Flare-On 5: Challenge 8 Solution – doogie.bin

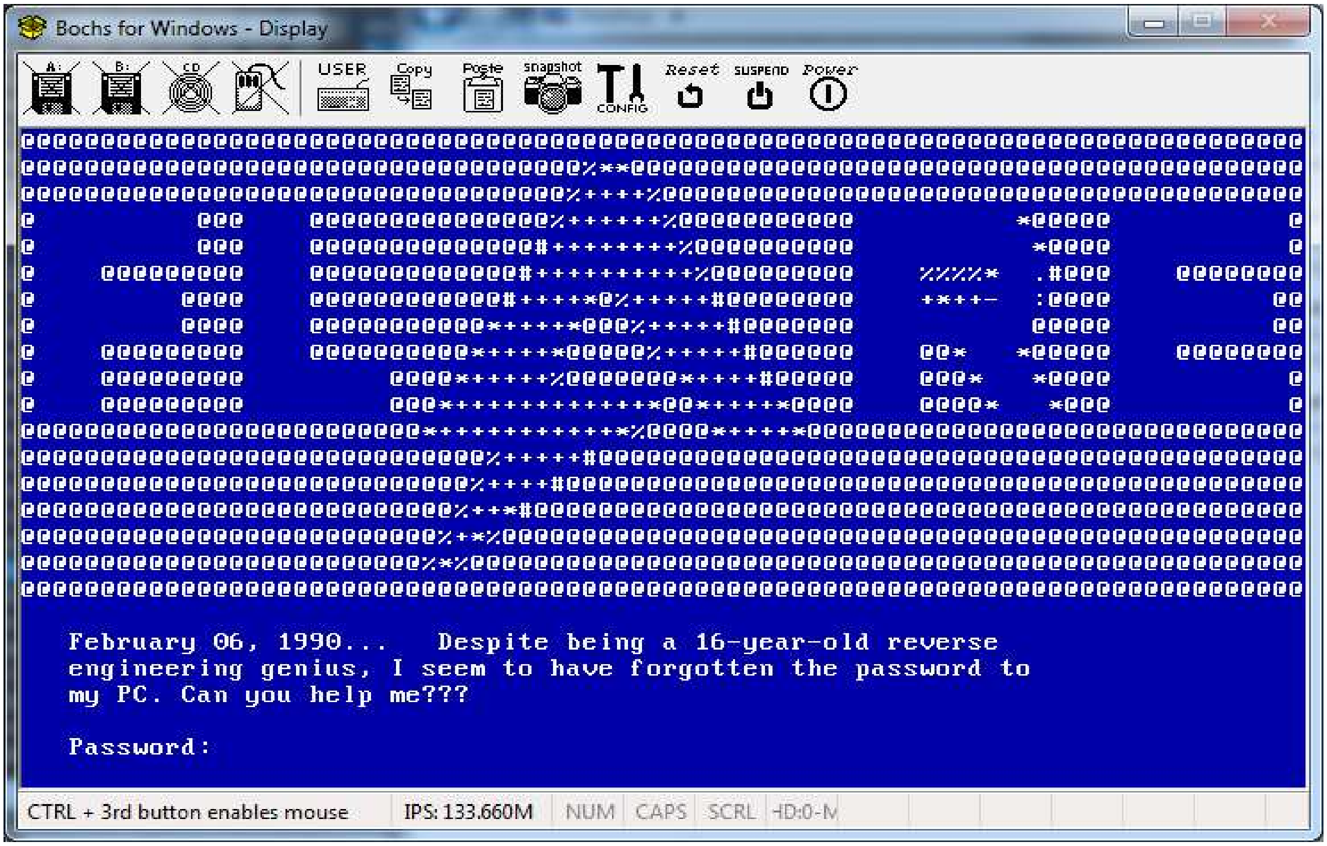
Challenge Author: Matt Williams (@0xmwilliams)

`doogie.bin` is a boot sector followed by additional supporting sectors.

```
$ file doogie.bin
doogie.bin: DOS/MBR boot sector
```

Figure 1 – `doogie.bin` identification

Using an emulator such as Bochs[1] to execute the boot code displays the prompt shown in Figure 2. Those familiar with a certain television show[2] from the early 90s may have appreciated the format of the journal entry.



```
February 06, 1990...   Despite being a 16-year-old reverse
engineering genius, I seem to have forgotten the password to
my PC. Can you help me???

Password:
```

