HackTheBox Cyber Apocalypse 2026

HackTheBox's Cyber Apocalypse 2026, hosted online.

Challenges and writeups available here: https://github.com/hackthebox/cyber-apocalypse-ctf-2026

- Fractured Seal

Fractured Seal

Written by `@flaberpengu`, solved by `@flaberpengu` and `@cookerflipper29`. Also hosted at flaberpengu.me.

```
Category: Crypto
Difficulty: Easy
```

Description

One of the Registry's oldest key-scrolls survived the fall of Crownspire, though time and fire spared only fragments of its writing, and most in the vault dismissed it as useless. Caldrin didn't. She always said a seal doesn't have to be whole to still remember the door it once opened.

Provided Files

We are given three files: `encrypt.py`, `flag.enc`, and `fractured_seal.pem`. Below are `encrypt.py` and `fractured_seal.pem`.

```
# encrypt.py
from Crypto.PublicKey import RSA
from Crypto.Util.number import long_to_bytes, bytes_to_long, getPrime

p = getPrime(1024)
q = getPrime(1024)
n = p * q
e = 0x10001
d = pow(e, -1, (p-1)*(q-1))

m = bytes_to_long(open('flag.txt', 'rb').read())

open('seal.pem', 'wb').write(RSA.construct((n, e, d)).export_key())
open('flag.enc', 'wb').write(long_to_bytes(pow(m, e, n)))
```

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAjK59ahXlX7a+oF+jt5icukpGeNXXgQO4D3jPeLaGAupJm6a6
9PnWCf0W3QDhmCPxLYWSr1C4DvxP23UlvP8Lcfu+w/oIS0jDHkfnv+m1Qku/ii9w
ehy/iRNuUeaH88K4AA0XFvJoak5I0Igi5ksP/CsyGMRJbUe898eLZjYOYAoA2+fi
LjjfYDBHliyXrva55W9O2ie2SNZ2Q859kAb0IQj/DK3LuAHqtuE7HVm2KDydU0jj
iMFz5uhwcU5njXnHdZRfrCTaUWBLLxmE8PCbIuy8hJqhL+iD1OcwTS8Yo8xoTtoc
7HFFfXYNoKjnqgp+LOmLwQGQeO2Yo1Hb8SCVd**************************
************************************************************
************************************************************
************************************************************
************************************************************
************************************************************
************************************************************
************************************************************
**************************************************2msCgYEAwLGx
cJ7/YCgq9GPDS16cHPNZEmYrbSX+atzUBBO2jLYg0QbXfitTIHfU+55DqIxFQOcu
+CahrPQQROoZAAIPg0LdaGd+3R3/ri**********************************
************************************************************
*****************************   □l□
**************************   □□□ □ □
*****************************   l□ ~□
*****************************   □□f_, )□
************************************************************
************************************************************
************************************************************
**************************************************
-----END RSA PRIVATE KEY-----
```

From reading `encrypt.py`, we can see that this challenge focuses on RSA. As expected, we learn that `flag.enc` is the target flag encrypted with standard RSA, no padding. We can also infer that if `seal.pem` is meant to be the exported RSA key, then our provided file `fractured_seal.pem` is probably a "broken" or partial version, which is what we can see in the file.

Understanding what PEM is

PEM stands for Privacy-Enhanced Mail and from Wikipedia we learn that it is a way of storing a key (or certificate) by base64-encoding the DER serialisation of an ASN.1 structure. Here, DER stands for Distinguished Encoding Rules and is a standard for serialising ASN.1 structures into binary, and

ASN.1 (or Abstract Syntax Notation One) is a standard language for describing and creating structures, usually in cryptography. Effectively, a PEM file is a way to easily store and transfer cryptographic primitive structures in a human-readable way - in this case, it is storing an RSA private key.

Reading our (partial) PEM file

The first step to reading our file is by finding out the structure of the output. The Python code uses `pycryptodome`, and if we check their documentation for the RSA structure, the default `PKCS` is `1` (which is the case in the provided code). The docs also list the variables, which match RFC2313, which is where the ASN.1 structure for PKCS#1 RSA private key is given. It says:

```
An RSA private key shall have ASN.1 type RSAPrivateKey:

   RSAPrivateKey ::= SEQUENCE {
     version Version,
     modulus INTEGER, -- n
     publicExponent INTEGER, -- e
     privateExponent INTEGER, -- d
     prime1 INTEGER, -- p
     prime2 INTEGER, -- q
     exponent1 INTEGER, -- d mod (p-1)
     exponent2 INTEGER, -- d mod (q-1)
     coefficient INTEGER -- (inverse of q) mod p }

   Version ::= INTEGER

   The fields of type RSAPrivateKey have the following meanings:

        o    version is the version number, for compatibility
             with future revisions of this document. It shall
             be 0 for this version of the document.

        o    modulus is the modulus n.

        o    publicExponent is the public exponent e.

        o    privateExponent is the private exponent d.
```

```
        o    prime1 is the prime factor p of n.

        o    prime2 is the prime factor q of n.

        o    exponent1 is d mod (p-1).

        o    exponent2 is d mod (q-1).

        o    coefficient is the Chinese Remainder Theorem
             coefficient q-1 mod p.
```

So we have some of this, in base64, with some parts missing, including between sections. At least until towards the end with the cat graphic, we made an assumption that every asterisk represented a single hex character.

In order to more properly understand what parts of our provided partial PEM file were present and what was missing, we used the provided `encrypt.py` file to generate a new, complete PEM file for a new key, which we then used with a decoder to work side-by-side with the partial file. We get the following:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAmD+svTVRHDvA2GES0sKr4F9DVAR/WcMX7CQGl7HIhs4EJZg9
EQUd4bqiwvBwapMpmjLSuEPXUQBKP4GixlYFRdMtR/ArC4q6BY6arl5VYgfL38mP
M3+hZiuuIMhmiWE3pyMdrFhqImocr6h99pbR00jAXQDJisl4ztlZcTGdGK8mCAHs
3lniST4oWqHYhNDdLrzOcPnVwX72QGelbAO+ERk1nB6CZ5UxVX7vfBHNtVtaHBVJ
kEjRCxl4vntlq91kWxvE9CiHIVWGUVMVbeUZxI9zVCPdK5kPwvooOdMVlPUu/l48
L41PbzgI1YDNZYTzYo7064eY/Yn/2h+uBiTixwIDAQABAoIBABIu0J+MRZKcpExz
U38uVr6mQfE/1EHRJgNCzjLBvdQzuJUQ2II23Tm/Q38KnjJsPvS2iyoMmCOJaFSr
U9hiH6EWNns3+LwZJlxRirG2fHYHvlaMyPh6jrkYEGTD6lMP9EiWzg1uRca0a5aj
BeA0ZPzZ3Hd/B5L2uuHhXWhCZ6Ox3reUaAeFckzBiAUvJQ0lyH8uHOSfFt1QHls/
pq+EbcvBtIgkqpSPC8LN2KTUkkPEq4G2BvzLQG3tr8t98I9kTwNLG64e7u4RpjnN
X5SwO7Yv+K67RISpO7WWX0uFQpk/Oa2inM8Uhiqy8h+smo1LZLYqLbffhn7+dpFv
xBRWefkCgYEAmgE8cMinmJUenpjZmWrIHvIrkFs3ULH2ChwBdftqxDNG5nxGGsvQ
1F1JBZ6W9SnCYS0psAQ/74dpKFQZciMYna2ZLWrJmrKvNvHlzc8G+qq0bSXKVH9s
sn7EEE4zIBvJ72qY7Z1PF/W2wUGi58JviP0yoNKfc7Bb2b/BDCSLwiUCgYEA/RSz
YK2r8tOPeESSwyoIjvnKtPO2TdZtsCUeVFxhdiXsuiXKpNyjSR8DqRUkAUE5KeqV
4kLcU4axqgcXil6UniI9Dp3K9D/LOhkm6fHihh1/FpO+dd5BnKi32N0/iKwZlr8p
7sgZ3u2+OBY22ny4Q9k2p02Ovp3FupCQGgMBv3sCgYBN4BokHhNBwQY+xKQkRC72
FxeYY8fQ8mysKAVFF+GhXRyZKHZ6zl3HRXTLdqp3RqqYYTIXY87A3QR79Cr54G1u
Ln6qzLktlPWtHx1JuUqaLwAVwWpS2CyrBhc295Q6xFKU0p6KaWW5g/LqPrqV32hv
```

```
KToUwO0In0GTFrgT39ccUQKBgFRkeMR+ZV5eCfOS+lW2SJBU37Gjq2EdZgpc9lYv
UBiz4NLaE08SfCi//Np/QDhi5YsCvORsY0g/HDOUldcAXxK0XCPpnmoAc+Z1yLIu
hNkwMNb4gYSQjtSMtnHMpYkPyqCoMnV7qJTcNphBYB1PhaqqXj0/8Z/CwNLLriHe
atDNAoGAPqXkeVmiW7lIv/APTtpnbbCpMRWr4rNkllD0uat64LdglQrhP42dIZzS
LTLN17tnU2Ywh/JpL51WtuCcmsm5or7gIQlD2DsyV+z7tio0k4COBqmRzKw/UDmz
6cIebjC/iPYfEXumTMHRL2dAdQVCYsExUTmEEuyqUawvLgWyAlk=
-----END RSA PRIVATE KEY-----
```

