Flare-On 10, Challenge 9, mbransom

Challenge Description

You're doing so great! Go out and celebrate. Take a day off kid, you've earned it. Watch your scoreboard position fall like the sand through the hourglass. Avoid this VM and feel the joy the outside world has to offer. Or crush this one and earn even more internet points, those will come in handy.

Files

Filename	Size	SHA256
hda.img	528,482,304 bytes	6a061453388e0313af11cab3db0178b6d8808c804a3ebf9933ee554fa4e95f86
TIPS.txt	319 bytes	95a4fb8af0ea9715796e6e29881e7222673378d03cc12218aa7a1f451c24a506
vmware\hda.vmdk	528,613,376 bytes	9abe5d685f25d6d9ac29615ed96e4fb40ddd69ab490ac566c018bb298ae39dce
vmware\mbransom.vmxf	271 bytes	daf6a43422350bd28385f5ec6d476156196019b2d5c3e858b3ac27ed89912b60
vmware\mbransom.vmx	1,095 bytes	66ee2e2930c8c05a6626a8ad8ee5adc53f269e963b56d9cd50dbe528bf97c476

High-Level Summary

- The harddrive disk image contains a packed ransomware in the boot sector code
- The Blowfish decryption key is XOR-derived from the victim ID, but two bytes were corrupted
- The key check function encrypts a test string and compares the ciphertext with an embedded value
    - This allows bruteforcing of the two corrupted key bytes

Analysis

Challenge Files

Leveraging `file` yields the disk geometry and identification.

```
$ file hda.img
hda.img: DOS/MBR boot sector; partition 1 : ID=0x6, active 0x81, start-CHS (0x0,1,1), end-
CHS (0x3ff,15,63), startsector 63, 1032129 sectors
```

First look at the partition table with the sleuth kit tools.

```
$ mmls hda.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000062   0000000063   Unallocated
002:  000:000   0000000063   0001032191   0001032129   DOS FAT16 (0x06)
```

Partition 2 is supposed to be `DOS FAT16` partition, let's try to inspect it.

```
$ fls -o 63 -f fat16 hda.img
Invalid magic value (Not a FATFS file system (magic))
```

Dumping the beginning of the supposed FAT16 partition (skip 63 sectors a 512 bytes).

```
$ xxd -s 0x7e00 -l 0x64 hda.img
00007e00: 3aa0 3bc2 e259 6a80 a449 1d5d 9e3d 8ae7  :.;..Yj..I.].=..
00007e10: d7c1 6ae2 0db1 e809 2c51 0eb2 c1fb c888  ..j.....,Q......
00007e20: c0b2 6086 e089 a728 005e b6c7 6782 19cf  ..`....(.^..g...
00007e30: 6927 8e4a 209f e56c b5d8 a950 cb8d bcac  i'.J ..l...P....
00007e40: 4e7d e376 5ab3 1294 d54e b643 51b3 f61f  N}.vZ....N.CQ...
00007e50: ac34 252a e0eb 4921 5c93 7beb 6930 56d1  .4%*..I!\.{.i0V.
00007e60: ad76 2c2d                                .v,-
```

Looks like the disk contents are already encrypted. Judging from the challenge name **mbransom**, which starts with **mbr** ... it might be a ransomware program hiding in boot code the Master Boort Record MBR

Getting some recon on the disk data

MBR = first 512 bytes, magic marker 0x55 0xaa at the end usually we then have slack space that is unused up to the first / primary partiton

but on this disk, the second sector seems to hold code or data

```
00000200: 6afd 506d 5015 08f5 fcf1 18d5 6e3a e7b2  j.PmP.......n:..
00000210: 0a48 7afe ed94 5d90 3652 0352 3781 7d14  .Hz...].6R.R7.}.
00000220: 0aee e5d7 463f 1ff9 874b d288 7d27 0582  ....F?...K..}'..
00000230: 3b85 3d53 4ed6 92c7 58d6 7510 6da5 b0de  ;.=SN...X.u.m...
[...]
up to here
00001c30: 9463 01dc 7307 63cd 3cdb c2d2 c35a 29c7  .c..s.c.<....Z).
00001c40: 38b9 ef09 9217 7c8b 39e4 245a 0000 0000  8.....|.9.$Z....
00001c50: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[...]
```

