Zach Marshall

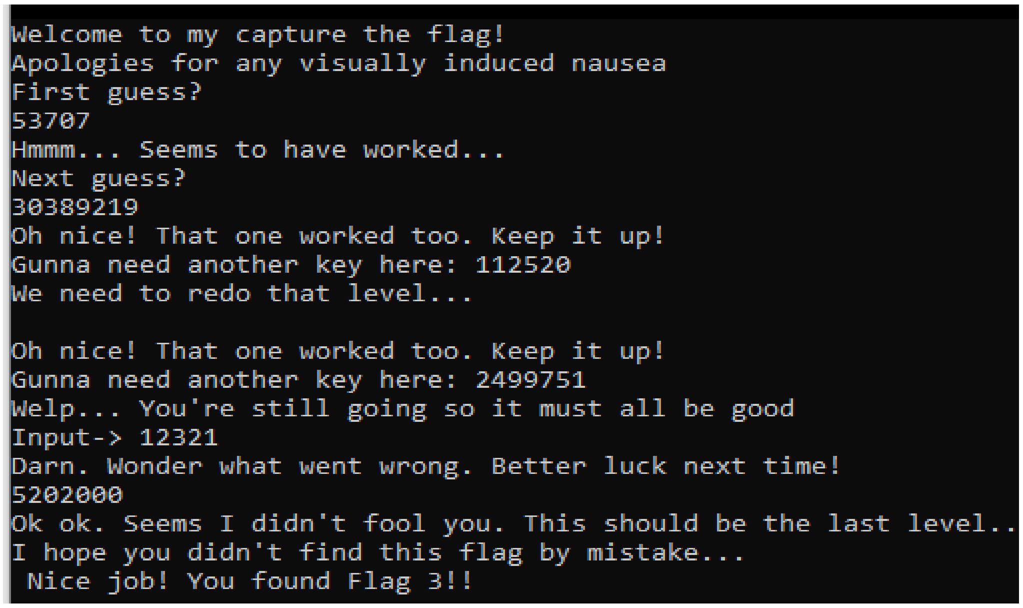
The Headache

Order:

The order I made this in was I wrote the code and then chose my values that worked per level and if they were close enough to working together, I altered the code so that it would. My main goal was obfuscation by making it hard to understand what is required to capture the flags even though it is easy to see. I then had to make minor changes so that it would transfer from windows to unix base. I have chosen to leave my code unstripped because it was already super messy in ghidra and my function names are misleading which is another layer of obfuscation.

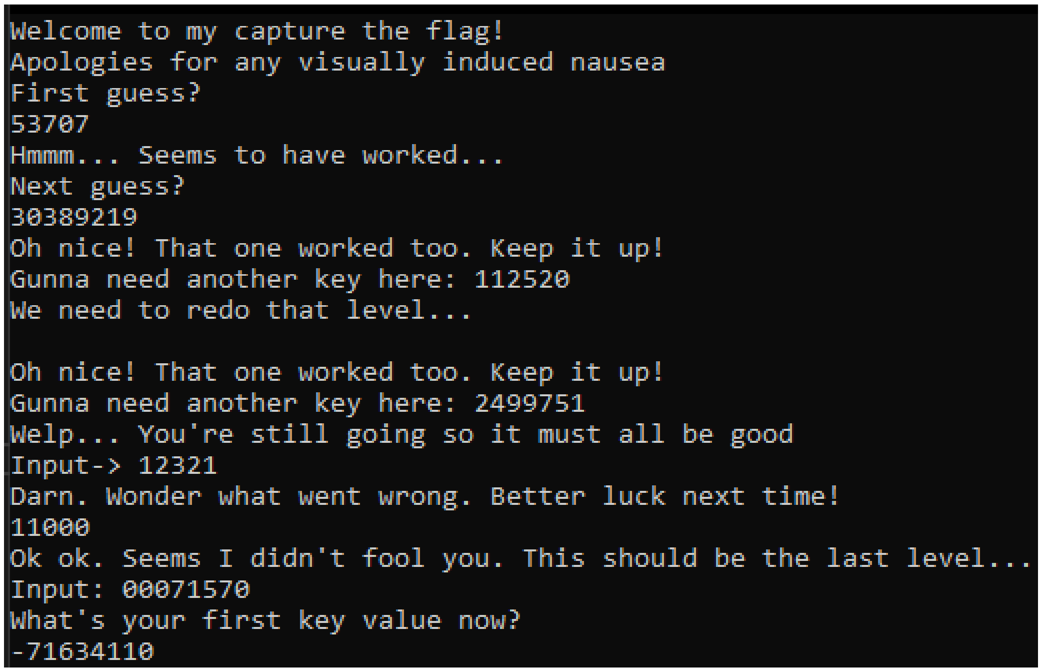
Solutions:

Below are my solutions to my capture-the-flag. Originally I had worked on this in windows because I wanted to use Visual Studios. I was able to alter the code to the necessary changes to make everything work the same and luckily the randoms worked similarly enough to allow me to have the exact same solutions in linux.

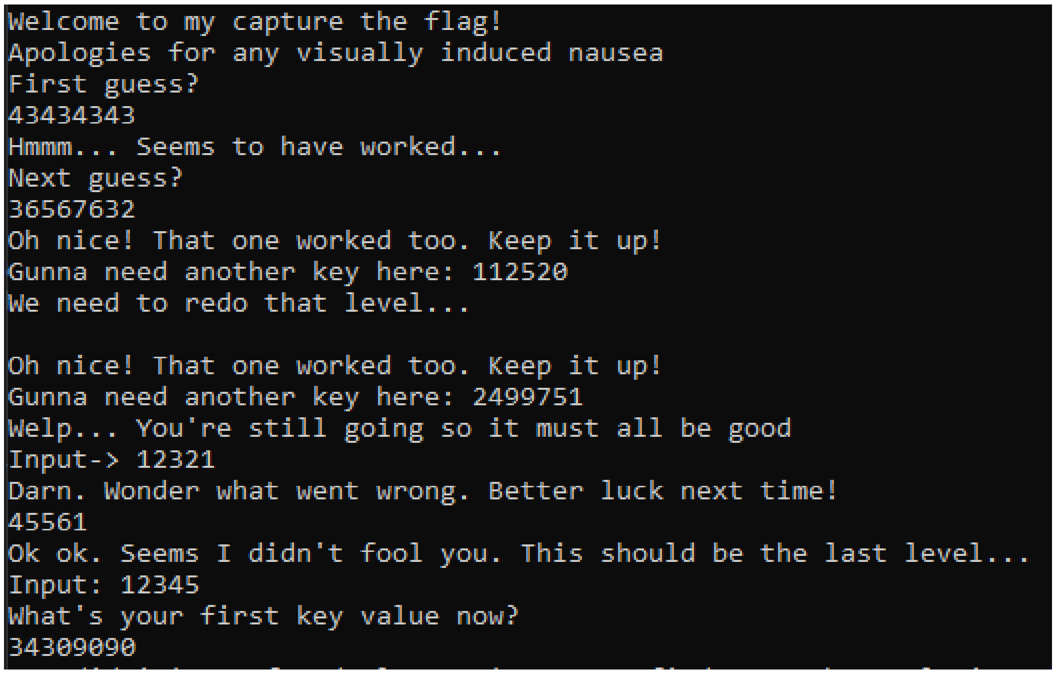


```
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
53707
Hmmm... Seems to have worked...
Next guess?
30389219
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
5202000
Ok ok. Seems I didn't fool you. This should be the last level..
I hope you didn't find this flag by mistake...
 Nice job! You found Flag 3!!
```

