Flare-On 10 Challenge 9: mbransom

By Jacob Thompson

Challenge Prompt

One of our legacy PCs supporting an old but expensive scientific machine seems to have been knocked off the network and won't do anything but play music and display a strange textual message on boot. We've imaged the disk; can you take a look and figure it out?

Solution

Overview

The ransomware has encrypted the entire C: partition using Blowfish-256-ECB, placed a decryption program in the remainder of Track 0, and rewritten the Master Boot Record. The new MBR checks the partition table for an active partition. If the first one found is unencrypted, the system boots normally. If it is encrypted (denoted by a 0x01 bit in the active field) the MBR instead loads the decryption program from Track 0 into memory, deobfuscates the decryption program by decrypting it with RC4 using the key `Obfuscation12345`, and passes control to the decryption program.

The decryption program displays a ransom note as shown in Figure 1. The decryption program gives a hint that the correct key consists of 16 hexadecimal digits and displays a 12-character victim ID. Upon typing 16 hex digits and pressing Enter, the program displays an Invalid Key error message. In fact, the victim ID just serves to restrict the brute force search space: the first 12 hex digits of the decryption key must correspond to the victim ID XOR 0x5 (i.e., 61D2E6E14A75); the last four digits can be totally random. A brute force attack against the remaining 16 bits using the key validation check incorporated into the decryption program is computationally trivial and produces the full key 61D2E6E14A754ADC. If the user enters this key, the program decrypts the disk as shown in Figure 2 and then prompts to reboot. The decrypted partition boots FreeDOS and there is a `C:\FLAG.TXT` containing the flag.

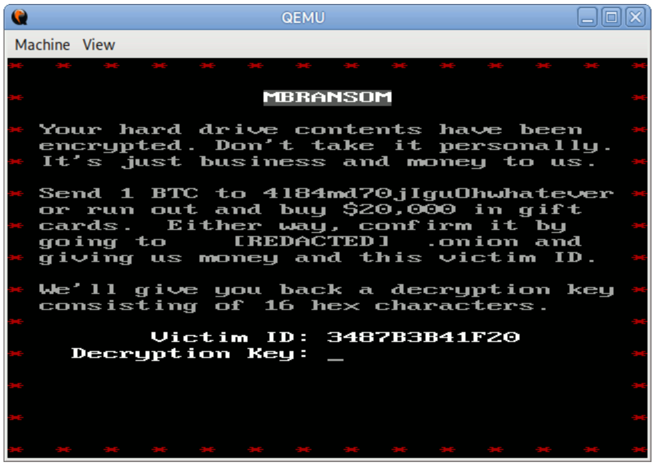

Figure 1. A ransom note displays when the system boots.

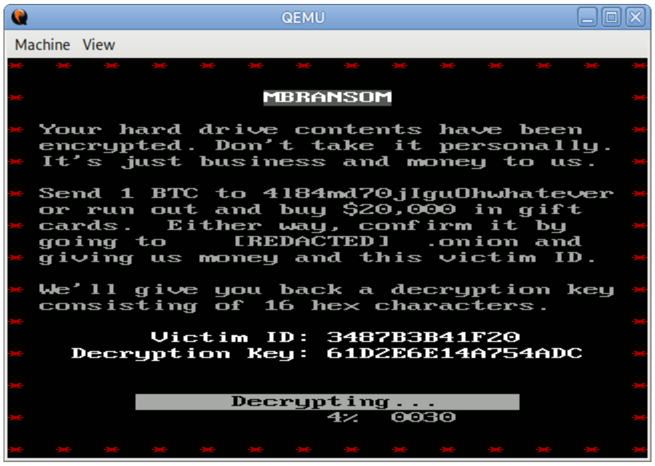

Figure 2. If the key entered passes the validation check, the decryption program decrypts the disk.

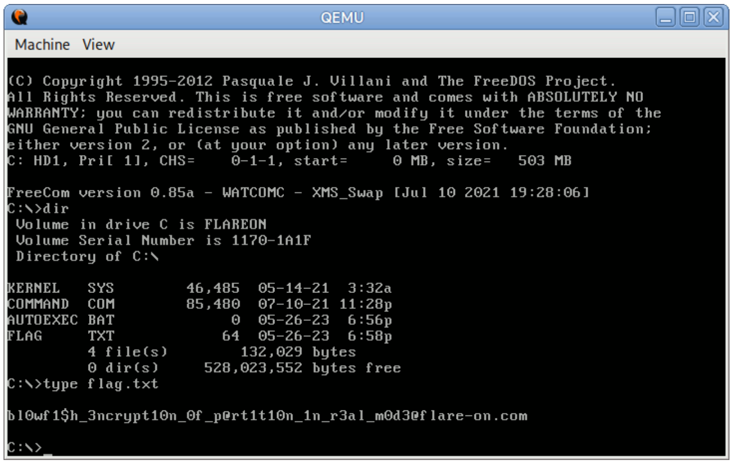

Figure 3. The decrypted partition boots into FreeDOS and contains flag.txt.