then all 0 (as expected) up to first / primary partition beginning at 0x7e00 (sector 63 - according to partition table)

```
00007e00: 3aa0 3bc2 e259 6a80 a449 1d5d 9e3d 8ae7  :.;..Yj..I.].=..
00007e10: d7c1 6ae2 0db1 e809 2c51 0eb2 c1fb c888  ..j.....,Q......
00007e20: c0b2 6086 e089 a728 005e b6c7 6782 19cf  ..`....(.^..g...
00007e30: 6927 8e4a 209f e56c b5d8 a950 cb8d bcac  i'.J ..l...P....
[...]
```

then beginning at 0x8028, there seems to be empty data / 16 byte repeating patterns

```
00008000: 312a cc56 6699 b391 c667 e031 19d0 50f4  1*.Vf....g.1..P.
00008010: 3130 7da3 1116 d6e9 3b4c ceb0 4790 a0b0  10}.....;L..G...
00008020: 12d9 1fc9 62db d2cd 47e2 f40d 7fcb a986  ....b...G.......
00008030: 47e2 f40d 7fcb a986 47e2 f40d 7fcb a986  G.......G.......
00008040: 47e2 f40d 7fcb a986 47e2 f40d 7fcb a986  G.......G.......
[...]
```

if the disk was XOR encrypted, this would be the key (as it encrypted lots of 0)

```
up to 0x27800
00027800: 312a cc56 6699 b391 c667 e031 19d0 50f4  1*.Vf....g.1..P.
00027810: 3130 7da3 1116 d6e9 3b4c ceb0 4790 a0b0  10}.....;L..G...
00027820: 12d9 1fc9 62db d2cd 47e2 f40d 7fcb a986  ....b...G.......

then again same pattern up to
00047000: 8d34 b3fd 2857 2a79 ab30 2742 3df2 03f0  .4..(W*y.0'B=...
00047010: d737 7b30 731f fcae 7d12 e3cd 879d 76df  .7{0s...}.....v.
00047020: de69 e069 87d8 a2f9 a501 5d7b 25c3 75b2  .i.i......]{%.u.
00047030: 75d8 b89f 55ec 5d7c 5539 d823 ce85 fffd  u...U.]|U9.#....
00047040: 96cd 7d06 a7b5 1f17 b5a9 14d5 2056 004a  ..}......... V.J
00047050: 0eba a561 a7fa b507 064d ec80 b427 ce56  ...a.....M...'.V
00047060: 5a1b def3 b72f cc8d aaf8 afdf 991c 1a9a  Z..../..........
00047070: e4d8 74cf adf3 a867 7d12 e3cd 879d 76df  ..t....g}.....v.
00047080: b916 3a90 1677 e63b 28a2 6d5f 2578 51a9  ..:..w.;(.m_%xQ.
00047090: afda f455 a988 d036 4ceb 2390 e9c9 742f  ...U...6L.#...t/

and again pattern up to
0004b000: 22c4 7cf1 e432 8317 8c14 aec4 c9ab ee35  ".|..2.........5
0004b010: 932f add1 b394 0f89 493f 0504 e39c f0c0  ./......I?......
0004b020: 6379 3413 243c 878f 50f9 28f5 1653 24c5  cy4.$<..P.(..S$.
0004b030: 207c 4857 5c55 46b9 a311 54fe aec4 6ed6   |HW\UF...T...n.
0004b040: ac0c 1260 a742 de07 1492 97b0 49d4 0d93  ...`.B......I...
```

back and forth and lots of that pattern

carve out the fat16 partition that is likely crypted

```
$ dd if=hda.img of=out.img bs=512 skip=63
1032129+0 records in
1032129+0 records out
528450048 bytes (528 MB, 504 MiB) copied, 133.091 s, 4.0 MB/s
```

trying to decrypt with xor key `47e2 f40d 7fcb a986 47e2 f40d 7fcb a986` does not yield anything useful -> so it's not just XOR, sad panda

looks like we have to understand the code in the boot sector! no shortcuts this way

and the challenge file `tips.txt` is there for a reason 😉

Setup Tips for debugging / booting the disk

```
qemu
====

qemu-system-i386 -soundhw pcspk -drive file=hda.img,format=raw

bochs
=====

bochs -n "ata0-master: type=disk, path=hda.img" "boot: disk"

vmware
======

import vmware/mbransom.vmx into VMware

other
=====

hda.img is a raw disk image. make sure the VM guest uses BIOS (not UEFI) boot
```

