Dissecting an Unknown Linux ELF - Static & Dynamic Analysis Report

Brief:

During a routine review of our network on 30 June 2022, we found the attached unknown binary on one of our Linux devices. It appears that the binary had just been run to create the following data: `6b5744775d5f38010b073e4325030e17`. The binary does not appear malicious, but we're not quite sure what it does.

Running the binary and supplying a 16-character input returns a transformed hexadecimal string:

```
master@myguy:~$ ./REcruitment
This is a flag encryption program, it makes flags safe. Enter your flag:
hellohellohelloh
flag: 5a505c5b5e5d0f0000006b160f030018
master@myguy:~$ ./REcruitment
This is a flag encryption program, it makes flags safe. Enter your flag:
hdhdhdksjdbskdbs
flag: 5a51585359510f0000006b00080b0d03
master@myguy:~$
```

Please review the binary and decode the data.

Tools Used

- **Radare2**: Utilised for static and dynamic binary analysis
- **Python**: Used to reverse the XOR encryption

Note: Although the brief recommended Ghidra and GDB, I chose Radare2 for its streamlined workflow (and my thorough experience with its comprehensive features). While I am also proficient with Ghidra and GDB, I find Radare2 more effective for simpler challenges that don't require extensive patching.

Please see the write-up on the next page.

The Write-Up

Using `radare2` we open up the binary in both `write` and `debug` mode. This allows us to both statically and dynamically analyse the binary, as well as patch it if necessary.

The command is `r2 -w -d ./REcruitment`

We then run `aa` to analyse the functions in the binary, and then `afl` to list them:

```
[master@myguy:~$ r2 -w -d ./REcruitment
 WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e bin.cache=true` next time
  -- Save your projects with 'Ps <project-filename>' and restore then with 'Po <project-filename>'
[[0x7c847f639540]> aa
 INFO: Analyze all flags starting with sym. and entry0 (aa)
 INFO: Analyze imports (af@@@i)
 INFO: Analyze entrypoint (af@ entry0)
 INFO: Analyze symbols (af@@@s)
 INFO: Recovering variables (afva@@@F)
 INFO: Analyze all functions arguments/locals (afva@@@F)
[[0x7c847f639540]> afl
 0x5b4db89b50e0    1     11 sym.imp.putchar
 0x5b4db89b50f0    1     11 sym.imp.localtime
 0x5b4db89b5100    1     11 sym.imp.puts
 0x5b4db89b5110    1     11 sym.imp.strlen
 0x5b4db89b5120    1     11 sym.imp.__stack_chk_fail
 0x5b4db89b5130    1     11 sym.imp.printf
 0x5b4db89b5140    1     11 sym.imp.uname
 0x5b4db89b5150    1     11 sym.imp.fgets
 0x5b4db89b5160    1     11 sym.imp.time
 0x5b4db89b5170    1     11 sym.imp.sprintf
 0x5b4db89b5180    1     46 entry0
 0x5b4db89b7fe0    1   4129 reloc.__libc_start_main
 0x5b4db89b5457    6    596 main
 0x5b4db89b5260    5     60 entry.init0
 0x5b4db89b5220    5     54 entry.fini0
 0x5b4db89b50d0    1     11 fcn.5b4db89b50d0
 0x5b4db89b51b0    4     34 fcn.5b4db89b51b0
```

There are a number of interesting function names here that give clues about the functionality of the binary, however we start with the main. You can run `s main` to 'seek' the main function and run `pdd` which will decompile the function (as long as you have r2dec installed). You can also run `pddf` which will decompile all the functions.

