
For my self-directed learning part I have decided to provide my writeups to some forensics challenges for **picoCTF'18** in Work Journal format along with reflections.

However respecting the *"Quality over Quantity"* recommendation, I decided to only include the Top **3** writeups (which I learnt the most from) even though I completed all of the forensics challenges.

Job #:	*z5087077_P1C0CTF18_001*		
Form Commenced By:	Ka Wing Ho	Date Commenced:	2018/10/25
Journal Type:	Evidence Acquisition		

Time (24HR)	Journal Notes, Screenshots, Attachments
22:45	

LoadSomeBits: (Link to image)

The challenge name acronym is LSB which hints very strongly at Least Significant Bit, which is a steganography method where the LSB of each byte in an image is altered to contain a message which can then be extracted by reversing the process

One mistake most people make is they include the header of the image which should not be included, so in my script later I will skip past the header bytes

I tried using the `zsteg` tool on the image but apparently bitmap files were not supported, therefore I decided to write my own script to do it

In the hexdump below you can see there are a lot of 1 and 0 bits followed by a seemingly endless chunk of NULL bytes:

```
00000000: 424d ea9b 2200 0000 0000 8a00 0000 7c00  BM.."........|.
00000010: 0000 b004 0000 7602 0000 0100 1800 0000  ......v.........
00000020: 0000 609b 2200 600f 0000 600f 0000 0000  ..`."`...`.....
00000030: 0000 0000 0000 0001 0101 0000 0000 0001  ................
00000040: 0100 0100 0001 0001 0100 0000 0101 0001  ................
00000050: 0100 0101 0101 0001 0000 0000 0101 0001  ................
00000060: 0001 0001 0000 0001 0000 0001 0100 0001  ................
00000070: 0101 0100 0101 0001 0101 0000 0101 0001  ................
00000080: 0101 0001 0000 0000 0101 0000 0000 0001  ................
00000090: 0101 0000 0100 0000 0101 0000 0101 0001  ................
000000a0: 0100 0001 0000 0001 0001 0101 0101 0001  ................
000000b0: 0100 0100 0001 0001 0000 0101 0100 0001  ................
000000c0: 0001 0101 0101 0001 0101 0001 0000 0001  ................
000000d0: 0000 0100 0000 0000 0101 0000 0101 0001  ................
000000e0: 0001 0101 0101 0001 0100 0101 0000 0000  ................
000000f0: 0101 0000 0101 0000 0101 0001 0000 0000  ................
00000100: 0101 0001 0001 0001 0101 0001 0000 0001  ................
00000110: 0001 0101 0101 0001 0101 0000 0101 0000  ................
00000120: 0101 0000 0001 0001 0100 0001 0101 0001  ................
00000130: 0100 0101 0100 0000 0101 0000 0001 0001  ................
00000140: 0100 0001 0100 0000 0101 0000 0001 0001  ................
00000150: 0100 0000 0101 0000 0101 0001 0000 0001  ................
00000160: 0100 0101 0100 0001 0001 0001 0000 0001  ................
00000170: 0001 0101 0101 0001 0100 0000 0100 0000  ................
00000180: 0101 0000 0001 0001 0101 0001 0000 0000  ................
00000190: 0101 0001 0001 0001 0001 0101 0101 0000  ................
000001a0: 0101 0100 0000 0000 0101 0100 0000 0000  ................
000001b0: 0101 0000 0100 0000 0101 0001 0101 0000  ................
000001c0: 0101 0001 0001 0000 0101 0001 0100 0000  ................
000001d0: 0101 0100 0001 0000 0101 0000 0000 0000  ................
000001e0: 0101 0000 0001 0001 0101 0101 0001 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000210: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

22:47	Either using `dd` or by hand, strip away the headers in the file: I decided to make a quick Python script for future-proofing as well as easy proof of concept
23:30	The completed script functions like this: 1) It opens the file `pico2018-special-logo.bmp` and reads bytes from it 2) It then gets the skips 54 bytes and reads til the 592nd byte (no point reading too long since it's all zeroes anyways) 3) Then for each byte in the truncated stream of bytes: a) convert each byte into 8 bits b) grab the lsb and add it to a result string 4) Then the resultant bit string is converted back into bytes 5) The bytes are then converted back into ascii The script was built it in a way where simply adjusting the byte offset in (2) quickly allowed me to find the flag (initially it was 14 instead of 52)
23:53	The flag was then obtained by running the script: **picoCTF{st0r3d_iN_tH3_l345t_s1gn1f1c4nT_b1t5_882756901}**
Reflection	● I think this challenge was quite satisfying because it would be hard if you are not comfortable with the concepts of bits and bytes and converting between the two (as well as ascii characters) ● I also learnt how to manipulate bits better in Python now ● Overall it took me about just over an hour to debug and make the script ● I found it interesting how this challenge was worth more (550) points than the other two below (200-300) but I spent considerably way less time on this one !

Job #:	*z5087077_P1C0CTF18_002*		
Form Commenced By:	Ka Wing Ho	Date Commenced:	2018/10/25
Journal Type:	Evidence Acquisition		

Time (24HR)	Journal Notes, Screenshots, Attachments
23:53	

Ext. Super Magic: (Link to raw image)

The challenge flavour text hinted that the image was broken somehow and needed repairing before the flag could be retrieved (the flag is in the form of a jpg file as shown below), this is apparent when trying to mount the image