Boch setup

IDA with bochs

- copy `dbg_bochs.cfg` from challenge bochs_setup dir to `$IDAUSR/cfg/dbg_bochs.cfg`
- check path for bochs installation dir

Boch's template -> IDA Pro dir -> cfg -> `bochsrc.cfg`

- use bochsrc from challenge dir next to hda.img
- open bochsrc with IDA as "PE"
- warning about memory visibility
    - go to debugger -> memory regions -> insert one 0-FFFFFFFE RWX, 16 bit segment
- another warning -> please setup compiler

Some bochs resources, Hexrays docu

Advanced Analysis

hex editor / 010

```
0000:011D  00 4F 62 66 75 73 63 61 74 69 6F 6E 31 32 33 34  .Obfuscation1234
0000:012D  35                                               5
```

funny string there

first run, see ransom note about the hard disk having been encrypted. We're supposed to pay and give the TA money and the victom id **3487B3B41F20** (6 bytes) We are asked for a decryption key in order to (hopefully) decrypt the disk Of course we don't pay but instead reverse the planet!

suspending bochs, we land at IP = 0xe86d with CS = 0xF000

looks like RC4 KSA loop @ 6bc and PRGA + decrypt @ 6d7, sbox @ 0x804, key @ 71f = Obfuscation12345 (16 bytes)

seems to decrypt code @ 0x1000

- decrypting that code, we now see strings like ransomnote and some that seems to be about key checking

second stage code @ 0x1000

```
- debug01:101F                call    display_ransom_note_1058
- debug01:102B                call    play_lovely_music_1175
- debug01:102E                call    sub_11BE
```

sub 11be

- di = 2a4c (inp buf?)
- int 16h @ 11ce -> read keyb char
- if cr is hit, compare di with 2a5c
- reads 16 chars, saves them to 0x2a4c
    - saves "A" and "a" as 0x0A
    - same for b-f
    - saves "0" as 0x0
    - same for 1-9
- after 16 chars and return
- calls process decr key 1296
    - bin2hex of inp -> store @ 2a5c (like from hex - each inp char byte is a hex digit, 16 chars -> 8 bytes)
        - xor that with something at 0x19fc 3487B3B41F209090 (xor key) / victim id
            - e.g. input a1fb, xor with 3487 = 95c7
                - checked with 0x5555 per word
        - so 0x19fc 8 bytes xor 55h should be the first 8 chars of the key
            - 61 d2 e6 e1 4a 75 c5 c5
        - next step
            - copy 4 words from 2a5c (the bin2hex inp key) to 2a64
            - and
            - xor again with 5555 (regains value from 19fc)
            - and then appends that beginning @ 2a6c
            - 2a64 then holds 61 D2 E6 E1 4A 75 C5 C5 34 87 B3 B4 1F 20 90 90 (16 bytes)

valid key -> has to fulfil the first check correct key -> has to fulfill the second check

- includes encrypting 8 byte "Test Str" from 0x19eb copied to stack
- and comparing that with 8 byte ciphertext @ 0x19f4
- if both ciphertexts are equal, the key is correct

what does sub 1674 do with our 16 bytes value @ 2a64 called @ 12d8?

- possibly some kind of KSA for some crypto alg
- it doesn't return anything that is used in the following crypt call for test str encryption
- but it modifies a possible crypto alg context
    - copies 0x824 words from 1a04 (up to 2a4c) to 2a78 (up to 3ac0)
    - calls enc 1573 @ 16db with di = 2a78 in a loop, 9 rounds
    - calls enc 1573 @ 16ef with di = 2ac0 in a loop, x rounds
    - the crypt internal func 152b uses 2ac0, 2ac2, 2ec0, 2ec2, 32c0, 32c2 and 36c0, 36c2 offsets + index