See the main function in radare2 on the following page:

```
/* name: main @ 0x1457 */
int32_t main (void) {
    time_t timer;
    tm* var_270h;
    int64_t var_268h;
    char * format;
    int64_t var_260h;
    int64_t var_25ch;
    int64_t var_258h;
    int64_t var_250h;
    int64_t var_248h;
    int64_t var_240h;
    int64_t var_230h;
    int64_t var_a0h;
    int64_t var_9ch;
    int64_t var_9ah;
    int64_t var_96h;
    int64_t var_94h;
    int64_t var_90h;
    int64_t var_8eh;
    int64_t var_8ah;
    int64_t var_88h;
    char * var_87h;
    char * s;
    int64_t var_78h;
    int64_t var_70h;
    int64_t var_60h;
    int64_t canary;
    rax = *(fs:0x28);
    *((rbp - 0x18)) = rax;
    eax = 0;
    *((rbp - 0x80)) = 0;
    *((rbp - 0x78)) = 0;
    *((rbp - 0x70)) = 0;
    puts ("This is a flag encryption program, it makes flags safe. Enter your flag:");
    rax = rbp - 0x80;
    fgets (rax, 0x11, *(obj.stdin));
    rax = rbp - 0x80;
    rax = strlen (rax);
    if (rax != 0x10) {
        puts ("Your flag must be 16 chars long for security or something");
        eax = 0;
    } else {
        *((rbp - 0x8e)) = 0;
        *((rbp - 0x8a)) = 0;
        *((rbp - 0x88)) = 0;
        *((rbp - 0xa0)) = 0;
        *((rbp - 0x9c)) = 0;
        *((rbp - 0x9a)) = 0;
        *((rbp - 0x96)) = 0;
        rax = rbp - 0x80;
        fcn_000012c8 (rax, rbp - 0x8e, rbp - 0xa0, rbp - 0x9a);
        *((rbp - 0x94)) = 0x6f6f6373;
        *((rbp - 0x90)) = 0x70;
        rax = rbp - 0x94;
        fcn_00001269 (rax, 5, rbp - 0x9a);
        rax = rbp - 0x230;
        rdi = rax;
        uname ();
        rax = rbp - 0x230;
        fcn_00001269 (rax, 5, rbp - 0xa0);
        rax = time (0);
        *((rbp - 0x278)) = rax;
        rax = rbp - 0x278;
        rax = localtime (rax);
        rcx = *(rax);
        rbx = *((rax + 8));
        *((rbp - 0x270)) = rcx;
        *((rbp - 0x268)) = rbx;
        rcx = *((rax + 0x10));
        rbx = *((rax + 0x18));
        *((rbp - 0x260)) = rcx;
        *((rbp - 0x258)) = rbx;
        rcx = *((rax + 0x20));
        rbx = *((rax + 0x28));
```

```
        *((rbp - 0x250)) = rcx;

        *((rbp - 0x248)) = rbx;
        rax = *((rax + 0x30));
        *((rbp - 0x240)) = rax;
        esi = *((rbp - 0x264));
        eax = *((rbp - 0x260));
        ecx = rax + 1;
        edx = *((rbp - 0x25c));
        rax = (int64_t) edx;
        rax *= 0x51eb851f;
        rax >>= 0x20;
        edi = eax;
        edi >>= 5;
        eax = edx;
        eax >>= 0x1f;
        edi -= eax;
        eax = edi;
        eax *= 0x64;
        edx -= eax;
        eax = edx;
        r8d = esi;
        eax = 0;
        sprintf (rbp - 0x87, "%d%02d%02d", eax);
        rax = rbp - 0x87;
        fcn_00001269 (rax, 6, rbp - 0x8e);
        rcx = rbp - 0x60;
        rax = rbp - 0x87;
        fcn_00001376 (rax, rcx, rbp - 0x94);
        eax = 0;
    }
    rbx = *((rbp - 0x18));
    rbx ^= *(fs:0x28);
    if (edx != 0) {
        stack_chk_fail ();
    }
    return rax;
}
```

Looking at the above pseudo C code, following the buffer and canary initialisation, we can see that the program uses `puts()` to ask for a flag in order to encrypt it. Following the request, it does the following:

```
rax = rbp - 0x80;
    fgets (rax, 0x11, *(obj.stdin));
    rax = rbp - 0x80;
    rax = strlen (rax);
```

It initialises memory space for `RAX` which is a register that holds return values for functions. Then `fgets()` is run which takes in user input (i.e. the flag). This value is then saved in `RAX`. `RAX` is then set to the beginning of the buffer of the string, and `strlen(rax)` is run to check the length of the data. The length is then saved to `RAX`.
It reads `0x11` bytes of data, which is `17` characters, however the 17th character would just be a null terminator.

```
if (rax != 0x10) {
        puts ("Your flag must be 16 chars long for security or
something");
        eax = 0;}
```