We came across a useful online ASN.1 JavaScript Decoder, though you could also use the command line tool provided as part of OpenSSL if you wished. We found the online interface nice and easy to use, so worked with this. Pasting in our full key, we got the following hex dump:

```
30 82 04 A2 02 01 00 02  82 01 01 00 98 3F AC BD
35 51 1C 3B C0 D8 61 12  D2 C2 AB E0 5F 43 54 04
7F 59 C3 17 EC 24 06 97  B1 C8 86 CE 04 25 98 3D
11 05 1D E1 BA A2 C2 F0  70 6A 93 29 9A 32 D2 B8
43 D7 51 00 4A 3F 81 A2  C6 56 05 45 D3 2D 47 F0
… skipping 160 bytes …
2F 8D 4F 6F 38 08 D5 80  CD 65 84 F3 62 8E F4 EB
87 98 FD 89 FF DA 1F AE  06 24 E2 C7 02 03 01 00
01 02 82 01 00 12 2E D0  9F 8C 45 92 9C A4 4C 73
53 7F 2E 56 BE A6 41 F1  3F D4 41 D1 26 03 42 CE
32 C1 BD D4 33 B8 95 10  D8 82 36 DD 39 BF 43 7F
0A 9E 32 6C 3E F4 B6 8B  2A 0C 98 23 89 68 54 AB
53 D8 62 1F A1 16 36 7B  37 F8 BC 19 26 5C 51 8A
… skipping 160 bytes …
AC 9A 8D 4B 64 B6 2A 2D  B7 DF 86 7E FE 76 91 6F
C4 14 56 79 F9 02 81 81  00 9A 01 3C 70 C8 A7 98
95 1E 9E 98 D9 99 6A C8  1E F2 2B 90 5B 37 50 B1
F6 0A 1C 01 75 FB 6A C4  33 46 E6 7C 46 1A CB D0
D4 5D 49 05 9E 96 F5 29  C2 61 2D 29 B0 04 3F EF
87 69 28 54 19 72 23 18  9D AD 99 2D 6A C9 9A B2
AF 36 F1 E5 CD CF 06 FA  AA B4 6D 25 CA 54 7F 6C
B2 7E C4 10 4E 33 20 1B  C9 EF 6A 98 ED 9D 4F 17
F5 B6 C1 41 A2 E7 C2 6F  88 FD 32 A0 D2 9F 73 B0
5B D9 BF C1 0C 24 8B C2  25 02 81 81 00 FD 14 B3
60 AD AB F2 D3 8F 78 44  92 C3 2A 08 8E F9 CA B4
F3 B6 4D D6 6D B0 25 1E  54 5C 61 76 25 EC BA 25