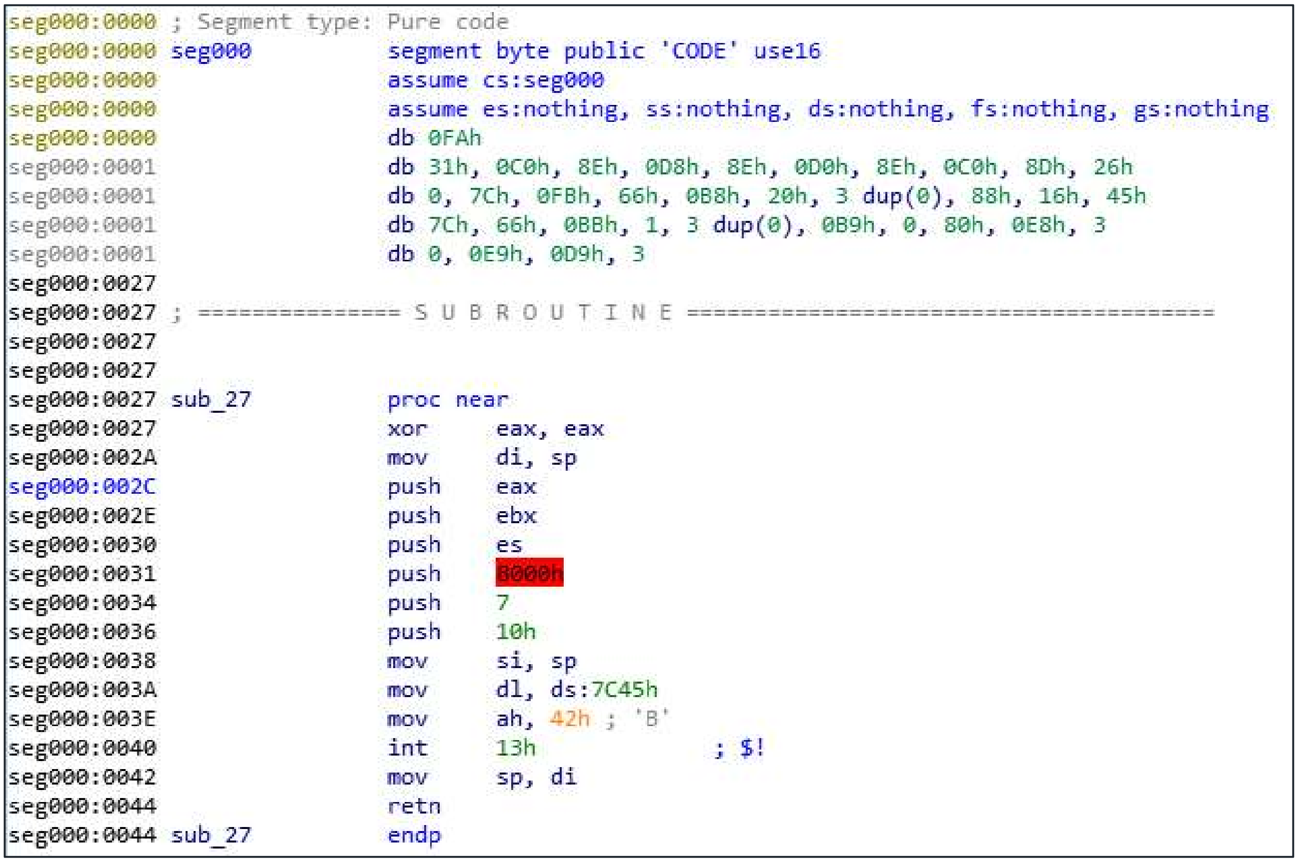
Figure 2 – Password prompt

---

[1] https://countuponsecurity.com/2017/07/02/analysis-of-a-master-boot-record-eternalpetya/

[2] https://en.wikipedia.org/wiki/Doogie_Howser,_M.D.

Disassembling `doogie.bin` using IDA Pro[3] in 16-bit mode results in the instructions shown in Figure 3.



```
seg000:0000 ; Segment type: Pure code
seg000:0000 seg000          segment byte public 'CODE' use16
seg000:0000                 assume cs:seg000
seg000:0000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000                 db 0FAh
seg000:0001                 db 31h, 0C0h, 8Eh, 0D8h, 8Eh, 0D0h, 8Eh, 0C0h, 8Dh, 26h
seg000:0001                 db 0, 7Ch, 0FBh, 66h, 0B8h, 20h, 3 dup(0), 88h, 16h, 45h
seg000:0001                 db 7Ch, 66h, 0BBh, 1, 3 dup(0), 0B9h, 0, 80h, 0E8h, 3
seg000:0001                 db 0, 0E9h, 0D9h, 3
seg000:0027
seg000:0027 ; =============== S U B R O U T I N E =======================================
seg000:0027
seg000:0027
seg000:0027 sub_27          proc near
seg000:0027                 xor     eax, eax
seg000:002A                 mov     di, sp
seg000:002C                 push    eax
seg000:002E                 push    ebx
seg000:0030                 push    es
seg000:0031                 push    8000h
seg000:0034                 push    7
seg000:0036                 push    10h
seg000:0038                 mov     si, sp
seg000:003A                 mov     dl, ds:7C45h
seg000:003E                 mov     ah, 42h ; 'B'
seg000:0040                 int     13h             ; $!
seg000:0042                 mov     sp, di
seg000:0044                 retn
seg000:0044 sub_27          endp
```

Figure 3 – Initial disassembly using IDA Pro

Converting the initial bytes to code yields instructions that eventually call the `sub_27` function. This function contains an interrupt `13h` instruction. Immediately before this interrupt, the value `42h` is moved into the AH register. This indicates an “extended read sectors from drive” operation[4]. As part of this operation a Disk Address Packet (DAP), which contains the arguments needed to perform the sector read operation, is built on the stack beginning at offset `0x2C`. Table 1 below lists the byte values associated with each DAP element. As a result, seven sectors, beginning at sector `1`, are read into memory address `0x8000`. The bootloader jumps to this address after returning from `sub_27`.

---

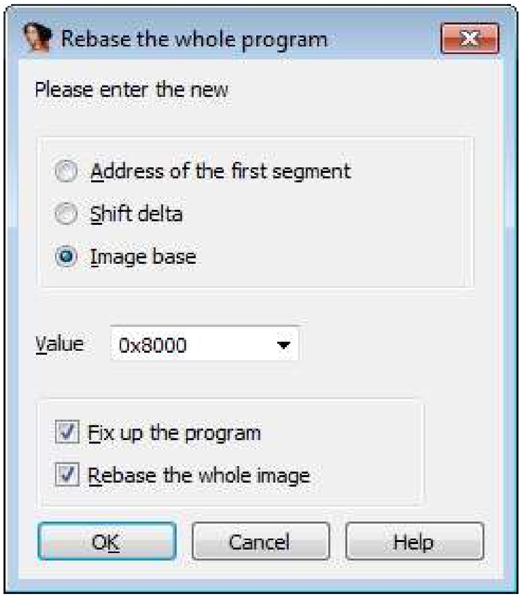
[3] https://www.hex-rays.com/products/ida/support/download_freeware.shtml

[4] https://en.wikipedia.org/wiki/INT_13H#INT_13h_AH=42h:_Extended_Read_Sectors_From_Drive

DAP Element	Hex Bytes
Size	`10`
Reserved	`00`
# Sectors to Read	`00 07`
Destination Address	`00 00 80 00`
Start Sector	`00 00 00 00 00 00 00 01`

Table 1: Initial sector read DAP elements

To clean up the disassembly of the seven sectors, we can extract them from `doogie.bin`, open them in a separate IDA Pro instance, and alter the base address to reflect the new location in memory. This is accomplished via the *Edit -> Segments -> Rebase program…* menu as shown in Figure 4 below.