If the length is not equal to `16` characters (`0x10` in hex is 16), then `eax` is set to `0` and the program presumably exits. I tested this, and it works as expected.

```
else {

        *((rbp - 0x8e)) = 0;
        *((rbp - 0x8a)) = 0;
        *((rbp - 0x88)) = 0;
        *((rbp - 0xa0)) = 0;
        *((rbp - 0x9c)) = 0;
        *((rbp - 0x9a)) = 0;
        *((rbp - 0x96)) = 0;

        rax = rbp - 0x80; //RAX is reset to the original flag
        fcn_000012c8 (rax, rbp - 0x8e, rbp - 0xa0, rbp - 0x9a);
```

If the number of characters is indeed 16 characters, it then starts the encryption process. Initially it zeroes some local buffers, and then it calls `fcn.000012c8` to split the 16 bytes into three parts.

See the decompiled version of `fcn.000012c8` below, I've added some comments for readability:

```
int64_t fcn_000012c8 (char * arg1, int64_t arg2, int64_t arg3, int64_t
arg4) {

    int64_t var_30h;
    int64_t var_28h;
    int64_t var_20h;
    char * var_18h;
    signed int64_t var_4h;

    rdi = arg1; //RAX is passed as arg1 (ie the original flag)
    rsi = arg2;
    rdx = arg3;
    rcx = arg4;

    *((rbp - 0x18)) = rdi;
    *((rbp - 0x20)) = rsi;
    *((rbp - 0x28)) = rdx;
    *((rbp - 0x30)) = rcx;
    *((rbp - 4)) = 0; //initialise loop counter to 0

    while (*((rbp - 4)) <= 0xf) { //loops up to 0xf which is 15
        if (*((rbp - 4)) <= 5) { //for the first 5 bytes copy to arg2
            eax = *((rbp - 4));
            rdx = (int64_t) eax;
            rax = *((rbp - 0x18));
```

```
            rax += rdx;
            edx = *((rbp - 4));
            rcx = (int64_t) edx;
            rdx = *((rbp - 0x20));
            rdx += rcx;
            eax = *(rax);
            *(rdx) = al;

        } else {
            if (*((rbp - 4)) > 5) {
                if (*((rbp - 4)) <= 0xa) { // for bytes 6 to 0xa which is 10, copy to arg3
                    eax = *((rbp - 4));
                    rdx = (int64_t) eax;
                    rax = *((rbp - 0x18));
                    rax += rdx; //increments rax address to loop through
                    edx = *((rbp - 4));
                    rdx = (int64_t) edx;
                    rcx = rdx - 6;
                    rdx = *((rbp - 0x28));
                    rdx += rcx;
                    eax = *(rax);
                    *(rdx) = al;
                }
            } else { // copy the remaining (ie counter 11-15) to arg4
                eax = *((rbp - 4));
                rdx = (int64_t) eax;
                rax = *((rbp - 0x18));
                rax += rdx;
                edx = *((rbp - 4));
                rdx = (int64_t) edx;
                rcx = rdx - 0xb;
                rdx = *((rbp - 0x30));
                rdx += rcx;
                eax = *(rax);
                *(rdx) = al;
            }
        }
        *((rbp - 4))++;
    }
    return rax;
}
```

Once the function has been returned, we now have three parts. Part 1 has 6 bytes, part 2 has 5 bytes, and part 3 also has 5 bytes, making a total of 16 bytes.

We can see where the program stores these parts via the function call:

```
fcn_000012c8 (rax, rbp - 0x8e, rbp - 0xa0, rbp - 0x9a);
```

