

Mark Seliternikov

picoCTF/ 2019/ Investigative Reversing 2

Challenge details:

Investigative Reversing 2

 | 350 points

Tags: Category: Forensics

AUTHOR: SANTIAGO C/DANNY T

Description

We have recovered a [binary](#) and an [image](#). See what you can make of it. There should be a flag somewhere.

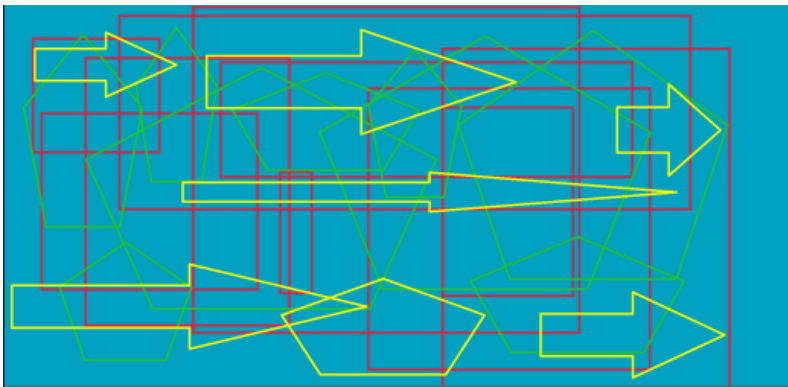
Hints

1 2 3

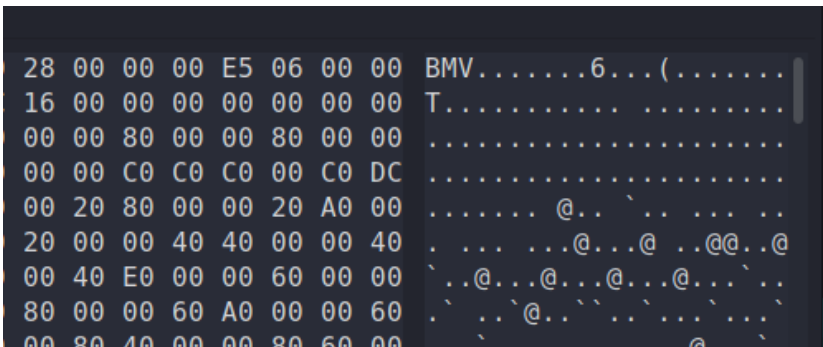
Hints:

- 1) Try using some forensics skills on the image
- 2) This problem requires both forensics and reversing skills
- 3) What is LSB encoding?

Alright so the first thing I wanted to try is looking at the image.



I couldn't discover any clues when looking at it like that so I've decided to open it in a hex editor. (I've used ghex)



I haven't discovered anything written to the file that can indicate right away that actual text string has been written to it, However one of the clues suggested that there's LSB (least significant bit) encoding going on.

So I kept that in mind... My next step was to reverse the 'mystery' file using NSA's program called 'ghidra', It attempts to rebuild the actual C source code so I wouldn't need to exhaust myself with assembly. I can do that because the 'mystery' file is an ELF file which is a linked library file indicating that it was compiled from C. (I changed the default variable names ghidra assigned to it so it'll be easier to understand).

Here are the interesting discoveries:

We can see that there were 3 files involved: 'flag.txt', 'original.bmp', 'encoded.bmp'.

```
local_60 = 0;
flag_pointer = fopen("flag.txt", "r");
orig_bmap_pointer = fopen("original.bmp", "r");
enc_bmap_pointer = fopen("encoded.bmp", "a");
if (flag_pointer == (FILE *)0x0) {
```

Then it proceeds to write 2000 bytes from the 'original.bmp' to 'encoded.bmp'.

```
num_read1 = fread(&buffer1, 1, 1, orig_bmap_pointer);
int_num_read1 = (int)num_read1;
len1 = 2000;
i = 0;
while (i < len1) {
    fputc((int)(char)buffer1, enc_bmap_pointer);
    num_read1 = fread(&buffer1, 1, 1, orig_bmap_pointer);
    int_num_read1 = (int)num_read1;
    i = i + 1;
}
```

Note that the buffer still holds a byte when it exits the loop.

