Yosef Goren & Tomer Bitan

April 30, 2023

Note to HW checker:
We found a backdoor so we could control the server and use it to solve port_c.
When asked, the tutor said we are still required to solve the rest of the homework normally, but we will get a bonus for finding the backdoor.
So our solution contains both the backdoor and the expected solution.

Also, you can see more of our solution in our github repo.

Contents

1 login page **1**

2 challange **2**
2.1 Solving the challenge (start) 2
2.2 Attempt at remote code execution (Unsuccessful) 2
2.3 Solving the challenge (cont) 3

3 recovery **3**
3.1 level_1 3

4 backdoor **4**

5 recovery (cont) **5**
5.1 level_2 5
5.2 level 3 6
5.3 level 4 7

1 login page

1. we see login page
2. attempt simple passwords to autenticate
3. to view the content of the page - download the html page for better accessability
4. trying to understand what does the 'login' button does when clicked.
   (a) learn about html elements, forms and how they interact with the javascript.
   (b) apperantly the button submits a form to the server.
   (c) stumble across an additional clickable button element which is disabled

2 challange

2.1 Solving the challenge (start)

1. the additional button we have found can be enabled by editing the html file
2. after enabling it, and clicking it in the browser - we see that a challange appears
3. we try to understand the required format of the challange. since the filename ends with '.exe' we assume two things:
   - we should submit an executable (rather than a python/javascript/C source file)
   - that executable should be compiled to a windows platform (linux typically does not have .exe)
4. we try to solve the challange
   (a) we first implement the challange in C++, but the executable is way too big, it needs to be smaller than 2KB
   (b) we now try implementing with C, still too big.
      - even a C executable that only has "`main(){return 0;}`" is 40KB in size!
      - this is likely due to libc code that runs before and after the main procedure, we cannot get rid of it and have a small enough executable using gcc.
      - we find a new compiler called tcc 'tiny C compiler'.
      - we compile our program with tcc and get an executable with 2KB size.
5. we submit our solution and we get an error that the challange failed - (so this is a different error than the one saying it's too big)
6. we are not sure if the server actually runs our code, so we experiment by giving it a text file.

2.2 Attempt at remote code execution (Unsuccessful)

1.
   - the text file is rejected since it is not executable.
   - we change the name of the text file to 'challange.exe'.
   - submitting this causes the server to yield 'HTTP error code 500' - meaning that an 'unexpected error' has occured.
   - this causes us to belive that the server indeed attempts to run our executable but fails due to the fact that it is actually a text file.
2. in an attempt to think of a new direction we might be able to hack to server using a bad usage of the HTTP protocol.
   - we learn about how HTTP works.
   - apparently web-pages are provided by using the 'HTTP GET request'.
   - we can attempt to run an 'HTTP GET request' on different pages.
     - we attempt to run 'HTTP GET' on paths like '`<server-ip>/success`' or '`<server-ip>/challange-success`', but all of these yield 'HTTP error 404' meaning that no such page exists.
     - interestingly, if we try '¡server-ip¿/admin' we don't get 'error 404', rather we get redirected to the login page, which indicated that maybe the 'HTTPT GET /admin' request is valid but requires authentication.
3. now that we knew our executable was being run, we could - in principle - run whatever we want on that executable - for example, open an interactive shell that will recive commands from us and run the on the rivals of catan server. In practice, we implementing this was very challanging and we didn't finish doing so.

2.3 Solving the challenge (cont)

1. after a few days of trying to find a backdoor to the website, we noticed that our code yields the wrong results if an endline is missing from the last line of the input file - this was because we ran 'strcmp' with strings like 'development
n', so we did not count the last line if it was missing a '
n' character.

2. after fixing the bug, we got access to the next stage.

3 recovery

1. the next stage landed us at a url with the '`/recovery`' suffix. considering what we have learned about HTTP GET, we attempted to simply write '`<rivals-of-catan-ip>/recovery`' into the browser url and this way we completely bypass the need for the catan executable (unfortunemtly we only found this out AFTER we have made it).