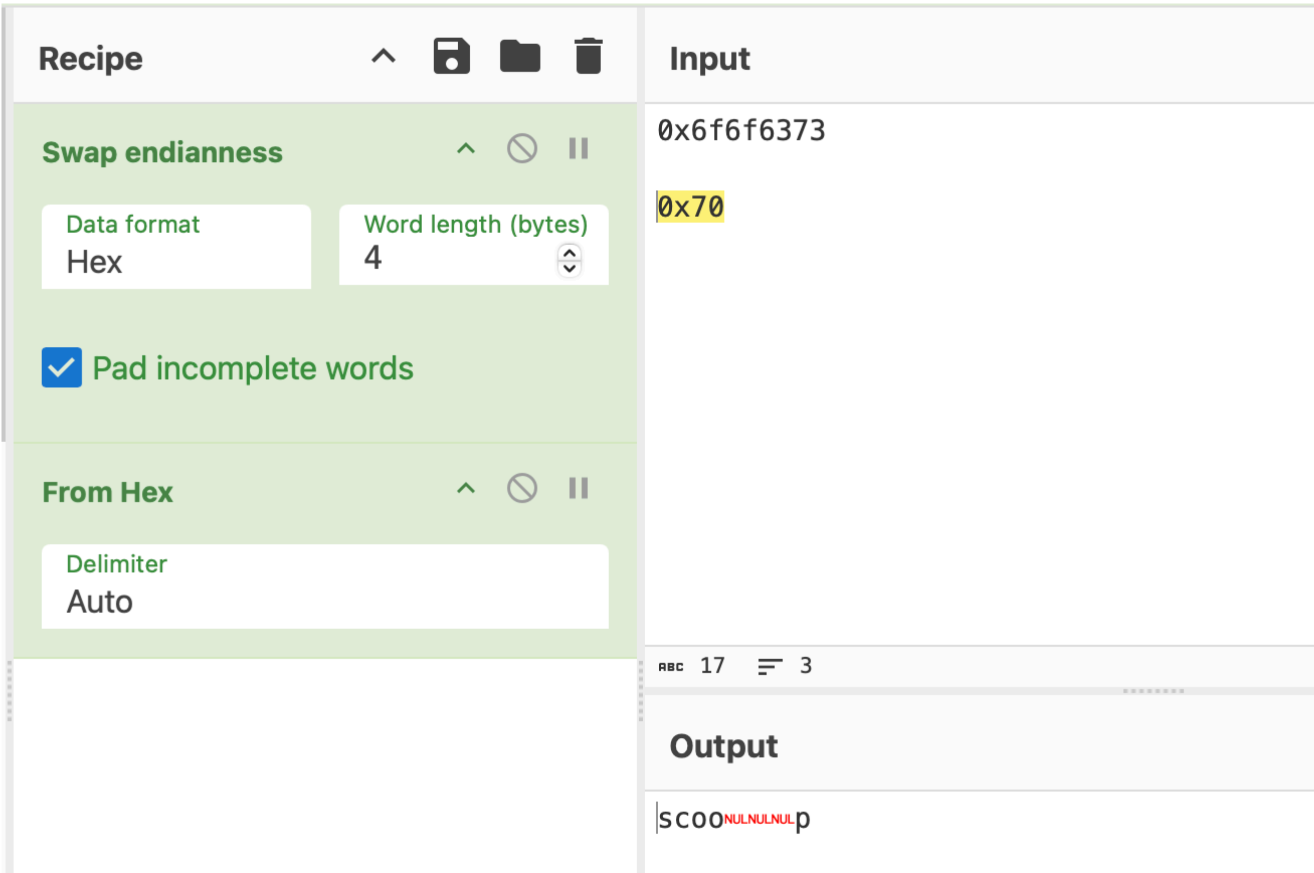
Part 1: `rbp - 0x8e`
Part 2: `rbp - 0xa0`
Part 3: `rbp - 0x9a`

Back to the main function, see the next part below:

```
*((rbp - 0x94)) = 0x6f6f6373;
*((rbp - 0x90)) = 0x70;

rax = rbp - 0x94;
fcn_00001269 (rax, 5, rbp - 0x9a);
```

`0x6f6f6373` and `0x70` are the string `scoop` in little endianness. We can quickly confirm this on cyberchef:



Therefore `rbp - 0x94` to `rbp - 0x90` now holds the string `scoop`.

We can also confirm this with live debugging on `radare2`. I set a breakpoint on `fcn_00001269` and when it hit I ran `px 6 @ rbp-0x94`:

```
[> px 6 @ rbp-0x94
- offset -       5C5D 5E5F 6061 6263 6465 6667 6869 6A6B  CDEF0123456789AB
0x7fffbc11525c  7363 6f6f 7000                           scoop.
```

As you can see, the string scoop is stored as expected with a null terminator to confirm the end of the string.

Then we see that the function `fcn_00001269 (rax, 5, rbp - 0x9a);` is called.

`RAX` passes in the string `scoop`. The length is set to 5, and `rbp - 0x9a` is part 3 of the flag.