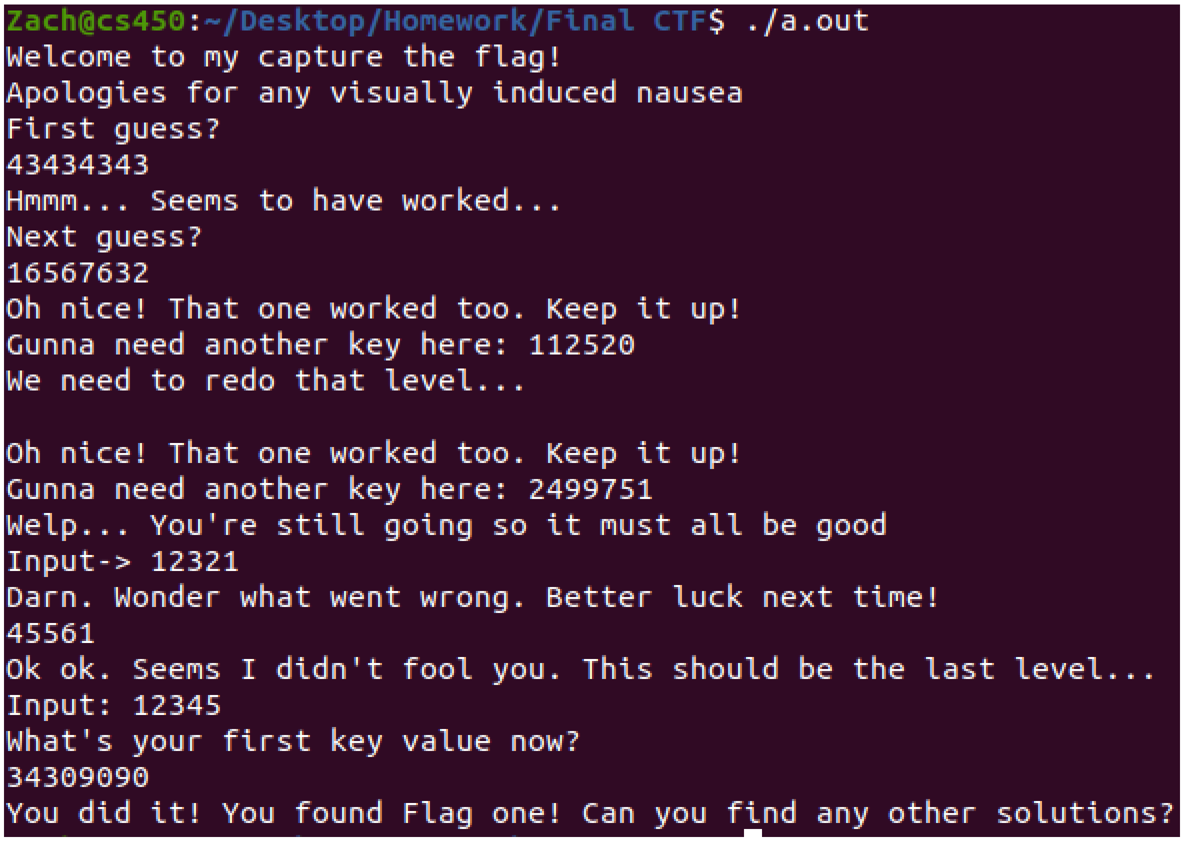
These windows outputs are in the order I made them work. I basically chose values that would work and when it didn't matter, did 112520, 3 of my favorite numbers, and my birthday for the special flag. Below is the continuation to the second flag which is hard but not as hard as the previous. The previous requires a bit of luck, or in my case force. This one is rather challenging on the last steps as I will point out in the breakdown.


```
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
53707
Hmmm... Seems to have worked...
Next guess?
30389219
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
11000
Ok ok. Seems I didn't fool you. This should be the last level...
Input: 00071570
What's your first key value now?
-71634110
```

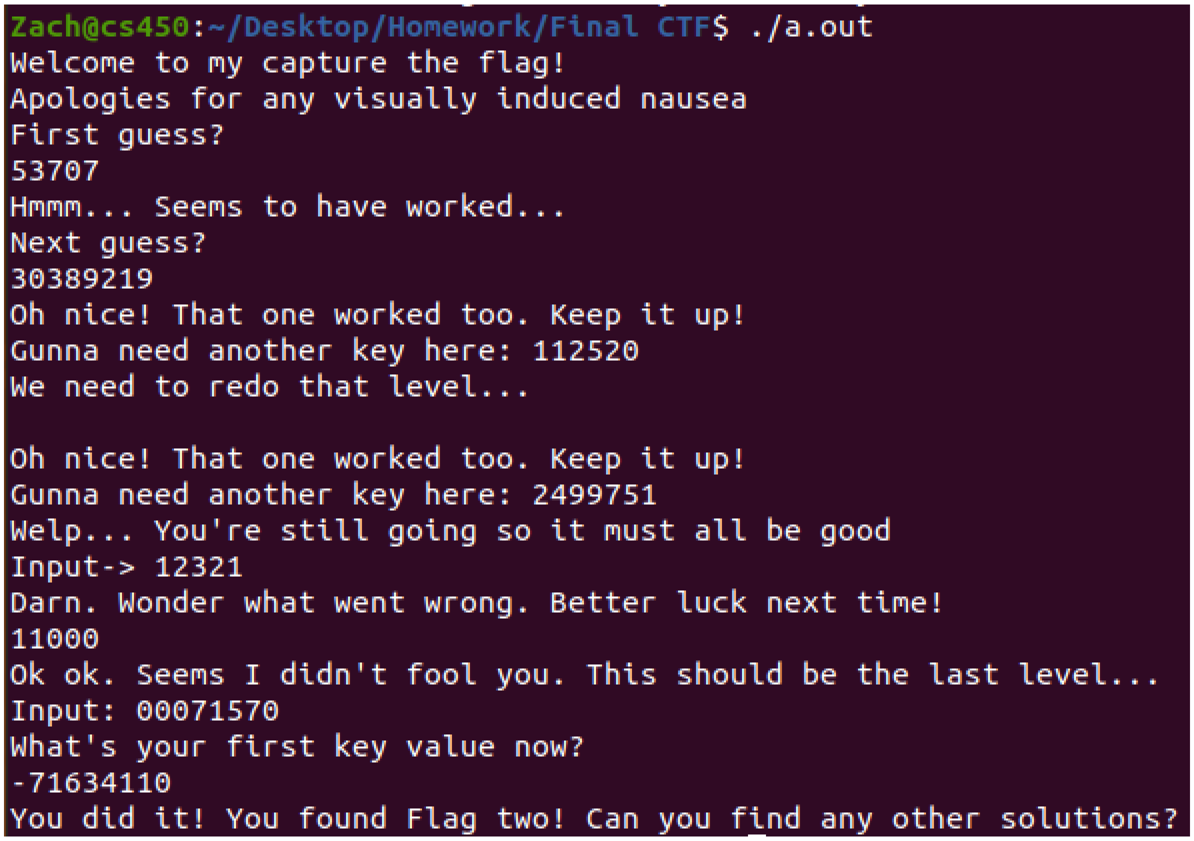
Last is the first flag that once this point is reached should be pretty easy but the hardest part is reaching the last level at all.


```
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
43434343
Hmmm... Seems to have worked...
Next guess?
36567632
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
45561
Ok ok. Seems I didn't fool you. This should be the last level...
Input: 12345
What's your first key value now?
34309090
```

