Report

חלק ב'-

זיהוי טעויות:

בקובץ $partb.opt.s$

זיהינו כי בשורה 45 קיימת הפקודה הבאה: $jge\ L2$,

הטעות היא שאם מתקיים התנאי לקפיצה על התוכנית לקפוץ ל – $L3$ ולא חזרה אל $L2$.

הסבר:

אם הקפיצה מתבצעת אל $L2$ התוכנית תחזור שוב על ההדפסה $LC0$ כלומר תבקש מספר מ-1 עד 100 , ולא תודיע למשתמש שהמספר שבחר הוא קטן מדיי או שניחש נכון את המספר.

לכן אם נשנה את הפקודה להיות: $jge\ L3$ התוכנית תמשיך בצורה הגיונית - תבצע את הפקודה $jle\ L5$ ומשם תנתב לflow נכון של התוכנית.

בנוסף, בשורה 39 מועבר כפרמטר לפונקצית printf מחרוזת LC0 במקום מחרוזת LC1.

בקובץ $partb.nopt.s$

בשורה מס' 58 מופיעה הפקודה $jmp\ L3$ במקום שיופיע $jmp\ L6$ - הריי אם הגענו לבצע את הפקודה בשורה מס' 58 אז המשתמש ניחש באופן שגוי את המספר שבחרה התוכנית ,

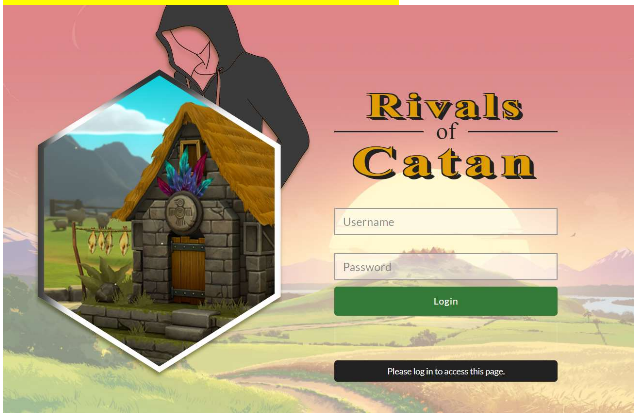
המשתמש בחר מספר גדול מדי.

ולכן על מנת שהתוכנית תמשיך לרוץ כמו שצריך יש לתקן ע"י החזרת המשתמש לשלב בו הוא בוחר מספר אחר כלומר ל - $L6$.

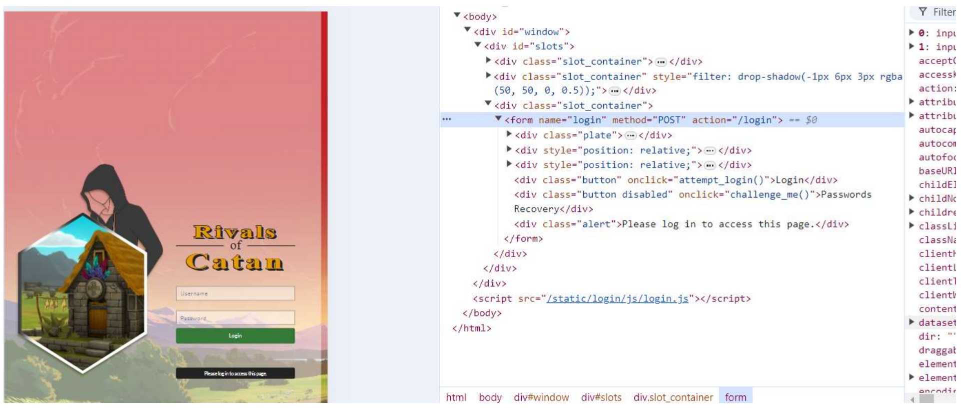
הדבר השגוי שהתוכנית מבצעת : היא בעצם בעת בחירת מספר שגוי במקרה הזה גדול מדיי התוכנית תקפוץ ל- $L3$ והתוכנית תסתיים.