Analyzing the Master Boot Record

A BIOS-based machine boots by loading the master boot record (MBR) from the first sector of a disk into memory and then passing control to the MBR. The MBR has the structure shown in Table 1 and a partition table entry has the structure shown in Table 2; for more information, see https://en.wikipedia.org/wiki/Master_boot_record.

Offset	Length	Description
0	0x1b8	Machine code
0x1b8	0x6	NT disk signature
0x1be	0x10	Primary partition entry 1
0x1ce	0x10	Primary partition entry 2
0x1de	0x10	Primary partition entry 3
0x1ee	0x10	Primary partition entry 4
0x1fe	0x2	Magic number 0xaa55

Table 1. Structure of the master boot record.

Offset	Length	Description
0	0x1	Active if 0x80
0x1	0x3	Starting address in CHS notation
0x4	0x1	Type
0x5	0x3	Ending address in CHS notation
0x8	0x4	Starting address in LBA notation
0xc	0x4	Length in sectors

Table 2. Structure of a primary partition entry.

To analyze the MBR, it is easiest to first extract it from the raw disk image:

```
$ dd if=hda.img bs=512 count=1 of=mbr.bin
```

By convention, the MBR executes in real mode and at the address 0000:7c00, so the correct IDA load settings are shown in Figure 4. The file must also be disassembled as 16-bit code.

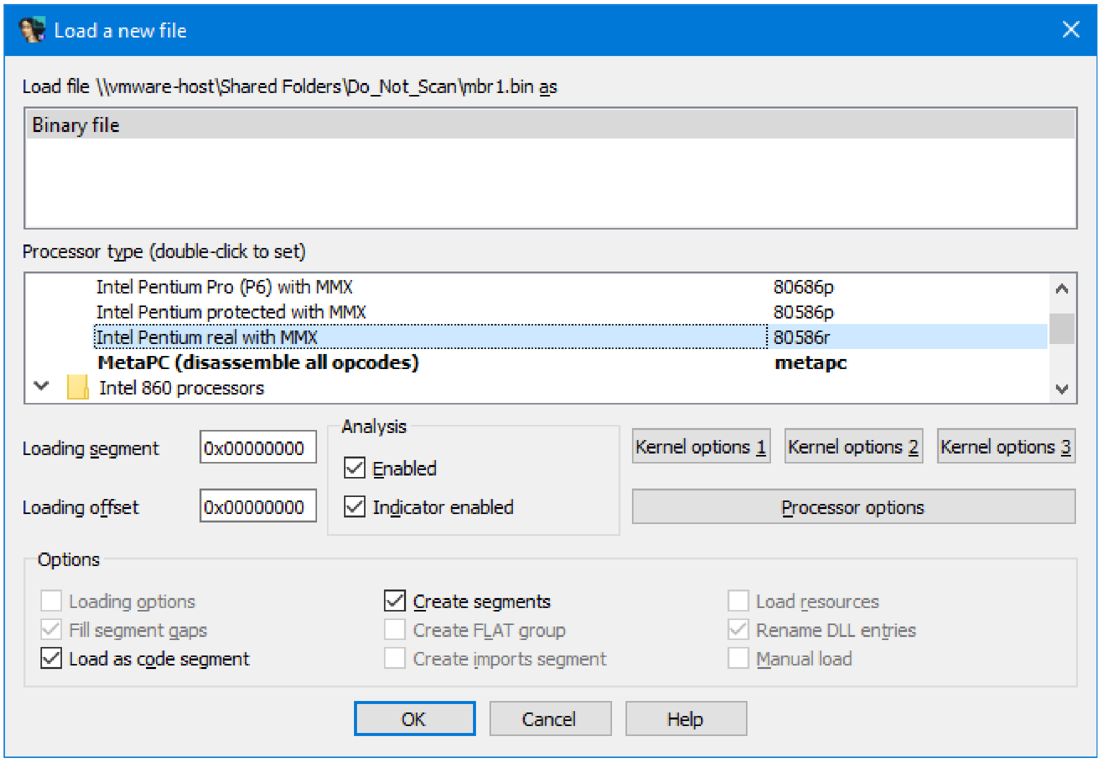

Figure 4. Correct IDA load settings for the Master Boot Record.