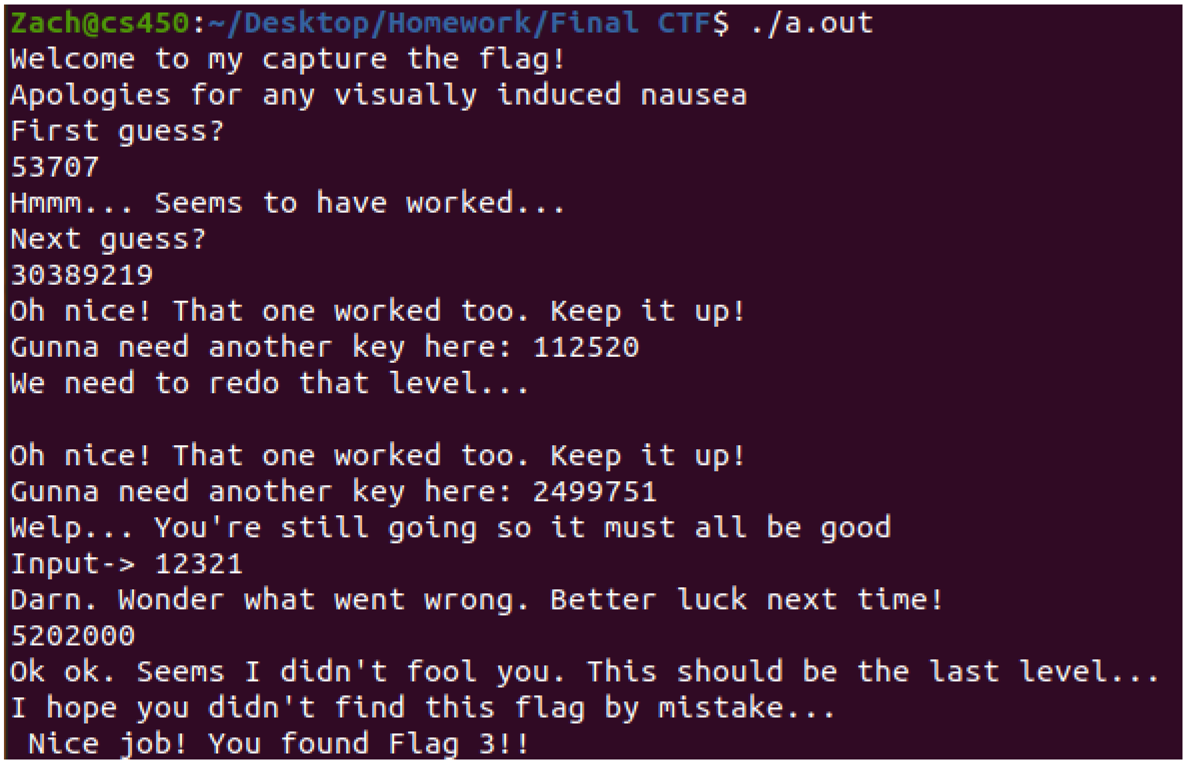
Below are the solutions to the flags in order on the virtual machine. As you can see they are the same for the same flags. I did this by setting up flag three down to one like before. Below is the correct order of the flags. As I will state later in more detail in the breakdown, the differences that were required for the switch from windows to unix was rand() was different. Luckily it wasn't as bad as I was thinking it would be or as bad as it could have been. I also was able to set up the alarm in linux to make the code exit if there isn't an answer after 5 seconds. This happens when we see the error message in the middle of the program near the end. Other than one more minor thing in the transfer from windows to linux, which will be explained later, just had to do with post increment and was actually a real headache which is funny because all it is a red herring since it should always be true but was false in linux but the solution was super simple.



```
Zach@cs450:~/Desktop/Homework/Final CTF$ ./a.out
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
43434343
Hmmm... Seems to have worked...
Next guess?
16567632
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
45561
Ok ok. Seems I didn't fool you. This should be the last level...
Input: 12345
What's your first key value now?
34309090
You did it! You found Flag one! Can you find any other solutions?
```



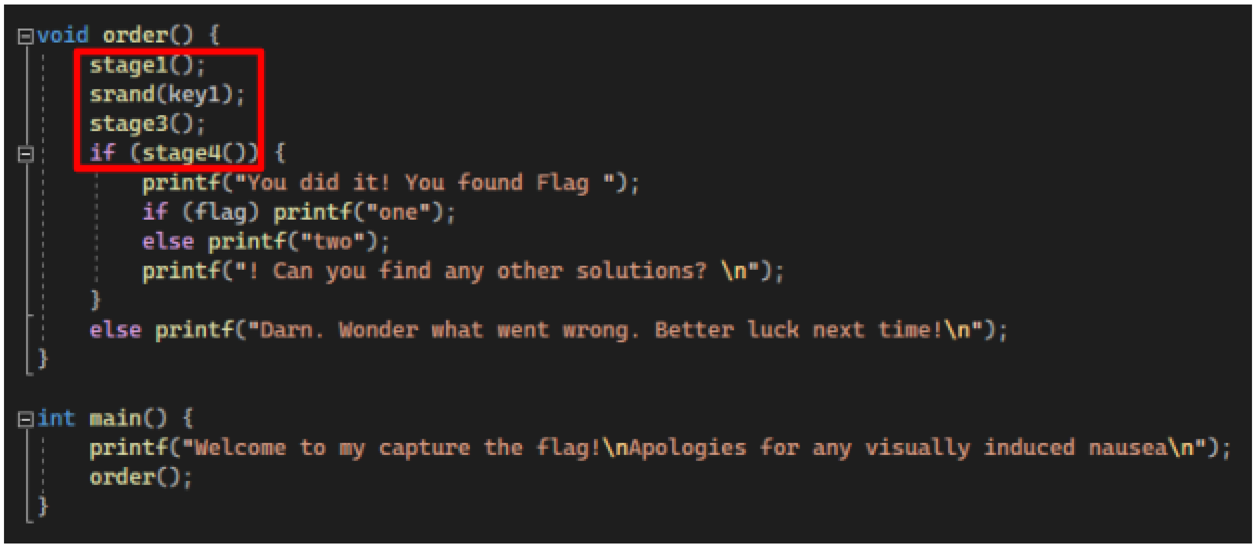
```
Zach@cs450:~/Desktop/Homework/Final CTF$ ./a.out
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
53707
Hmmm... Seems to have worked...
Next guess?
30389219
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
11000
Ok ok. Seems I didn't fool you. This should be the last level...
Input: 00071570
What's your first key value now?
-71634110
You did it! You found Flag two! Can you find any other solutions?
```



```
Zach@cs450:~/Desktop/Homework/Final CTF$ ./a.out
Welcome to my capture the flag!
Apologies for any visually induced nausea
First guess?
53707
Hmmm... Seems to have worked...
Next guess?
30389219
Oh nice! That one worked too. Keep it up!
Gunna need another key here: 112520
We need to redo that level...

Oh nice! That one worked too. Keep it up!
Gunna need another key here: 2499751
Welp... You're still going so it must all be good
Input-> 12321
Darn. Wonder what went wrong. Better luck next time!
5202000
Ok ok. Seems I didn't fool you. This should be the last level...
I hope you didn't find this flag by mistake...
 Nice job! You found Flag 3!!
```

Breakdown:

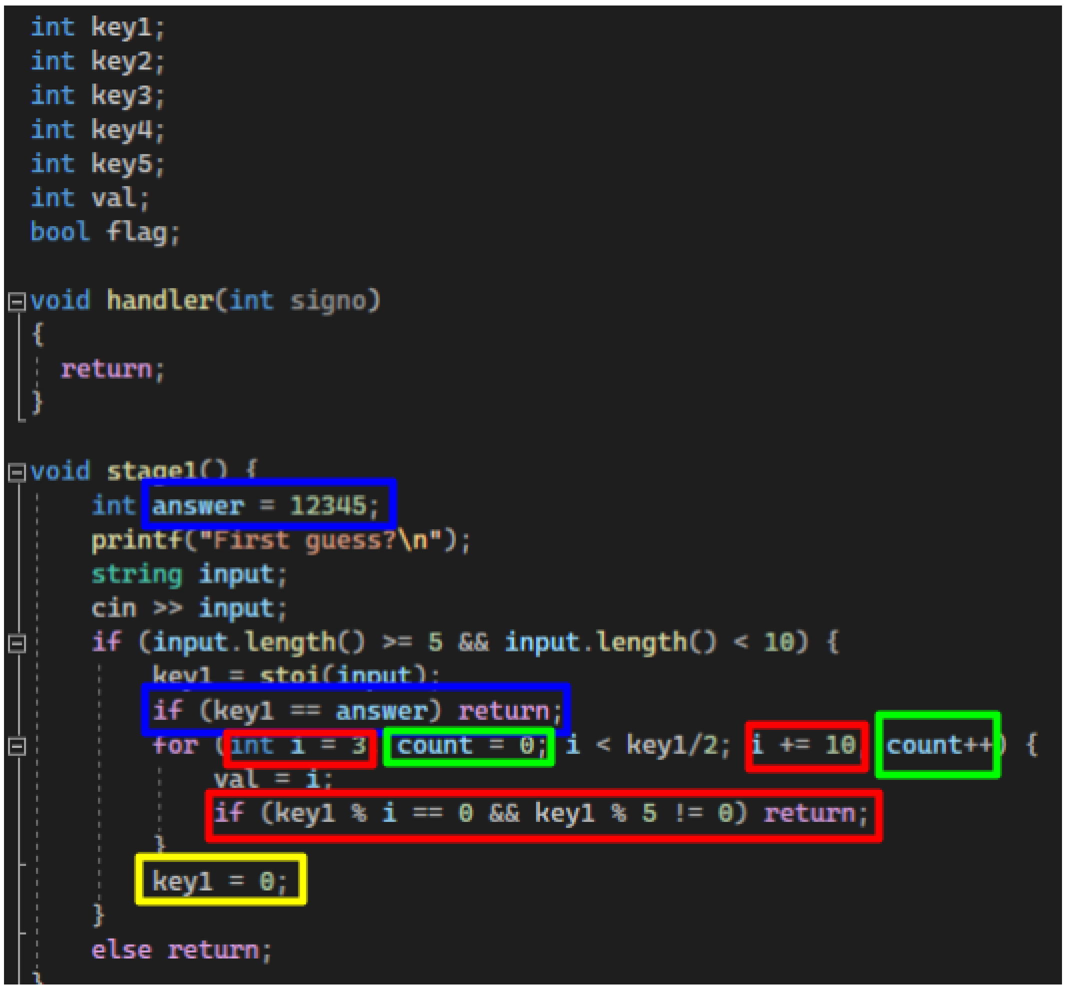
I'll do the breakdown by looking at my code since it will also be more visually pleasing while I explain this. We will see later a bit of what ghidra looks like.



```
void order() {
    stage1();
    srand(key1);
    stage3();
    if (stage4()) {
        printf("You did it! You found Flag ");
        if (flag) printf("one");
        else printf("two");
        printf("! Can you find any other solutions? \n");
    }
    else printf("Darn. Wonder what went wrong. Better luck next time!\n");
}

int main() {
    printf("Welcome to my capture the flag!\nApologies for any visually induced nausea\n");
    order();
}
```

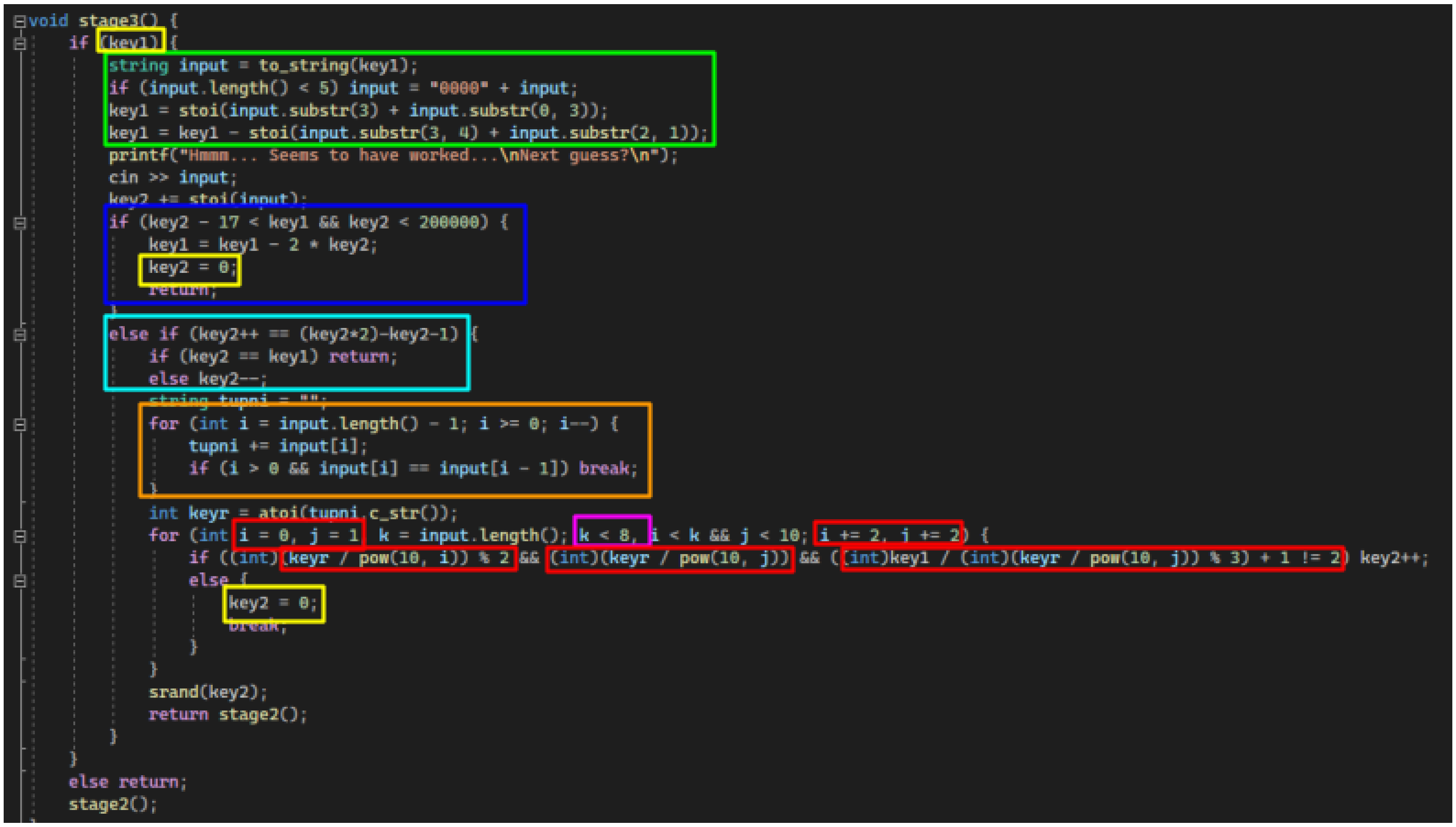
To start, it is rather simple that we can see the order of the functions… kinda. I try not to be very tricky with stage one but as we will see the actual order of these functions is stage1, stage3, stage2, stage5, stage4 which can be misleading in the order function at least when it comes to stage 4 and 5. This is another reason I wanted to leave the function names in the compiled file. In the snippet above we see a small layout of the functions and that after stage four we can win the game a couple ways or lose. We see the srand() in the box which I actually forgot I put there and doesn't do anything if you do stage3 correctly. Other than that it is rather straightforward here.

Below we have stage1 and the global variables as well as a simple function that is necessary for the interrupt for the later task. The only way to fail this task is to reach the yellow box which we will see later. In the blue box we can see that we can avoid the yellow box but later we will see that this is a bad decision. In the green box we can see that the variable count is absolutely useless. The red boxes are actually what define this level. You'll see below that it will only take values that are divisible by numbers ending in 3 but do not end in a 0 or 5. It's hard to see but we can see that our global variable “val” will be set to that value. I also make sure the value is between 5 and 10 digits long throughout this assignment to make sure nothing gets too wild.



```
int key1;
int key2;
int key3;
int key4;
int key5;
int val;
bool flag;

void handler(int signo)
{
    return;
}

void stage1() {
    int answer = 12345;
    printf("First guess?\n");
    string input;
    cin >> input;
    if (input.length() >= 5 && input.length() < 10) {
        key1 = stoi(input);
        if (key1 == answer) return;
        for (int i = 3; count = 0; i < key1/2; i += 10, count++) {
            val = i;
            if (key1 % i == 0 && key1 % 5 != 0) return;
        }
        key1 = 0;
    }
    else return;
}
```

