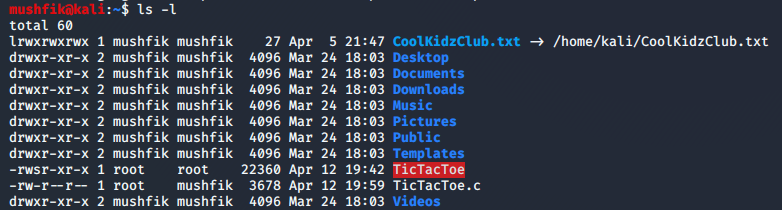
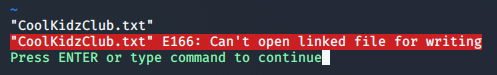
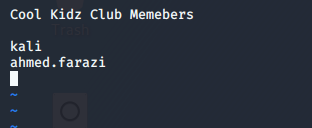
In this tutorial, we are tasked with winning a game of TicTacToe against an AI.

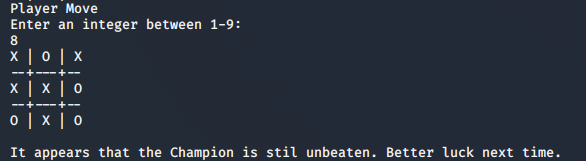
After logging into Kali and performing ‘ls –l’ on our home directory, we find a link to a file titled ‘CoolKidzClub.txt’ and a binary executable titled ‘TicTacToe’. We can also find the source code of the TicTacToe binary.



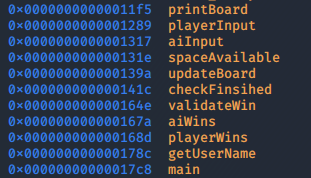
Opening the text file in Vim shows us some names of other users. However, trying to edit the file gives us an error. It appears that we do not have write permission to the actual text file this link is pointing us to.



After launching the ‘TicTacToe’ binary, we are greeted with a TicTacToe board on the console and a challenge to beat the ‘unbeatable AI’ to join the ‘CoolKidzClub’. Recalling from the ‘ls -l’ output, the binary is owned by root and it has SUID permission enbaled. This implies that we can run this binary as root and write our name to the CoolKidzClub.txt file by beating the AI in a game of TicTacToe.



In the TicTacToe game, we can fill a space in the board by typing in any number between 1-9. It gives us an error for any invalid move (including trying to fill an already filled up space) and waits until we finally provide a valid move. After a few games, it becomes evident that the AI plays optimally every time and the game only results in a draw or a loss for us should we make any mistake.

We open the binary in GDB and look for all the function names using ‘info functions’. Aside from the library functions, we find symbols which are possibly defined by the developer. Among these, ‘playerWins’ and ‘playerInput’ stands out. Using ‘disassemble playerWins’ and ‘disassemble playerInput’, we peek into the assembly code for both of these function.

After opeing the source code ‘TicTacToe.c’, we look for the two functions.

void playerWins(){

    printf("\nWow, you actually won. Adding you to the CoolKidzClub...\n");

    FILE \*fp;

    char line[64];

    int alreadyIn = 0, len;

    fp = fopen("/home/kali/CoolKidzClub.txt", "a+");

    fgets(line, sizeof(line), fp);

    while (fgets(line, sizeof(line), fp)){

        len = strlen(line);

        if(line[len-1] == '\n') line[len-1] = 0;

        if(strcmp(username, line) == 0) {

            printf("You are already in the CoolKidzClub.\n");

            alreadyIn = 1;

            break;

        }

    }

    if(!alreadyIn) fprintf(fp, "%s", name);

    printf("Done.\n");

    fclose(fp);

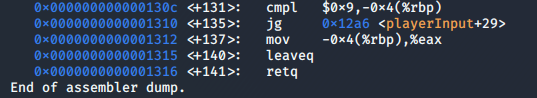
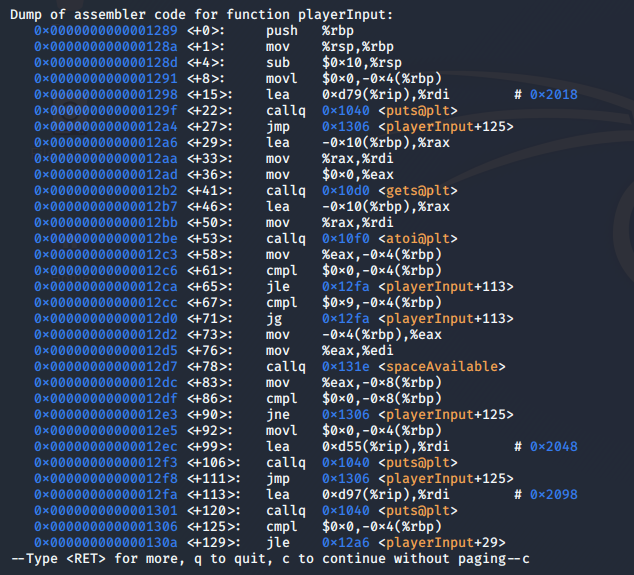
}

Upon inspection, we find that the ‘playerWins’ functions is called after the program detects that a player has won. The function opens the ‘CoolKidzClub.txt’ file, checks if the user is already added to the list or not. If the username of the user is not present in the file already, the function writes their name to the file.

