Challenge Name: Bootstrap Bill's Lock

Description:

A notorious band of digital pirates, confident in their cryptographic prowess, have locked away their treasure map with a custom piece of ransomware—daring any would-be treasure hunters to break their scheme....all that remains is a single encrypted file and the original malware binary—rumored to be the handiwork of the infamous Bootstrap Bill himself.

Files provided:

- blackpearl_curse (unstripped Go binary)
- whatsthis.jpg.enc (base64 encrypted flag image)

Part 1: Initial Recon

Firstly starting out with running strings for any hints, no flag present, nor typical CTF markers. However, you do see some pirate references; the flavor text like, "Ye map be cursed! The bravest pirates may yet break its spell..." and also references to Go standard library symbols, including crypto/des.NewTripleDESCipher, encodeBase64, and error/panic strings.

Next up,

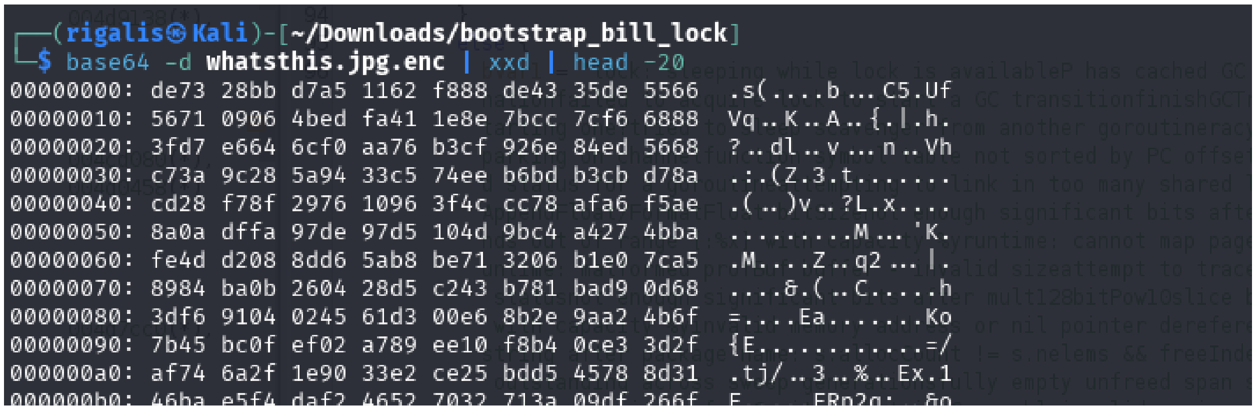
Running `nm blackpearl_curse`

And `ltrace ./blackpearl_curse`

Symbol table confirms the presence of named functions: main.main, main.encrypt, and more. ltrace traces typical file IO and Go runtime calls, but nothing is revealed, no secrets, keys, or syscalls are helpful.

Part 2: Artifact Triage

Before diving into the binary, it's better to understand the encrypted flag file's structure. Based on the challenge text, it's protected using some crypto.

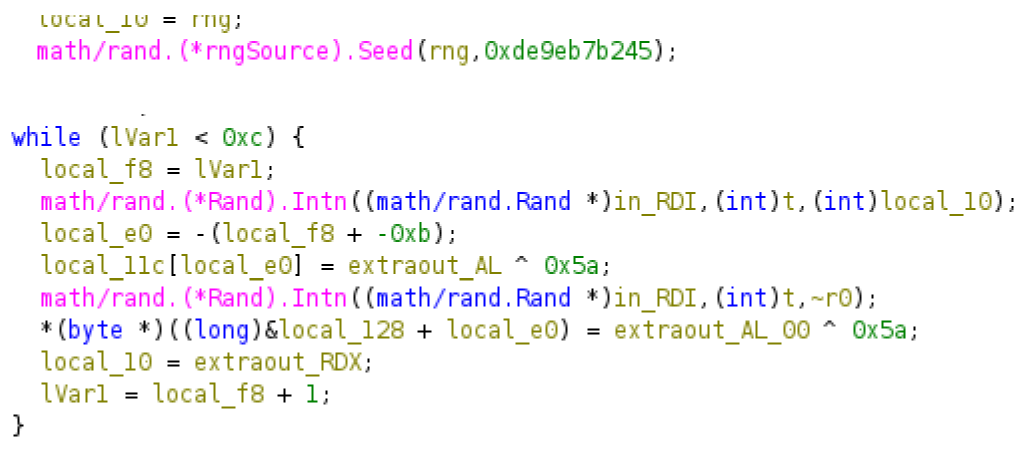


```
┌──(rigalis㉿Kali)-[~/Downloads/bootstrap_bill_lock]
└─$ base64 -d whatsthis.jpg.enc | xxd | head -20
00000000: de73 28bb d7a5 1162 f888 de43 35de 5566  .s(....b...C5.Uf
00000010: 5671 0906 4bed fa41 1e8e 7bcc 7cf6 6888  Vq..K..A..{.|.h.
00000020: 3fd7 e664 6cf0 aa76 b3cf 926e 84ed 5668  ?..dl..v...n..Vh
00000030: c73a 9c28 5a94 33c5 74ee b6bd b3cb d78a  .:.(Z.3.t.......
00000040: cd28 f78f 2976 1096 3f4c cc78 afa6 f5ae  .(..)v..?L.x....
00000050: 8a0a dffa 97de 97d5 104d 9bc4 a427 4bba  .........M...'K.
00000060: fe4d d208 8dd6 5ab8 be71 3206 b1e0 7ca5  .M....Z..q2...|.
00000070: 8984 ba0b 2604 28d5 c243 b781 bad9 0d68  ....&.(..C.....h
00000080: 3df6 9104 0245 61d3 00e6 8b2e 9aa2 4b6f  =....Ea.......Ko
00000090: 7b45 bc0f ef02 a789 ee10 f8b4 0ce3 3d2f  {E............=/
000000a0: af74 6a2f 1e90 33e2 ce25 bdd5 4578 8d31  .tj/..3..%..Ex.1
000000b0: 46ba e5f4 daf2 4652 7032 713a 09df 266f  F.....FRp2q:..&o
```