Rebase the whole program

Please enter the new

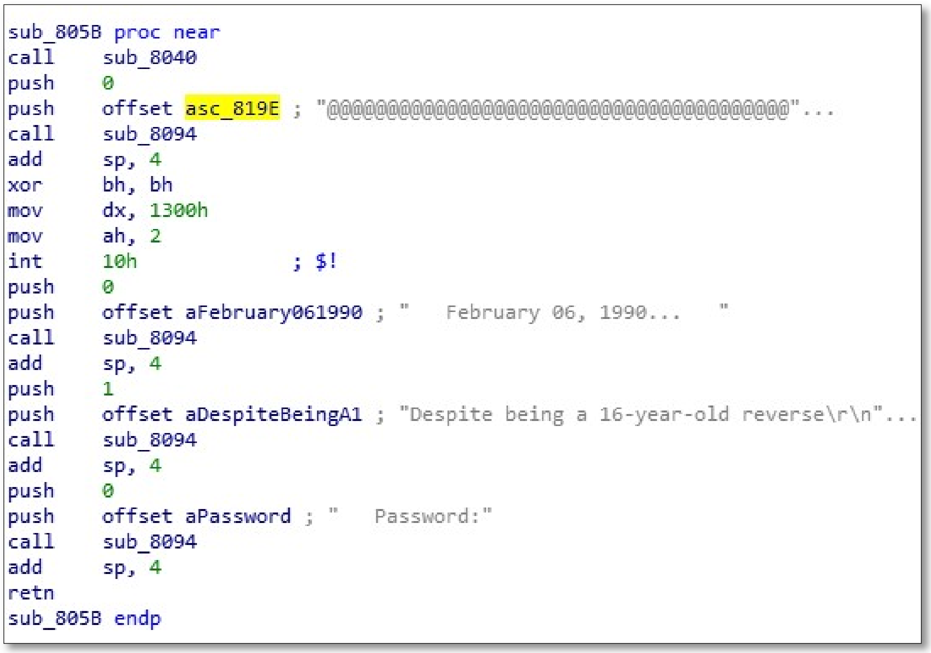
○ Address of the first segment
○ Shift delta
● Image base

Value 0x8000

☑ Fix up the program
☑ Rebase the whole image

Figure 4 – Rebasing to `0x8000`

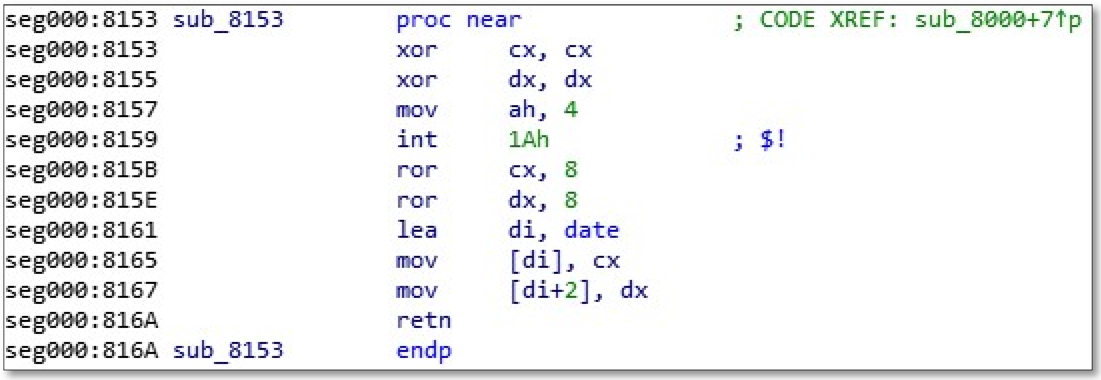
The first function called in the rebased disassembly is `sub_805B`. Note IDA Pro does not interpret memory addresses in the `0x8000` range correctly. To remedy this, convert the values to offsets using the “O” key. The data stored at these memory addresses can be converted to strings using the “A” key. As a result, the `sub_805B` disassembly should appear similar to Figure 5 below, which allows us to quickly recognize this function is responsible for printing the password prompt shown in Figure 2.



```
sub_805B proc near
call    sub_8040
push    0
push    offset asc_819E ; "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"...
call    sub_8094
add     sp, 4
xor     bh, bh
mov     dx, 1300h
mov     ah, 2
int     10h             ; $!
push    0
push    offset aFebruary061990 ; "   February 06, 1990...   "
call    sub_8094
add     sp, 4
push    1
push    offset aDespiteBeingA1 ; "Despite being a 16-year-old reverse\r\n"...
call    sub_8094
add     sp, 4
push    0
push    offset aPassword ; "   Password:"
call    sub_8094
add     sp, 4
retn
sub_805B endp
```

Figure 5 – Prompt function

The next function of interest is `sub_8153` whose disassembly is shown in Figure 6 below.



```
seg000:8153 sub_8153        proc near               ; CODE XREF: sub_8000+7↑p
seg000:8153                 xor     cx, cx
seg000:8155                 xor     dx, dx
seg000:8157                 mov     ah, 4
seg000:8159                 int     1Ah             ; $!
seg000:815B                 ror     cx, 8
seg000:815E                 ror     dx, 8
seg000:8161                 lea     di, date
seg000:8165                 mov     [di], cx
seg000:8167                 mov     [di+2], dx
seg000:816A                 retn
seg000:816A sub_8153        endp
```

Figure 6 – sub_8153 disassembly

At `0x8159`, interrupt 1Ah[5] reads the current date from the real time clock. By convention, the month and day are stored in the DX register and the year is stored in CX. The four-byte date value is written to `0x87EE` using the big-endian byte format YY YY MM DD.

