SKR CTF Write-up

Name / Nickname : Ariff / Rydzze

Challenge Category : Reverse Engineering

Challenge Name : Cr4ck M3!

SKR User / Team : [rydzze / Sternritter]

Challenge : Cr4ck M3!

📚 Overview

Crack me if you can!

🤔 Hint

➢ *"I wonder if I can solve this with math."*

✨ Solution

```
from z3 import *
import string

s = Solver()
F = [BitVec(i, 8) for i in range(24)]

COF = [0x6f, 0x23, 0x60, 0x73, 0xfd  #and so on]
SUM = [0x1260, 0xd6c3, 0x9964        #and so on]

for c in F:
    s.add(Or([c == ord(char) for char in string.printable[:-6]]))

for i in range(len(COF)):
    signed_integer = (int(COF[i]) + 2**7) % 2**8 - 2**7
    COF[i] = signed_integer

for i in range(24):
    s.add(SUM[i] == sum(COF[24 * i + j] * F[j] for j in range(24)) & 0xFFFF)

flag = ""
if s.check() == sat:
    solution = s.model()
    for i in range(24):
        flag += chr(int(str(solution[F[i]])))
    print("The flag is SKR{" + flag + "}")
else:
    print("No flag for you 😢")
```

This challenge took me around 2 months to solve it. Debug with GDB, add 24 equations, watch CTF walkthroughs, and I decided to take a break a month ago. Then, a revelation come with those write-ups, and I able to recognise my mistake 🙏🏼 Thanks for the wisdom! `:D`

🛠️ Source Code Breakdown

```
from z3 import *
import string

s = Solver()
F = [BitVec(i, 8) for i in range(24)]

COF = [0x6f, 0x23, 0x60, 0x73, 0xfd  #and so on]
SUM = [0x1260, 0xd6c3, 0x9964        #and so on]
```

In this challenge, I solved it using Z3 Theorem Prover that can automate the process to get the flag by providing it with some constraints. Firstly, I declared a list called `F` using BitVec that creates a **bit-vector** with **a width of 8-bits** for each character in flag (since the range of **ASCII characters** = **0x00 – 0x7F**). Then, I obtained data for `COF` (*stand for coefficient I guess*) and `SUM` from the binary file.

```
for c in F:
    s.add(Or([c == ord(char) for char in string.printable[:-6]]))
```

Next, this **for loop** will adds constraints where **each character in the flag** will only **corresponds** to **printable ASCII characters**. During my attempts, I stumbled across this video (͡° ͜ʖ ͡°) that implemented the code snippet above. (*Thanks for the walkthrough!*)

```
for i in range(len(COF)):
    signed_integer = (int(COF[i]) + 2**7) % 2**8 - 2**7
    COF[i] = signed_integer
```

In the code snippet above, it will convert **every hexadecimal number** in `COF` into a **signed integer** (*This is why I couldn't solve the challenge back then*). How do I know that `COF` value is actually a signed value?

```
[─────────────────────────────────registers─────────────────────────────────]
RAX: 0x fd
RBX: 0x7fffffffded8 ──> 0x7fffffffe251 ("/home/kali/Downloads/crackme")
RCX: 0x4
RDX: 0xfffd
RSI: 0x4455 ('UD')
RDI: 0x7fffffffd800 ──> 0x5f00455441564952 ('RIVATE')
RBP: 0x7fffffffddc0 ──> 0x1
RSP: 0x7fffffffdd40 ──> 0x7fffffffded8 ──> 0x7fffffffe251 ("/home/kali/Downloads/crackme")
RIP: 0x555555555273 (<main+202>:        mov     eax,DWORD PTR [rbp-0x64])
[────────────────────────────────────code────────────────────────────────────]
   0x555555555269 <main+192>:   add    rax,rdx
   0x55555555526c <main+195>:   movzx  eax,BYTE PTR [rax]
   0x55555555526f <main+198>:   movsx  dx,al
=> 0x555555555273 <main+202>:   mov    eax,DWORD PTR [rbp-0x64]
   0x555555555276 <main+205>:   cdqe
   0x555555555278 <main+207>:   movzx  eax,BYTE PTR [rbp+rax*1-0x30]
   0x55555555527d <main+212>:   cbw
   0x55555555527f <main+214>:   imul   eax,edx
```

After **`movsx dx, al`** executed, the value **`0xFD`** become **`0xFFFD`**. The instruction **movsx** (move with sign-extend) will **sign-extend** the value **referring** to the **MSB** of the binary. In short, the binary **`1111 1101`** extended to **`1111 1111 1111 1101`** since **`AL`** is an **8-bit register** and **`DX`** is a **16-bit register**. The decimal value of **`0xFD`** in this context is -3, not 253.

```python
for i in range(24):
    s.add(SUM[i] == sum(COF[24 * i + j] * F[j] for j in range(24)) &
        0xFFFF)
```

Lastly, this **for loop** will generate, add **24 constraints** to the solver where each constraint:

- Have the **sum of 24 *products*** between **`COF`** and **`flagCharacter (F)`**.
- Perform a **bitwise AND operation** between the **sum** and **0xFFFF**.
- Then, it will be **equal to** the **`SUM`**. (*The value of SUM is 16-bit only, that's why we perform bitwise AND operation before, to accept only 16 LSB*)

After that, we let the code run and wait for the solution :).

```
┌──(kali㉿kali)-[~/Downloads]
└─$ python solver.py
The flag is SKR{M4th_1n_Cr@cK_m3_3azyPzz}

┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme
Enter password: M4th_1n_Cr@cK_m3_3azyPzz
Correct password! Flag is SKR{M4th_1n_Cr@cK_m3_3azyPzz}
```

Flag

Hence, the flag is indeed **`SKR{M4th_1n_Cr@cK_m3_3azyPzz}`**.