```
$ file ext-super-magic.img
ext-super-magic.img: data

$ strings ext-super-magic.img | grep flag
flag.jpg

$ sudo mkdir /mnt/magic
$ sudo mount ext-super-magic.img /mnt/magic
mount: wrong fs type, bad option, bad superblock on /dev/loop0,
       missing codepage or helper program, or other error

       In some cases useful info is found in syslog - try
       dmesg | tail or so.
```

Running `xxd` shows that there are many many images, but most likely red herrings:

```
00031b30: a401 0000 1800 0d01 6669 6c6c 6572 2d38  ........filler-8
00031b40: 352e 6a70 6700 0000 ad01 0000 1000 0801  5.jpg...........
00031b50: 666c 6167 2e6a 7067 ae01 0000 1800 0d01  flag.jpg........
00031b60: 6669 6c6c 6572 2d37 302e 6a70 672e 6a70  filler-70.jpg.jp
00031b70: c601 0000 1800 0e01 6669 6c6c 6572 2d33  ........filler-3
00031b80: 3635 2e6a 7067 6a70 f901 0000 1800 0d01  65.jpgjp........
00031b90: 6669 6c6c 6572 2d38 382e 6a70 6734 2e6a  filler-88.jpg4.j
00031ba0: 0002 0000 6000 0e01 6669 6c6c 6572 2d33  ....`...filler-3
00031bb0: 3634 2e6a 7067 6a70 6700 0e01 0000 0000  64.jpgjpg.......
00031bc0: 1800 0e01 6669 6c6c 6572 2d33 3235 2e6a  ....filler-325.j
00031bd0: 7067 0e01 0000 0000 2c00 0e01 6669 6c6c  pg......,...fill
00031be0: 6572 2d31 3930 2e6a 7067 0e01 6669 6c6c  er-190.jpg..fill
00031bf0: 6572 2d34 3834 2e6a 7067 0000 0000 0000  er-484.jpg......
00031c00: 2200 0000 1800 0e01 6669 6c6c 6572 2d32  "........filler-2
00031c10: 3737 2e6a 7067 0000 2800 0000 1800 0e01  77.jpg..(.......
00031c20: 6669 6c6c 6572 2d31 3935 2e6a 7067 0000  filler-195.jpg..
00031c30: 0e00 0000 1800 0e01 6669 6c6c 6572 2d32  ........filler-2
00031c40: 3836 2e6a 7067 0000 3100 0000 1800 0e01  86.jpg..1.......
00031c50: 6669 6c6c 6572 2d32 3731 2e6a 7067 0000  filler-271.jpg..
00031c60: 3800 0000 1800 0e01 6669 6c6c 6572 2d32  8.......filler-2
00031c70: 3237 2e6a 7067 0000 4600 0000 1800 0e01  27.jpg..F.......
00031c80: 6669 6c6c 6572 2d31 3639 2e6a 7067 d243  filler-169.jpg.C
00031c90: 4a00 0000 1800 0e01 6669 6c6c 6572 2d33  J.......filler-3
```

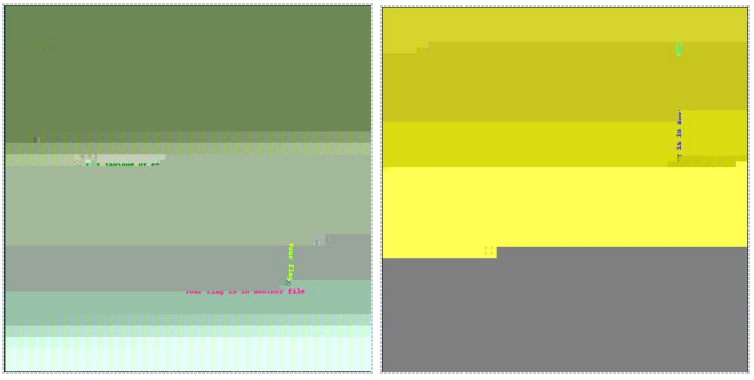
26/10
00:36

Tried using `foremost` to extract the images out, but the flag image was not in the directory of extracted images:

