# Based Emoji

## Task

* Category: Van Halen Radiation Belt
* Points: 128
* Solves: 28

Description:

🫠

🕳

💻😀

🧮😥

🇾 🇴 🇺 ⬛ 🇲 🇮 🇬 🇭 🇹 ⬛ 🇼 🇦 🇳 🇹 ⬛ 🇹 🇴 ⬛ 🇺 🇸 🇪 ⬛ 🇦 ⬛ 🇹 🇪 🇷 🇲 🇮 🇳 🇦 🇱 ⬛ 🇹 🇭 🇦 🇹 ⬛ 🇭 🇦 🇸 ⬛ 🇪 🇲 🇴 🇯 🇮 🇸

👍 🆗 🌈 💖

Upon connecting to the challenge and providing our ticket we are greeted with this rather cursed challenge:

❗ ❗ ❗ 🇹 🇭 🇮 🇸 🇮 🇸 🇦 🇹 🇪 🇸 🇹 🇴 🇫 🇹 🇭 🇪 🇪 🇲 🇴 🇯 🇮 🇧 🇷 🇴 🇦 🇩 🇨 🇦 🇸 🇹 🇸 🇾 🇸 🇹 🇪 🇲 ❗ ❗ ❗   
🇼 🇪 🇦 🇷 🇪 🇺 🇸 🇮 🇳 🇬 🇧 🇦 🇸 🇪 ➖ 3⃣ 2⃣ 🇪 🇲 🇴 🇯 🇮 🇪 🇳 🇨 🇴 🇩 🇮 🇳 🇬   
🇹 🇭 🇦 🇹 🇲 🇪 🇦 🇳 🇸 🇪 🇦 🇨 🇭 🇪 🇲 🇴 🇯 🇮 🇷 🇪 🇵 🇷 🇪 🇸 🇪 🇳 🇹 🇸 5⃣ 🇧 🇮 🇹 🇸   
🇧 🇪 🇱 🇴 🇼 🇮 🇸 🇹 🇭 🇪 🇮 🇳 🇫 🇴 🇷 🇲 🇦 🇹 🇮 🇴 🇳 🇳 🇪 🇪 🇩 🇪 🇩 🇹 🇴 🇸 🇴 🇱 🇻 🇪 🇹 🇭 🇪 🇨 🇭 🇦 🇱 🇱 🇪 🇳 🇬 🇪   
🇳 🟰 📡🤯🎉🤟🌛👄🌝🚀🧨🤟🎉🥵🌟💾🚀🧂🌝🗿🙌🌞🚀🌛🚀🧠🛸💵🌝🥃🛰🔥🔥🍌🤟💵🎉🦍💽👽👄🌎🧨💥👄👽💥📼🥵🙌🦍📼🧂🌎🛸🧠🍌📼🧠🥃👄🤯🛰🤯💽🧂🌎🍌🦍👄🧂👽💵💽🌝🔥🌟💥🌞💾💵💥🌝🧂🌞🌟🧂🌞👄🧠🧂🧠🛰🌞📼👀🌛🛸🌞🛰🔥📡📡🧨🛰🛰🛰🤟👽💥🛰🥃🦍🌝💽🍌🌎💾🗿💯🤯💵🧠🗿🍌🥵🛰🌎🔥🌝🧂🧨🎉🚀👄🛸🤯🤯🤟🎉🛰🌞🌎💵🌝👄🌟🌎💽🌝💵📡🌛🤯🛸📼💯📼🧂🎉🛰🥵🧨👄💵🌝🌟💩🤟🙌🛰👽🧨🥵🚀🧂🌛🥃🍌🗿🥵🌎💥🥵📡🙌🌝🤟💽👽🧠🧂🙌💵🌝🌛💽🌛🍌🧨🌎🎉🚀🦍💽💽🚀🦍🛸🧂🤯🎉💽💵🤟👄💥🚀🦍💯🎉🎉💾🌞📼👄🌞💽👽🧠🤯👀📡🥵💩🔥🌛🥵🧠💽🚀🤯🙌📼💥🚀🧨🌎🤯🥵💩🛰🦍🌟📡🌎👽🥵🗿🛰📡💽🧠💾🌎🌛🥃👽📼🛸💵💽🛸🗿📼👄🧠🙌💾🤯🎉📡💯🎉🎉🌎🧠🤟💽🧂🌎🌟🦍🌛💾🥃🙌🧂🤯🛸🙌🧨🤟🎉🌛🥃🦍💵🦍🌎🛰🌟🔥👄🥵💵📼🌝🛰💾📡🌟🥃🍌💥🧨🤯👄💽🧂🦍🌎🌟📼💯🌎💵🌟🚀🌞🥃📼🧂💾🎉🤟🤟🛰🌛🛸🛰👀🌛🧨📡📼🌛💾📼🧨🎉💯🧨🌞💩📡👀💥🧠💾🎉🌞📡🌎🌝🎉👄🛰👽💥🌞💵🌎🛸🥃🍌💽🛸💯👽🔥💥🌝🙌🥃🌛🌝🥃👽🛰🛸🌞🌟💩🤯🤯🥃🧂🤟🤯🎉👄  
🇪 🟰 🥃🎉🎉🛰  
  