Figure 5 shows that first, the machine code jumps to 0000:7C05, to ensure that the CS register is zeroed as there are many other real-mode aliases of the linear address 0x7C05. Next, the machine code copies itself from 0000:7C00 to 0000:0600, then jumps to the next instruction in the relocated copy at 0000:061D. This is necessary because the Volume Boot Record (VBR) from the active partition will also be loaded at 0000:7C00, which would overwrite the MBR. Since the machine code will actually execute from 0000:0600, it is easiest to close the IDB at this point and reopen it, instructing IDA to use the load address 0000:0600.

```
; =============== S U B R O U T I N E ====

; Attributes: noreturn

sub_7C00        proc near
                jmp     far ptr sub_7C05
sub_7C00        endp


; =============== S U B R O U T I N E ====

; Attributes: noreturn

sub_7C05        proc near                   ;
                sti
                cld
                xor     ax, ax
                mov     ds, ax
                mov     es, ax
                mov     bx, 7C00h
                mov     si, bx
                mov     di, 600h
                mov     cx, 100h
                rep movsw
                jmp     near ptr 61Dh
sub_7C05        endp

; ---------------------------------------
                db 0BEh
```

Figure 5. The machine code copies itself to 0000:0600 and jumps there.

The code from 0x61D-0x645 (Figure 6) parses all four primary partition entries. If an active partition (status 0x80) is found, it is checked to see if it is encrypted (status 0x81) and if it is unencrypted, the machine code calls to BIOS to read the Volume Boot Record (first sector of the partition) and jumps to the VBR. if the first primary partition found is encrypted, the code passes control to address 0x655.

```
; =============== S U B R O U T I N E =======================================


sub_61D         proc near
                mov     si, 7BEh        ; first primary partition entry

loc_620:                                ; CODE XREF: sub_61D+F↓j
                test    byte ptr [si], 80h ; is the active bit set?
                jnz     short loc_633   ; jump if it's active
                add     si, 10h         ; move on to the next one
                cmp     si, 7EEh        ; already past the fourth?
                jbe     short loc_620   ; no, parse this one
                mov     si, offset aNoActivePartit ; "No active partition"
                jmp     short loc_64A   ; couldn't find any active partition
; ---------------------------------------------------------------------------

loc_633:                                ; CODE XREF: sub_61D+6↑j
                test    byte ptr [si], 1 ; is it an encrypted partition?
                jnz     short loc_655   ; jump if it's encrypted
                mov     ax, 201h        ; it's unencrypted; read the VBR
                mov     dh, [si+1]
                mov     cx, [si+2]
                int     13h             ; DISK - READ SECTORS INTO MEMORY
                                        ; AL = number of sectors to read, CH = track, CL = sector
                                        ; DH = head, DL = drive, ES:BX -> buffer to fill
                                        ; Return: CF set on error, AH = status, AL = number of sectors read
                jb      short loc_647
                jmp     bx              ; jump to the VBR from the active partition
; ---------------------------------------------------------------------------

loc_647:                                ; CODE XREF: sub_61D+26↑j
                mov     si, offset aDiskError ; "Disk error"

loc_64A:                                ; CODE XREF: sub_61D+14↑j
                                        ; sub_61D+52↓j ...
                mov     ah, 0Eh

loc_64C:                                ; CODE XREF: sub_61D+36↓j
                lodsb
                test    al, al
```

Figure 6. The machine code first locates the first active partition and checks whether it is encrypted.

Figure 7 shows that from 0x655-0x690, the machine code queries the BIOS for the number of sectors per track for the disk. The machine code decrements one from this number (to exclude the MBR which is the first sector of the first track) and loads the remainder of the sectors making up the first track into the address 0000:1000. As the first partition cannot begin before the second track of the disk, the sectors making up the first track (aside from the MBR itself) are unallocated and might be used for other bootloader related purposes, or as will be shown, for the ransomware decryption program.

```
loc_655:                                ; CODE XREF: sub_61D+19↑j
                cli
                mov     ss, ax          ; move the stack to 64K so it does not overlap
                mov     sp, 0FFFEh      ; with decryption program
                sti
                mov     ds:802h, dx     ; save drive number
                mov     ds:800h, si     ; save pointer to partition entry
                mov     ah, 8
                xor     di, di          ; get the number of sectors per track
                int     13h             ; DISK - DISK - GET CURRENT DRIVE PARAMETERS (XT,AT,XT286,CONV,PS)
                                        ; DL = drive number
                                        ; Return: CF set on error, AH = status code, BL = drive type
                                        ; DL = number of consecutive drives
                                        ; DH = maximum value for head number, ES:DI -> drive parameter
                jnb     short loc_671
                mov     si, offset aDiskError ; "Disk error"
                jmp     short loc_64A
; ---------------------------------------------------------------------------

loc_671:                                ; CODE XREF: sub_61D+4D↑j
                xor     ax, ax
                mov     es, ax
                mov     al, 2           ; set CL = 2, the starting sector
                xchg    ax, cx          ; CL = 2, the starting sector
                and     al, 3Fh         ; (sector numbering is one-based)
                dec     ax              ; excluding the MBR itself
                mov     di, ax          ; save number of sectors to read
                mov     ah, cl          ; AH=2 read sectors
                cwd                     ; DX=0
                mov     dl, ds:802h     ; drive number
                mov     bx, 1000h       ; destination buffer
                int     13h             ; DISK -
                jnb     short loc_690
                mov     si, offset aDiskError ; "Disk error"
                jmp     short loc_64A
; ---------------------------------------------------------------------------
```