```
$ foremost ext-super-magic.img
Processing: ext-super-magic.img
|*|
$ cd output/jpg/
$ ls
00001026.jpg  00002506.jpg  00002964.jpg
00003232.jpg  00003704.jpg  00004106.jpg
00005328.jpg  00002066.jpg  00002538.jpg
00003026.jpg  00003280.jpg  00003740.jpg
00004198.jpg  00008194.jpg 00002140.jpg
00002602.jpg  00003092.jpg  00003288.jpg
00003808.jpg  00004356.jpg  00008564.jpg
00002206.jpg  00002672.jpg  00003110.jpg
00003386.jpg  00003834.jpg  00004470.jpg
00008886.jpg 00002412.jpg  00002752.jpg
00003148.jpg  00003472.jpg  00003980.jpg
00004740.jpg  00009002.jpg 00002440.jpg
00002940.jpg  00003174.jpg  00003484.jpg
00004052.jpg  00005156.jpg  00009128.jpg
$ for x in *; do feh $x; done
```

The images all appear corrupted and when closely scrutinized say the message *"Your flag is in another file"*

As an example:



00:53

The challenge hinted at the `fsck` tool so I decided to check it out:

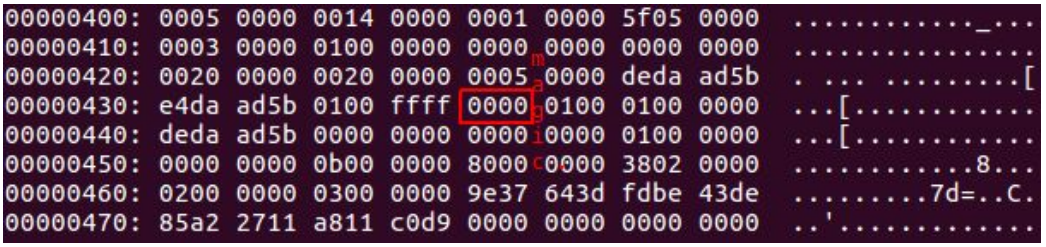
```
$ e2fsck ext-super-magic.img
e2fsck 1.42.13 (17-May-2015)
ext2fs_open2: Bad magic number in super-block
e2fsck: Superblock invalid, trying backup blocks...
e2fsck: Bad magic number in super-block while trying to open ext-super-magic.img

