

Richi-challenge CTF writeup

This is *my* personal solution, the challenge could be solved in many other ways.

Per prima cosa dopo aver dato un'occhiata al sito web bisogna scaricare l'immagine: `curl -O https://dave997.github.io/rs3.jpg`.

Tramite il tool **binwalk** è possibile cercare eventuali bytes estranei presenti all'interno dell'immagine.

```
binwalk rs3.jpg

922716      0xE145C      Zip archive data, encrypted at least v2.0 to
extract, compressed size: 231, uncompressed size: 288, name: lets_go_deeper
923139      0xE1603      End of Zip archive
```

Vediamo che ha trovato un file zip chiamato `lets_go_deeper` nascosto all'interno dell'immagine. Per estrarlo usiamo il tool **foremost**, dando come query di ricerca *zip files*.

```
foremost -t zip -i rs3.jpg

Processing: rs3.jpg
|foundat=lets_go_deeperUT
*|
```

Analizzando il file con il comando `file lets_go_deeper`, possiamo vedere che ci sono diverse compressioni, in successione. Con questo script è possibile scompattare il file rimuovendo tutti gli strati di compressione:

```
#!/bin/sh

# 4. XZ compression
mv lets_go_deeper lets_go_deeper.xz
xz -d lets_go_deeper.xz

# 3. BUNZIP2 compression
mv lets_go_deeper lets_go_deeper.bz2
bzip2 -d lets_go_deeper.bz2

# 2. GUNZIP compression
mv lets_go_deeper lets_go_deeper.tar.gz
tar -xzf lets_go_deeper.tar.gz

# 1. ZIP compression
file="lets_go_deeper.zip"
name="lets_go_deeper"
i="1"
while [ $? -eq 0 ]; do
```

```
    echo "$1"
    i=$((i+1))

    mv $name $file
    unzip $file

done
```

A questo punto vediamo che lo script si blocca perchè il file è protetto da una password, quindi si può provare un bruteforce utilizzando un noto dizionario `rockyou.txt`:

```
# extract if needed
gunzip /usr/share/wordlists/rockyou.txt.gz

# crack password
fcrackzip -u -D -p /usr/share/wordlists/rockyou.txt lets_go_deeper.zip

PASSWORD FOUND!!!!: pw == kari

# unzip it
unzip lets_go_deeper.zip

Archive:  lets_go_deeper.zip
[lets_go_deeper.zip] key password:
  inflating: key
```

Il file estratto si chiama **key**, ed è un file binario. Provando ad eseguirlo vediamo che per ottenere la password bisogna conoscere la parola magica, quindi per scoprirla andiamo ad eseguire il reverse engineering del binario:

```
$> gdb key
(gdb)
# set a better readable format for perations
(gdb) set disassembly-flavor intel
# this command return a list of all the functions present inside the binary, there
is also a function called "print_key", but its a rickroll that I've voluntarily
created
(gdb) info functions
# so let's disassemble the main to understand how it works:
(gdb) disas main
```

```

File Edit View Search Terminal Help
0x0000000000001170 frame_dummy
0x0000000000001179 print_key
0x0000000000001191 main
0x0000000000001250 _libc_csu_init
0x00000000000012c0 _libc_csu_fini
0x00000000000012c8 _fini
(gdb) disas main
Dump of assembler code for function main:
0x0000000000001191 <+0>:    push    rbp
0x0000000000001192 <+1>:    mov     rbp, rsp
0x0000000000001195 <+4>:    add     rsp, 0xffffffffffff80
0x0000000000001199 <+8>:    mov     DWORD PTR [rbp-0x74], edi
0x000000000000119c <+11>:   mov     QWORD PTR [rbp-0x80], rsi
0x00000000000011a0 <+15>:   mov     rax, QWORD PTR fs:0x28
0x00000000000011a9 <+24>:   mov     QWORD PTR [rbp-0x8], rax
0x00000000000011ad <+28>:   xor     eax, eax
0x00000000000011af <+30>:   lea     rdi, [rip+0xe99]          # 0x204f
0x00000000000011b6 <+37>:   call    0x1030 <puts@plt>
0x00000000000011bb <+42>:   lea     rdi, [rip+0xe96]          # 0x2058
0x00000000000011c2 <+49>:   call    0x1030 <puts@plt>
0x00000000000011c7 <+54>:   lea     rdi, [rip+0xeb0]          # 0x207e
0x00000000000011ce <+61>:   call    0x1030 <puts@plt>
0x00000000000011d3 <+66>:   lea     rdi, [rip+0xec0]          # 0x209a
0x00000000000011da <+73>:   mov     eax, 0x0
0x00000000000011df <+78>:   call    0x1050 <printf@plt>
0x00000000000011e4 <+83>:   lea     rax, [rbp-0x70]
0x00000000000011e8 <+87>:   mov     rdi, rax
0x00000000000011eb <+90>:   mov     eax, 0x0
0x00000000000011f0 <+95>:   call    0x1070 <gets@plt>
0x00000000000011f5 <+100>:  lea     rax, [rbp-0x70]
0x00000000000011f9 <+104>:  lea     rsi, [rip+0xeb5]          # 0x20b5
0x0000000000001200 <+111>:  mov     rdi, rax
0x0000000000001203 <+114>:  call    0x1060 <strcmp@plt>
0x0000000000001208 <+119>:  test    eax, eax
0x000000000000120a <+121>:  jne     0x121a <main+137>
0x000000000000120c <+123>:  lea     rdi, [rip+0xe9d]          # 0x20c0
0x0000000000001213 <+130>:  call    0x1030 <puts@plt>
0x0000000000001218 <+135>:  jmp     0x1226 <main+149>
0x000000000000121a <+137>:  lea     rdi, [rip+0xebf]          # 0x20e0
0x0000000000001221 <+144>:  call    0x1030 <puts@plt>
0x0000000000001226 <+149>:  mov     eax, 0x0
0x000000000000122b <+154>:  mov     rdx, QWORD PTR [rbp-0x8]
0x000000000000122f <+158>:  xor     rdx, QWORD PTR fs:0x28
0x0000000000001238 <+167>:  je      0x123f <main+174>
0x000000000000123a <+169>:  call    0x1040 <__stack_chk_fail@plt>
0x000000000000123f <+174>:  leave
0x0000000000001240 <+175>:  ret
End of assembler dump.
(gdb)

```

Print initial text

Read input

Check the input

Print the key!

A questo punto per poter bypassare il controllo sulla parola magica ho creato un breakpoint esattamente sul comando che setta il flag risultante dalla comparazione, che verrà poi letto dal comando *jump* che andrà ad eseguire il pezzo di codice corretto.

Quindi ho impostato manualmente il flag a 0, il quale significa che l'input e la parola magica corrispondono.

N.B. Un'altra possibile soluzione poteva essere impostare l'Instruction Pointer direttamente sull'indirizzo del print della chiave

```

# 0x00005555555555208 <+119>: test    eax, eax
(gdb) break *0x00005555555555208
# input inserted!
(gdb) run
(gdb) set $eax=0
(gdb) s
Single stepping until exit from function main,

```

```
which has no line number information.  
Well done!  
The code is: 190316  
(gdb)
```

Well Done! Il codice è 190316

Complimenti Dottore!