בשורה מס׳ 56 מתבצעת קריאה ל- $scanf$ ללא העברת פרמטרים.

Part 2 - Rivals of Catan Site

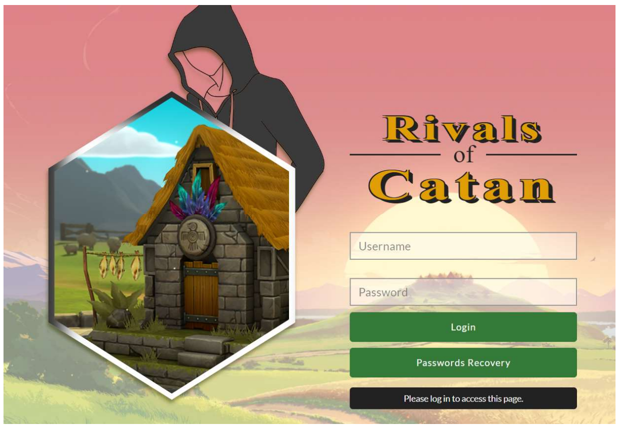


At first we inspected the site (F12) and we managed to see that 1 of the buttons are disabled, all we did is to change it to : button.

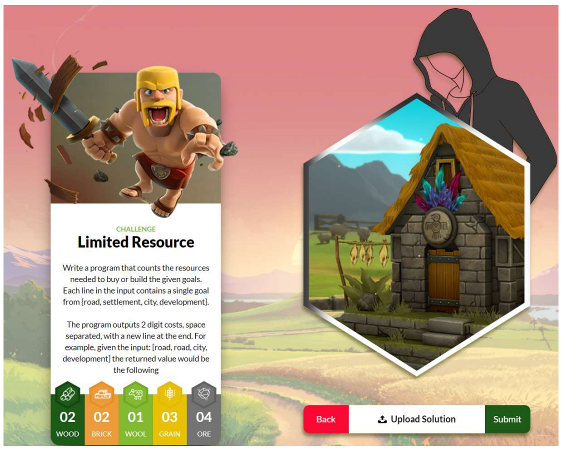


```
▼<body>
  ▼<div id="window">
    ▼<div id="slots">
      ▶<div class="slot_container">⋯</div>
      ▶<div class="slot_container" style="filter: drop-shadow(-1px 6px 3px rgba
       (50, 50, 0, 0.5));">⋯</div>
      ▼<div class="slot_container">
...     ▼<form name="login" method="POST" action="/login"> == $0
          ▶<div class="plate">⋯</div>
          ▶<div style="position: relative;">⋯</div>
          ▶<div style="position: relative;">⋯</div>
           <div class="button" onclick="attempt_login()">Login</div>
           <div class="button disabled" onclick="challenge_me()">Passwords
           Recovery</div>
           <div class="alert">Please log in to access this page.</div>
         </form>
       </div>
     </div>
   </div>
   <script src="/static/login/js/login.js"></script>
 </body>
</html>
```

A new button revealed to us :



we click on it - passing us to the upload page of our challenge.exe .


CHALLENGE

Limited Resource

Write a program that counts the resources needed to buy or build the given goals. Each line in the input contains a single goal from {road, settlement, city, development}.

The program outputs 2 digit costs, space separated, with a new line at the end. For example, given the input: [road, road, city, development] the returned value would be the following

02	02	01	03	04
WOOD	BRICK	WOOL	GRAIN	ORE

challenge.exe:

gcc print.S -o print_gcc.exe

But the exe file is too big -

We uploaded but the size of the .exe was too big - we most upload .exe file that weights less then 2048B(2K).

So as we learned in the tutorials we took out all the metadata which includes the libraries that were loaded. So basically we skinned code from metadata using "objcopy" command .