🇭 🇪 🇷 🇪 🇦 🇷 🇪 🇦 🇫 🇪 🇼 🆓 🐝🐝🐝 🇹 🇴 🇬 🇪 🇹 🇾 🇴 🇺 🇸 🇹 🇦 🇷 🇹 🇪 🇩  
0⃣ 🟰 🎉  
1⃣ 🟰 🛰  
2⃣ 🟰 🥃  
🇹 🇴 🇬 🇪 🇹 🇹 🇭 🇪 🚩 🇾 🇴 🇺 🇲 🇺 🇸 🇹 🇩 🇪 🇨 🇷 🇾 🇵 🇹 🇹 🇭 🇪 🚩  
🔐 🚩 👇  
🇨 🟰 🥃🥵🥃📡💾🙌💵🌎🙌🚀🧠💾💽👄🛸🚀🙌💾🔥🥵💥🛰👀🥵🌝🧠👽🛸🛸🌛🛰👄🙌📼💵🌞🙌🌎💥💥📼🤯💯🤟💥🔥🧂📼🛸💯👽🌞👀🌝🙌🌟📡👀🗿💥🌎🚀📼🌟🦍🦍🛸👽🧂🥵💾🧠🧠👽🤟🛸🎉🥃🙌🛰📡🤟💵👀🦍🔥👽🍌🛰🧠🗿🧠💾🌛🌞🧂📡🌞💾🔥🎉🦍🔥🤟📼💯🌎🤯🎉💥🌛🍌🛰💯🎉👄🧠🌝👽🙌🧠🧂🦍🧂💽📼🎉🌎🥃💾👄🍌🎉📡🌟🛸🤯💵🌞💾🚀🌟🌛🛸🧠💵👀🍌🤟🌛🌛🌝👄💩🌟🤟🌎🦍🤟🙌💯👀🥃🌝🔥💯💾🔥💥🔥🎉🚀🌟🌝🙌🌝🤟🙌🌝📡👽🧠🦍💩🥵👀🎉🥃🌟🤯💯🥃🎉🧨🤯🗿👽🧨🛰📼🛸🧂💵🤟📡🌎📡👄🌎🙌🌞💽🥃🍌🌎🧂🌟🗿💾💯🌞👀🌎🔥👄🙌💾🧠💥💽🧨🎉🗿🌛💾🌛👽🛰🥃🧨🤟🌝🥵🌞🤟🎉📼🌝👽👀💯🦍🤟🧨🧨👽🥃🌝🌟📼👽🌞🍌🧠💥🔥🔥🥃🥃📡📼💵🧠🌟🥃🌛📡💾🍌💥🙌💥👄💯🧨💽🚀📡🥃🥵💽📼💽🗿🗿🌎🗿👄👽🌝👽🤯🙌🥵🧠💥🌛🌞🦍🌛🛸🙌💾🛸🗿💥👄👄🌎🔥👄📡🥃🍌🧂🤯🤟💽🚀🙌🛸📡🧨🥵🛸💾🛸🌛🎉🧨🚀🌝💵💥🔥💥💵📡🌞🌎🤟🥵🍌💯💵🧂🧠🌟🔥🎉📡👄🤟🌛📼🛰🌛🧂🙌🚀🌞🙌🥃🍌💩🥵🗿🍌🛸🌝🦍🙌🎉🔥💯🌝👀💾🥵🌞🦍🌞🛰🍌🍌🦍💯🥃🌝👽🌝🦍🧠👄🥃🌛💵💵💾🤯  
🇲 🇪 🇳 🇺   
🇵 🇷 🇪 🇸 🇸 1⃣ 🇹 🇴 🇬 🇪 🇹 🇹 🇭 🇪 🇫 🇦 🇨 🇹 🇴 🇷 🇸 🇴 🇫 🇦 🇳 🇪 🇲 🇴 🇯 🇮   
🇵 🇷 🇪 🇸 🇸 2⃣ 🇹 🇴 🇬 🇪 🇹 🇦 🇷 🇦 🇳 🇩 🇴 🇲 🇵 🇷 🇮 🇲 🇪 🇧 🇾 🇧 🇮 🇹 🇸  
🇵 🇷 🇪 🇸 🇸 3⃣ 🇹 🇴 🇬 🇪 🇹 🇹 🇭 🇪 🇫 🇦 🇨 🇹 🇴 🇷 🇸 🇴 🇫 🇦 🇷 🇦 🇳 🇩 🇴 🇲 🇳 🇺 🇲 🇧 🇪 🇷 🇹 🇭 🇦 🇹 🇮 🇸 🇦 🇹 🇴 🇻 🇪 🇷 5⃣ 🇧 🇮 🇹 🇸 🇮 🇳 🇸 🇮 🇿 🇪   
🇵 🇷 🇪 🇸 🇸 🇦 🇳 🇾 🇹 🇭 🇮 🇳 🇬 🇪 🇱 🇸 🇪 🇹 🇴 🇪 🇽 🇮 🇹