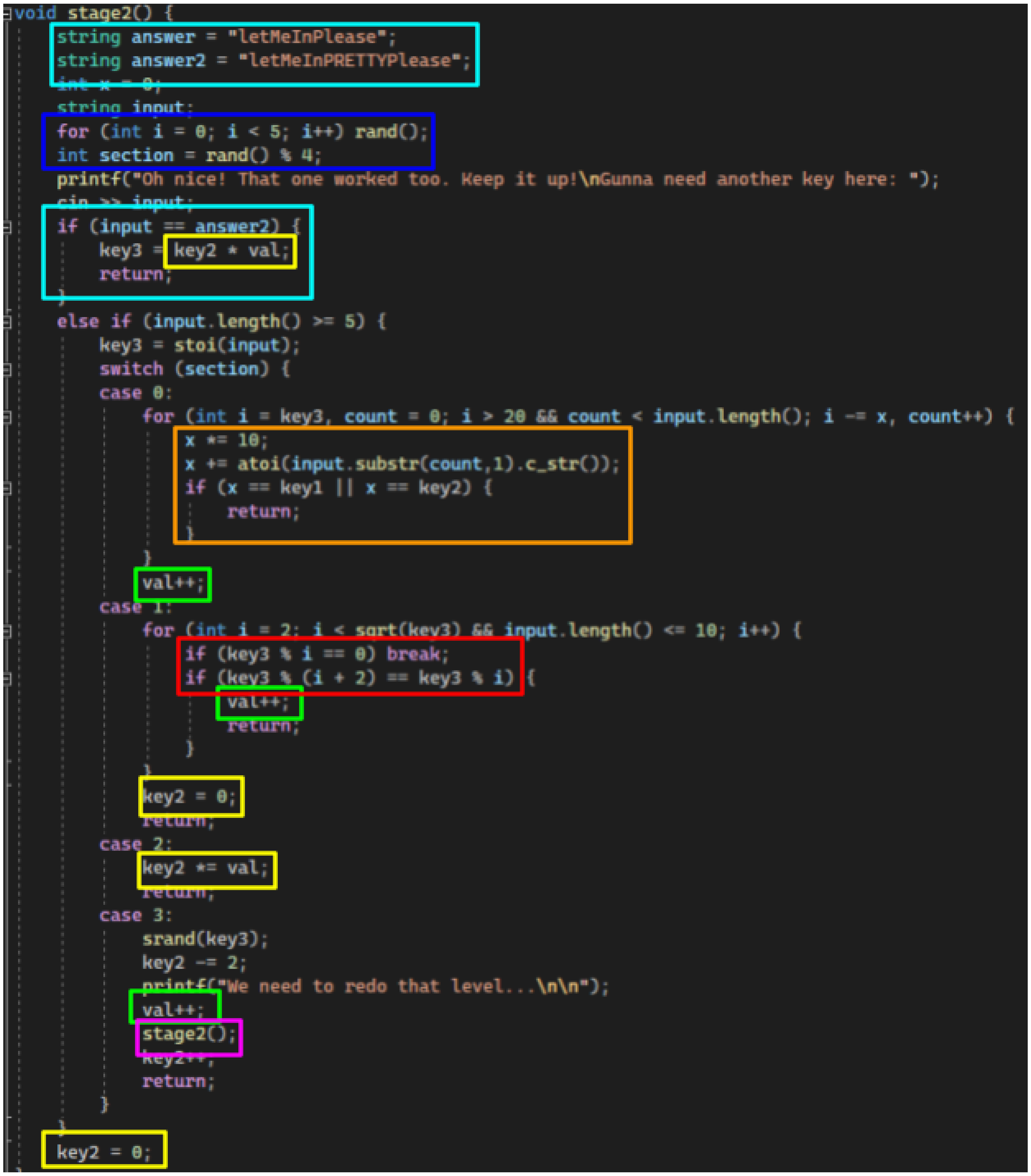
Moving on to stage3 since it’s our second step, we can see that things start ramping up. It might be a little hard to see so apologies for that. Reaching a yellow box means failing. We can see that the previous yellow box from stage1 causes us to fail here since if it’s 0 (false) it will skip this function (and therefore stage2) and will fail out with a similar check for stage5 as we will see later. To start, the green box takes the answer from the last section and does something very funky with it. First, it makes the answer greater than 5 long since they could put in something like “00003” for their answer and key1 would only be “3”. Then, it takes the first three numbers and puts them at the back of the number (e.g. 1234567 → 4567123). Then, it take that value and subtracts the numbers from position 3-7 (substr takes position then number past it as arguments) and puts the number at position 2 (e.g. (input=1234567) 4567123 - (4567 concatenated with 3)). So key1 goes from 1234567 → 4521450. Ugly. but the point. Then in the blue box we will continue the code but fail later since key2 will be 0 or less as we will see in stage3. The blue box is pretty easy to avoid but still a small threat. The light blue box is smoke and mirrors because I am comparing key2 to itself in a very ugly way. For linux I had to add that minus one at the end because the post increment actually affects key2 before the next check whereas it doesn't until after in windows. Small fix but was making me fail every time. I used gdb to track this down. Below you can see key2 can't equal key1 which is also very easy to avoid but still a way to fail later because key3 will be zero.



```
void stage3() {
    if (key1) {
        string input = to_string(key1);
        if (input.length() < 5) input = "0000" + input;
        key1 = stoi(input.substr(3) + input.substr(0, 3));
        key1 = key1 - stoi(input.substr(3, 4) + input.substr(2, 1));
        printf("Hmmm... Seems to have worked...\nNext guess?\n");
        cin >> input;
        key2 += stoi(input);
        if (key2 - 17 < key1 && key2 < 200000) {
            key1 = key1 - 2 * key2;
            key2 = 0;
            return;
        }
        else if (key2++ == (key2*2)-key2-1) {
            if (key2 == key1) return;
            else key2--;
            string tupni = "";
            for (int i = input.length() - 1; i >= 0; i--) {
                tupni += input[i];
                if (i > 0 && input[i] == input[i - 1]) break;
            }
            int keyr = atoi(tupni.c_str());
            for (int i = 0, j = 1, k = input.length(); k < 8, i < k && j < 10; i += 2, j += 2) {
                if ((int)(keyr / pow(10, i)) % 2 && (int)(keyr / pow(10, j)) && ((int)key1 / (int)(keyr / pow(10, j)) % 3) + 1 != 2) key2++;
                else {
                    key2 = 0;
                    break;
                }
            }
            srand(key2);
            return stage2();
        }
    }
    else return;
    stage2();
```

In the orange box I set up keyr which is just the reverse of key2 that was entered. In the magenta box is code that is allowed in c++ (I found out during this creation) that will never be evaluated but is ok with letting me have it there. In other words, it sounds perfect for this assignment. In order to pass this next step a few ugly things happen. The top two red boxes show i and j where i is all the even values and j is all the odd. Looking at the bottom red boxes now, the left box basically checks that every other value is odd starting on the first one… but it's of the reversed key so starting with the furthest left value. The next red box is just to make sure you don't have zero as the division for the next modulo. The last red box is a jumbled mess. It is checking that key1 integer division by reverse key2, chopped at the bits that i is not looking at (12345→54321 so 5432 → 54 → 0 (so it can't be an odd amount of digits)), mod 3 doesn't equal 1. I believe that it is easier to guess and check than to do the math for this. With gdb you can see what iteration makes it fail and you can just add one there and it should slowly fix it (while remembering that it needs to be odd every other and an even amount of digits). Seen at the bottom of that one is srand(key2). This is used to start stage3. This runs whether key2 is 0 or not. Although not necessary, srand(0) ruins your chances for later even if it is made to later steps which will be explained later.

Now we dive into stage2. Just like before the yellow boxes lead to failing later as we will see. We have seen key2=0 as bad before but what's wrong with the other ones? We will see in stage 5 that if any number is divisible by our global val then it will fail. The third stage is only really tricky because of the randomness. In the light blue we can see that we have another false lead even though we will make it to the next stage. In the dark blue box we can see that the section we enter is based on randomness of the seed set by key2 or key1 if you messed up. There isn't a way to salvage key2 when you fail early in stage3 because key2 will still be 0. The for loop is to make sure that the values I wanted to have work will do so. Each of the cases are very straightforward. Case 0 is going to exit early if your key three is the same as either key1 or key2 maybe with something else at the front. This may be a strategy as I will talk about the strategies for this in a second. Case 0 will actually fall to case 1 if it makes it through the loop. Case one checks if the number is prime essentially because I'm pretty certain that the next if will always run at some point. Case 2 will fail every time and really doesn't do anything else. In the magenta box we can see that case 3 causes recursion with some altering. Mess with key2 but it really doesn't change much but still important in stage 5.

