littlerev

The program must be launched with a second argument, that is the password to be guessed.

```
Usage: ./littlerev PASSWORD
```

Build

Compiled with gcc 7.5.0 (Ubuntu 18.04)

```
gcc -s -z execstack -no-pie -o littlerev littlerev.c
```

Writeup

Since it appears to be a reverse engineering challenge, let's straight up disassemble the executable with Ghidra.

The first thing that can be noticed in *main()* is a long **if** condition where the elements of argv[1] (that is, the characters of the password we have to provide) are compared to a mathematical expression. As an example:

```
// condition on the last character of the password
argv[1][12] == (uint)(byte)(DAT_006010d4 * DAT_006010c4 ^ 0xea)
```

Each expression takes one byte from each of the two global arrays `DAT_006010d4` and `DAT_006010c4`, multiplies them together and then performs a XOR with a hardcoded value. The result is casted as byte and is checked against the n-th character of the inserted password. In total there are 13 expressions like this, so presumably the password must be 13 characters long. We'll discuss how to crack this password in a moment.

Anti-debugging

Before the big if condition we see that the function `ptrace` is called, followed by a check on its return value: if it returns **-1** the program terminates. This is a common *anti-debugging technique* because debuggers on Unix-like platforms use the `ptrace` syscall to control the process to be debugged and, by design, a process can be "ptraced" only by a single process. In this case, if we ran the program through gdb, the `ptrace` call at the beginning of *main* would return the error value -1 and the execution would terminate, leaving us with little dynamic analysis options.

Fortunately, this obstacle can be easily overcome by patching the binary with **NOP** instructions in place of the problematic block. In this case we substitute all the bytes from 0x629 to 0x66d included *(offsets from ELF base)* with the value `0x90`.

Password cracking

Now that we have a debuggable program, let's crack the password.

I see tree main paths:

1. **Manual calculations**
   Naive approach: take all the values that are used in the expressions and calculate the result. It is surely doable, the characters are not too many but it is a pain and not elegant.
2. **GDB assisted**
   - *Manual*
     Run gdb and step manually until the correct value is stored in a register, then print it as a character. Better than doing calculations manually but can still be tedious.
   - *Automated with .gdbinit*
     gdb commands can be automated with a .gdbinit file. As an example, here are the commands for the first character of the password:

     ```
     b *0x40069c

     commands 2
       silent
       printf "PASSWORD: %c", $rdx
       set $rax=$rdx
       c
     end
     ```

     Notice that we also need to set **RAX** with the correct value in order to pass the check and continue the execution to the next one.
3. **Symbolic execution**

   Use *angr* to simulate the execution with symbolic input. When the simulation reaches the addresses at the end of *main()*, evaluate the symbolic input.

In the end, we obtain the password `g1mm3fl4g_plz`.

Is this a packer?

Running the program with the correct password we finally get inside the if statement. Here two functions are called, both of which are given argv[1] (*i.e.* the password we inserted) as an argument:

- the first one is modifying a global array by XORing its elements with the password
- the second one is actually calling the global array as a function

This means that the global array contains some packed code that gets unpacked only if we provide the correct password. Ok, so let's get to the point where the code has been unpacked and print the instructions with gdb.

Inspecting the assembly, we see another call to ptrace. Wow, this program really doesn't want to be debugged. It performs two comparisons and, depending on the results:

- prints "Get out!" and jumps straight to the end of the function: CHECKS KO
- does something else: CHECKS OK

We want to see what happens when the checks are passed, so we step with gdb and when we reach a comparison we modify the registers so that the execution does not jump to the end of the function. Actually, the only change needed is at this instructon:

```
0x601120: cmp    eax,0xffffffff
```

If eax is different from -1, the function doesn't jump to the end. Let's set it to 0 then and continue stepping with gdb.

A new password

The program now loads the loads the 14th character of the password and compares it to the value 0x21 (ASCII for `!`). If the character is different, it jumps to the end again.

```
// in al there is the 14th character of the inserted password
0x60113e: cmp    al,0x21
```

Ok, so the complete password is 14 characters long with `!` as last character.

After setting **AL** = 0x21 the program prints the characters of the real flag one by one.

Comments

- It's interesting to see that if the program is run with the full password `g1mm3fl4_plz!` as input, it doesn't print the flag. This is because of the checks at the beginning of the unpacked code where the second `ptrace` is used: they never allow for the flag to be printed. To get the flag, the registers must be set manually with gdb (or with a new .gdbinit).

  In the exploit folder you can find `.gdbinit2_flag.txt` that shows a .gdbinit script that makes the original binary print the flag without applying any patches to it.
- If you pay attention to the bytes of the packed code in the binary, you can actually see the password characters (!):

```
006010e0 32 79 e4       XOR         BH,byte ptr [RCX + -0x1c]
006010e3 88 7b e5       MOV         byte ptr [RBX + -0x1b],BH
006010e6 80 14 2f d6    ADC         byte ptr [RDI + RBP*0x1],0x
006010ea 0d 84 f1       OR          EAX,0x8762f184
         62 87
006010ef 6d             INSD        RDI,DX
006010f0 6d             INSD        RDI,DX
006010f1 33 e3          XOR         ESP,EBX
006010f3 ac             LODSB       RSI
006010f4 41             ??          41h    A
006010f5 43             ??          43h    C
006010f6 e6             ??          E6h
006010f7 70             ??          70h    p
006010f8 6c             ??          6Ch    l
006010f9 7a             ??          7Ah    z
006010fa 67             ??          67h    g
006010fb 8b             ??          8Bh
006010fc 6c             ??          6Ch    l
006010fd 6d             ??          6Dh    m
006010fe 33             ??          33h    3
006010ff 66             ??          66h    f
00601100 d2             ??          D2h
00601101 34             ??          34h    4
00601102 67             ??          67h    g
00601103 5f             ??          5Fh    _
00601104 70             ??          70h    p
00601105 d3             ??          D3h
00601106 7a             ??          7Ah    z
00601107 67             ??          67h    g
00601108 31             ??          31h    1
00601109 6d             ??          6Dh    m
0060110a d5             ??          D5h
0060110b 33             ??          33h    3
0060110c 66             ??          66h    f
0060110d 6c             ??          6Ch    l
0060110e 34             ??          34h    4
0060110f 8f             ??          8Fh
00601110 33             ??          33h    3
00601111 83             ??          83h
```

This happens because the code is obscured by means of just a XOR with the password: some assembly instructions in the code may have a number of null bytes in their operands, so XORing them with flag bytes will leave the flag bytes unchanged.
These are the specific instructions:

```
0x6010f6: mov    ecx,0x0          # b9 00 00 00 00
0x6010fb: mov    edx,0x1          # ba 01 00 00 00
0x601100: mov    esi,0x0          # be 00 00 00 00
0x601105: mov    edi,0x0          # bf 00 00 00 00
0x60110a: mov    eax,0x0          # b8 00 00 00 00
```