After those bytes it begins to use the 'flag.txt' and 'original.bmp' to create encoded bytes to write to the 'encoded.bmp' file.

```
num_read1 = fread(flag_buffer, 0x32, 1, flag_pointer);
local_64 = (int)num_read1;
if (local_64 < 1) {
    puts("flag is not 50 chars");
    /* WARNING: Subroutine does not return */
    exit(0);
}
ii = 0;
while (ii < 0x32) {
    j = 0;
    while (j < 8) {
        coded = codedChar(j, flag_buffer[ii] - 5, buffer1);
        fputc((int)(char)coded, enc_bmap_pointer);
        fread(&buffer1, 1, 1, orig_bmap_pointer);
        j = j + 1;
    }
    ii = ii + 1;
}
```

Note that it also then proceeds to read another byte from the original.

What's also interesting is that for each 'char' in the flag array the inner loop is executed 8 times.

```
while (j < 8) {
    coded = codedChar(
```

It's very much the number of bits in a single byte! (foreshadowing???)

When I looked into 'codedChar' I've seen something interesting.

```

byte codedChar(int j, byte flag[i]-5, byte buffer1)
{
    byte var;

    var = flag[i]-5;
    if (j != 0) {
        var = (byte)((int)(char)flag[i]-5 >> ((byte)j & 0x1f));
    }
    return buffer1 & 0xfe | var & 1;
}

```

You can see that it does 2 interesting things.

The first highlighted line shows that as the parameter 'j' that gets passed to 'codedChar' grows, it shifts the bits accordingly to the right (note that I named the parameter 'j' to make it easier to understand, same for the rest of the variables/parameters).

Then the return has 2 parts, the first AND. (buffer1 & 0xfe).

What happens there is basically 'original byte' & [11111110], which means that all the bits are being used except the LSB bit which is changed to 0 always!

The second part does the opposite, from the byte of the flag (which is being shifted according to 'j') we can see that only the LSB is being used!

Then both these parts are being merged and result in an "original byte" that had his LSB changed.

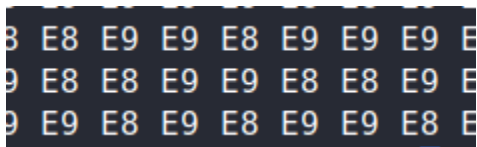
Then what happens is just writing the rest of the 'original.bmp' into 'encoded.bmp' without any change.

```

}
while (int_num_read1 == 1) {
    fputc((int)(char)buffer1, enc_bmap_pointer);
    num_read1 = fread(&buffer1, 1, 1, orig_bmap_pointer);
    int_num_read1 = (int)num_read1;
}
fclose(enc_bmap_pointer);
fclose(orig_bmap_pointer);
fclose(flag_pointer);

```

After that what I did was to check the bit map in a hex editor to see that pattern, and BOOM i've got you!



Can you see it? The value of the bytes generally change in value by just 1! Just like binary 0 and 1 :)

Now all I had to do is write a script that would automate the whole decoding process for me.

This is what I came up with:

(see next page)

```

import codecs

with open('encoded.bmp','rb') as f:

    # goes over the first 2000 bytes
    for i in range(0, 2000):

        cur = f.read(1)

    # loop over 50 times because of the flag.txt length
    for i in range(0, 50):

        # fresh 0x0 byte
        unencoded = 0

        # reads the relevant bytes
        for j in range(0, 8):

            cur = f.read(1)

            cur = int.from_bytes(cur, 'little')

            # get the LSB of the byte (will be 0 if 0 and 1 if 1)
            temp = int(cur) & 1

            # move bit 'j' times accordingly
            temp = temp << j

            # add it to the unencoded byte
            unencoded += temp

        print(chr(unencoded + 5), end='')

print()

```

It's a small and cute script that did the job!