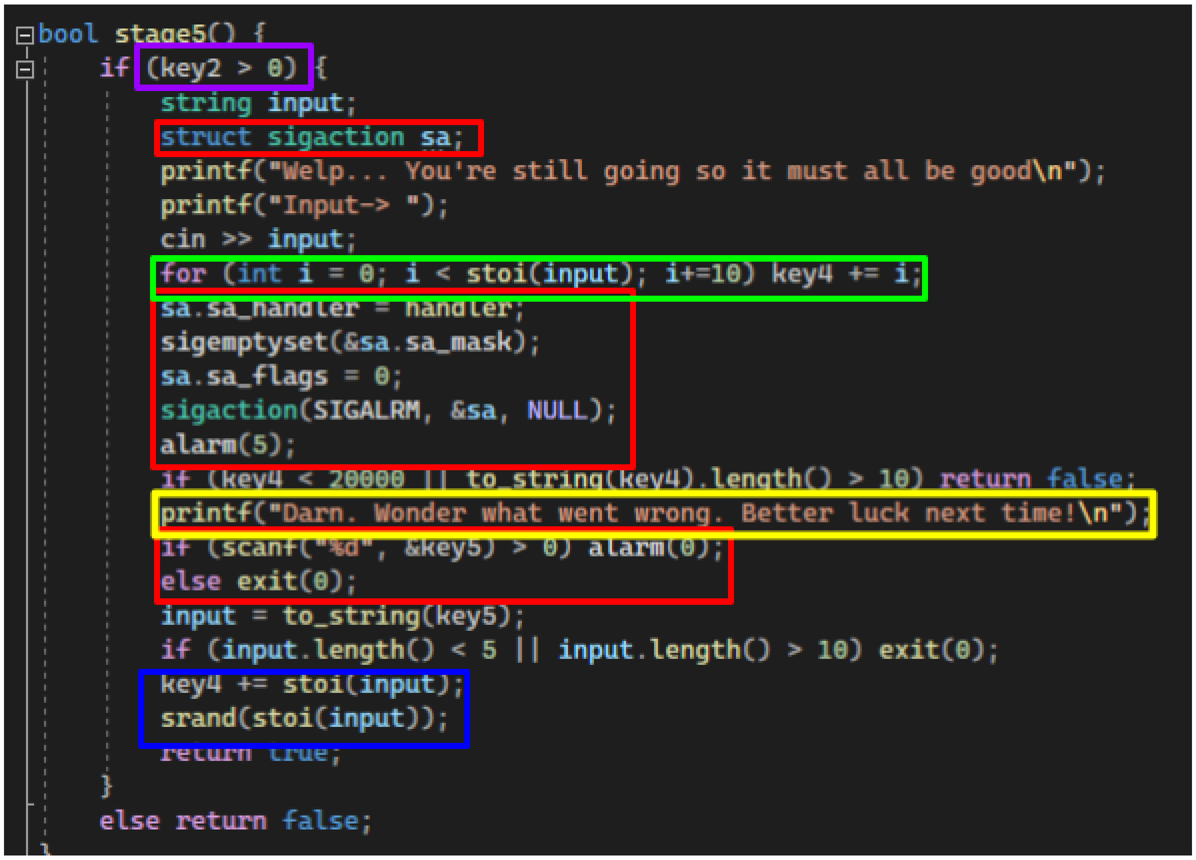


```
void stage2() {
    string answer = "letMeInPlease";
    string answer2 = "letMeInPRETTYPlease";
    int x = 0;
    string input;
    for (int i = 0; i < 5; i++) rand();
    int section = rand() % 4;
    printf("Oh nice! That one worked too. Keep it up!\nGunna need another key here: ");
    cin >> input;
    if (input == answer2) {
        key3 = key2 * val;
        return;
    }
    else if (input.length() >= 5) {
        key3 = stoi(input);
        switch (section) {
        case 0:
            for (int i = key3, count = 0; i > 20 && count < input.length(); i -= x, count++) {
                x *= 10;
                x += atoi(input.substr(count,1).c_str());
                if (x == key1 || x == key2) {
                    return;
                }
            }
            val++;
        case 1:
            for (int i = 2; i < sqrt(key3) && input.length() <= 10; i++) {
                if (key3 % i == 0) break;
                if (key3 % (i + 2) == key3 % i) {
                    val++;
                    return;
                }
            }
            key2 = 0;
            return;
        case 2:
            key2 *= val;
            return;
        case 3:
            srand(key3);
            key2 -= 2;
            printf("We need to redo that level...\n\n");
            val++;
            stage2();
            key2++;
            return;
        }
    }
    key2 = 0;
```

With this, there is potential to bring key2 back from a zero with case 2 since it'll add one and therefore surely not be divisible by val but I’m not convinced it matters because it will be negative since it’s *=.

As you can see in the green boxes, val increases often. In stage 5 we will see that val needs to be prime. Since val will end in 3 (3,13,23,43,53,73 (33 and 63 are impossible since it would exit on 3 first)) then it is very likely that it will already be prime so keeping val the same number is a good strategy. It is likely that if you subtract 2 then it will be prime as well (1, 11, not 21, 41, 51) or you can try to shoot for a number ending in 7 or 9 but with the randoms this will be pretty hard… but possible. In stage 5 (stage4) val -= 3 before the check of it’s primeness. The strategy to keep val itself is straight forward. You need to get the recursion in case 3 and then case 0 to fall to 1 with a prime. You could also get recursion twice and get case 1 or get recursion 3 times and exit through case 0. There are many options here which is fair I think. What you don’t want is to just get an even amount of increase because then it will stay odd but subtract 3 and it will be even and will always be divisible by 2. I stated earlier that seed 0 was nice (for me) because it will make you go to case 0 so if you chose to fall through val will be even and you will fail. You can make key3 get you out of that with the return in case 0 but key2 will be 0 and remain 0 so it will be for not.