The first 8 bytes look random, not a JPEG file header (ffd8...). The rest are also non-printable. This is consistent with the pattern where a block cipher (possibly 3DES in CFB mode) encrypts a file and prepends the IV (Initialization Vector) to the ciphertext, then base64 encodes the whole buffer.

Part 3: Disassembly

Opening blackpearl_curse in Ghidra,since the binary is unstripped. You can easily locate the main entry at main.main, with helpers named main.encrypt, main.encodeBase64, and references to crypto/des.NewTripleDESCipher.



```
    local_10 = rng;
    math/rand.(*rngSource).Seed(rng,0xde9eb7b245);


  while (lVar1 < 0xc) {
    local_f8 = lVar1;
    math/rand.(*Rand).Intn((math/rand.Rand *)in_RDI,(int)t,(int)local_10);
    local_e0 = -(local_f8 + -0xb);
    local_11c[local_e0] = extraout_AL ^ 0x5a;
    math/rand.(*Rand).Intn((math/rand.Rand *)in_RDI,(int)t,~r0);
    *(byte *)((long)&local_128 + local_e0) = extraout_AL_00 ^ 0x5a;
    local_10 = extraout_RDX;
    lVar1 = local_f8 + 1;
  }
```

- The function initializes two 12-byte arrays (local_11c and local_128) with zeros and then enters a loop to fill each with pseudo-random bytes derived from Go's math/rand, XORed with 0x5a. The key part is that the filling happens in *reverse order* (end-to-front), as evidenced by:
- The random source is seeded with the pirate-themed constant (0xde9eb7b245).
- After filling, the two slices are concatenated to make a 24-byte key for TripleDES. *Any error in PRNG, XOR, or reversal will yield the wrong key*.

Ghidra Stack Frame Mapping

- local_11c at offset -0x11c: is your left slice, always written reversed.
- local_128 at offset -0x128: is your right slice.