CA A4 DC A3 49 1F 03 A9  15 24 01 41 39 29 EA 95
E2 42 DC 53 86 B1 AA 07  17 8A 5E 94 9E 22 3D 0E
```

```
9D CA F4 3F CB 3A 19 26  E9 F1 E2 86 1D 7F 16 93
BE 75 DE 41 9C A8 B7 D8  DD 3F 88 AC 19 96 BF 29
EE C8 19 DE ED BE 38 16  36 DA 7C B8 43 D9 36 A7
4D 8E BE 9D C5 BA 90 90  1A 03 01 BF 7B 02 81 80
4D E0 1A 24 1E 13 41 C1  06 3E C4 A4 24 44 2E F6
17 17 98 63 C7 D0 F2 6C  AC 28 05 45 17 E1 A1 5D
1C 99 28 76 7A CE 5D C7  45 74 CB 76 AA 77 46 AA
98 61 32 17 63 CE C0 DD  04 7B F4 2A F9 E0 6D 6E
2E 7E AA CC B9 2D 94 F5  AD 1F 1D 49 B9 4A 9A 2F
00 15 C1 6A 52 D8 2C AB  06 17 36 F7 94 3A C4 52
94 D2 9E 8A 69 65 B9 83  F2 EA 3E BA 95 DF 68 6F
29 3A 14 C0 ED 08 9F 41  93 16 B8 13 DF D7 1C 51
02 81 80 54 64 78 C4 7E  65 5E 5E 09 F3 92 FA 55
B6 48 90 54 DF B1 A3 AB  61 1D 66 0A 5C F6 56 2F
50 18 B3 E0 D2 DA 13 4F  12 7C 28 BF FC DA 7F 40
38 62 E5 8B 02 BC E4 6C  63 48 3F 1C 33 94 95 D7
00 5F 12 B4 5C 23 E9 9E  6A 00 73 E6 75 C8 B2 2E
84 D9 30 30 D6 F8 81 84  90 8E D4 8C B6 71 CC A5
89 0F CA A0 A8 32 75 7B  A8 94 DC 36 98 41 60 1D
4F 85 AA AA 5E 3D 3F F1  9F C2 C0 D2 CB AE 21 DE
6A D0 CD 02 81 80 3E A5  E4 79 59 A2 5B B9 48 BF
F0 0F 4E DA 67 6D B0 A9  31 15 AB E2 B3 64 96 50
F4 B9 AB 7A E0 B7 60 95  0A E1 3F 8D 9D 21 9C D2
2D 32 CD D7 BB 67 53 66  30 87 F2 69 2F 9D 56 B6
E0 9C 9A C9 B9 A2 BE E0  21 09 43 D8 3B 32 57 EC
FB B6 2A 34 93 80 8E 06  A9 91 CC AC 3F 50 39 B3
E9 C2 1E 6E 30 BF 88 F6  1F 11 7B A6 4C C1 D1 2F
67 40 75 05 42 62 C1 31  51 39 84 12 EC AA 51 AC
2F 2E 05 B2 02 59
```