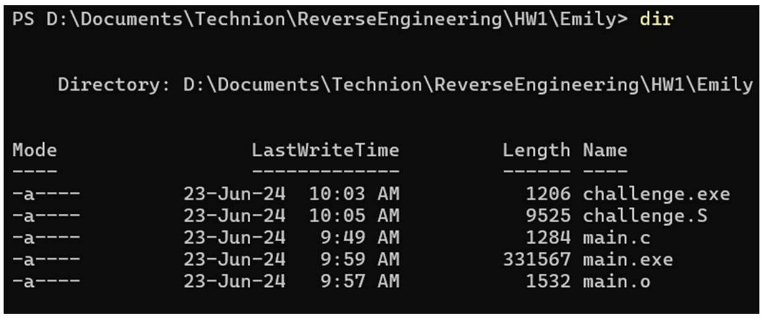
We used the find_function which used the load_library and get_proc_address in order to get specific function and not the all dll.

now we had to prepare a PE header so we used the "dir" command to know the size of our code and then used CFF Explorer to adjust our entry point to the right location by

changing the raw and virtual to the size of the file.

then we used the "type" command to append the PE header we created and the binary of:

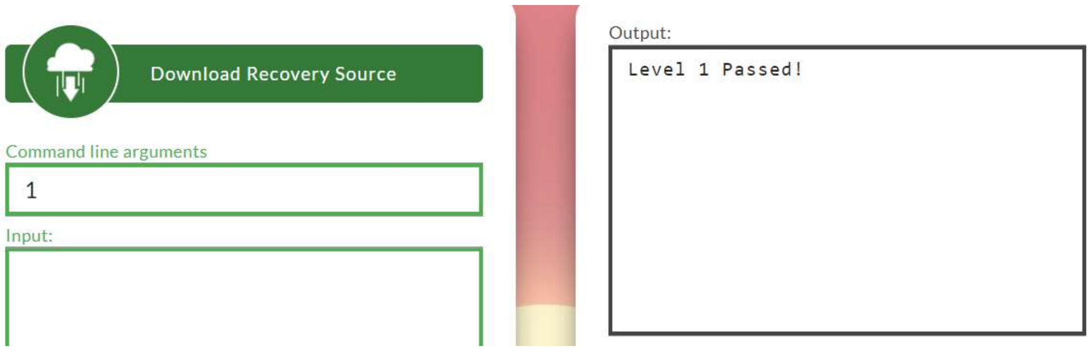
type PE.bin challenge.bin find_function.bin > challenge.bin and created a challenge.exe which weights: 1206B


```
PS D:\Documents\Technion\ReverseEngineering\HW1\Emily> dir


    Directory: D:\Documents\Technion\ReverseEngineering\HW1\Emily


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         23-Jun-24  10:03 AM           1206 challenge.exe
-a----         23-Jun-24  10:05 AM           9525 challenge.S
-a----         23-Jun-24   9:49 AM           1284 main.c
-a----         23-Jun-24   9:59 AM         331567 main.exe
-a----         23-Jun-24   9:57 AM           1532 main.o
```

Then we uploaded it and moved to the CRACKME part.



```
Level 1 Passed!
```

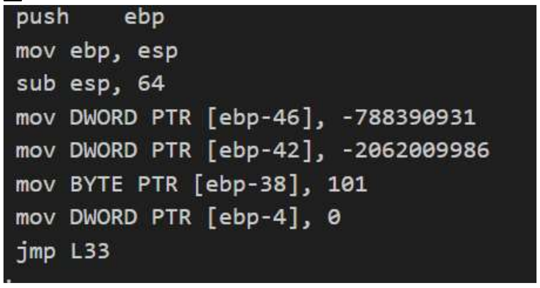
LVL1:

After examining the code, we understood we need to provide at least 1 argument to pass Level1.
We choose the argument "1".
And We passed Level1.