The superblock could not be read or does not describe a valid ext2/ext3/ext4
filesystem.  If the device is valid and it really contains an ext2/ext3/ext4
filesystem (and not swap or ufs or something else), then the superblock
is corrupt, and you might try running e2fsck with an alternate superblock:
        e2fsck -b 8193 <device>
 or
        e2fsck -b 32768 <device>
```

	Trying with different alternate superblocks as suggested did not change the error message At this point my thought process was to try and repair the superblock by finding an intact copy and replacing the entire block

01:23	I discovered `lost+found` in the hexdump earlier, which is usually created at the root directory after a restoration has been completed.
	
	Ran mke2fs and it seemed to have generated a filesystem on the raw image file itself, I then mounted it successfully (?) but no files could be found aside from `lost+found`
	
	$ cp ext-super-magic.img test.img
	$ mke2fs test.img
	mke2fs 1.42.13 (17-May-2015)
	Creating filesystem with 5120 1k blocks and 1280 inodes
	
	Allocating group tables: done
	Writing inode tables: done
	Writing superblocks and filesystem accounting information: done
	
	$ file test.img
	test.img: Linux rev 1.0 ext2 filesystem data,
	UUID=9d3df05f-5175-4954-b854-7756881d6552 (large files)
	
	$ e2fsck test.img
	e2fsck 1.42.13 (17-May-2015)
	test.img: clean, 11/1280 files, 198/5120 blocks
	
	However I don't think this is the way to solve the challenge so I stopped pursuing this route

02:00	Paused the investigation
10:59	Resumed the investigation The ext2 filesystem is most certainly there as fdisk detects the sector size and all: $ fdisk -l ext-super-magic.img Disk ext-super-magic.img: 5 MiB, 5242880 bytes, 10240 sectors Units: sectors of 1 * 512 = 512 bytes Sector size (logical/physical): 512 bytes / 512 bytes I/O size (minimum/optimal): 512 bytes / 512 bytes I tried running tools like `ddrescue` (it has a interface similar to Photorec which is pretty neat) but it did nothing useful
11:25	Ran `fdisk` and it created a DOS partition table, but this again is modifying the image even further which we should avoid I played around with different options but left it at that.

11:49	Reading up the documentation on ext2fs it seems that there are multiple copies of the superblock in the block groups, so I plan to use dd to extract and replace the main superblock at offset 1024 to try and manually repair the superblock
	
	Tried running `mke2fs -n ext-super-magic.img` to print out all stored superblocks but got no result
	
	Tried running `testdisk` on the image as well:
	`$ cp ext-super-magic.ext test.img`
	`$ testdisk test.img`
	`> Disk test.img - 5242 KB / 5120 KiB`
	`> [EFI GPT] EFI GPT partition map (Mac i386, some x86_64...)`
	`> [ Analyse ] Analyse current partition structure and search for lost partitions`
	`Disk test.img - 5242 KB / 5120 KiB - CHS 1 255 63`
	`Current partition structure:`
	`Partition Start End Size in sectors`
	
	`Bad GPT partition, invalid signature.`
	`Trying alternate GPT`
	`Bad GPT partition, invalid signature.`
	
	Also found another command meant to dump out the filesystem information which didn't help much:
	`$ dumpe2fs test.img`
	`dumpe2fs 1.42.13 (17-May-2015)`
	`dumpe2fs: Bad magic number in super-block while trying to open test.img`
	`Couldn't find valid filesystem superblock.`

12:30	According to this link, the magic bytes of EXT2 should be ***0xEF53***