        - these could be like S-boxes or somesuch
        - googling for the first 8 bytes 886A3F24 D308A385 yields a hint to blowfish pbox (little endian dwords)
            - too bad all findcrypt / ida plugins crash ida on 16 bit mode
- "valid" key = 61 D2 E6 E1 4A 75 C5 C5 34 87 B3 B4 1F 20 90 90 (victim id xor 0x55 with victim id appended)
- stack params: first: key buf, second key len

crypto algo id

- first hint was blowfish pbox when googling for 886A3F24D308A385
- Blowfish's key schedule starts by initializing the P-array and S-boxes with values derived from the hexadecimal digits of pi.
- The secret key is then, byte by byte, cycling the key if necessary, XORed with all the P-entries in order. A 64-bit all-zero block is then encrypted with the algorithm as it stands. The resultant ciphertext replaces P1 and P2. The same ciphertext is then encrypted again with the new subkeys, and the new ciphertext replaces P3 and P4. This continues, replacing the entire P-array and all the S-box entries. In all, the Blowfish encryption algorithm will run 521 times to generate all the subkeys – about 4 KB of data is processed.
- p array has size 576 bits = 0x48 bytes
    - fits to the 0x48 byte gap between 2a78 (begin of copy buffer from 1a04) and indexing in sub 152b begins @ 2ac0
    - 2a78 + 0x48 = 2ac0
    - first 0x48 = `243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E6C89452821E638D01377BE5466CF34E90C6CC0AC29B7C97C50DD3F84D5B5B54709179216D5D98979FB1`
- so we have p-array / p-box @ 0x2a78 size 0x48
- and then s-boxes from 0x2ac0 up to 3ac0 = size 0x1000 = 4 KiB
- fits to pseudo code

```
uint32_t P[18];   // dwords little endian! define the variables in ida
uint32_t S[4][256];   // dwords little endian!
```

- the pbox values correspond to what we would expect, first dword 243f6a88
- internal sub 152b = feistel function f
    - `void __usercall f_152B(int x@<ax>);`

for all we know, this is good enough to identify the crypto algo as Blowfish)

- Blowfish sub 1619 is used to encrypt the Test Str for key correctness check that is also used in the decrypt disk sub 132f (call to parent sub @ 13d4 to 1660 to 1619)
    - ax is used as a function pointer that is different between encrypt and decrypt
    - encrypt: ax = offset sub 1573 (used for enc Test Str)
        - enc func has si 2a78
    - decrypt: ax = offset sub 15c5 (used to decrypt disk sectors)
        - dec func has si 2abe

    - both enc and dec functions use an internal crypt sub 1528
- key len 16 bytes
- enc test str @ 19f4 2E2157823EA96C6E
- encrypt("Test Str") == 2E2157823EA96C6E?
- block size seems to be 8 bytes, if it is a block cipher

feistel function 1573 enc and 15c5 dec? test str @ 19eb cipher test str @ 19f4

summary so far

- there are 2 key checks
    - validness
        - key = xor(victim id, 0x55)
        - thing is, the validness check only loops 3 times (word sized), so the last two key bytes are not check
        - the 2 bytes behind the 6 byte victim id @ 19fc are 0x90 0x90.
        - these bytes could be random garbage. or at least not be relevant
        - so technically any input key of 61 d2 e6 e1 4a 75 00 00 to 61 d2 e6 e1 4a 75 ff ff would be "valid"
    - correctness
        - enc(Test Str) == 2E2157823EA96C6E
        - this is not fulfilled with the blowfish key 61 D2 E6 E1 4A 75 C5 C5 34 87 B3 B4 1F 20 90 90

approach

- brute force last 2 input key bytes (256 * 256 combinations)
- put them into key bytes xx, and put yy = xx ^ 0x55 at the end (the corrupted bytes?)
- 61 D2 E6 E1 4A 75 xx xx 34 87 B3 B4 1F 20 yy yy
- assert(enc(Test Str) == 2E2157823EA96C6E)

```
> py .\crack_blowfish_key.py
[*] YAY, found key b'61d2e6e14a754adc3487b3b41f201f89', decryption key =
b'61d2e6e14a754adc', cipher = b'2e2157823ea96c6e'
```

See script

run the program, enter the cracked key, it starts decrypting

reboot system -> land in a dos command prompt -> dir -> type flag.txt

```
C:\>dir
 Volume in drive C is FLAREON
 Volume Serial Number is 1170-1A1F

 Directory of C:\

KERNEL   SYS        46,485  05-14-21  3:32a
COMMAND  COM        85,480  07-10-21 11:28p
AUTOEXEC BAT             0  05-26-23  6:56p
FLAG     TXT            64  05-26-23  6:58p
         4 file(s)        132,029 bytes
         0 dir(s)     528,023,552 bytes free
C:\>type flag.txt
```

```
bl0wf1$h_3ncrypt10n_0f_p@rt1t10n_1n_r3al_m0d3@flare-on.com
```