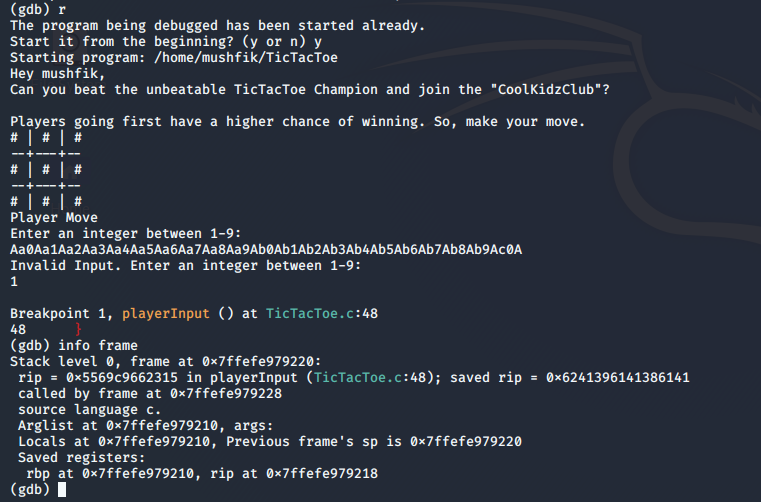
Looking into ‘playerInput’, we realize that this function is used for taking valid input from the user. It asks the user to input a number, calls a library ‘atoi’ to turn it into an integer, checks if the integer is in a valid range and returns it after checking the space on the board corresponding to that number is free or not. The function has three local variables, *char c[8]*, *int s* and *int p.* The user input is stored in a char array using ‘gets()’, which is an extremely vulnerable funtion and opens up possibilities of a buffer overflow attack.

Back in GDB, we disassemble ‘playerInput’ function with the command ‘disassemble playerInput’ to identify a breakpoint right after we provide our input to inpect the stack frame at that point. The gets() is called playerInput+41. However, since the gets() is called inside a while loop, it is best to inspect the stack right before the function returns to caller which is at playerInput+140.

We add a breakpoint there with the command ‘b \* playerInput+140’



We now run the program in gdb. Like usual, we get the prompt to play the game against the AI. However, we are now interested in overflowing the buffer. Since there are 3 local variable totalling 16 (8+4+4) bytes of data excluding eip and ebp, we want to input a metasploit string for a length over 16 bytes. We can generate such a string from this [online tool](https://zerosum0x0.blogspot.com/2016/11/overflow-exploit-pattern-generator.html)1.



Using a 64 byte string, we manage to get a Segmentation Fault. We check the stack frame info with the command ‘info frame’. The saved eip value on stack has been overwritten with the hex value ‘0x6241396141386141’. Using that value, we can find the buffer offset for the exploit.

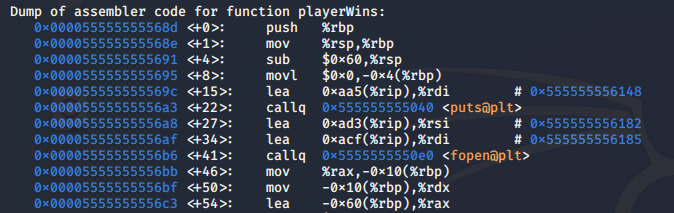
Since the hex value is in little-endian format, we need to convert it to big-endian and then covert the hex value to ASCII. Using the following python3 command line instruction, we can get the ASCII value.

mushfik@kali:~$ python3 -c "print((0x6241396141386141).to\_bytes(8, byteorder='little'))"

b’Aa8Aa9Ab’

After putting the value in the online tool, we get the buffer offset value at 24. If we can put the instruction address of playerWins function after 24 offset values, we can have the playerInput function return to playerWins function directly which will then write our name to the CoolKidzClub.txt file.

At this point, we also disassemble playerWins function to find the instruction pointer to the start of that function with the command ‘disassemble playerWins’. The first instruction is at ‘0x000055555555568d’.

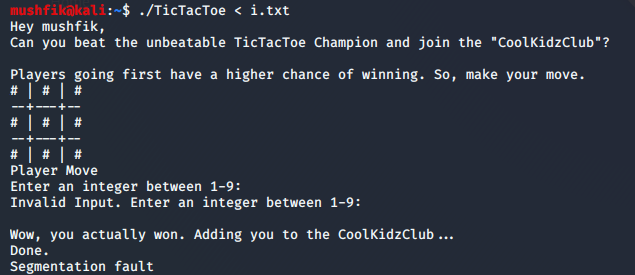


As the machine architecture supports little-endian memory addresses, we have to convert 0x00005569c966268d to little endian format and append it after 24 random characters. Using the following python command, we can write our payload to ‘i.txt’ file.

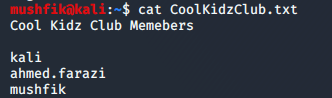
mushfik@kali:~$ python3 -c "f = open('i.txt','wb') ; f.write(b'A'\*24+(0x000055555555568d).to\_bytes(8, byteorder='little')+b'\0\n1\n') ; f.close()"

We can quickly check if ASLR (Address Space Layout Randomization) is enabled on this machine with the command ‘cat /proc/sys/kernel/randomize\_va\_space’. This prints ‘0’ on the console, indicating that ASLR is disabled the and the instruction address of playerWins function will remain the same every time we run the program.

We can now safely start the TicTacToe game feedin it the i.txt file containing our buffer overflow exploit.



Looks like we managed to successfully overflow the buffer and had our program flow get into the playerInput function, evident by the output strings we saw in the source code. We can now check the CoolKidzClub.txt file to verify if the program actually wrote our name to the file or not. And surely, we find our username added to the file.



1. <https://zerosum0x0.blogspot.com/2016/11/overflow-exploit-pattern-generator.html>