Figure 7. The machine code loads the decryption program from Track 0 into the address 0000:1000.

From 0x690-0x6B2, the machine code records the number of bytes (number of sectors multiplied by 512 bytes, the assumed sector size) read from the first track. Next, the code fills a 256-byte array that will come to be named S, according to the formula S[i] = i (Figure 8).

```
loc_690:                                ; CODE XREF: sub_61D+6C↑j
                cbw                     ; AH=0
                test    ax, ax          ; if the BIOS claims zero sectors were read,
                jnz     short loc_698   ; but returned success, assume we got the
                mov     ax, di          ; number of sectors we asked for.
                cbw                     ; AH=0

loc_698:                                ; CODE XREF: sub_61D+76↑j
                xchg    ah, al          ; AX <<= 9
                shl     ax, 1           ; (convert from sectors to bytes)
                xchg    ax, cx          ; CX=AX
                mov     ax, 100h
                mov     dx, 202h
                mov     bx, 804h        ; S
                mov     di, bx

loc_6A8:                                ; CODE XREF: sub_61D+8E↓j
                stosw                   ; S[i] = i
                add     ax, dx
                jnb     short loc_6A8
                cwd
                mov     di, bx
                jmp     short loc_6B7
; ---------------------------------------------------------------------------
```

Figure 8. The machine code records the number of bytes making up the decryption program and begins RC4 key scheduling.

Continuing to follow the machine code, it performs the RC4 key scheduling algorithm using the key Obfuscation12345 (Figure 9).

```
loc_6B2:                                ; CODE XREF: sub_61D+9D↓j
                                        ; sub_61D+B4↓j
                lodsb                   ; al = *key++
                test    al, al          ; check for \0
                jnz     short loc_6BC   ; j += key[i % keylen]

loc_6B7:                                ; CODE XREF: sub_61D+93↑j
                mov     si, offset aObfuscation123 ; "Obfuscation12345"
                jmp     short loc_6B2
; ---------------------------------------------------------------------------

loc_6BC:                                ; CODE XREF: sub_61D+98↑j
                add     dl, al          ; j += key[i % keylen]
                mov     al, dh          ; al = i
                xlat                    ; al = S[i]
                add     dl, al          ; j += S[i]
                xor     bx, bx
                mov     bl, dl          ; bx = j
                xchg    al, [bx+di]     ; al <=> S[j]
                mov     bl, dh          ; bx = i
                mov     [bx+di], al     ; S[i] = al
                mov     bx, di          ; S
                inc     dh              ; ++i
                jnz     short loc_6B2   ; jump if i < 256
```

Figure 9. Machine code runs the RC4 key scheduling algorithm.

Finally, the code at 0x6D3-0x6FC performs RC4 decryption on the Track 0 contents that were previously loaded at 0000:1000, and when decryption is complete, the code jumps to 0000:1000 (Figure 10). The code at that location is the ransomware decryption program.

```
                mov     si, 1000h          ; Track 0 buffer
                cwd                        ; i=j=0

loc_6D7:                                   ; CODE XREF: sub_61D+D5↓j
                inc     dh                 ; ++i
                mov     al, dh
                xlat                       ; al = S[i]
                add     dl, al             ; j += S[i]
                xor     bx, bx
                mov     bl, dl             ; bx = j
                mov     ah, al
                xchg    al, [bx+di]        ; S[i] <=> S[j]
                mov     bl, dh
                mov     [bx+di], al
                mov     bx, di             ; S
                add     al, ah             ; S[i] + S[j]
                xlat                       ; S[S[i] + S[j]]
                xor     [si], al           ; XOR keystream with buffer
                inc     si
                loop    loc_6D7            ; while(--cx)
                mov     si, ds:800h        ; restore pointer to partition table entry
                mov     dx, ds:802h        ; restore drive number in DL
                jmp     near ptr 1000h     ; jump to the ransomware decryptor!
sub_61D         endp
```

Figure 10. The machine code performs RC4 decryption of the decryption program and then jumps to the decryption program.