This is the function implementation shown by r2dec, with my added comments to aid understanding:

```
/* name: fcn.00001269 @ 0x1269 */

int64_t fcn_00001269 (char * arg1, signed int64_t arg2, int64_t arg3) {

    int64_t var_28h;
    signed int64_t var_1ch;
    char * var_18h;
    int64_t var_4h;
    rdi = arg1;
    rsi = arg2;
    rdx = arg3;
    *((rbp - 0x18)) = rdi; //arg1
    *((rbp - 0x1c)) = esi; //arg2
    *((rbp - 0x28)) = rdx; //arg3
    *((rbp - 4)) = 0;

    while (eax < *((rbp - 0x1c))) { //loops up to arg2 (i.e. 5)
        eax = *((rbp - 4));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x18)); // loads 'scoop'
        rax += rdx;
        esi = *(rax);
        eax = *((rbp - 4));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x28)); //loads part 3 of the flag.
        rax += rdx; //incremements pointer to address to allow for looping
        ecx = *(rax);
        eax = *((rbp - 4));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x18));
        rax += rdx;
        esi ^= ecx; // bitwise XOR operation
        edx = esi;
        *(rax) = dl; //writes dl (lower 8 bits of EDX) back to arg1 (i.e
replaces 'scoop' with the XOR result)
        *((rbp - 4))++;
        eax = *((rbp - 4));
    }
```

```
    return rax;
}
```

So the function bitwise XORs each byte in `scoop` with each byte in part three of the flag. (This just means converting each to binary, and comparing each byte to ensure at least one of them is 1). It then stores the result in `rbp - 0x94` to `rbp - 0x90` which now holds `scoop` XOR `part 3`.

Going back to the main, this is the next part:

```
rax = rbp - 0x230;
rdi = rax;
uname ();
rax = rbp - 0x230;
fcn_00001269 (rax, 5, rbp - 0xa0);
```

This calls `uname()` which fills `RDI` (which points to `rbp - 0x230`). This returns strings related to the operating system. In the challenge, we're given a clue that the binary was found on `one of our Linux devices`. Given that `uname()` typically gives strings like `sysname` which is often `Linux`, we could possibly guess that the string being pointed to by the `RAX` register is `Linux`.

However, it's better to debug the program and check:

```
[> px 6 @ rbp-0x230
- offset -       C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x7fffbc1150c0  4c69 6e75 7800                            Linux.
```

As we can see the string is Linux (with a null terminator to confirm the end of the string). In order to get this, you should wait until the breakpoint at `fcn_00001269` is hit a **second** time, and then run `px 6 @ rbp-0x230`.

The function `fcn_00001269 (rax, 5, rbp - 0xa0);` is then called again but this time with different arguments. This time it XORs the string `Linux` (pointed to by RAX), and part 2 of the flag which is stored in `rbp - 0x230`. The result then replaces the string `Linux` just as it did previously with `scoop`.

Continuing with the main function:

```
rax = time (0); // get the current time in seconds since Unix epoch
*((rbp - 0x278)) = rax;
rax = rbp - 0x278;
rax = localtime (rax); //then converts it to a localtime struct
rcx = *(rax);
rbx = *((rax + 8));
*((rbp - 0x270)) = rcx;
*((rbp - 0x268)) = rbx;
rcx = *((rax + 0x10));
```

```
rbx = *((rax + 0x18));
*((rbp - 0x260)) = rcx; // store month
*((rbp - 0x258)) = rbx; // store day
rcx = *((rax + 0x20));
rbx = *((rax + 0x28));
*((rbp - 0x250)) = rcx;
*((rbp - 0x248)) = rbx; // store year
rax = *((rax + 0x30));
*((rbp - 0x240)) = rax;
esi = *((rbp - 0x264));
eax = *((rbp - 0x260));
ecx = rax + 1;
edx = *((rbp - 0x25c));
rax = (int64_t) edx;
rax *= 0x51eb851f; // multiply by magic number constant
rax >>= 0x20; // shift right by 32 bits (division part)
edi = eax;
edi >>= 5; // shift the answer of the division right by 5 bits (divide by
32)
eax = edx;
eax >>= 0x1f; // extract sign bit
edi -= eax;
eax = edi;
eax *= 0x64; // multiply by 100
edx -= eax; // get the remainder
eax = edx; // now holds remainder of the division used below to format the
time
r8d = esi;
eax = 0;
sprintf (rbp - 0x87, "%d%02d%02d", eax); //formats a date string into rbp
- 0x87
rax = rbp - 0x87;
fcn_00001269 (rax, 6, rbp - 0x8e); //passes date string to the XOR
function
```