## Solution

The challenge consists of reversing the base32 encoding. We can do so by utilising the three options giving to us below.

### Inherent constraints

The easiest way to solve this is using z3 as we found, in order to do so we need to build as many constraints as we can, so even before having a look at the functions we can add constraints for all emojis to encode unique numbers in base32 (5 bits).

io, n, e, c = connect()  
emojis = set(n)  
z3Vars = {x: z3.BitVec(x, 5) for x in emojis}  
assert (len(z3Vars) == 32)  
  
s = z3.Solver()  
  
# Known values  
for k, v in cachedMappings.items():  
 s.add(z3Vars[k] == v)  
  
# All are known to be different  
for k1, v1 in z3Vars.items():  
 for k2, v2 in z3Vars.items():  
 if k1 == k2:  
 continue  
 s.add(v1 != v2)

### Factors

Using the first provided method we can easily get all the factors and add constraints for them:

def get\_factors(io, emoji):  
 io.sendlineafter("🇪 🇽 🇮 🇹".encode("UTF-8"), "1⃣".encode("UTF-8"))  
 io.sendlineafter("🇫 🇦 🇨 🇹 🇴 🇷 🇸 🇴 🇫 ❓".encode(  
 "UTF-8"), emoji.encode("UTF-8"))  
 io.recvline()  
 return io.recvline().strip().decode("UTF-8").split(" ")  
  
# Collect all factors  
for k, v in z3Vars.items():  
 if k == ZERO:  
 continue  
  
 if k in cachedFactors:  
 f = cachedFactors[k]  
 else:  
 f = get\_factors(io, k)  
 cachedFactors[k] = f  
 print(f"Factors ({len(cachedFactors)}/31): {k} => {f}")  
  
 # Factor contraints  
 for x in f:  
 s.add(z3.URem(v, z3Vars[x]) == 0)

### Primes

The second method can be utilised to obtain a list of all primes, this in turn can be used to add contraints restricting the n-bit primes to a specific set of numbers, as we know all n-bit primes.

