Introduction

First thing, the description of this challenge is

> You can prove you know the flag to anybody without revealing any information thanks to my To-the-moon circuit! Do you want to look at the constraints ?

This didn't help much at first. There was a file attached with extension `.r1cs`, something we never saw before. After a quick search we found out that R1CS stands for "Rank 1 Constraint System" and it is related to "Zero knowledge proofs".

- What is a zero knowledge proof?

  A zero knowledge proof is a method that allows proving a statement without disclosing any information other than the statement truth. In particular, there's a *prover* and a *verifier*: the prover has to prove the statement, while the verifier has to verify it is true. The prover has some information that the verifier doesn't have, but they can use a zero knowledge proof protocol to prove their statement without giving any additional information to the verifier. In practice, the most common calls of protocols for using zero knowledge proof in crypto is zk-SNARK.
- What is a rank 1 constraint system?

  A Rank 1 Constraint System is an essential component of zk-SNARK.

  It is a system of equations in $\mathbb{R}[x_1, \dots, x_n]$, where $x_1, \dots, x_n$ are the unknowns and each equation is of the form $hk = l$, where $h, k, l \in \mathbb{R}[x_1, \dots, x_n]$ and can have degree at most 1 (they are monomials in some unknown $x_1, \dots, x_n$ plus a constant).

  For each equation of the type $hk = l$, we can rewrite h, k, l as a linear combination of these elements $\{1, x_1, \dots, x_n\}$, we call the vector $w = (1, x_1, \dots, x_n)$ witness. So, if we have m equations of the type $h_i k_i = l_i$, where $i \in \{1, \dots, m\}$, there exist one unique linear combination $h_i = a_{i1}w_1 + a_{i2}w_2 + \dots + a_{1(n+1)}w_{n+1}$, so we can define a $m \times (n+1)$ matrix A with values a_{ij}. We can do the same for the values k_i, getting another $m \times (n+1)$ matrix B with coefficients b_{ij} and again we can define a matrix C with coefficients c_{ij} for the values l_i. Overall you can rewrite the whole system as following:

  $$Aw \cdot Bw = Cw$$

  where w is a vertical vector and $\cdot$ is the elementwise multiplication.
- Back to the challenge

  Recall the challenge description: we have to "prove we have the flag without showing the flag"; that means "solve this system, a solution is the flag". To solve such a system we have to find the witness vector.

Parsing

We found the specification of `.r1cs` files online [1] and put together a simple script to parse it:

```python
from pathlib import Path
import sys


get_int = lambda x: int.from_bytes(x, "little")

field_size = None
prime = None
nWires = None
nPubOut = None
nPubIn = None
nPrvIn = None
```

```python
nLabels = None
mConstraints = None
A = []
B = []
C = []
labelIDs = []


def parse_r1cs(path: str):
    with open(path, mode="br") as fs:
        byt = fs.read()

    if byt[0:4] != b"\x72\x31\x63\x73":
        print(f"File {path} is not correct r1cs format")
    else:
        print(f"Parsing file {path}")

    if byt[4:8] != b"\x01\x00\x00\x00":
        print(f"Sorry, I don't know this version")
    else:
        print(f"Version 1")

    parse_version_1(byt[8:])


def parse_version_1(or_byt: bytes):
    sections_n = get_int(or_byt[:4])
    byt = or_byt[4:]
    for _ in range(sections_n):
        section_type = get_int(byt[0:4])
        section_size = get_int(byt[4:12])
        byt = byt[12:]
        if section_type == 1:
            print("1")
            parse_header(byt[:section_size])
            break
        byt = byt[section_size:]

    byt = or_byt[4:]
    for _ in range(sections_n):
        section_type = get_int(byt[0:4])
        section_size = get_int(byt[4:12])
        byt = byt[12:]

        match section_type:
            case 2:
                parse_constraint(byt[:section_size])
            case 3:
                parse_wire2labelid_map(byt, section_size)
            case _:
                print(f"Ignoring section type {section_type}")

        byt = byt[section_size:]

    if byt != b"":
        print(
            f"Beware, there's some surplus bytes that don't belong to a section:\n{byt}"
        )


def parse_header(byt: bytes):
    global field_size, prime, nWires, nPubOut, nPubIn, nPrvIn, nLabels, mConstraints
    field_size = get_int(byt[:4])
    prime = get_int(byt[4 : field_size + 4])
    byt = byt[field_size + 4 :]
    nWires = get_int(byt[:4])
    nPubOut = get_int(byt[4:8])
    nPubIn = get_int(byt[8:12])
    nPrvIn = get_int(byt[12:16])
    nLabels = get_int(byt[16:24])
    mConstraints = get_int(byt[24:28])

    if byt[28:] != b"":
        print("WTF")
        exit()
    return


def parse_constraint(byt: bytes):
    global A, B, C

    for k in range(mConstraints):
        for i in range(3):  # A, B and C
            tmp_M = []
            nFactors = get_int(byt[:4])
            byt = byt[4:]

            for j in range(nWires):
                w_id = get_int(byt[:4])
                if w_id != j:
                    tmp_M.append(0)
                    continue
                val = get_int(byt[4 : 4 + field_size])
                tmp_M.append(val)
                byt = byt[4 + field_size :]

            match i:
                case 0:
                    A.append(tmp_M)
                case 1:
                    B.append(tmp_M)
                case 2:
                    C.append(tmp_M)


def parse_wire2labelid_map(byt: bytes, section_size: int):
    global labelIDs

    while byt != b"":
        labelIDs.append(byt[:8])
        byt = byt[8:]


if __name__ == "__main__":
    parse_r1cs(sys.argv[1])
    print(f"{A = }, {B = }, {C = }")
    print(f"{prime = }")
```