which decodes to

```
RSAPrivateKey SEQUENCE (9 elem)
  version Version INTEGER 0
  modulus INTEGER (2048 bit) 19219621550291068719218081674499601195559721340472645393653138314824O…
  publicExponent INTEGER 65537
  privateExponent INTEGER (2045 bit) 22953747939961883343045901592429982842920173174219051146088944197953 4…
  prime1 INTEGER (1024 bit) 10814586833191694236942933831042218231564371133499834383935705309944 2…
  prime2 INTEGER (1024 bit) 17771942513146207795872951094393961955443180520978279384764897197571 9…
  exponent1 INTEGER (1023 bit) 54685964821699611976790031151981188060796688796280652377073908474839 2…
  exponent2 INTEGER (1023 bit) 59262406225841467441446430751619031169302114394229109918771390719095 6…
  coefficient INTEGER (1022 bit) 43992933964454103940135611979598013252989131444310501009866723241166 0…
```

I used CyberChef to decode the base64 provided in our partial PEM, pasted this into VS Code along with the same number of asterisks, and then used the full key's hex dump section view to split out

the partial dump. If you do this, using the same delimiters as the online decoder lists, you get the following split of the file:

```
308204a3 = length (1187 bytes)
020100 = version (0)
02820101 = starting integer (modulus)
008cae7d6a15e55fb6bea05fa3b7989cba4a4678d5d78103b80f78cf78b68602ea499ba6baf4f9d609fd16dd00e198
23f12d8592af50b80efc4fdb7525bcff0b71fbbec3fa084b48c31e47e7bfe9b5424bbf8a2f707a1cbf89136e51e687
f3c2b8000d1716f2686a4e48d08822e64b0ffc2b3218c4496d47bcf7c78b66360e600a00dbe7e22e38df603047962c
97aef6b9e56f4eda27b648d67643ce7d9006f42108ff0cadcbb801eab6e13b1d59b6283c9d5348e388c173e6e87071
4e678d79c775945fac24da51604b2f1984f0f09b22ecbc849aa12fe883d4e7304d2f18a3cc684eda1cec71457d760d
a0a8e7aa0a7e2ce98bc1019078ed98a351dbf12095** = n
********** = e
********************************************************************************************
********************************************************************************************
********************************************************************************************
********************************************************************************************
********************************************************************************************
*******************************************da6b = d
02818100c0b1b1709eff60282af463c34b5e9c1cf35912662b6d25fe6adcd40413b68cb620d106d77e2b532077d4fb
9e43a88c4540e72ef826a1acf41044ea1900020f8342dd68677edd1dffae**********************************
************************************************************************** = p
********************************************************************************************
********************************************************************************************
************************************************************************** = q
********************************************************************************************
********************************************************************************************
***********************************
```

We can see that we are provided with all of `n` except the final byte, the final two bytes of `d`, and ~50% of `p`. We also know `e` from the Python code - it is `65537`. At this point, we recognised that we are dealing with a classic Coppersmith attack with partial known `p`.

Coppersmith to retrieve the key

Coppersmith's attack is a class of attacks on RSA cryptosystems that use the Coppersmith method underneath. This writeup won't try to explain the maths behind this as it is quite involved, but what is useful to know for us is that they use lattice-based techniques to recover private keys based on partial information. It is also useful for us to know that there exist multiple implementations in

Python (specifically in Sage) that save us a lot of time now that we have figured out what information we actually have. We opted to use [jvdsn's crypto-attacks] as the backbone for our solve script.

The main things we need to handle in this solve script are:

- We don't have all of `n` - we are missing the final byte. So we will need to try all possible final bytes until one works.
- We need to use the `PartialInteger` type, so we need to be certain on how many bytes we have.
- Our retrieved partial `p` contains the delimiting bytes `02818100`, so we should not include that in our `p`.