	The documentation also shows that the magic bytes are at offset 1024 bytes from the start of the file, however closer inspection shows that the magic bytes themselves are 56 bytes offset from the start of the superblock which means the the magic bytes are at offset 1080 (or 0x438 in hex)from the start of the file
	
	This is what that looks like in the hexdump:
	 
	`00000400: 0005 0000 0014 0000 0001 0000 5f05 0000 ............_...`
	`00000410: 0003 0000 0100 0000 0000 0000 0000 0000 ................`
	`00000420: 0020 0000 0020 0000 0005 0000 deda ad5b . ... .........[`
	`00000430: e4da ad5b 0100 ffff 0000 0100 0100 0000 ...[............`
	`00000440: deda ad5b 0000 0000 0000 0000 0100 0000 ...[............`
	`00000450: 0000 0000 0b00 0000 8000 0000 3802 0000 ............8...`
	`00000460: 0200 0000 0300 0000 9e37 643d fdbe 43de .........7d=..C.`
	`00000470: 85a2 2711 a811 c0d9 0000 0000 0000 0000 ..'.............`
	
	So I will try to fix these two bytes back to be 0xEF53

13:00	Using `hexedit` , I edited the two bytes at offset 438 to be **53 EF** (little endian) and then saved the file
	`$ hexedit ext-super-magic.img`
	<Press Enter to search for new position and Type in **438**>
	<Selection now jumps to 0x438, now type in **53 EF**>
	<It should look like this: >
	00000438 **53 EF** 01 00 01 00 00 00 DE DA AD 5B 00 00 00 00 00 00 00 00 S.....[.....
	<Then hit **CTRL+X** to save>
	
	Now running file on the image shows the correct magic filetype !
	`$ file ext-super-magic.img`
	ext-super-magic.img: Linux rev 1.0 ext2 filesystem data, UUID=9e37643d-fdbe-43de-85a2-2711a811c0d9 (large files)
	
	Now mounting the image is successful
	`$ sudo mount ext-super-magic.img /mnt/magic`
	`$ cd /mnt/magic && ls`
	-- snipped--
	filler-172.jpg filler-248.jpg filler-323.jpg filler-39.jpg filler-475.jpg **flag.jpg**
	filler-173.jpg filler-249.jpg filler-324.jpg filler-3.jpg filler-476.jpg lost+found
	filler-174.jpg filler-24.jpg filler-325.jpg filler-400.jpg filler-477.jpg
	filler-175.jpg filler-250.jpg filler-326.jpg filler-401.jpg filler-478.jpg
	
	The flag is now visible, opening it shows the flag in a really cool image:
	`Your flag is: "picoCTF{ab0CD63BC762514ea2f4fc9eDEC8cb1E}"`
	
	Thus the flag is:
	picoCTF{ab0CD63BC762514ea2f4fc9eDEC8cb1E}

Reflection	• This challenge was quite hard for me because I kept getting muddled up between choosing which ways I wanted to approach the challenge • I tried a whole bunch of different methods but the actual solution was much easier than I imagined • This challenge took me about 4-5 hours to solve • This challenge also taught me more about filesystem repairing/destroying tools such as fsck and mkefs • It's kind of scary but also funny how a corruption of two bytes (magic header in this case) can cause the whole file to be unusable/ unmountable • It also taught me that tools can only make your life easier but not all the time, in this case the tools were useless because they were crippled by not being able to detect the magic bytes, in this case that as a hint that the magic bytes needed repairing but I didn't catch on until much later

Job #:	*z5087077_P1C0CTF18_003*		
Form Commenced By:	Ka Wing Ho	Date Commenced:	2018/10/26
Journal Type:	Evidence Acquisition		

Time (24HR)	Journal Notes, Screenshots, Attachments
12:30	

core: (Link to binary) (Link to core file)

This challenge was about finding the flag that was supposedly loaded into memory but the program was interrupted before the flag could be printed. This somewhat crosses-over slightly into Reversing territory but was interesting nonetheless.

The two files are as follows:

```
$ file print_flag
print_flag: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=87da2b5b238201d6e071e3189ddef79979bbc723, not stripped

$ file core
core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from '/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag'
```

Tried `strings` and found some information:

```
$ strings core | grep pico
your flag is: picoCTF{%s}
your flag is: picoCTF{%s}
SUDO_COMMAND=/usr/local/bin/shell_manager deploy -r -n 5 -b pico2018

$ strings core | grep flag
print_flag
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
./flag
Failed to open flag file, exiting
Failed to read entire_flag, exiting
your flag is: picoCTF{%s}
./flag
Failed to open flag file, exiting
Failed to read entire_flag, exiting
your flag is: picoCTF{%s}
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag
```

So there appears to be some flag file that the flag is read from before being read into memory

12:45	Loading the binary into GDB as well as the core file shows some more clues in the disassembly:

```
$ gdb -c core print_flag
Reading symbols from print_flag...done.
[New LWP 59747]
Core was generated by
`/opt/hacksports/staging/core_3_928148685553025/problem_files/print_flag'.
Program terminated with signal SIGTRAP, Trace/breakpoint trap.
#0  print_flag () at ./print_flag.c:90
90    ./print_flag.c: No such file or directory.

gdb-peda$ disas print_flag
Dump of assembler code for function print_flag:
=> 0x080487c1 <+0>:    push   ebp                (Execution has stopped here)
   0x080487c2 <+1>:    mov        ebp,esp
   0x080487c4 <+3>:    sub        esp,0x18
   0x080487c7 <+6>:    mov        DWORD PTR [ebp-0xc],0x539
   0x080487ce <+13>:    mov       eax,DWORD PTR [ebp-0xc]
   0x080487d1 <+16>:    mov       eax,DWORD PTR [eax*4+0x804a080]
   0x080487d8 <+23>:    sub       esp,0x8
   0x080487db <+26>:    push   eax
   0x080487dc <+27>:    push   0x804894c
   0x080487e1 <+32>:    call   0x8048410 <printf@plt>
   0x080487e6 <+37>:    add       esp,0x10
   0x080487e9 <+40>:    nop
   0x080487ea <+41>:    leave
   0x080487eb <+42>:    ret
End of assembler dump.

gdb-peda$ info registers
eax       0x270f    0x270f                (Actually this value is not important at all)
ecx       0xd1d32a9    0xd1d32a9
edx       0x80b5a60    0x80b5a60
ebx       0x0    0x0
esp       0xffffd65c    0xffffd65c
ebp               0xffffd668    0xffffd668
esi       0xf7fc6000    0xf7fc6000
edi       0xf7fc6000    0xf7fc6000
eip       0x80487c1    0x80487c1 <print_flag>
eflags            0x212    [ AF IF ]
cs        0x23    0x23
ss        0x2b    0x2b
ds        0x2b    0x2b
es        0x2b    0x2b
fs        0x0    0x0
gs        0x63    0x63
k0        0x0    0x0
k1        0x0    0x0
k2        0x0    0x0
k3        0x0    0x0
k4        0x0    0x0
k5        0x0    0x0
k6        0x0    0x0
k7        0x0    0x0
```

I tried to do some `memsearch` during runtime but nothing useful was found

I noticed that the the value of eax register was **0x270f** before the SIGTRAP, so I tried to recalculate and obtain the flag

```
gdb-peda$ p $eax
$7 = 0x270f
gdb-peda$ p $eax*4+0x804a080
$8 = 0x8053cbc
gdb-peda$ x/s $8
0x8053cbc <strs+39996>: "`Z\v\b"
```

Hmm interesting, there is a strings variable, printing it produces:

```
gdb-peda$ p strs
$1 = {0x8054008 "2c4bf247ebba0ee3d26980cb3dd1ca9e",
      0x8054030 "6da50ecea79a57d293b237d74a8142fb",
      0x8054058 "f90f8de247cae9c8c8aff7642f561410",
      0x8054080 "e206fe8354d3dab92581df5fea5ff7fd",
      0x80540a8 "9f38dbb2f96c31557d64cc3a6895f928",
      0x80540d0 "85060c8f5eb800d7e3bafe566eb67ee0",
       ....
      }
```

So it's an array of hash-like strings located in memory that the flag is probably one of them.

I tried to hex-decode one of them but it appears they are not ascii characters

```
(Python)
>>> s = "2c4bf247ebba0ee3d26980cb3dd1ca9e"
>>> s.decode('hex')
',K\xf2G\xeb\xba\x0e\xe3\xd2i\x80\xcb=\xd1\xca\x9e'
```

Getting more curious I decided to inspect other things like using `backtrace` and `info locals`:

```
gdb-peda$ whatis strs
type = char *[10000]
gdb-peda$ info locals
flag_idx = 0x80b5a60
gdb-peda$ x/xs 0x80b5a60
0x80b5a60:      "742100740f1df143e8952f5b4cf32b49"

gdb-peda$ bt full
#0  print_flag () at ./print_flag.c:90
            flag_idx = 0x80b5a60
#1  0x08048807 in main () at ./print_flag.c:98
No locals.
#2  0xf7e2e637 in __libc_start_main () from /lib32/libc.so.6
No symbol table info available.
#3  0x080484e1 in _start ()
No symbol table info available.
```

This unfortunately was not the flag...

13:00	I found out that in order to run the program properly the nevironment needed to be set up such that **SEED_ENV** was not not and there was a **flag file** which contained 32 bytes of information:
	`$ ./print_flag`
	`Unable to seed prng, exiting`
	`$ export SEED_ENV='test'`
	
	`$ ./print_flag`
	`Failed to open flag file, exiting`
	
	`$ touch flag`
	`$ ./print_flag`
	`Failed to read entire_flag, exiting`
	
	`$ python -c "print 'A'*32" > flag`
	`$ ./print_flag`
	`your flag is: picoCTF{AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA}`
	
	I was absolutely blown away at what I found because all this time I tried submitting the hash string itself as the flag without enclosing it in the **picoCTF{...}**

13:30	Another interesting discovery was that the original SEED_ENV was discovered in the *corefile*:
	`$ strings core | g SEED_ENV`
	`SEED_ENV`
	`SEED_ENV`
	`SEED_ENV=0x46b6615f`

14:28	FInally solved the challenge , with a bit of reversing knowledge...
	
	Referring back to the disassembly above:
	`gdb-peda$ disas print_flag`
	`Dump of assembler code for function print_flag:`
	`=> 0x080487c1 <+0>: push ebp (Execution has stopped here)`
	`0x080487c2 <+1>: mov ebp,esp`
	`0x080487c4 <+3>: sub esp,0x18`
	`0x080487c7 <+6>: mov DWORD PTR [ebp-0xc],0x539`
	`0x080487ce <+13>: mov eax,DWORD PTR [ebp-0xc]`
	`0x080487d1 <+16>: mov eax,DWORD PTR [eax*4+0x804a080]`
	`0x080487d8 <+23>: sub esp,0x8`
	`0x080487db <+26>: push eax`
	`0x080487dc <+27>: push 0x804894c`
	`0x080487e1 <+32>: call 0x8048410 <printf@plt>`
	`0x080487e6 <+37>: add esp,0x10`
	`0x080487e9 <+40>: nop`
	`0x080487ea <+41>: leave`
	`0x080487eb <+42>: ret`
	`End of assembler dump.`
	
	• The value of eax is actually hardcoded to always be 0x539 !
	• It is then multiplied by 4 and added to a constant address to lookup the fag
	• We simply print out whatever is in eax at the time
	• And then dereference the pointer to get the flag !

<table>
<tr><td></td><td>In GDB:
<pre>
gdb-peda$ set $hardcoded = 0x539
gdb-peda$ set $index = 4*$hardcoded + 0x804a080
gdb-peda$ p $index
$1 = 0x804b564
gdb-peda$ x/xw $index
0x804b564 <strs+5348>:   0x080610f0
gdb-peda$ x/s 0x080610f0
0x80610f0:   "8a1f03cbcf407a296fa0bcf149fc5879"
</pre>

This the flag was:
picoCTF{8a1f03cbcf407a296fa0bcf149fc5879}</td></tr>
<tr><td>Reflection</td><td>

This challenge was hard because of the additional complexities made to throw people like me who like to over analyse the situation off
The following parts of the code had no influence on print_flag at all:

SEED_ENV being random
The flag being read from a file
The entire load_string function

I should have focused more on reading the disassembly at the beggining instead of being led on a wild goose chase
I also should have been more aware that the hash string was meant to be submitted with the flag format as well

</td></tr>
</table>