Analyzing the Decryption Program

Through analysis in the prior section it is known that:

- The decryption program is located in sectors 2 through 63 of the first track (as MBR-compatible disks have, at most, 63 sectors per track)
- The decryption program on-disk is encrypted using RC4 using the key Obfuscation12345.
- The decryption program will be loaded into memory and will execute at address 0000:1000.

The unencrypted code making up the decryption program could be recovered dynamically using a debugger, or statically by extracting and decrypting it directly from the disk image:

```
$ dd if=hda.img bs=512 skip=1 count=62 | openssl rc4 -K \
4f626675736361746696f6e3132333435 -out decryptor.bin
```

As the decryption program produces a ransom note and offers to decrypt the disk in exchange for the correct key, it is likely to contain cryptographic code. Despite not being a PE, the PEiD program's KANAL plugin will accept decryptor.bin and identifies constants related to the Blowfish algorithm as shown in Figure 11. This could come in great use later.

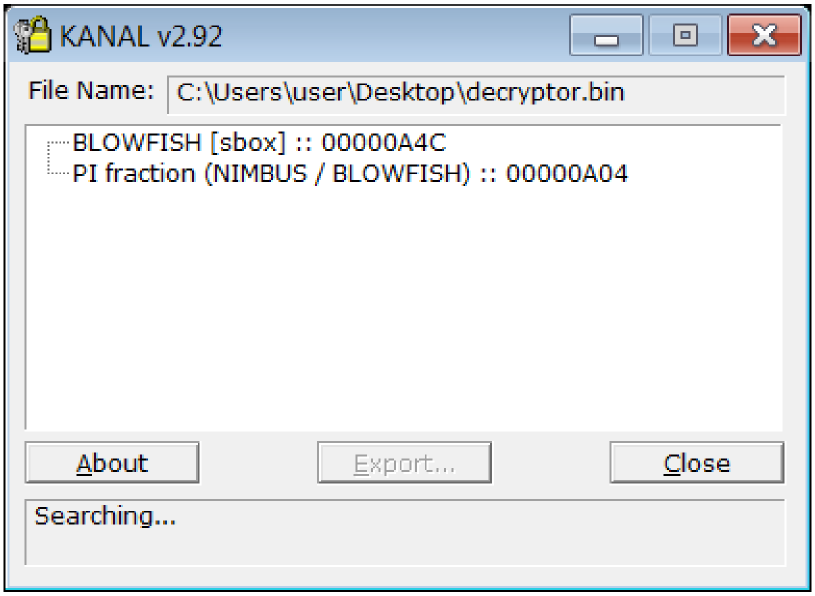

Figure 11. The decryption program contains constants related to Blowfish.

As with the MBR, the decryption program can be loaded into IDA. Select Real Mode Pentium with MMX, a loading offset of 0x1000, and 16-bit code. Based on the structure of the MBR, the decryption program's entry point is right at the beginning of the file, at 0000:1000. The code performs some initialization of registers and the screen. As Figure 12 shows, the first function call is to 0x1058. The function at 0x1058 prints the * border around the edges of the screen, prints the ransom note, and loads the victim ID from the location 0x19FC and prints it to the screen in hexadecimal. Then, the program calls the function 0x1175.

```
sub_1000        proc near
                sti
                cld
                mov     ax, cs          ; ensure DS = CS = SS = 0
                mov     ds, ax
                mov     es, ax
                cli
                mov     ss, ax
                mov     sp, 0FFFEh      ; place stack at 64KB
                sti
                xor     dh, dh
                push    dx              ; save DX, SI on stack for later
                push    si
                mov     ax, 1           ; clear the screen
                int     10h             ; - VIDEO - SET VIDEO MODE
                                        ; AL = mode
                mov     ax, 1003h
                mov     bl, 1           ; enable blinking mode
                int     10h             ; - VIDEO - TOGGLE INTENSITY/BLINKING BIT
                                        ; BL = 00h enable background intensity
                                        ; = 01h enable blink
                call    sub_1058
                mov     ah, 2
                xor     bx, bx
                mov     dx, 1214h
                int     10h             ; - VIDEO - SET CURSOR POSITION
                                        ; DH,DL = row, column (0,0 = upper left)
                                        ; BH = page number
                call    sub_1175
                call    sub_11BE
                pop     si
                pop     dx
                push    dx
                call    sub_132F
                jnb     short loc_103B
```

Figure 12. The decryption program performs initialization, and then calls several functions.