2. in the new page we can download an assembly source file and the layout suggests that by filling a form and clicking a button we can run the compiled source code on input of our choice.

3. ofcourse we attempt analyze the assembly file on our local machine

   - doing so first we attempt to compile it locally
   - this yields a linker error compialing that some sqlite3 library symbols are missing
   - without readin any assembly we can tell that the code interacts with some databasem likely containing user authentication info
   - unfortunetly this also means that running the code locally will not be very helpful since we do not have the appropriate database

4. analyzing the assembly itself brings a few additional conclusions

   - in the predefined strings we see one for "`----------`" and another for " `|` " this likely means the result might generate some type of table.
   - we look at _main: we see that the first few commands interpret the first argument as an integer, and appear to save that integer as a local variable

3.1 level_1

1.
   - it appears the level_1 procedure recives 'argc, and argv' - as-is from the main procedure, they are the only arguments.
   - the procedure roughly executes '`if(argc <= 1) exit(0); printf("level_1 passed");`'.
   - so to pass this test we simply need to give at-least one additional argument.

2. starting level_2:

   - this procedure recives one argument being '`atoi(argv[1])`'.
   - it also appears this procedure is one of the most incomprehensive peices of code we had the misfortune to encounter.

4 backdoor

1. We notice that this 'crackme.S' asm file can also be downloaded directly by running `HTTP GET /crackme.S`. perhaps we can edit 'crackme.S' by running `system("echo ̈<asm line>̈> crackme.S")` within our `challenge.exe` solution.

2. To test this theory we build an executable that edits a file in it's directory by the name of 'crackme.S' by inserting our own line to it.

3. Our theory proves to be correct, which was verifyied by uploading our new 'challenge.exe', and then downloading 'crackme.S' again where we saw that indeed the content of 'crackme.S' changed.

4. At this point, we want to make sure that if we make changes, it's not to the whole website, so that our solution does not ruin it for the other students. We verify this is not the case by asking a friend to download the 'crackme.S' file, and we see that his is the original, unmodified version.

5. The next step was to try to use this to pass the recovery stage

   - our new 'challenge.exe' editor now inserts the two commands: `mov DWORD PTR [esp], OFFSET FLAT:LC1` and `call _puts` right after `_main`.
   - this should make the recovery executable print ADMIN before it does anything else.
   - the only problem is that it is not recompiled, so we cannot see if our change has worked.

6. To get back some more information, we change our challenge such that it will create a new text file in it's local folder, then we try to fetch it with HTTP GET ¡new_file_name.txt¿. or an unknown reason this fails: when we try to download the new file we get HTTP 404.

7. Our next way to attempt to communicate back to ourselves is to append comments to the 'crackme.S' file. this method did work; we can see the comment we added at the end of 'crackme.S'

8. Since we are always working with 'crackme.S', we want to save it's version so we can always go back to it, so we make a new file on the server named 'crackme_save.S' and verify we can indeed recover the original file from it.

9. the next step is to snoop around and see what files exist, by running "dir ¿¿ crackme.S", now we can see what files exist:

```
        04/13/2023  05:28 PM    <DIR>          .
04/13/2023  05:28 PM    <DIR>          ..
04/13/2023  05:28 PM             2,048 challenge.exe
04/02/2023  08:07 AM             1,217 challenge.py
04/02/2023  08:07 AM             3,729 common.py
04/02/2023  08:07 AM           764,928 crackme.exe
04/02/2023  08:07 AM               395 crackme.out    <----- probobly the expected output
04/13/2023  04:39 PM            12,068 crackme.S
04/13/2023  04:39 PM            12,068 crackme\_save.S <----- our backup
04/13/2023  04:32 PM                 9 hello778.txt   <----- this one we created before for testing
04/02/2023  08:07 AM            32,768 users.db       <----- database prob contains solution to homework
04/11/2023  08:44 PM    <DIR>          \_\_pycache\_\_
9 File(s)        829,230 bytes
3 Dir(s)  86,185,775,104 bytes free
```