Here it skips the first 2000 bytes which were irrelevant:

```

# goes over the first 2000 bytes
for i in range(0, 2000):

    cur = f.read(1)

```

Then I'm looping 50 times because the flag consists of 50 bytes:

```

# loop over 50 times because of the flag.txt length
for i in range(0, 50):

```

I'm looping again for 8 times because each flag byte was spread to 8 LSBs.

```

# reads the relevant bytes
for j in range(0, 8):

```

Then its reading the bytes and constructs the unencoded (more like rebuilt) byte and then print the final result

```
# reads the relevant bytes
for j in range(0, 8):

    cur = f.read(1)

    cur = int.from_bytes(cur, 'little')

    # get the LSB of the byte (will be 0 if 0 and 1 if 1)
    temp = int(cur) & 1

    # move bit 'j' times accordingly
    temp = temp << j

    # add it to the unencoded byte
    unencoded += temp

print(chr(unencoded + 5), end='')
```

Now it is time to see the result... (drum roll please...)

```
(kali@kali) ~/7 Desktop
$ python3 test.py
picoCTF{n3xt_0n300000000000000000000000000000c5c7dd39}
```

Yeah that's the flag right there...

LETS GOOOOOOOOOO



Ghidra's original output:

Main:

```
undefined8 main(void)
{
    size_t sVar1;
    long in_FS_OFFSET;
    byte local_7e;
    byte local_7d;
    int local_7c;
    int local_78;
    int local_74;
    int local_70;
    undefined4 local_6c;
    int local_68;
    int local_64;
    FILE *local_60;
    FILE *local_58;
    FILE *local_50;
    char local_48 [56];
    long local_10;

    local_10 = *(long *) (in_FS_OFFSET + 0x28);
    local_6c = 0;
    local_60 = fopen("flag.txt","r");
    local_58 = fopen("original.bmp","r");
    local_50 = fopen("encoded.bmp","a");
    if (local_60 == (FILE *)0x0) {
        puts("No flag found, please make sure this is run on the server");
    }
    if (local_58 == (FILE *)0x0) {
        puts("original.bmp is missing, please run this on the server");
    }
    sVar1 = fread(&local_7e,1,1,local_58);
    local_7c = (int)sVar1;
    local_68 = 2000;
    local_78 = 0;
    while (local_78 < local_68) {
        fputc((int)(char)local_7e,local_50);
        sVar1 = fread(&local_7e,1,1,local_58);
        local_7c = (int)sVar1;
        local_78 = local_78 + 1;
    }
    sVar1 = fread(local_48,0x32,1,local_60);
    local_64 = (int)sVar1;
    if (local_64 < 1) {
        puts("flag is not 50 chars");
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    local_74 = 0;
    while (local_74 < 0x32) {
        local_70 = 0;
```

```

    local_74 = 0;
    while (local_74 < 0x32) {
        local_70 = 0;
        while (local_70 < 8) {
            local_7d = codedChar(local_70, local_48[local_74] - 5, local_7e);
            fputc((int)(char)local_7d, local_50);
            fread(&local_7e, 1, 1, local_58);
            local_70 = local_70 + 1;
        }
        local_74 = local_74 + 1;
    }
    while (local_7c == 1) {
        fputc((int)(char)local_7e, local_50);
        sVar1 = fread(&local_7e, 1, 1, local_58);
        local_7c = (int)sVar1;
    }
    fclose(local_50);
    fclose(local_58);
    fclose(local_60);
    if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
        return 0;
    }

    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
}

```

codedChar:

```

1 |
2 | byte codedChar(int param_1, byte param_2, byte param_3)
3 |
4 | {
5 |     byte local_20;
6 |
7 |     local_20 = param_2;
8 |     if (param_1 != 0) {
9 |         local_20 = (byte)((int)(char)param_2 >> ((byte)param_1 & 0x1f));
10 |     }
11 |     return param_3 & 0xfe | local_20 & 1;
12 | }

```