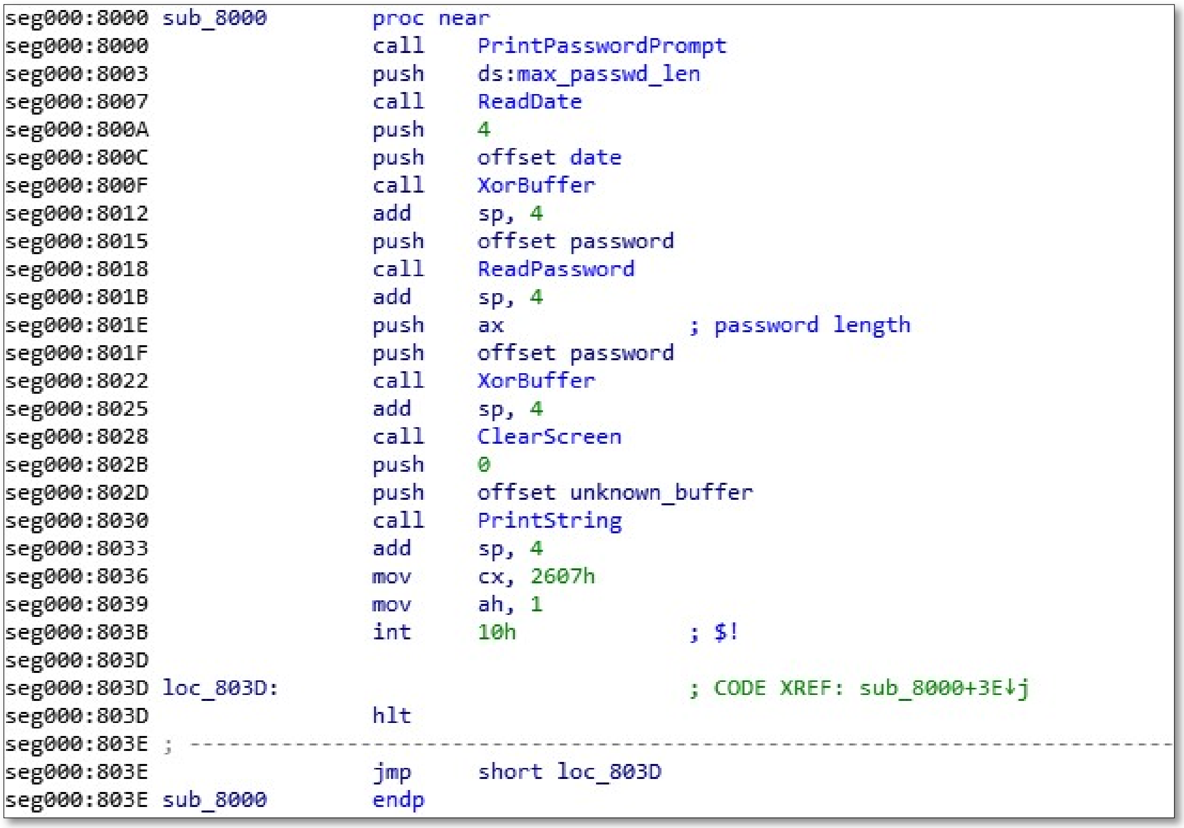
The next function, `sub_816B`, accepts the date value and the hard-coded value 4 as arguments. It uses the date value as a multi-byte repeating XOR key with length 4 to decrypt a NULL-terminated buffer at `0x8809`. Clearly the XOR key varies based on the current date; therefore, you may have correctly assumed the date from the journal entry is needed to properly decrypt the unknown buffer. This

---

[5] https://en.wikipedia.org/wiki/BIOS_interrupt_call#Interrupt_table

assumption is addressed later in the solution.

After XORing the unknown buffer at `0x8809`, the sample attempts to read user input in `sub_80D5`. Examination of the function reveals it accepts a password string of maximum length 20 (`WORD` value pushed at `0x8003`) which it stores at `0x87F4`. The length of the string entered by the user is returned in the AX register. At this stage our marked-up disassembly of `sub_8000` might look something like Figure 7.



```
seg000:8000 sub_8000        proc near
seg000:8000                 call    PrintPasswordPrompt
seg000:8003                 push    ds:max_passwd_len
seg000:8007                 call    ReadDate
seg000:800A                 push    4
seg000:800C                 push    offset date
seg000:800F                 call    XorBuffer
seg000:8012                 add     sp, 4
seg000:8015                 push    offset password
seg000:8018                 call    ReadPassword
seg000:801B                 add     sp, 4
seg000:801E                 push    ax              ; password length
seg000:801F                 push    offset password
seg000:8022                 call    XorBuffer
seg000:8025                 add     sp, 4
seg000:8028                 call    ClearScreen
seg000:802B                 push    0
seg000:802D                 push    offset unknown_buffer
seg000:8030                 call    PrintString
seg000:8033                 add     sp, 4
seg000:8036                 mov     cx, 2607h
seg000:8039                 mov     ah, 1
seg000:803B                 int     10h             ; $!
seg000:803D
seg000:803D loc_803D:                               ; CODE XREF: sub_8000+3E↓j
seg000:803D                 hlt
seg000:803E ; ---------------------------------------------------------------------------
seg000:803E                 jmp     short loc_803D
seg000:803E sub_8000        endp
```

Figure 7 – `sub_8000` marked-up disassembly

After reading the password at `0x8018`, the `XorBuffer` function (`sub_816B`) uses the password as the multi-byte repeating XOR key to further decrypt the unknown buffer at `0x8809`. After clearing the screen, the unknown buffer is printed and the sample enters an infinite `hlt` loop.

Thus far we understand two multi-byte XOR keys are used to decrypt the bytes at `0x8809`, which are likely an encrypted solution of some kind. We also understand this unknown buffer is hard-coded and does not update itself based on the current date; therefore, a specific date is needed to properly decrypt the solution. Also, given the size of the NULL-terminated buffer (`0x49A` bytes), the decrypted solution is unlikely to be the email address we're after. Instead, considering the function that clears the screen at `0x8028`, the solution is likely to be in the form of ASCII art.