def get\_prime(io, bits):  
 io.sendlineafter("🇪 🇽 🇮 🇹".encode("UTF-8"), "2⃣".encode("UTF-8"))  
 io.sendlineafter("🇧 🇮 🇹 🇸 ❓".encode("UTF-8"),  
 str(bits).encode("UTF-8") + b"\xe2\x83\xa3")  
 return io.recvline\_contains("🟰".encode("UTF-8")).strip().decode(  
 "UTF-8").split("🟰")[1].lstrip()  
  
# Find all primes  
for i in range(2, 6):  
 hits = cachedPrimeHits[i] if i in cachedPrimeHits else set()  
 while len(hits) < len(primes[i]):  
 prime = get\_prime(io, i)  
 if prime in hits:  
 continue  
  
 hits.add(prime)  
 print(f"{i}-bit prime ({len(hits)}/{len(primes[i])}): {prime}")  
 s.add(z3.Or([z3Vars[prime] == x for x in primes[i]]))

### Primes, again..

Getting the model count for this unfortunately still yields loads of results. Luckily for us there is still a third method, which allows us to get the factors of a random n-bit number, whereas 6 <= n <= 9. This can be used to build a multitude of constraints, such as more factorisation based ones etc, but an easy one is just finding new primes.

def get\_rand\_factors(io, bits):  
 io.sendlineafter("🇪 🇽 🇮 🇹".encode("UTF-8"), "3⃣".encode("UTF-8"))  
 io.sendlineafter("🇧 🇮 🇹 🇸 ❓".encode("UTF-8"),  
 str(bits).encode("UTF-8") + b"\xe2\x83\xa3")  
 io.recvline()  
 return io.recvline().strip().decode("UTF-8").split(" ")  
  
# Grab top bits of numbers and primes >5 bit  
hits = set()  
limit = 100  
for i in range(6, 10):  
 primeHits = cachedPrimeHits[i] if i in cachedPrimeHits else set()  
 while len(hits) < (2 \*\* (i - 5)) - 1:  
 f = get\_rand\_factors(io, i)  
 if len(f) == 2:  
 for x in f:  
 if x == ONE:  
 continue  
  
 cVar = z3.Concat(\*[z3Vars[y] for y in x])  
 if x in primeHits:  
 continue  
  
 primeHits.add(x)  
 print(  
 f"{i}-bit prime ({len(primeHits)}/{len(primes[i])}): {cVar}")  
 s.add(z3.Or(\*[cVar == x for x in primes[i]]))  
  
 try:  
 models = get\_all\_models(s, limit)  
 print("Possible solutions: ", len(models))  
 if len(models) == 1:  
 break  
 except:  
 print(f"Possible solutions: >{limit}")  
  
 for x in f:  
 if len(x) != 2:  
 continue  
  
 x = x[0]  
 if x in hits:  
 continue  
  
 hits.add(x)  
 print(f"{i - 5}-bit int ({len(hits)}/{(2 \*\* (i - 5)) - 1}): {x}")  
 s.add(z3.ULT(z3Vars[x], 2\*\*(i - 5)))  
 s.add(z3.UGE(z3Vars[x], 2\*\*(i - 6)))  
 else:  
 if len(primeHits) > len(cachedPrimeHits.get(i, [])):  
 print(f"cachedPrimeHits[{i}] = ", end="")  
 pprint(primeHits)  
   
 continue  
 break

### Decrypting the flag

Turns out the above methods result in only one viable model, so now we’re able to decode anything encoded using *emoji*-base32. What really threw us of here was that the flag was labelled as c, which is commonly used for the *ciphertext* in RSA and N didn’t seem to be weak at all. After some toying around we finally tried decrypting using e and it turned out the flag was actually **signed** using the private key, **not encrypted**.

def decrypt\_with\_model(emojis, vars, model, n, e, c):  
 mapping = {k: model[vars[k]].as\_long() for k in emojis}  
 n = parse\_with\_mapping(n, mapping)  
 e = parse\_with\_mapping(e, mapping)  
 c = parse\_with\_mapping(c, mapping)  
 return long\_to\_bytes(pow(c, e, n))[::-1]

Flag: flag{hotel475384oscar4:GLI7XUJ06xDF8ebbGFiZi5jmfuDVjK02ME07iN-p7Ms3W39eQssXC370rd0imSQKf3HWRWuwjfabRflrS29Y\_WQ}