LVL2:

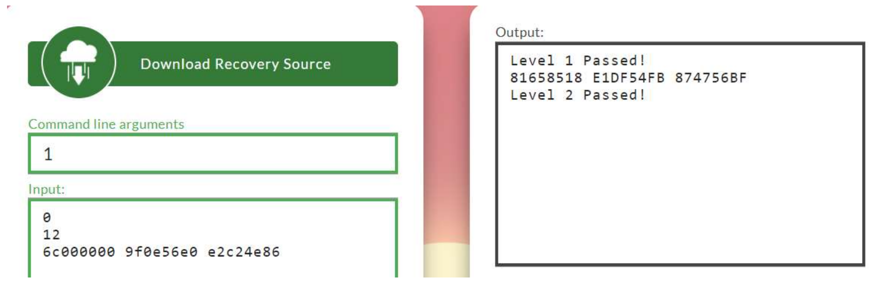
what the function does:
It generates 8 random numbers (32bits) and after a few math manipulations, it takes only the 8 lower bits, stores them in each cell of the array which is in [ebp-37] address.
Then the function asks for input of 2 numbers.
We tried to enter different numbers and we understood that the numbers will be used in printArray function as boundaries from which cell to print and when to stop.
So the first num we chose is "0".
The second number we chose is "12" and this is because the code performs alignment (modulo 4), and for us to see the whole numbers in the array (we need to see 9 numbers) we must have a number which modulo 4 of it we be minimum 3.

Each print prints 8 HEX numbers (4bytes) and we want to see at least the first 9 bytes.
After the print we need to enter another input which takes each byte in the input and XOR it with the random number in each cell at the array and then insert the answer of XOR to the same cell in the array.
 After 9 iterations the code continues to compare each cell in the array with the 9 bytes of HEX numbers at the stack which provided at the start of _level2 function:



```
push    ebp
mov ebp, esp
sub esp, 64
mov DWORD PTR [ebp-46], -788390931
mov DWORD PTR [ebp-42], -2062009986
mov BYTE PTR [ebp-38], 101
mov DWORD PTR [ebp-4], 0
jmp L33
```

We understood that our input should be a hex number which XOR with it will be the same hex number at the stack - byte after byte.



Command line arguments

```
1
```

Input:

```
0
12
6c000000 9f0e56e0 e2c24e86
```

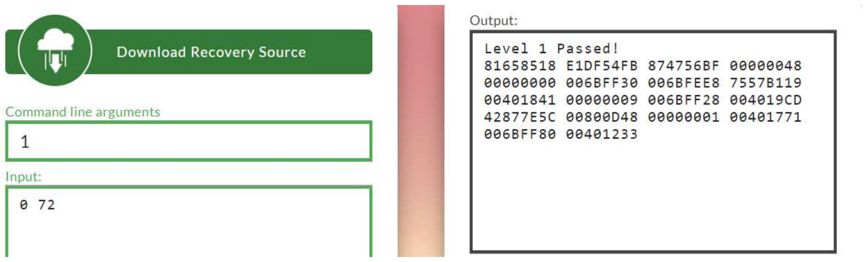
Output:

```
Level 1 Passed!
81658518 E1DF54FB 874756BF
Level 2 Passed!
```

LVL3:
Looking for "Level 3 Passed!" we found it in _dummy function.
We can see after lvl2 is complete the programs ends.
We thought to our that somehow we need to make _dummy function to be called.
One option is to override the return address of _level2 function with the address of _dummy function.
So we might want to change the second number on input in lvl2 function to understand from which point we see the return address at the stack and then override it with _dummy address.
Therefore, we calculated the offset between the start of the array until the ret address of the main function(first we tried until the ret address of _level2 but there was more code in the main which we lost by that - and