Assuming the date from the journal entry is the first key, we could XOR-decrypt the unknown buffer using hexadecimal key `0x19900206`. Doing so results in the bytes shown below in Figure 8. Non-printable characters are represented with a "`.`" and new-line characters were removed for readability. Keen observers may have spotted repeated strings like "`MALWARE`" or "`OPERATE`" and derived the correct second XOR key based on surrounding text.

```
QWH]JYL.ANMAB.YJ]Q^ERATEONMALWYJ]QWH].OTEWVUALWAREQWH]JYL.AcgYTOARE0WH.R.L]?NM8TO.REIOPER
ATEWVUALW8J].OH]JATEONMYTOARG0WH..kL]WNMALOYJEIOPERO.]W>MALWAREIOP]JYTEONUYTWYJ]IOPERAL]W
NMALOYJhcWH]RAT.WV=ALWAR]QWHGRATEONMALWAJ]QOPE.YL5OVUYTO.\EIWH]RATEWVUlfOYJ]QWH5PATEONMAN
.Y.KIOPERATEONUYTOYJ]9MPEJYLEMVU.LOYJEIOP]JYyoWVUA8OY.EIOH]JATEOVUYLWAREIOPERYL]ONMALWAR]
QWPEJYLEWVUALWAJ]Qbz]JYTE;VU.LW8J].OP.JY$EONMALWAREQWHERATEONMYTOAR]QWP]JYTEAUY<zkJ]QOPE&
YL.ONO8TOYJ5KOP]JYL]WVUATOYREIOPERAL]WNMYTOAJ]QWH]J1VhecgAB.YJ]QWH].OTEO@YTOAJ]QOPERATEON
QOP.YJ5KOPEP8L]NYT'CREQWHERATEONMALzkJ]QOP.J.TEWVUATOYREIOH]JATEONMALWA.oQWHERYL]ONUYTWYJ
]QWHEJYLEOVUYT.OREQWH.JYLEAUY.YAREIOPERATK.VU.BWAJ]QWH.\ATheVUYLWYJ]..H]"AL]WNMALOYJEIOPE
WNM.OVUY<UAR.Q?PE+Y.EONMALWA.]QMR]J.T]WVMCTO.RhcWH]RA-
]WVU1NWAJ]QOPERYL]O@YTOYJ]IWH]RATEWVUYTOYJEQWH]JYT]WVMATOYR]QWPEJYLEbd4YT.OREIOPK.YT]WVMA
WVMA]IWH]RAL]WNUYTWARE0W.KRATEONMALWAR<QW^KJY$EWVUALOYJEdePG+YL]WVUYT'CR]QWPERAL]WNO8TOYJ
]QOH]JATEONO8TOYJEIOPERATEOL4YT'CREQWHERYL]OcglfWAREIA.]JYL.O@YT.OREQWH]J.Z.WV.Oa}AREI.H]
"CTEOUYNUYJ.IWH]RCL]WNOYT.lxEIOP]JYTEONMYTOAR]QWP]JYTEWVUALOYJhc.H.R8L]@MAL.YJKGWH5RYL]ON
ONUYTW]Qbz<J1TEM7UYTO1RG0WH5PAT]WVMATOYREQWHhx
```

Figure 8 – XOR-decrypted bytes using journal date (modified for readability)

Our task is to determine the multi-byte XOR key that will successfully decrypt what is likely an ASCII art solution. For those new to breaking a repeating multi-byte XOR key (or cryptography in general), I encourage you to review challenges published by http://cryptopals.com. One of the challenges[6] specifically addresses our current situation. There are numerous write-ups published for these challenges that cover various approaches to solving this problem.

A common first step in breaking a multi-byte XOR key is determining a probable key length. One tool that assists with this is `xortool`[7]. Executing it against our ciphertext using the maximum key length 20 produces the results shown in Figure 9, which indicates the probable key length is 17 bytes. We could brute-force all 256 one-byte keys, but this does not produce meaningful results. For those curious about determining probable key lengths using tools like `xortool`, I encourage to review this blog post[8] from Dave Hull.

```
> python xortool -m 20 ciphertext.bin
The most probable key lengths:
   1:   44.0%
  17:   56.0%
Key-length can be 17*n
```

---

[6] http://cryptopals.com/sets/1/challenges/6

[7] https://github.com/hellman/xortool

[8] https://trustedsignal.blogspot.com/2015/06/xord-play-normalized-hamming-distance.html

```
Most possible char is needed to guess the key!
```

Figure 9 – `xortool` *key length result*

Unfortunately, providing `xortool` with a probable most-frequent character (e.g., `0x20`) does not produce an obvious solution as shown in Figure 10.

```
> python xortool -l 17 -c 0x20 xored.bin
4 possible key(s) of length 17:
qwh}jyteonuatoyj}
qwh}jyteonuytoyj}
qwh}ryteonuatoyj}
qwh}ryteonuytoyj}
Found 4 plaintexts with 95.0%+ printable characters
See files filename-key.csv, filename-char_used-perc_printable.csv
```

Figure 10 – `xortool` *brute-force result*

The same can be said for `xortool`'s "`-o`" option, which brute-forces the XOR key by restricting the plaintext result to printable ASCII characters. However, it does produce possible keys that contain partial or full strings observed previously, as shown in the `xortool` output from Figure 11. This output offers a hint about the correct XOR key.

```
<cut>
xortool_out\032.out;ioperal7dwvmylware
xortool_out\033.out;ioperal7dwvmalware
xortool_out\034.out;iopejal7dwvmylware
xortool_out\035.out;iopejal7dwvmalware
<cut>
```