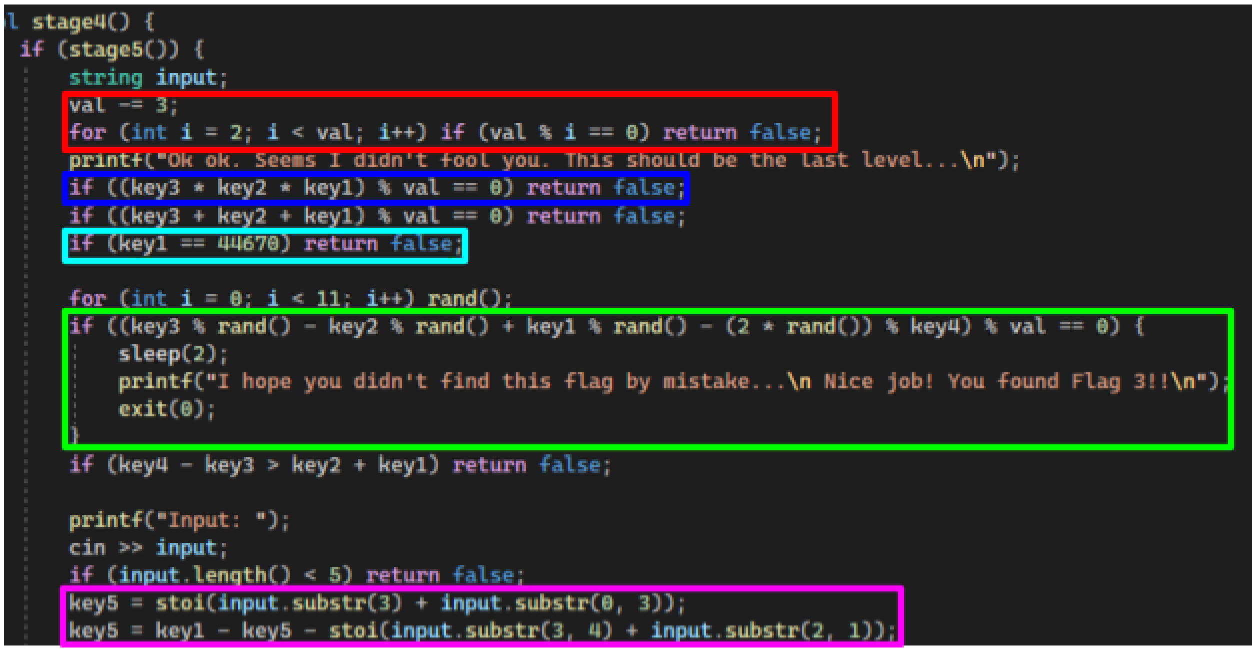
Here, we are now looking at the fourth stage, stage5. As you can see the only real trickey parts are trying to make you assume you failed and a couple things that are pretty easy to avoid. We can see that this is finally where key2 being positive matters in the purple box. The red boxes are what I use for the interrupt so if they don’t answer quickly it will fail. The green box sets up key 4 as basically a tenth of the summation of the input. The blue box is just adding the second value to key4 and we use it as the seed for stage4. Everything else is just making key4 bound to be large enough and relatively small like the other inputs.



```
bool stage5() {
    if (key2 > 0) {
        string input;
        struct sigaction sa;
        printf("Welp... You're still going so it must all be good\n");
        printf("Input-> ");
        cin >> input;
        for (int i = 0; i < stoi(input); i+=10) key4 += i;
        sa.sa_handler = handler;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sigaction(SIGALRM, &sa, NULL);
        alarm(5);
        if (key4 < 20000 || to_string(key4).length() > 10) return false;
        printf("Darn. Wonder what went wrong. Better luck next time!\n");
        if (scanf("%d", &key5) > 0) alarm(0);
        else exit(0);
        input = to_string(key5);
        if (input.length() < 5 || input.length() > 10) exit(0);
        key4 += stoi(input);
        srand(stoi(input));
        return true;
    }
    else return false;
```

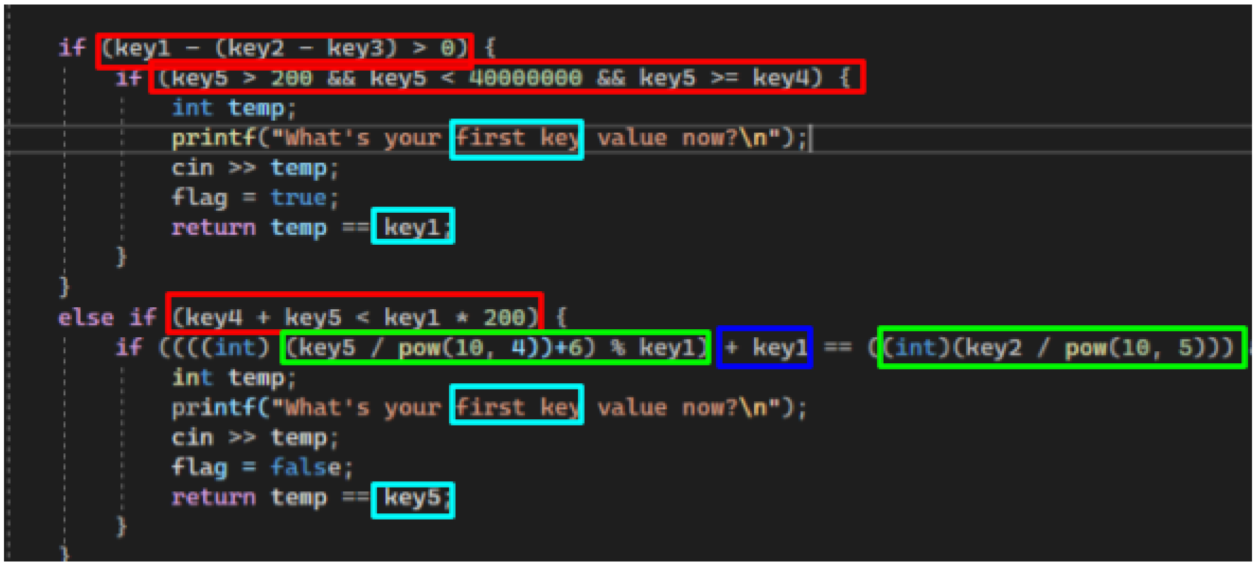
As we will see, stage5 is used as the boolean for stage4 so we return true to continue and falses to exit or simply exit so that the fake fail message in the yellow box seems more legit maybe.

In the last step we have a lot of checks to get through. We can see that since stage5 is used for whether stage4 runs or not, it will run before stage4 which helps with the confusion. In the red box I am testing if val is prime. Then in the light blue box I am finally catching if they entered 12345 for the first key since that’s what it would be now that it was changed horribly a while ago. In between those two boxes is just a random test that is very unlikely but still possible so why not. In the green box we have the third flag which if someone wanted to go for I wish them the best of luck. As you can see, each key messes with a random value. The for loop above is how I force it to work for my solution. After the box is just another random test. Now in the magenta box I am doing something similar to key5 that I did to key1 with the small difference that I subtract the modified key5 minus the stuff from key1. A small key1 will most likely mean a negative key5 which is ok..



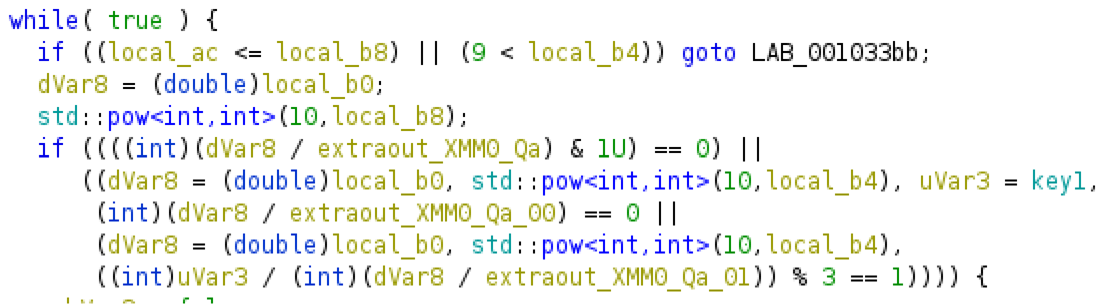
```
l stage4() {
 if (stage5()) {
     string input;
     val -= 3;
     for (int i = 2; i < val; i++) if (val % i == 0) return false;
     printf("Ok ok. Seems I didn't fool you. This should be the last level...\n");
     if ((key3 * key2 * key1) % val == 0) return false;
     if ((key3 + key2 + key1) % val == 0) return false;
     if (key1 == 44670) return false;

     for (int i = 0; i < 11; i++) rand();
     if ((key3 % rand() - key2 % rand() + key1 % rand() - (2 * rand()) % key4) % val == 0) {
         sleep(2);
         printf("I hope you didn't find this flag by mistake...\n Nice job! You found Flag 3!!\n");
         exit(0);
     }
     if (key4 - key3 > key2 + key1) return false;

     printf("Input: ");
     cin >> input;
     if (input.length() < 5) return false;
     key5 = stoi(input.substr(3) + input.substr(0, 3));
     key5 = key1 - key5 - stoi(input.substr(3, 4) + input.substr(2, 1));
```

Below are the two other flags to get. The top one requires almost nothing from key5 but the hardest part is that key1 will need to be large enough in the first red box to even reach the second red box which is simple. For my first run, 53707 was nowhere large enough which is why I needed 43434343. Small key2 and key3 is another strategy but that seems harder than a large key1. After that, all you need to know is what your key1 was transformed to so long ago.