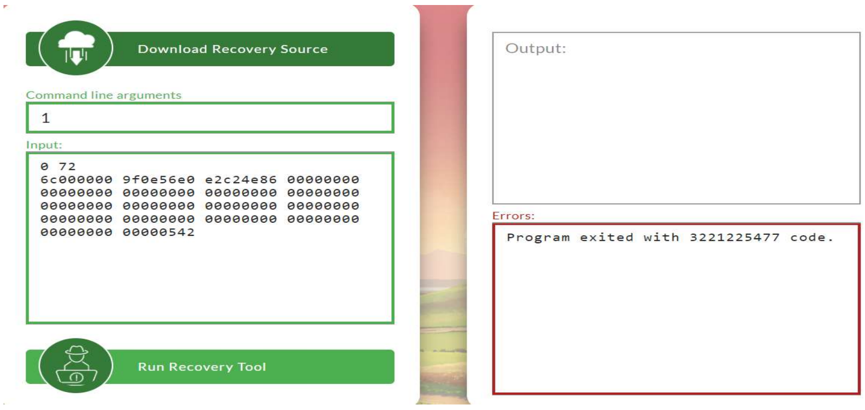
therefore we decided to override/overflow the main ret address).
We chose num1=0 and num2=72 because of the offset between array and ret address.

and we got that:



Command line arguments
```
1
```
Input:
```
0 72
```
Output:
```
Level 1 Passed!
81658518 E1DF54FB 874756BF 00000048
00000000 006BFF30 006BFEE8 7557B119
00401841 00000009 006BFF28 004019CD
42877E5C 00800D48 00000001 00401771
006BFF80 00401233
```

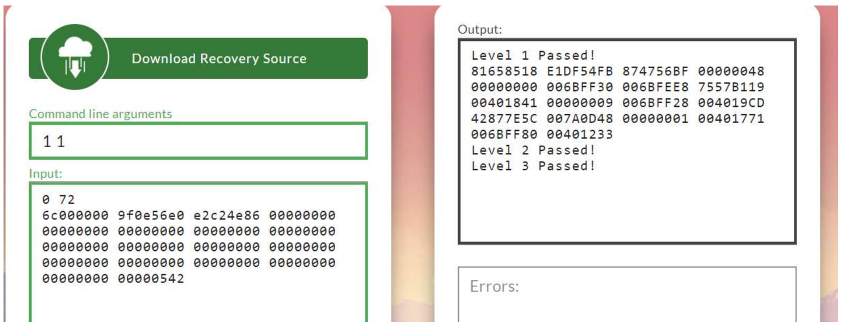
Because of the calculation we can decide where _dummy address is: 0x401771
and we need to discard the main ret address which is: 0x401233
So we did the next calculation of XOR:
(17)xor(12) = 05
(71)xor(33) = 42
And we entered a new input:



Command line arguments
```
1
```
Input:
```
0 72
6c000000 9f0e56e0 e2c24e86 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000542
```
Output:

Errors:
```
Program exited with 3221225477 code.
```

Everything cloupsed , but after few hours we understood we need 1 more input (the code that executed after _level2 function in the main.
Then we add 1 more argument "1":



Command line arguments

```
1 1
```

Input:

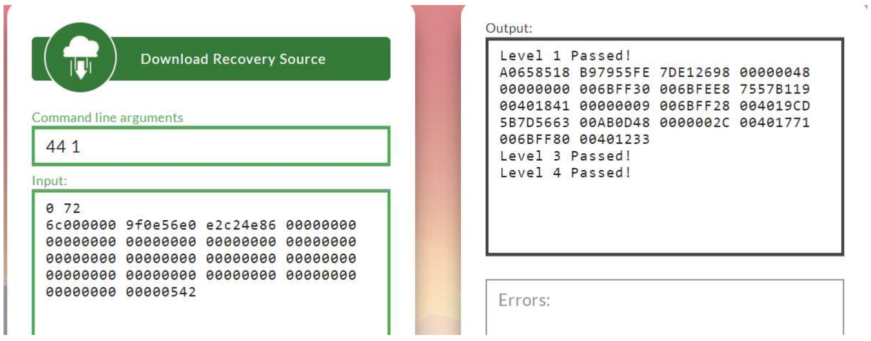
```
0 72
6c000000 9f0e56e0 e2c24e86 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000542
```