Figure 11 – `xortool` *brute-force possible keys*

At this stage it appears we need to devise our own brute-force solution. This will allow us to tailor our input and desired output, which may reveal interesting patterns that can be used to discover the correct key. Using the bytes produced by XORing the unknown buffer with the journal date (see Figure 8), we can collect encrypted bytes associated with a given XOR key position within the 17-byte repeating XOR key. For example, we'll combine the bytes at offsets `0x00`, `0x11`, `0x22`, etc. into a single sequence. These bytes will be XORed with the first character of the XOR key. The same will be done for bytes at offsets `0x01`, `0x12`, `0x23`, etc., which will be XORed with the second byte of the XOR key. The Python snippet in Figure 12 demonstrates collecting these encrypted sequences for each of the 17 XOR key positions.

```python
KEY_LENGTH = 17

# read current ciphertext from provided file
with open(sys.argv[1], 'rb') as f:
    ciphertext = f.read()
```

```
# initialize key position dictionary
encrypted_bytes_by_key_position = {x: "" for x in range(KEY_LENGTH)}

# collect encrypted bytes for each key position
for i in range(len(ciphertext)):
    encrypted_bytes_by_key_position[i % KEY_LENGTH] += ciphertext[i]
```

Figure 12 – Collecting encrypted bytes at each key position

To further illustrate the collection process, Figure 13 below shows encrypted bytes within the ciphertext to be decrypted by the first byte of the 17-byte XOR key. Notice two characters have a much higher frequency than others.

```
00000000  51 51 51 51 30 49 0B 30 49 49 49 63 51 51 49 51  QQQQ0I.0IIIcQQIQ

00000010  49 39 49 49 49 51 51 0B 51 51 51 4B 49 51 51 51  I9IIIQQ.QQQKIQQQ

00000020  4B 51 51 49 51 51 51 49 51 0B 49 51 51 63 51 49  KQQIQQQIQ.IQQcQI

00000030  51 51 49 49 30 51 64 51 51 49 51 49 51 49 49 49  QQII0QdQQIQIQIII

00000040  51 63 47 51 30 51                                QcGQ0Q
```

Figure 13 – Encrypted bytes at key position 0

Once we’ve collected all 17 encrypted byte sequences, we can treat each sequence as being encrypted with a *single-byte* XOR key. From there we can determine which single-byte XOR keys produces only printable ASCII output for each encrypted sequence. The Python snippet in Figure 14 performs this process. Note the `xor_buffer` and `is_printable` functions are not shown for brevity.

```
# determine which keys in range 0x20-0x7E produce ASCII output
for key_pos, encrypted_bytes in encrypted_bytes_by_key_position.iteritems():
    for key_candidate in range(0x20, 0x7E):
        # XOR encrypted bytes with key candidate
        xor_result = xor_buffer(encrypted_bytes, chr(key_candidate))

        # is the resulting string printable?
        if is_printable(xor_result):
            key_candidates_by_position[key_pos].append(chr(key_candidate))
```

Figure 14 – Finding keys that product printable output

Printing the character representation for each candidate in `key_candidates_by_position` (Figure 14) results in the list shown in Figure 15. Most key positions have three or fewer possible candidates, which significantly reduces the number of possible 17-byte keys.

```
Key Position 0: i, n
Key Position 1: 5, 6, h, o
Key Position 2:  , ", #, $, %, &, ', (, ), *, +, ,, ., 0, 1, 2, 3, 4, 5, 6, 8, 9, :, ;,
<, =, >, ?, p, s, w
Key Position 3: a, b, e
```

```
Key Position 4: r, u
Key Position 5: a, b, f
Key Position 6: p, s, t
Key Position 7: 8, <, ?, b, e
Key Position 8: 6, h, o
Key Position 9: i, n
Key Position 10: j, m, n
Key Position 11: a, e, f
Key Position 12: 5, k, l
Key Position 13: p, w
Key Position 14: a, f
Key Position 15:  , !, ", $, %, &, ', (, ), *, +, ,, ., /, 0, 1, 2, 3, 4, 6, 7, 8, 9, :,
;, <, =, >, ?, r, u
Key Position 16: b, e
```

Figure 15 – Key candidates by position

Next, we’ll examine the plaintext produced by each one of these candidates. Because ASCII art commonly uses repeating characters, we’ll measure character frequency in the generated plaintext. The Python snippet in Figure 16 performs these steps and uses the `collections` module to print the three most-frequent characters in the plaintext produced by each key position candidate.

```python
for key_pos, encrypted_bytes in encrypted_bytes_by_key_position.iteritems():
    print 'Key Position %d' % key_pos
    for key_candidate in key_candidates_by_position[key_pos]:
        xor_result = xor_buffer(encrypted_bytes, key_candidate)
        top_3_chars = collections.Counter(xor_result).most_common(3)
        print 'Character: %c -> %s' % (key_candidate, top_3_chars)
```

Figure 16 – Printing character frequency for each key position candidate

Partial results of the Figure 16 code snippet are shown in Figure 17. They reveal common most-frequent characters across all 17 key positions. Namely, “8” and the space character occur most frequently for at least one candidate at each position. The same can be said for “?” and “’”.

```
Key Position 0
Character: i -> [('8', 34), (' ', 21), ('Y', 4)]
Character: n -> [('?', 34), ("'", 21), ('^', 4)]
Key Position 1
Character: 5 -> [('b', 34), ('z', 25), ('>', 3)]
Character: 6 -> [('a', 34), ('y', 25), ('=', 3)]
Character: h -> [('?', 34), ("'", 25), ('c', 3)]
Character: o -> [('8', 34), (' ', 25), ('d', 3)]
Key Position 2
       <cut>
Key Position 3
Character: a -> [('<', 28), ('$', 25), ('f', 4)]
Character: b -> [('?', 28), ("'", 25), ('e', 4)]
Character: e -> [('8', 28), (' ', 25), ('b', 4)]
Key Position 4
Character: r -> [(' ', 27), ('8', 27), ('"', 4)]
Character: u -> [("'", 27), ('?', 27), ('%', 4)]
Key Position 5
Character: a -> [('8', 29), (' ', 26), ('b', 3)]
Character: b -> [(';', 29), ('#', 26), ('a', 3)]
Character: f -> [('?', 29), ("'", 26), ('e', 3)]
```