The function at 0x1175 calls the function 0x119D four times, with the arguments in BX of 4063, 4304, 4560, and 4831 (Figure 13). The function 0x119D, in turn, uses the timer and PC speaker to beep the speaker at 1193180/BX Hz (based on 1/12 of the standard 14.31818 MHz motherboard oscillator frequency for the 8254 timer). Using this conversion, the speaker beeps at 293 Hz, 277 Hz, 261 Hz, and 246 Hz. These are the notes D4, C#4, C4, and B3, which generates the “sad trombone” sound played when the ransom note opens. After playing the sound, the decryption program next calls 0x11BE.

```
sub_1175        proc near               ; CODE XREF: sub_1000+2B↑p
                mov     cx, 7
                mov     dx, 0A120h
                mov     bx, 4063
                call    sub_119D
                mov     bx, 4304
                call    sub_119D
                mov     bx, 4560
                call    sub_119D
                mov     bx, 4831
                call    sub_119D
                retn
sub_1175        endp
```

Figure 13. The function 0x1175 generates the "sad trombone" sound.

Function 0x11BE is complex because it is the “main loop” of the decryption program, containing the code to parse keyboard input and continue prompting the user to enter a digit until the correct encryption key is entered. It is easier to follow by noting that the function maintains a pointer to an error message in SI so that if any operation fails, the error message is already loaded. As shown in Figure 14, the function loops on keyboard input, and the first keys checked for are Backspace and ENTER, since they require special handling. Otherwise, the code branches to 0x121C, which begins a hexadecimal conversion.