With this in mind, we end up with the script `factorise.py`:

```
import sys
sys.path.insert(0, "/path/to/crypto-attacks")

from shared.partial_integer import PartialInteger
from attacks.factorization.coppersmith import factorize_p

e = 65537

## n_known_hex is n with the last byte unknown
## n_prefix is shifted hex to have space for final byte when iterating all options
n_known_hex =
"008cae7d6a15e55fb6bea05fa3b7989cba4a4678d5d78103b80f78cf78b68602ea499ba6baf4f9d609fd16dd00e19
823f12d8592af50b80efc4fdb7525bcff0b71fbbec3fa084b48c31e47e7bfe9b5424bbf8a2f707a1cbf89136e51e68
7f3c2b8000d1716f2686a4e48d08822e64b0ffc2b3218c4496d47bcf7c78b66360e600a00dbe7e22e38df603047962
c97aef6b9e56f4eda27b648d67643ce7d9006f42108ff0cadcbb801eab6e13b1d59b6283c9d5348e388c173e6e8707
14e678d79c775945fac24da51604b2f1984f0f09b22ecbc849aa12fe883d4e7304d2f18a3cc684eda1cec71457d760
da0a8e7aa0a7e2ce98bc1019078ed98a351dbf12095"
n_shifted = int(n_known_hex, 16) << 8

p_hex =
("c0b1b1709eff60282af463c34b5e9c1cf35912662b6d25fe6adcd40413b68cb620d106d77e2b532077d4fb9e43a8
8c4540e72ef826a1acf41044ea1900020f8342dd68677edd1dffae" + "?" * 110)

partial_p = PartialInteger.from_hex_be(p_hex)
assert partial_p.bit_length == 1024

for last_byte in range(256):
```

```
    N = n_shifted + last_byte
    print(f"trying byte {last_byte:#04x}")
    result = factorize_p(N, partial_p)
    if result:
        p, q = result
        print("FOUND:", last_byte, p, q)
        break
```

where `/path/to/crypto-attacks` is the location at which you `git clone https://github.com/jvdsn/crypto-attacks` - we're just adding it to the path temporarily, though you could easily just add it permanently.

In defining `p_hex`, we use `? * 110` - from the `PartialInteger` definition, it uses `?` to mark an unknown nibble, so we require two per missing byte. From the original `encrypt.py`, we're working with 1024-bit primes (= 128 bytes), and we have 73 bytes of `p`, so we need 55 unknown bytes, hence 110 `?` characters.

Install SageMath (and activate it if necessary) and run `sage -python factorise.py`. It should return

```
FOUND: 117
p:
135314408378842790751605878050931209066067635249717498350882274491410611188834198512433542860977980915267615830180088553515126808474341043736459805810824761780913391116795622398843468715298669440308270490800437972694168370812052926427757058006436117836843232703533488083597801200757836873180405061962240493471
q:
131245497686548195467002565571865142560301544570020751232142961289012921376593159901052103730440250925007836457109179886547836486011666467584064970241380093385446276958000532368427648668071765509981171770398020329521298809677306252117144794197756373244812597628909948298104892770922899165466399775874596133483
```

after a few seconds. So for our `n` with last byte `117` (or `0x75`), we get `p` and `q` via Coppersmith as above. We made a second script (that doesn't require Sage but does use `pycryptodome` - getting these to work together threw errors for some reason) called `decrypt.py` that simply decrypts the flag:

```
from Crypto.Util.number import long_to_bytes, bytes_to_long


p =
135314408378842790751605878050931209066067635249717498350882274491410611188834198512433542860977980915267615830180088553515126808474341043736459805810824761780913391116795622398843468715298669440308270490800437972694168370812052926427757058006436117836843232703533488083597801200757
```

```
836873180405061962240493471
q =
131245497686548195467002565571865142560301544570020751232142961289012921376593159901052103730 4
402509250078364571091798865478364860116664675840649702413800933854462769580005323684276486680 7
176550998117177039802032952129880967730625211714479419775637324481259762890994829810489277092 2
899165466399775874596133483
n = p * q
e = 0x10001
d = pow(e, -1, (p-1)*(q-1))

c = bytes_to_long(open('flag.enc', 'rb').read())
m = pow(c, d, n)
print(f"m: {long_to_bytes(m)}")
```

Finally, we get the flag `HTB{r3c0v3r1ng_RSA_k3ys___l1k3___Me0w___me0o00o0o0w___Me0w}`.

Official Writeup

HackTheBox have also released their official writeup, which differs slightly from our findings. We are unsure on how they split their data to get what they got, since the provided partial PEM file certainly misses a byte of `n` and there are not enough redacted bytes between the sections of hex to skip `p` and line up for `q`, and they also seem to think none of `d` is leaked. However, the broad idea is the same and we got the flag anyway so it's fine.