Flag

Flag: `bl0wf1$h_3ncrypt10n_0f_p@rt1t10n_1n_r3al_m0d3@flare-on.com`

PS - Disk Decryption Code

in the disk image, flag.txt resides @ 0x6d000

```
$ xxd -s 0x6cff0 -l 0x60 ../hda.img
0006cff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0006d000: 0d0a 626c 3077 6631 2468 5f33 6e63 7279  ..bl0wf1$h_3ncry
0006d010: 7074 3130 6e5f 3066 5f70 4072 7431 7431  pt10n_0f_p@rt1t1
0006d020: 306e 5f31 6e5f 7233 616c 5f6d 3064 3340  0n_1n_r3al_m0d3@
0006d030: 666c 6172 652d 6f6e 2e63 6f6d 0d0a 0d0a  flare-on.com....
0006d040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

in the crypted image, that was

```
$ xxd -s 0x6cff0 -l 0x60 hda.img
0006cff0: 47e2 f40d 7fcb a986 47e2 f40d 7fcb a986  G.......G.......
0006d000: cd48 a6eb 177c 7143 e28b fc37 600d f50d  .H...|qC...7`...
0006d010: f7e0 7bd5 b019 d857 bca8 2f36 487d 5213  ..{....W../6H}R.
0006d020: c4d4 8ebf d0f8 65df 4a41 8124 70cf c353  ......e.JA.$p..S
0006d030: e596 7144 6443 0945 2b58 7158 977e c58e  ..qDdC.E+XqX.~..
0006d040: 47e2 f40d 7fcb a986 47e2 f40d 7fcb a986  G.......G.......
```

once that key correctness check is passed. what happens there can be deduced from int 13h syscalls and error strings

```
- debug01:1034 E8 F8 02               call    decrypt_disk_132F
- debug01:103C E8 98 04               call    repair_mbr_14D7
```

high level

```
debug01:101F               call    display_ransom_note_1058
debug01:1022               mov     ah, 2
debug01:1024               xor     bx, bx
debug01:1026               mov     dx, 1214h
debug01:1029               int     10h             ; - VIDEO - SET CURSOR POSITION
debug01:1029                                       ; DH,DL = row, column (0,0 = upper
left)
debug01:1029                                       ; BH = page number
debug01:102B               call    play_lovely_music_1175
debug01:102E               call    read_and_validate_key_11BE
debug01:1031               pop     si
debug01:1032               pop     dx
debug01:1033               push    dx
debug01:1034               call    decrypt_disk_132F
```

```
debug01:1037                 jnb     short loc_103B
debug01:1039
debug01:1039 hang_1039:                              ; CODE XREF: debug01:hang_1039↓j
debug01:1039                 jmp     short hang_1039
debug01:103B ; ---------------------------------------------------------------------------
debug01:103B
debug01:103B loc_103B:                               ; CODE XREF: debug01:1037↑j
debug01:103B                 pop     dx
debug01:103C                 call    restore_old_mbr_14D7
debug01:103F                 mov     si, 1973h       ; Hit enter to reboot disk
debug01:1042                 call    print_str_130D  ; Hit enter to reboot disk
debug01:1045
debug01:1045 read_char_1045:                         ; CODE XREF: debug01:1050↓j
debug01:1045                 xor     ah, ah
debug01:1047                 int     16h             ; KEYBOARD - READ CHAR FROM BUFFER,
WAIT IF EMPTY
debug01:1047                                         ; Return: AH = scan code, AL =
character
debug01:1049                 cmp     al, 0Dh
debug01:104B                 jz      short reboot_1052
debug01:104D                 call    set_ax_a97_cx_1_dx_86a0_1194 ; this beeps for some
reason
debug01:1050                 jmp     short read_char_1045
debug01:1052 ; ---------------------------------------------------------------------------
debug01:1052
debug01:1052 reboot_1052:                            ; CODE XREF: debug01:104B↑j
debug01:1052                 cli
debug01:1053                 jmp     far ptr loc_FFFF0
```