The easiest way to confirm the exact date format is to view it during live debugging. I set a breakpoint on `fcn_00001269` (using `radare2`) and entered `dc` three times in order to continue the flow till we reach `fcn_00001269` for the **third** time. At this point, I simply inspect `rbp - 0x87` to view the exact date format:

```
[> px 7 @ rbp-0x87
- offset -       C9CA CBCC CDCE CFD0 D1D2 D3D4 D5D6 D7D8  9ABCDEF012345678
0x7ffc8ef5aec9   3235 3031 3233 00                        250123.
```

As we can see the date format is YYMMDD. This is important as it helps us to understand exactly how the data is XOR'd. The last byte is a null terminator ensuring we have the full date.

As before, `RAX` points to the string that is used to XOR the flag. It is passed in and this time bitwise XOR'd with part 1 of the flag stored in `rbp - 0x8e`. It returns and replaces the date with the XOR'd data in `rbp - 0x87`.

These are the addresses for the encrypted flag parts:
Part 1: `rbp - 0x94` (5 bytes)
Part 2: `rbp - 0x230` (5 bytes)
Part 3: `rbp - 0x87` (6 bytes)

All that is left now is to print them out!

```
rcx = rbp - 0x60;
rax = rbp - 0x87;
fcn_00001376 (rax, rcx, rbp - 0x94);
eax = 0;
```

`Part1` and `Part3` are passed to the function which prints them out, however the 5 bytes at `part2` are printed out from `rbp - 0x60`, but the encrypted bytes are held in `rbp - 0x230`. I suspect this could be an obfuscation technique to stop the flag being brute-forced perhaps, (but it could also be passed onto the function for printing in another way that I haven't seen). `Part3` is passed first, then `part2`, then `part1`. This means that the first 6 bytes of the flag are dependant on the correct time.

```
/* name: fcn.00001376 @ 0x1376 */
int64_t fcn_00001376 (char * arg1, int64_t arg2, int64_t arg3) {

    int64_t var_28h;
    int64_t var_20h;
    char * var_18h;
    signed int64_t var_ch;
    signed int64_t var_8h;
    signed int64_t var_4h;
    rdi = arg1;
    rsi = arg2;
    rdx = arg3;
    *((rbp - 0x18)) = rdi;
    *((rbp - 0x20)) = rsi;
    *((rbp - 0x28)) = rdx;
    eax = 0;
    printf ("flag:"); // print the hardcoded string "flag:"
    *((rbp - 0xc)) = 0;

    while (*((rbp - 0xc)) <= 5) { //prints the 6 bytes from arg1 in hex
        eax = *((rbp - 0xc));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x18));
        rax += rdx;
```

```
        eax = *(rax);
        eax = (int32_t) al;
        esi = eax;
        eax = 0;
        printf ("%02x"); //prints in 2 digit hex pattern
        *((rbp - 0xc))++;

    }

    *((rbp - 8)) = 0;
    while (*((rbp - 8)) <= 4) { //prints the 5 bytes from arg2 in hex

        eax = *((rbp - 8));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x20));
        rax += rdx;
        eax = *(rax);
        eax = (int32_t) al;
        esi = eax;
        eax = 0;
        printf ("%02x"); // prints in 2 digit hex pattern
        *((rbp - 8))++;

    }

    *((rbp - 4)) = 0;
    while (*((rbp - 4)) <= 4) { //prints the 5 bytes from arg3 in hex
        eax = *((rbp - 4));
        rdx = (int64_t) eax;
        rax = *((rbp - 0x28));
        rax += rdx;
        eax = *(rax);
        eax = (int32_t) al;
        esi = eax;
        eax = 0;
        printf ("%02x"); // prints in 2 digit hex pattern
        *((rbp - 4))++;

    }

    putchar (0xa);
    return rax;

}
```

The code above loops through the three encrypted flag parts and prints them out in hex, giving us our encrypted flag.