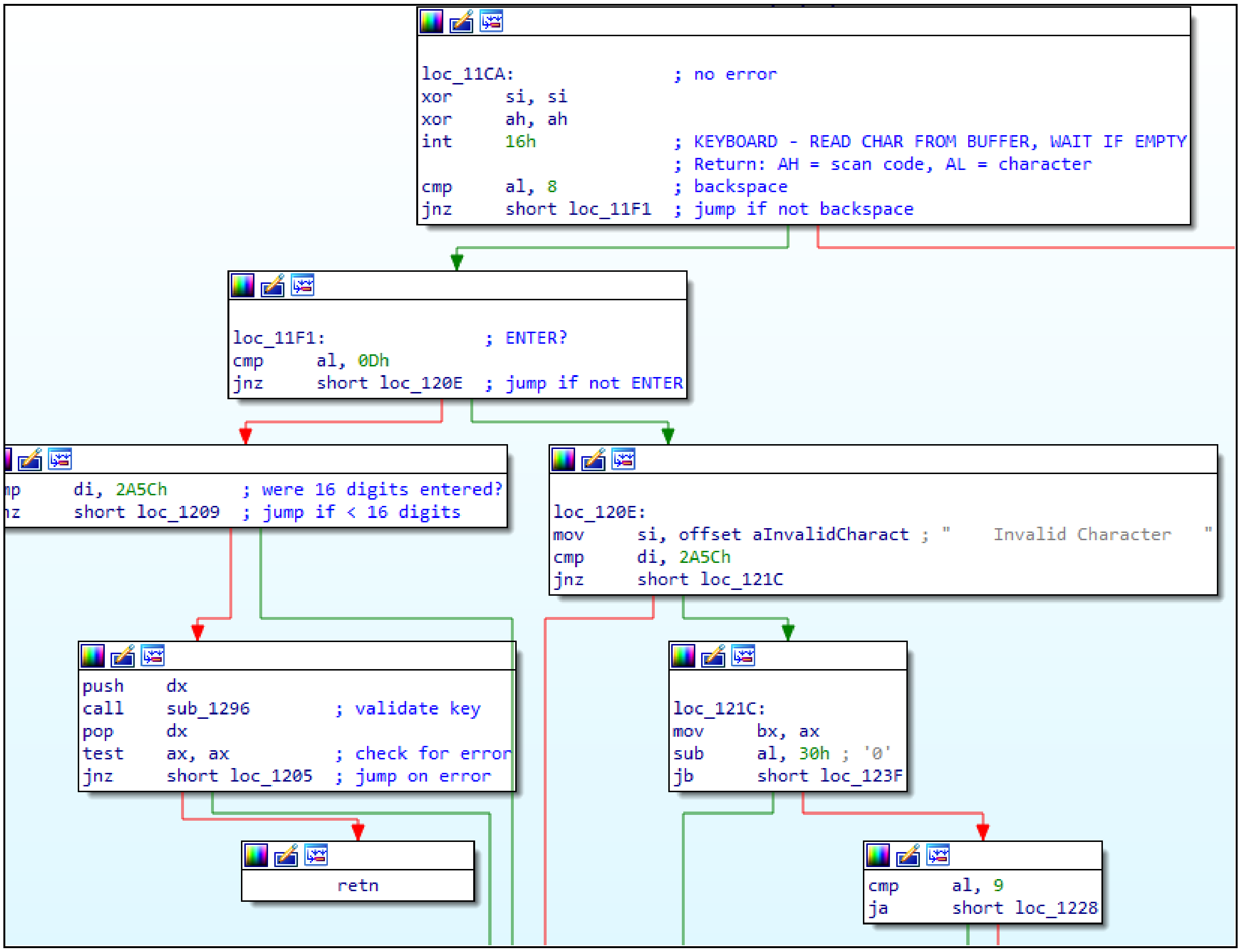

Figure 14. The function 0x11BE loops on keyboard input until the user enters the correct decryption key.

Further analysis of the function 0x11BE shows that as displayed in Figure 14, the interesting code, including the call to function 0x1296, is only reached if the user presses ENTER after already having successfully entered sixteen hexadecimal digits. The function 0x1296 must be a “key validation” function that returns zero or (as further analysis of 0x11BE demonstrates) a pointer to an error message in AX.

From 0x1296-0x12D3, the key validation function compacts the key input by the user that was stored as sixteen nibbles at 0x2A4C (likely to make “backspace” functionality and binary to ASCII formatting easier) into eight bytes at 0x2A5C. Next, the function XORs the first six bytes of the compacted key together with the six-byte victim ID at 0x19FC and ensures the result is 0x555555555555. If it is not, the function generates the message “Invalid Key.” Note that the last four hex digits of the entered key play no role in the “Invalid Key” message. The function expands the eight-byte compacted key into a sixteen-byte Blowfish decryption key, by concatenating onto the original eight bytes, those bytes XOR 0x555555555555 (Figure 15).

```
sub_1296        proc near               ; CODE XREF: sub_11BE+3E↑p
                push    si
                push    di
                sub     sp, 8
                mov     si, 2A4Ch       ; input key from user as 16 nibbles
                mov     di, 2A5Ch       ; key from user as 8 bytes
                mov     cx, 804h

loc_12A4:                               ; CODE XREF: sub_1296+16↓j
                lodsw                   ; get two nibbles in AL, AH
                shl     al, cl
                or      al, ah          ; pack them both into AL
                stosb                   ; store them at di++
                dec     ch
                jnz     short loc_12A4
                mov     si, 2A5Ch       ; key from user as 8 bytes
                mov     di, 19FCh       ; victim ID (embedded in program) as 8 bytes
                dec     cx              ; cx=3

loc_12B5:                               ; CODE XREF: sub_1296+29↓j
                lodsw
                xor     ax, [di]        ; ax = userkey[i] ^ victimid[i]
                inc     di
                inc     di              ; each hex digit of the input key from
                cmp     ax, 5555h       ; the user, XOR the victim ID, must be 0x5
                jnz     short loc_1304  ; if not, it's a bad key
                loop    loc_12B5        ; this only applies to the first six bytes (12 hex digits)
                mov     si, 2A5Ch       ; key from user as 8 bytes
                mov     di, 2A64h       ; Blowfish decryption key
                mov     cl, 4

loc_12C9:                               ; CODE XREF: sub_1296+3B↓j
                lodsw                   ; blowfish_key[i] = userkey[i]
                stosw
                xor     ax, 5555h
                mov     [di+6], ax      ; blowfish_key[i+8]=userkey[i]^0x55
                loop    loc_12C9        ; for four words or eight bytes
                mov     ax, 10h         ; blowfish_key is the userkey, but it is expanded to
                push    ax              ; 16 bytes by XORing each byte in 2nd copy by 0x55
```

Figure 15. The function 0x1296 converts from nibbles to bytes, performs an XOR validation against the victim ID, then expands from eight to sixteen bytes using another XOR.