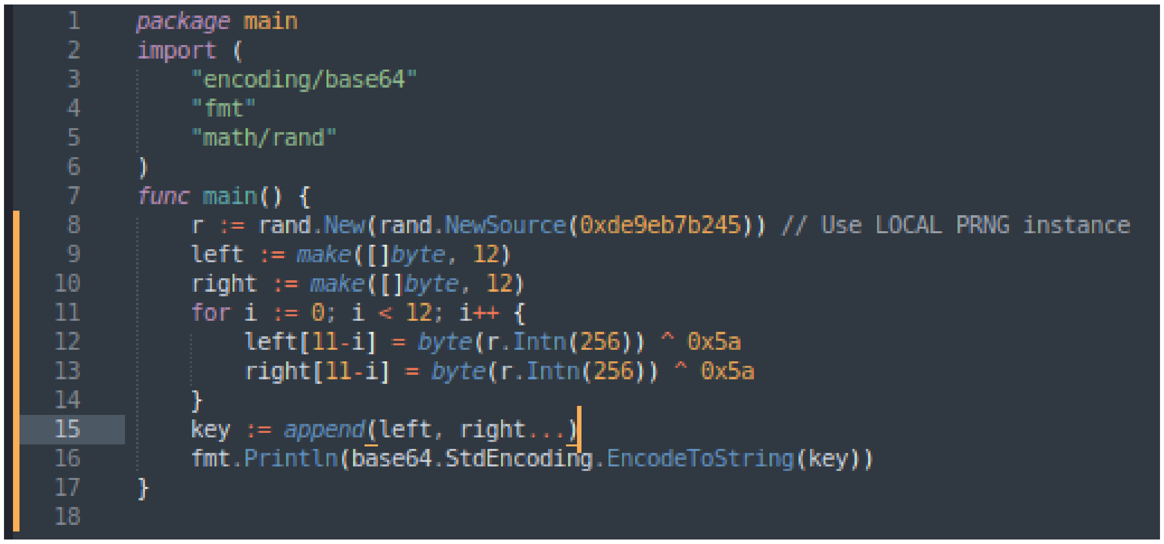
Part 4: Working Out the Key

After analyzing the key generation logic in main.main, you can replicated the process by seeding Go's math/rand PRNG with the extracted constant (0xde9eb7b245), filling two 12-byte slices in reverse order, each byte XOR'd with 0x5a, and concatenating the results to form the 24-byte 3DES key. **This exact process is critical for key correctness, due to an important language-level detail.**

Go's binary uses a local PRNG instance (r := rand.New(rand.NewSource(seed))), which means the random byte sequence is unique to Go's implementation of math/rand. Although the logic for generating the key appears simple and could be transcribed into Python or other languages, attempting to do so will always produce a different key: **Python's random module and Go's math/rand do *not* generate the same sequence, even with the same seed and operations.**

Therefore, to faithfully recover the correct key required for decryption, the keygen must be written and executed in Go, mirroring the binary's behavior. Only this approach yields the reproducible, deterministic base64 key matching the one used by the encrypting binary. With the correct Go keygen, every run produces PyWWS9K/pb/OwteTpc4E4BROtpD/VCKW—the only key that will decrypt the challenge artifact.

Keygen script



```go
package main
import (
    "encoding/base64"
    "fmt"
    "math/rand"
)
func main() {
    r := rand.New(rand.NewSource(0xde9eb7b245)) // Use LOCAL PRNG instance
    left := make([]byte, 12)
    right := make([]byte, 12)
    for i := 0; i < 12; i++ {
        left[11-i] = byte(r.Intn(256)) ^ 0x5a
        right[11-i] = byte(r.Intn(256)) ^ 0x5a
    }
    key := append(left, right...)
    fmt.Println(base64.StdEncoding.EncodeToString(key))
}
```

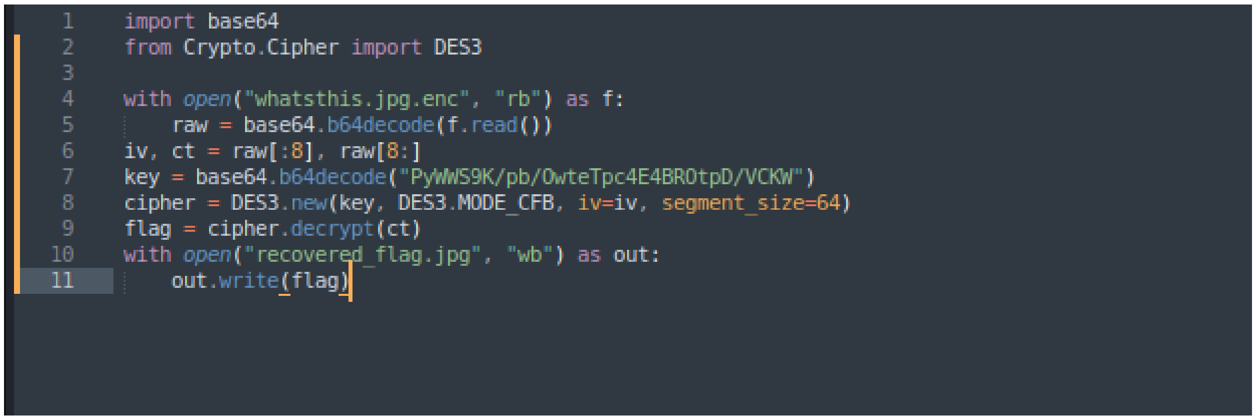
Once key is generated,

Part 5: Extracting IV and Decrypting

After base64-decoding your .enc file:

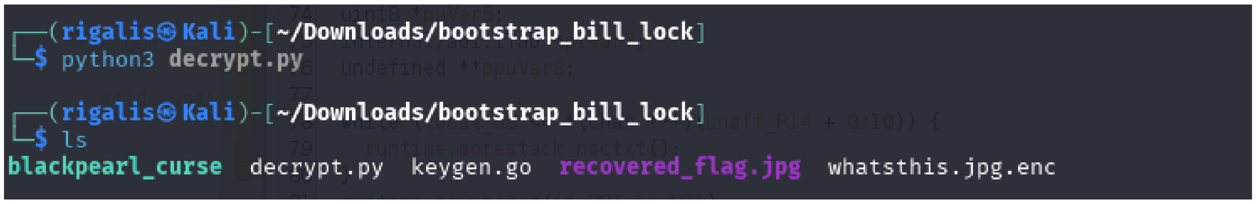
- The first 8 bytes (de 73 28 bb d7 a5 11 62) are your IV for CFB mode.
- The remainder is ciphertext.

Decryption Script



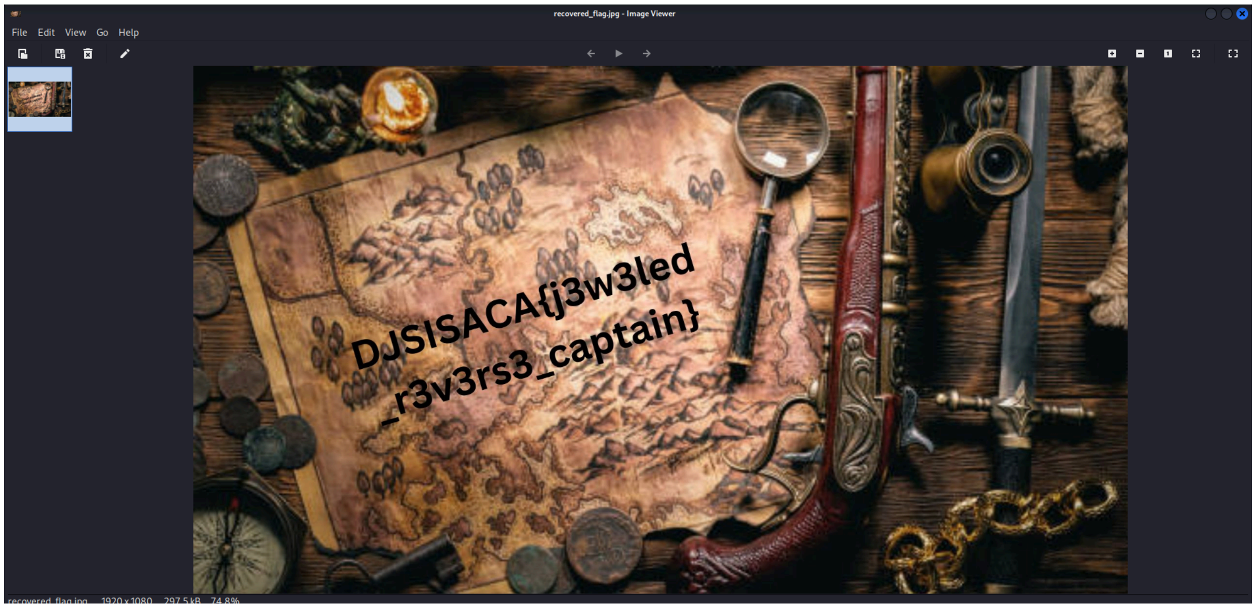
```python
import base64
from Crypto.Cipher import DES3

with open("whatsthis.jpg.enc", "rb") as f:
    raw = base64.b64decode(f.read())
iv, ct = raw[:8], raw[8:]
key = base64.b64decode("PyWWS9K/pb/OwteTpc4E4BROtpD/VCKW")
cipher = DES3.new(key, DES3.MODE_CFB, iv=iv, segment_size=64)
flag = cipher.decrypt(ct)
with open("recovered_flag.jpg", "wb") as out:
    out.write(flag)
```



```
┌──(rigalis㉿Kali)-[~/Downloads/bootstrap_bill_lock]
└─$ python3 decrypt.py

┌──(rigalis㉿Kali)-[~/Downloads/bootstrap_bill_lock]
└─$ ls
blackpearl_curse  decrypt.py  keygen.go  recovered_flag.jpg  whatsthis.jpg.enc
```


recovered_flag.jpg - Image Viewer
File Edit View Go Help
DJSISACA{j3w3led
_r3v3rs3_captain}
recovered_flag.jpg 1920 x 1080 297.5 kB 74.8%

What's this, that's your flag :D
FLAG: DJSISACA{j3w3led_r3v3rs3_captain}