With this, we now have

```
A = [
  [0, 21888242871839275222246405745257275088548364400416034343698204186575808495616]
]
B = [
  [0, 1]
]
C = [
  [7358504996770508486687187130827958137520805565857056985433965719766776637594, 0]
]
prime = 21888242871839275222246405745257275088548364400416034343698204186575808495617
```

Solve

A, B, C have just one row and 2 columns, which means there's just one equation and the witness vector has 2 elements (therefore there's only one unknown). There's quite a bit of zero values, so let's simplify the equations and write them explicitly:

$$(a_{11} a_{12}) \begin{pmatrix} 1 \\ x_1 \end{pmatrix} \cdot (b_{11} b_{12}) \begin{pmatrix} 1 \\ x_1 \end{pmatrix} = (c_{11} c_{12}) \begin{pmatrix} 1 \\ x_1 \end{pmatrix} \iff$$

$$\iff (a_{11} + a_{12}x_1)(b_{11} + b_{12}x_1) = c_{11} + c_{12}x_1$$

We also know that $a_{11} = 0,\ b_{11} = 0,\ b_{12} = 1,\ c_{12} = 0$

$$\iff a_{12}x_1^2 = c_{11}\,.$$

At this point the answer looks trivial, but there's a small detail that I didn't mention. With parsing we also found a prime number. That means that all the equations that I previously described in the form of $hk = l$ are actually in this form $hk = l \mod p$. The actual equation to solve is

$$a_{12}x_1^2 = c_{11} \mod p$$

Since we are in a prime field, every element is invertible

$$x_1^2 = a_{12}^{-1}c_{11} \mod p$$

To solve a square root mod p we can use the Tonelli-Shanks algorithm that ~~is~~ executes in polynomial time (because p is prime). We get one possible value of x_1, let it be x_1', then we calculate $x_1'' = -x_1' = p - x_1'$ (mod p of course), so we have the two possible values that solve the equation

```
from Crypto.Util.number import inverse
from sympy.ntheory import sqrt_mod

A = [[0, 21888242871839275222246405745257275088548364400416034343698204186575808495616]]
B = [[0, 1]]
C = [[7358504996770508486687187130827958137520805565857056985433965719766776637594, 0]]

p = 21888242871839275222246405745257275088548364400416034343698204186575808495617

h = (inverse(A[0][1], p) * C[0][0]) % p
x_1 = sqrt_mod(h, p)
```

We can print them as bytes

```
print(long_to_bytes(x_1))
print(long_to_bytes(p - x_1))
```

And sure enough, one of them is the flag

```
b'INS{Nothing_to_hide_in_R1CS!!!}'
b'0\x1b\x00\x1fe\xe30\xb5O\xe6\xd7O"\x0c\xe8\xfd\xbf\xca\x83\xe3\x1aP\x021\xf1\xb0\xb2@\xce\xde
\xde\x84'
```

Here it is!

---

[1] https://github.com/iden3/r1csfile/blob/master/doc/r1cs_bin_format.md