybe, it is expected that we might need to do several attempts to break the given ECDSA system.

Full Script

I use sage to run this script

```python
from pwn import *
from Crypto.Util.number import *
import json
from hashlib import sha1
from ecdsa.ecdsa import generator_256, Public_key, Private_key, Signature
import binascii

def retrieve_file(fname):
    try:
        dt = open(fname, 'rb').read()
        return dt.hex()
    except:
        return 'The file does not exist!'

class ECDSA:
    def __init__(self, key=-1):
        self.G = generator_256
        self.n = self.G.order()
        if key == -1:
            self.key = randint(1, self.n - 1)
        else:
            self.key = key
        self.pubkey = Public_key(self.G, self.key * self.G)
        self.privkey = Private_key(self.pubkey, self.key)

    def sign(self, fname):
        h = sha1(fname).digest()
        nonce = randint(1, self.n - 1)
        sig = self.privkey.sign(bytes_to_long(h), nonce)
        return {"r": hex(sig.r)[2:], "s": hex(sig.s)[2:], "nonce": hex(nonce)[2:]}

    def verify(self, fname, r, s):
        h = bytes_to_long(sha1(fname).digest())
        r = int(r, 16)
        s = int(s, 16)
        sig = Signature(r, s)

        if self.pubkey.verifies(h, sig):
            return retrieve_file(fname)
        else:
            return 'Signature is not valid\n'

def list_files(conn):
    conn.sendlineafter(b'a file\n', b'{"option": "list"}')
    response = json.loads(conn.recvuntil(b'\n').strip())
    return response['files']

def access_file(conn, fname, r, s):
    payload = json.dumps({
        'option': 'access',
        'fname': fname,
        'r': r,
        's': s,
    })
    conn.sendlineafter(b'a file\n', payload.encode())
    response = conn.recvuntil(b'\n')
    return json.loads(response)

```

```python
url = 'localhost:1337'
host = url.split(':')[0]
port = int(url.split(':')[1])
conn = remote(host, port)
files = list_files(conn)

q = 115792089210356248762697446949407573529996955224135760342422259061068512044369

# Get first signature with its leaked nonce
fname1 = (files[0].split('_')[0] + '_' + files[0].split('_')[1]).encode()
h1 = bytes_to_long(sha1(fname1).digest())
r1 = int(files[0].split('_')[2], 16)
s1 = int(files[0].split('_')[3], 16)
leaked1 = int(files[0].split('_')[4], 16)
a1 = leaked1*(2**56)

# Get second signature with its leaked nonce
fname2 = (files[1].split('_')[0] + '_' + files[1].split('_')[1]).encode()
h2 = bytes_to_long(sha1(fname2).digest())
r2 = int(files[1].split('_')[2], 16)
s2 = int(files[1].split('_')[3], 16)
leaked2 = int(files[1].split('_')[4], 16)
a2 = leaked2*(2**56)

# Craft our lattice
k = 2**56
inv_s1 = int(inverse_mod(s1, q))
inv_r2 = int(inverse_mod(r2, q))
t = (-inv_s1*s2*r1*inv_r2) % q
u = (inv_s1*r1*h2*inv_r2 - inv_s1*h1) % q
uu = a1 + t*a2 + u
m = Matrix([
    [k, k*2**200, k*t, k*t*2**192, uu],
    [0, k*q, 0, 0, 0],
    [0, 0, k*q, 0, 0],
    [0, 0, 0, k*q, 0],
    [0, 0, 0, 0,   q],
])
m = m.LLL()

# Now, from the reduced basis, we try to solve the equations
new_m = []
res = []
for row in m[:-1]:
    real_row = []
    for val in row[:-1]:
        real_row.append(val / k)
    new_m.append(real_row)
    res.append(-row[-1])
new_m = Matrix(new_m)
res = vector(res)
ans = new_m.solve_right(res)

# Now, we can get the nonce
k1 = ans[0] + ans[1]*2**200 + a1
k2 = ans[2] + ans[3]*2**192 + a2
print('nonce_1:', hex(k1))
print('nonce_2:', hex(k2))

# From the retrieved nonce, we get the secret (key)
secret = ((k1 * s1 - h1) * inverse_mod(r1, q)) % q
print('secret:', secret)

```

```
# Sign 'subject_danbeer'
new_ecc = ECDSA(secret)
filename = b'subject_danbeer'
signature = new_ecc.sign(b'subject_danbeer')

# Send its signature to the server
resp = access_file(conn, filename.decode(), signature['r'], signature['s'])
print('\nResponse:')
print(binascii.unhexlify(resp['data']).decode())
```

Social Media

Follow me on twitter

Writeup htb CyberApocalypseCTF RSA ECDSA LLL HNP crypto 2022