From 0x12D3-0x12FE, the key validation function takes the sixteen-byte expanded key that was previously stored at 0x2A64 and uses it to Blowfish-256-ECB-encrypt (as could be determined through further analysis of the functions 0x1674, 0x1573, and 0x1619) the string “Test Str”. The resulting ciphertext is compared to that stored at 0x19F4 which is: 2E 21 57 82 3E A9 6C 6E. If this point is reached but the ciphertext fails to match, the error generated is “Incorrect Key.” Otherwise, the key is considered correct.

```
                mov     ax, 2A64h       ; Blowfish encryption key
                push    ax
                call    sub_1674        ; Blowfish init function taking the key
                mov     si, offset aTestStr ; "Test Str"
                mov     di, sp
                movsw
                movsw
                movsw
                movsw
                mov     bx, sp
                mov     ax, 1573h
                call    sub_1619
                mov     si, 19F4h
                mov     di, sp
                mov     cx, 4
                repe cmpsw
                jz      short loc_1300
                mov     ax, offset aIncorrectKey ; "      Incorrect Key     "
                jmp     short loc_1307
; ---------------------------------------------------------------------------

loc_1300:                               ; CODE XREF: sub_1296+63↑j
                xor     ax, ax
                jmp     short loc_1307
; ---------------------------------------------------------------------------

loc_1304:                               ; CODE XREF: sub_1296+27↑j
                mov     ax, offset aInvalidKey ; "       Invalid Key      "

loc_1307:                               ; CODE XREF: sub_1296+68↑j
                                        ; sub_1296+6C↑j
                add     sp, 8
                pop     di
                pop     si
                retn
```

Figure 16. The function 0x1296 encrypts the string "Test Str" using the derived key, and compares the ciphertext with a hard-coded value in the program.

Given the information:

- The encryption is Blowfish-256-ECB and so the key is 16 bytes and the block size is 8 bytes
- The victim ID is 34 87 B3 B4 1F 20.
- The first six bytes of the key must be the bytes of victim ID XOR 0x55, or 61 D2 E6 E1 4A 75.
    - The message “Invalid Key” means that this check failed and the Blowfish encryption check was not even attempted.
- The last eight bytes of the key are derived from the first eight via XOR with 0x55. Therefore, the bytes at offsets 8 through 13 the victim ID, and the bytes at offsets 14 and 15 correspond to the bytes at offsets 6 and 7 XOR 0x55.
- Bytes 6 and 7 are unknown and must be brute-forced to match the ciphertext when the string “Test Str” is encrypted.
    - The message “Incorrect Key” means that this ciphertext check was attempted, but the key did not match.

A brute force attack against the key completes within a few seconds, using a program like the one given in Figure 17.

```
$ python3 crack.py
b'61d2e6e14a754adc3487b3b41f201f89'
```

Since the second half of the key is derived from the first XOR 0x55, the key that must actually be entered into the decryption program is 61D2E6E14A754ADC.

```
#!/usr/bin/python3
import binascii
import struct
from Crypto.Cipher import Blowfish

VICTIM_ID = b"\x34\x87\xB3\xB4\x1F\x20"
PLAINTEXT = b"Test Str"
CIPHERTEXT = b"\x2E\x21\x57\x82\x3E\xA9\x6C\x6E"

xor_victim_id = b""
for c in VICTIM_ID:
    xor_victim_id += struct.pack("B", c ^ 0x55)

for b0 in range(256):
    for b1 in range(256):
        key = xor_victim_id + struct.pack("BB", b0, b1) \
            + VICTIM_ID + struct.pack("BB", b0 ^ 0x55, b1 ^ 0x55)
        cipher = Blowfish.new(key, Blowfish.MODE_ECB)
        ct = cipher.encrypt(PLAINTEXT)
        if ct == CIPHERTEXT:
            print(binascii.b2a_hex(key))
            break
```

Figure 17. Brute force attack program to discover the key.

The key 61D2E6E14A754ADC does cause the decryption program to decrypt the C: partition, as shown in Figure 2.
To assist in further study of the decryption program, a list of functions is given in Table 3.

Offset	Description
0x1000	main
0x1058	Display ransom note
0x1163	INT 10H call with preserved registers
0x1175	Play trombone sound
0x1194	Error beep
0x119d	Play 1193180 / bx Hz for cx:dx us
0x11be	Keep asking for key until it is correct
0x1296	Check whether the entered key is correct
0x130d	Update error/status line at bottom of screen
0x132f	Decrypt the disk

0x13fd	Recalculate the percentage done
0x142b	Print the percentage done
0x146d	Print the cylinders done
0x14d7	Remove the “encrypted” bit from the “active” field in the partition table
0x152b	Blowfish “F” function on dx:ax
0x1573	Blowfish “encipher” function on pointer to left/right values in bx
0x15c5	Blowfish “decipher” function on pointer to left/right values in bx
0x1619	Blowfish “crypt” function taking buffer in bx and pointer to encipher/decipher in ax
0x1660	Blowfish-ECB decrypt
0x1674	Blowfish “init” function

Table 3. List of functions in the decryption program.

Final Flag

bl0wf1$h_3ncrypt10n_0f_p@rt1t10n_1n_r3al_m0d3@flare-on.com