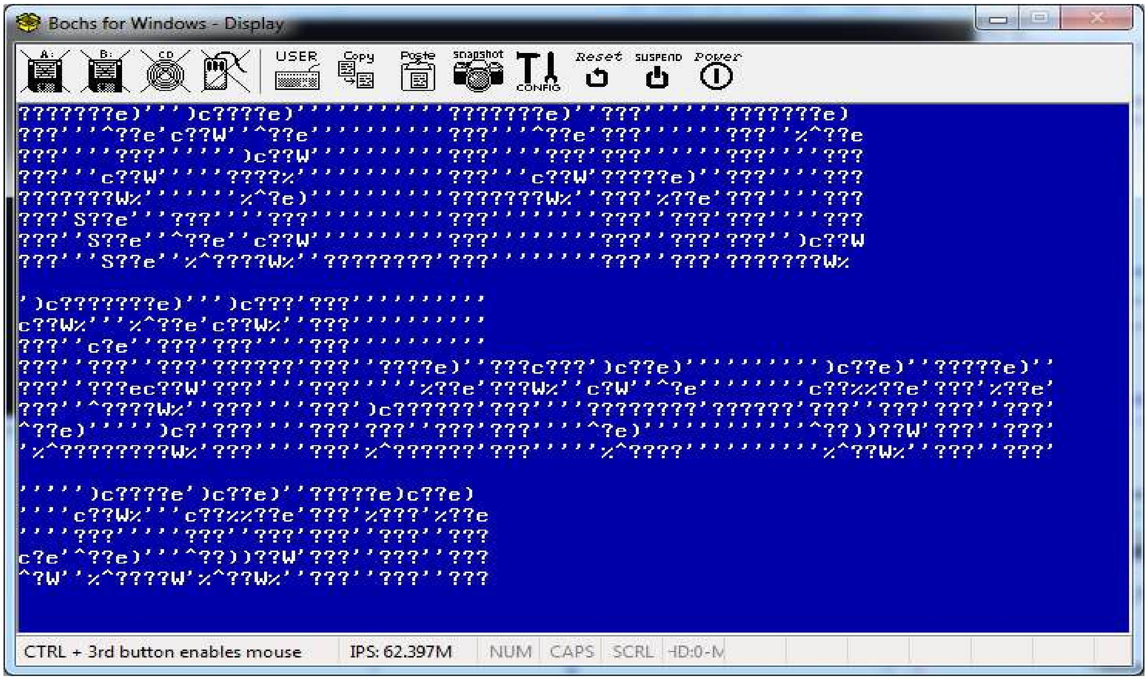
Figure 17 – Character frequency

Assuming the ASCII art will contain a significant number of repeated characters, we can group key position candidates that produced the same most-frequent characters. For example, we can select all key candidates whose most-frequent characters match “?” and “’”. At key position 1 we’d select candidate “n”, at position 2 we’d select “h”, and so on. Doing so produces two possible XOR keys listed in Figure 18.

```
nhwbufsbhijfkpfub
ioperateonmalware
```

Figure 18 – Possible XOR keys

Conveniently, both keys produce a legible ASCII art solution as shown in Figure 19 and Figure 20.

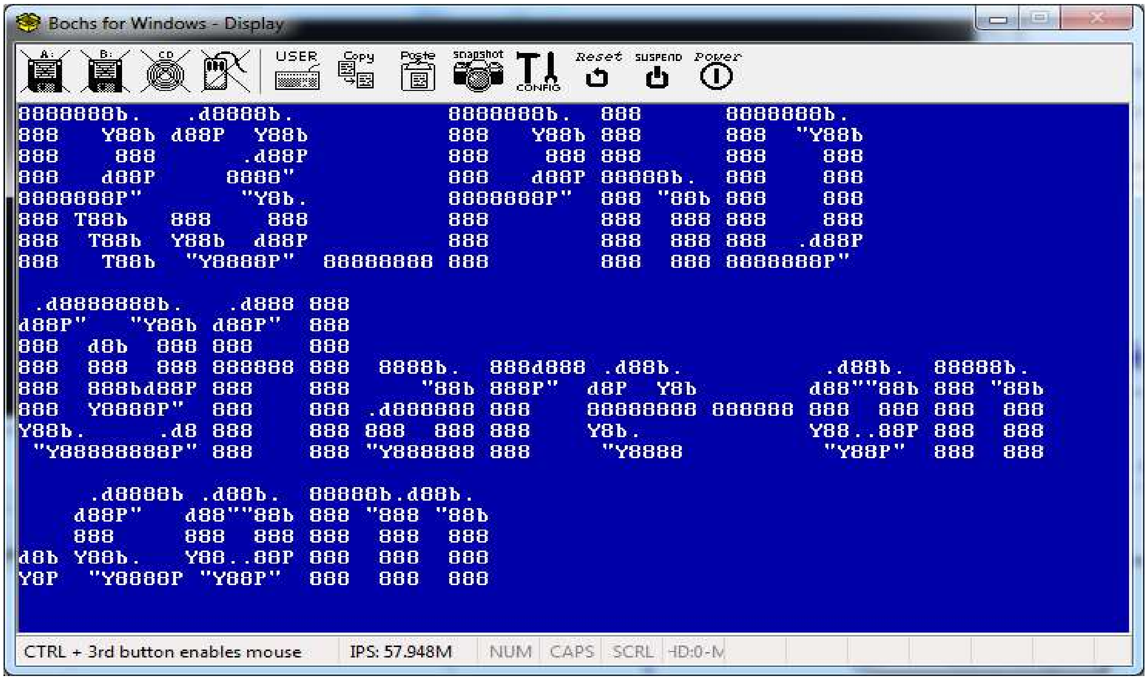


```
???????e)''')c????e)'''''''''''???????e)''???''''''???????e)
???'''^??e'c??W''^??e''''''''''???'''^??e'???''''''???''%^??e
???''''???'''''')c??W''''''''''???''''???'???''''''???''''???
???'''c??W'''''????%'''''''''''???'''c??W'?????e)''???''''???
???????W%'''''''%^?e)''''''''''???????W%''???'%??e'???''''???
???'S??e'''???''''???''''''''''???''''''''???''???'???''''???
???''S??e''^??e''c??W''''''''''???''''''''???''???'???'')c??W
???'''S??e''%^????W%''????????'???''''''''???''???'???????W%

')c???????e)''')c???'???''''''''''
c??W%'''%^??e'c??W%''???''''''''''
???''c?e''???'???''''???''''''''''
???''???''???'??????'???''????e)''???c???')c??e)'''''''''')c??e)''?????e)''
???''???ec??W'???''''???'''''%??e'???W%''c?W''^?e''''''''c??%%??e'???'%??e'
???''^????W%''???''''???')c??????'???''''????????'??????'???''???'???''???'
^??e)''''')c?'???''''???'???''???'???''''^?e)''''''''''''^??))??W'???''???'
'%^????????W%'???''''???'%^??????'???'''''%^????''''''''''%^??W%''???''???'

''''')c????e')c??e)''?????e)c??e)
''''c??W%'''c??%%??e'???'%???'%??e
''''???'''''???''???'???''???''???
c?e'^??e)'''^??))??W'???''???''???
^?W''%^????W'%^??W%''???''???''???
```