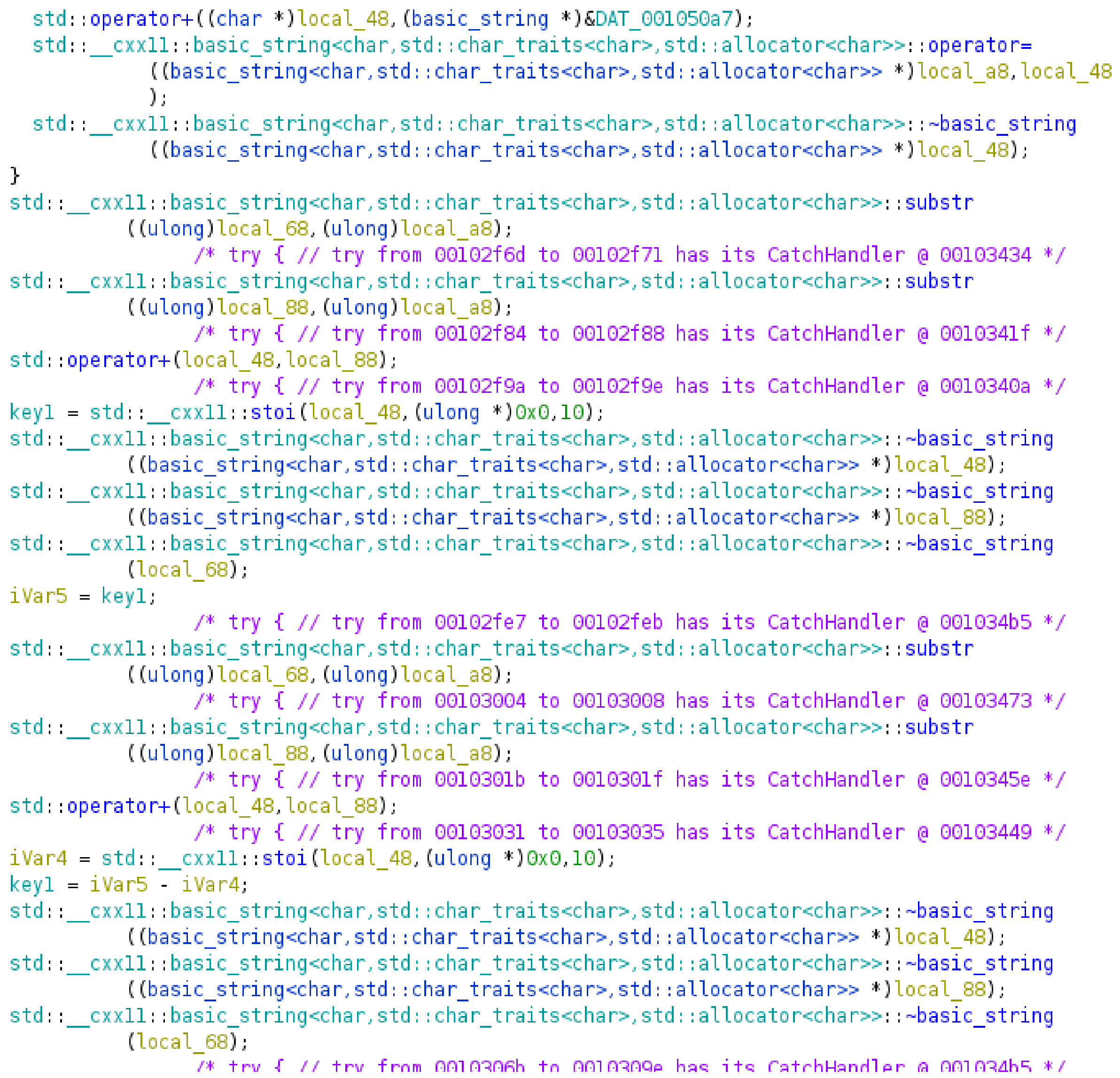
```
    if (key1 - (key2 - key3) > 0) {
        if (key5 > 200 && key5 < 40000000 && key5 >= key4) {
            int temp;
            printf("What's your first key value now?\n");
            cin >> temp;
            flag = true;
            return temp == key1;
        }
    }
    else if (key4 + key5 < key1 * 200) {
        if ((((int) (key5 / pow(10, 4))+6) % key1) + key1 == ((int)(key2 / pow(10, 5)))
            int temp;
            printf("What's your first key value now?\n");
            cin >> temp;
            flag = false;
            return temp == key5;
        }
    }
```

You can see how in the light blue box there is something tricky for flag two. I ask for key1 but check against key5. I accidentally left what's in the yellow box in the code when I was testing something but what's the harm in leaving useless stuff? The red box for flag two requires small key5 (and when it's negative it's useful for this) and medium key1 since if it's large it might go to the first flag instead. The green boxes compare the the top 6 (if key5 is 10 in length) of key5 minus 6 (which was necessary to make mine work) and the top 5 (if there are all 10 digits possible for key2) are equal. The first dark blue box is incase key5 is negative it will bring it back positive. That blue box actually makes this a lot harder because you can't just make key5 and key2 small enough so that these operations make them 0 ((int) 12345 / pow(10,5) = .12345 → 0), which is an easy way to make the sides match, because it is impossible for key1 to be zero and it is impossible to have the result of that mod equal to negative key1 because it would instead take "another" key1 out and leave you with 0 plus key1 and therefore no easy check. Below we can see that the ghidra on the second stage of this mess becomes much more of a mess. I think that since I am using string objects instead of char arrays it looks like this. I think that this is just more proof as to why my ctf should be called the headache. I changed the intro that can be seen in my solutions to say the name of my ctf. In ghidra things are a bit funky but everything looks fine. I think that my ctf will be challenging while still fair. It might not, however, be fair to ask them to get all three flags. The hardest part about this is how all the parts play together and how there are traps since I allow wrong entries because they just fail later.



```
while( true ) {
  if ((local_ac <= local_b8) || (9 < local_b4)) goto LAB_001033bb;
  dVar8 = (double)local_b0;
  std::pow<int,int>(10,local_b8);
  if ((((int)(dVar8 / extraout_XMM0_Qa) & 1U) == 0) ||
     ((dVar8 = (double)local_b0, std::pow<int,int>(10,local_b4), uVar3 = key1,
      (int)(dVar8 / extraout_XMM0_Qa_00) == 0 ||
      (dVar8 = (double)local_b0, std::pow<int,int>(10,local_b4),
      ((int)uVar3 / (int)(dVar8 / extraout_XMM0_Qa_01)) % 3 == 1)))) {
```

Above is the if for stage3. Below is the string changing for key1 in stage3. I wasn't sure if this would make it too hard so I included headache_G which is compiled with the -g option for gdb.



```
  std::operator+((char *)local_48,(basic_string *)&DAT_001050a7);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator=
            ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_a8,local_48
            );
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
            ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_48);
}
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::substr
          ((ulong)local_68,(ulong)local_a8);
                /* try { // try from 00102f6d to 00102f71 has its CatchHandler @ 00103434 */
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::substr
          ((ulong)local_88,(ulong)local_a8);
                /* try { // try from 00102f84 to 00102f88 has its CatchHandler @ 0010341f */
std::operator+(local_48,local_88);
                /* try { // try from 00102f9a to 00102f9e has its CatchHandler @ 0010340a */
key1 = std::__cxx11::stoi(local_48,(ulong *)0x0,10);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_48);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_88);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          (local_68);
iVar5 = key1;
                /* try { // try from 00102fe7 to 00102feb has its CatchHandler @ 001034b5 */
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::substr
          ((ulong)local_68,(ulong)local_a8);
                /* try { // try from 00103004 to 00103008 has its CatchHandler @ 00103473 */
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::substr
          ((ulong)local_88,(ulong)local_a8);
                /* try { // try from 0010301b to 0010301f has its CatchHandler @ 0010345e */
std::operator+(local_48,local_88);
                /* try { // try from 00103031 to 00103035 has its CatchHandler @ 00103449 */
iVar4 = std::__cxx11::stoi(local_48,(ulong *)0x0,10);
key1 = iVar5 - iVar4;
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_48);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_88);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
          (local_68);
                /* try { // try from 0010306b to 0010309e has its CatchHandler @ 001034b5 */
```