Output:

```
Level 1 Passed!
81658518 E1DF54FB 874756BF 00000048
00000000 006BFF30 006BFEE8 7557B119
00401841 00000009 006BFF28 004019CD
42877E5C 007A0D48 00000001 00401771
006BFF80 00401233
Level 2 Passed!
Level 3 Passed!
```

Errors:

LVL4 :
After looking for "Level 4 Passed" , we found it in the handler function - which means we must divide a number by 0.
We found a "div" command in the _dummy_ function.
The value which the global variable "_divider" holds is the first argument we give to the program and it going throw a few math manipulation.
It changed everything - we tried to do a quick scan for a number between 0-100 by creating a small program with do the exec commands - it didn't work so we started to do manually - 44 is a number that can help us.
Now our code will divide by 0 and then a SIG fault #8 will rise and a handler function will run. It will print us the string :" Level 4 Passed!"
But our program will end without giving us the list of users/passwaords.
it will output :



Command line arguments

```
44 1
```

Input:

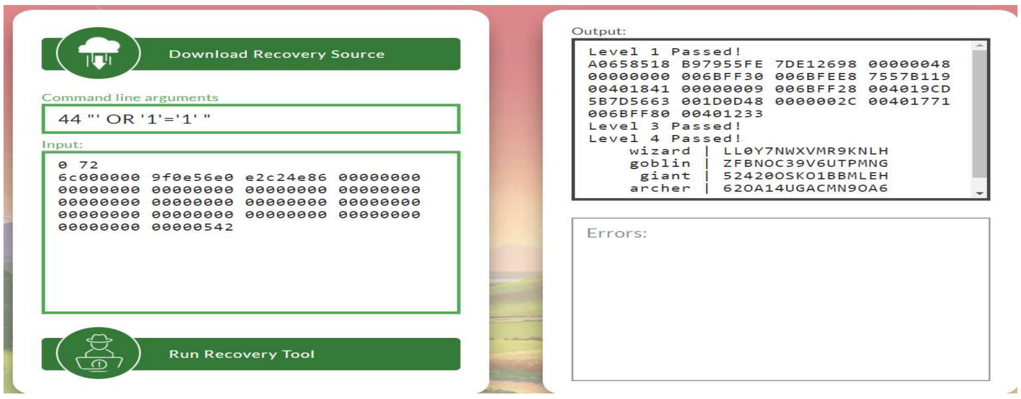
```
0 72
6c000000 9f0e56e0 e2c24e86 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000542
```

Output:

```
Level 1 Passed!
A0658518 B97955FE 7DE12698 00000048
00000000 006BFF30 006BFEE8 7557B119
00401841 00000009 006BFF28 004019CD
5B7D5663 00AB0D48 0000002C 00401771
006BFF80 00401233
Level 3 Passed!
Level 4 Passed!
```

Errors:

Note: when we changed the first argument it changed all our random numbers and we had to insert a new input.
Looking at the _db_access we can see there is a SQL injection that can be performed because "arg_0 " is appending to the string we send to the database.
So when you are performing SQL injection, we can use the basic method of appending the "' OR '1'='1' " string and the idea it is that the expression after the OR is always true and that is why we managed to pass the table from the DB.



```
44 "' OR '1'='1' "
```

```
0 72
6c000000 9f0e56e0 e2c24e86 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000542
```

```
Level 1 Passed!
A0658518 B97955FE 7DE12698 00000048
00000000 006BFF30 006BFEE8 7557B119
00401841 00000009 006BFF28 004019CD
5B7D5663 001D0D48 0000002C 00401771
006BFF80 00401233
Level 3 Passed!
Level 4 Passed!
    wizard | LL0Y7NWXVMR9KNLH
    goblin | ZFBNOC39V6UTPMNG
     giant | 52420OSKO1BBMLEH
    archer | 62OA14UGACMN9OA6
```