Figure 19 – ASCII art solution generated by the key `nhwbufsbhijfkpfub`



```
8888888b.   .d8888b.           8888888b.  888      8888888b.
888   Y88b d88P  Y88b          888   Y88b 888      888  "Y88b
888    888      .d88P          888    888 888      888    888
888   d88P     8888"           888   d88P 88888b.  888    888
8888888P"       "Y8b.          8888888P"  888 "88b 888    888
888 T88b   888    888          888        888  888 888    888
888  T88b  Y88b  d88P          888        888  888 888  .d88P
888   T88b  "Y8888P"  88888888 888        888  888 8888888P"

 .d8888888b.   .d888 888
d88P"   "Y88b d88P"  888
888  d8b  888 888    888
888  888  888 888888 888  8888b.  888d888 .d88b.          .d88b.  88888b.
888  888bd88P 888    888     "88b 888P"  d8P  Y8b        d88""88b 888 "88b
888  Y8888P"  888    888 .d888888 888    88888888 888888 888  888 888  888
Y88b.     .d8 888    888 888  888 888    Y8b.            Y88..88P 888  888
 "Y88888888P" 888    888 "Y888888 888     "Y8888          "Y88P"  888  888

     .d8888b .d88b.  88888b.d88b.
    d88P"   d88""88b 888 "888 "88b
    888     888  888 888  888  888
d8b Y88b.   Y88..88P 888  888  888
Y8P  "Y8888P "Y88P"  888  888  888
```

Figure 20 – ASCII art solution generated by the key `ioperateonmalware`

Given the more legible solution (Figure 20), we can understand why the uppercase strings “OPERATE” and “MALWARE” were present in the ciphertext (Figure 8). This is due to the significant number of spaces (hex value: `0x20`) in the final solution. The result of XORing a lowercase character with `0x20` produces the uppercase equivalent and vice versa. Therefore, the proper key contains the lowercase strings "`operate`" and "`malware`".

Figure 21 below contains a Python script that may be used to reproduce the brute-force approach outlined above.

```python
import sys
import collections

KEY_LENGTH = 17


def is_printable(buf):
    for c in buf:
        if ' ' <= c <= '~' or c in '\n\r\t':
            continue
        else:
            return False

    return True


def xor_buffer(buf, key):
    key_len = len(key)
    xored_buf = ""
    for i in range(len(buf)):
        xored_buf += chr((ord(buf[i]) ^ ord(key[i % key_len])))

    return xored_buf


def gen_key_candidate_char_frequency(encrypted_bytes_by_key_position, key_candidates_by_position):
    for key_pos, encrypted_bytes in encrypted_bytes_by_key_position.iteritems():
        print 'Key Position %d' % key_pos
        for key_candidate in key_candidates_by_position[key_pos]:
            xor_result = xor_buffer(encrypted_bytes, key_candidate)
            top_3_chars = collections.Counter(xor_result).most_common(3)
            print 'Character: %c -> %s' % (key_candidate, top_3_chars)


def find_key_candidates_by_position(encrypted_bytes_by_key_position):
    # initialize list of candidates for each key position
    key_candidates_by_position = {x: [] for x in range(KEY_LENGTH)}

    # determine which keys in range 0x20-0x7E produce ASCII output
    for key_pos, encrypted_bytes in encrypted_bytes_by_key_position.iteritems():
        for key_candidate in range(0x20, 0x7E):
            # XOR encrypted bytes with key candidate
            xor_result = xor_buffer(encrypted_bytes, chr(key_candidate))

            # is the resulting string printable?
            if is_printable(xor_result):
                key_candidates_by_position[key_pos].append(chr(key_candidate))
```

```
    return key_candidates_by_position


def extract_encrypted_sequences(ciphertext):
    # initialize key position dictionary
    encrypted_bytes_by_key_position = {x: "" for x in range(KEY_LENGTH)}

    # collect encrypted bytes for each key position
    for i in range(len(ciphertext)):
        encrypted_bytes_by_key_position[i % KEY_LENGTH] += ciphertext[i]

    return encrypted_bytes_by_key_position


def main():
    with open('doogie.bin', 'rb') as f:
        doogie_bytes = f.read()

    # decrypt encrypted buffer using journal date
    date_decrypted = xor_buffer(doogie_bytes[0xA09:0xEA3], '19900206'.decode('hex'))

    encrypted_bytes_by_key_position = extract_encrypted_sequences(date_decrypted)
    key_candidates_by_position = find_key_candidates_by_position(encrypted_bytes_by_key_position)
    gen_key_candidate_char_frequency(encrypted_bytes_by_key_position, key_candidates_by_position)


if __name__ == "__main__":
    sys.exit(main())
```

Figure 21 – Python script to brute-force possible keys and character frequencies