Now in order to reverse this process, we can write a python script:

```
#!/usr/bin/env python3

import binascii

def xor_bytes(data, key):

return bytes(a ^ b for a, b in zip(data, key)) # XOR each byte in the data with the corresponding byte in the key. Zip is used to pair the bytes together.

# The encrypted data in hex:

encrypted_hex = "6b5744775d5f38010b073e4325030e17"
encrypted_bytes = binascii.unhexlify(encrypted_hex)

dateStringKey = b"220630" # Matches the date format (YYMMDD)

unameStringKey = b"Linux" # Matches the uname output

scoopStringKey = b"scoop" # Matches the scoop string

# Split the flag into 6 bytes, 5 bytes, and 5 bytes
part1 = encrypted_bytes[0:6]
part2 = encrypted_bytes[6:11]
part3 = encrypted_bytes[11:16]

# XOR each part with its corresponding key
decoded1 = xor_bytes(part1, dateStringKey)
decoded2 = xor_bytes(part2, unameStringKey)
decoded3 = xor_bytes(part3, scoopStringKey)

# Combine and print the original flag
original_flag = decoded1 + decoded2 + decoded3
print("Decoded:", original_flag.decode("utf-8"))
```

The output gives us:

```
Decoded: YetAnotherF0Flag
```

So the flag is `YetAnotherF0Flag`

However if we try to insert the discovered flag back into the program, we don't get the original encrypted string given in the challenge. This is due to the fact that the date is used to encrypt the flag.

So the easiest way to check this would be to alter the memory, and change the date to `220630`.

By pressing `v` in `Radare2`, you can see the live debugger:

```
0x5cb79069064b      e820fbffff       call sym.imp.sprintf    ;[1] ; int sprintf(char *s, const char *f
0x5cb790690650      488d9572ff..     lea rdx, [var_8eh]
0x5cb790690657      488d8579ff..     lea rax, [var_87h]
;-- rip:
0x5cb79069065e b    be06000000       mov esi, 6
0x5cb790690663      4889c7           mov rdi, rax
0x5cb790690666      e8fefbffff       call 0x5cb790690269     ;[2]
```

We set a breakpoint just before the XOR function in the main, with the aim of tapping into the memory and altering the date:

```
[> dc
This is a flag encryption program, it makes flags safe. Enter your flag:
[YetAnotherF0Flag
INFO: hit breakpoint at: 0x5a3327a4b663
[> px 7 @ rbp-0x87
- offset -       C9CA CBCC CDCE CFD0 D1D2 D3D4 D5D6 D7D8  9ABCDEF012345678
0x7ffc8ef5aec9  3235 3031 3233 00                         250123.
[> wx 32 32 30 36 33 30 00 @ rbp-0x87
[> px 7 @ rbp-0x87
- offset -       C9CA CBCC CDCE CFD0 D1D2 D3D4 D5D6 D7D8  9ABCDEF012345678
0x7ffc8ef5aec9  3232 3036 3330 00                         220630.
```

As you can see I have altered the date to `220630`. We continue execution and wait for the XOR function to run. I set a breakpoint just before the print function:

```
INFO: hit breakpoint at: 0x55d5445c0680
[> px 7 @ rbp-0x87
- offset -       898A 8B8C 8D8E 8F90 9192 9394 9596 9798  9ABCDEF012345678
0x7fff15a01789  6b57 4477 5d5f 00                         kWDw]_.
[> px 7 @ rbp-0x230
- offset -       E0E1 E2E3 E4E5 E6E7 E8E9 EAEB ECED EEEF  0123456789ABCDEF
0x7fff15a015e0  3801 0b07 3e00 00                         8...>..
[> px 7 @ rbp-0x94
- offset -       7C7D 7E7F 8081 8283 8485 8687 8889 8A8B  CDEF0123456789AB
0x7fff15a0177c  4325 030e 1700 59                         C%....Y
```

Encrypted flag: `6b5744775d5f38010b073e4325030e17`

As you can see above, we now have the encrypted flag again in Hex. This confirms our discovered flag!

Additional Observation:

If we try to print out the flag, part 2 of the outputted flag is always `f0f0000006b`:

```
[> dc
flag: 6b5744775d5f0f0000006b4325030e17
(9442) Process exited with status=0x0
```

It seems that the print function uses `rbp - 0x60` rather than `rbp - 0x230`. This might be an obfuscation attempt to prevent brute-force techniques to get the flag. While patching the print function so that it reads from `rbp - 0x230` would likely restore the binary's complete functionality, it is not needed to solve the challenge.