- the next step is to run: 'type crackme.out ¿ crackme.S', and we see:

```
Level 1 Passed!
04BE9463 5FAC5A1F 04AF1079 00000050
00000000 006BFF38 006BFEE8 778DB119
00401841 00000009 006BFF28 004019CD
B904C0C7 00801500 00000000 00401771
```

```
006BFF80 00401233 00000003 00801500
Level 2 Passed!
Level 3 Passed!
Level 4 Passed!
    wizard | ME7WS9H9UXV9DND4
    goblin | 34U97VEYPNODNGZS
giant | QVN4ZXKH38PGDGS2
    archer | FKGXJP0OCE1LKT3D
```

- Now we try to log in using the credentials we have found, and authentication is successful.
- Similarly, we can also get the content of the database file and open it locally: `"copy users.db crackme.S"`, download crackme.S once more then rename it back to 'users.db'. Now open it locally with sqlite and we can see the content of the database.

5 recovery (cont)

5.1 level_2

- From reading through the code we understand that this code looks at a bunch of seemingly random numbers that are XORed with the input of numbers writen in Hex returned from a scanf. Then comparred to some predetermand byte sequence.
- The numbers that our input is XORed with seem to be determend by rand32 with a seed of out first command line argument `atoi(argv[1])`.
- To identify the numers we need to input we use IDA stack view to debug a compiled version of the code with the command line argument.
  1. we can see in the memory that the numbers we are XORed with appear on the stack and stay consistant with the same seed.
  2. note from later: we started with 1 as the command line argument and later after identifying that 5 works better for level 4 redid this part with that seed.
- Since we see that the result of the XOR is printed to the screen we can input 0s in the scanf and the resulting print will be the numbers that our iput is XORed with. we can use the fact that $a \otimes b = c \Leftrightarrow b \otimes c = a$ to calculate what our input needs to be (b) given that we know a (given numbres) and c (the random numbers)

```
Memory Areas: (for seed 1)
0x4A ^ 0x81 = 0xcb
0x0E ^ 0xFB = 0xf5
0xB4 ^ 0x54 = 0xe0
0x1B ^ 0xDF = 0xc4
0xB6 ^ 0xE1 = 0x57
0xA8 ^ 0xBF = 0x17
0x63 ^ 0x56 = 0x35
0x94 ^ 0x47 = 0xd3
0xBE ^ 0x87 = 0x39

New Memory Areas: (for seed 5)
0x4A ^ 0x78 = 0x32
0x0E ^ 0x6C = 0x62
0xB4 ^ 0x3C = 0x88
0x1B ^ 0xAC = 0xB7
0xB6 ^ 0xE9 = 0x5F
0xA8 ^ 0xD9 = 0x71
0x63 ^ 0x6F = 0x0C
```

```
0x94 ^ 0xA9 = 0x3D
0xBE ^ 0x92 = 0x2C

32000000
5FB78862
2C3D0C71

32000000 5FB78862 2C3D0C71
```

- We can see in the the code required the input to be aligned properly in order to lineup with the numbers for the XOR.

```
        Memory Alignment:
Note: Little Endian == MSB in highest address.

???8: ?? <---- start
???9: ??
???A: ??
???B: cb

???b f5  <---- second dword
???c e0
???d c4
???e 57

???f 17  <---- third dword
???0 35
???1 d3
???2 39

cb000000
57c4e0f5
39d33517

Final input:
0 12 cb000000 57c4e0f5 39d33517
```

5.2 level 3

- At this point we look for the string of 'success lvl 3' e.t.c, and we see there is no 'reasobable' way to get to it from the code. What we can do, is exploit our ability to modify the stack using the writing XOR's mechanism from level 2; so - we use the prints of lvl2 to what values we want to write, and then we use it to change the return address of _level2 such that it will return to _dummy instead of returning to main. This causes us to 'pass' level 3:

- Want to override return address, and get _dummy_ address:

```
        0 64
0x00: cb000000 57c4e0f5 39d33517 00000000
0x10: 00000000 00000000 00000000 00000000
0x20: 00000000 00000000 00000000 <\_lvl2\_r>
0x30: 00000000 00000000 00000000 <\_dummy>
```

- We first run locally so we can look at the memory in debug mode and understand where to look for the addresses: Locally:

```
wanted ret addr = 0x004017C1
```

```
existing addr = 0x00401A1D
should xor with = 0x00000DDC
```

Input to disvocer wanted and existing addresses:

```
0 64 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Input to pass:

```
0 48 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000DDC
```

- After we get it working locally, we try the same method on the server, only this time we already know at what offsets we can find the procedure offserts we are looking for. On server:

```
wanted ret addr = 0x00401771
existing addr = 0x004019CD
should xor with = 0x00000EBC
```

Input to disvocer wanted and existing addresses:

```
0 64 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Input to pass:

```
0 48 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000EBC
```

5.3 level 4

- Now we notice that level 3 seems to work properly on our computer, however when running on the server the code crashes and doesn't print the message.
  We think this is because we ruind the return address. and _dummy_ doesnt know where to return to.

- Want to override return address, and get _dummy_ address:

```
0 72
0x00: cb000000 57c4e0f5 39d33517 00000000
0x10: 00000000 00000000 00000000 00000000
0x20: 00000000 00000000 00000000 00000000
0x30: 00000000 00000000 00000000 <\_dummy>
0x40: <ebp-mc> <main-r>
```

- Locally (seed 1):

```
wanted ret addr = 0x004017C1
existing addr = 0x00401288
should xor with = 0x0000549
0 72 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000549
```

- On server (seed 1):

```
wanted ret addr = 0x00401771
existing addr = 0x00401233
should xor with = 0x00000542
0 72 CB000000 57C4E0F5 39D33517 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000542
```

- On server (seed 5):

```
0 72 32000000 5FB78862 2C3D0C71 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000542
```

- This as well as reading the code further trying to understand how to get from level 3 to level 4 we identify _divider and handler.
- We understand handler is set up to be a sig 8 (floating point exception) handler. - So we conclude we need to cause the 'divider' to have a 0 in it so that the div command to throw an hardware exception calling handler and passing level 4.
- after analyzing the code, we notice that after returning from level_2, 'main' will use the first argument (interpreted as a number) as a seed for the 'rand_32' function, and the take the result mod 4 as the new value of the 'divider'. Hence - assuming rand_32 is really random w.r to the seed - we only need to guess about 2 or 3 different seeds before we come across one that will cause 'main' to put 0 inside 'divider'.
- So far, we have used seed=1, but it did not cause us to pass level4, so we tried seed=0 next, but this caused us to pass levels 1, 4 and 3 - but not 2. This was because we needed to scanf numbers for the XOR of part 2 to work with the new seed (different seed different rand_32 for lvl2 comparisons).
- When running this code on the server we see that we are still getting the crash after due to somthing with the db.
- While reading through the db_access code we found that _arg takes another comand line argument and concatinates it to the LC7 string before its sent to the databes. so we attempted to concat a string that would serve as a legal sql command that requests all the users passwords.
  - The string we concatinated was "' or true; "to complete the query to be:" select username, password from users where username=" or true; "
  - This fits in the 11 character limit but although it prints out the requested username and passwords it also had an error with some unknown token '.
  - Turs out the unknown token is another ' char that is concatinated to the end of the query that we missed when reading through the code. to fix this we add /* to the end of the query to make anything after true; a comment including the other '
- Final Solution:
  - Comand line args:
    `5 "'or true;/*"`
  - Stdin:

```
0 72 32000000 5FB78862 2C3D0C71 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000542
```