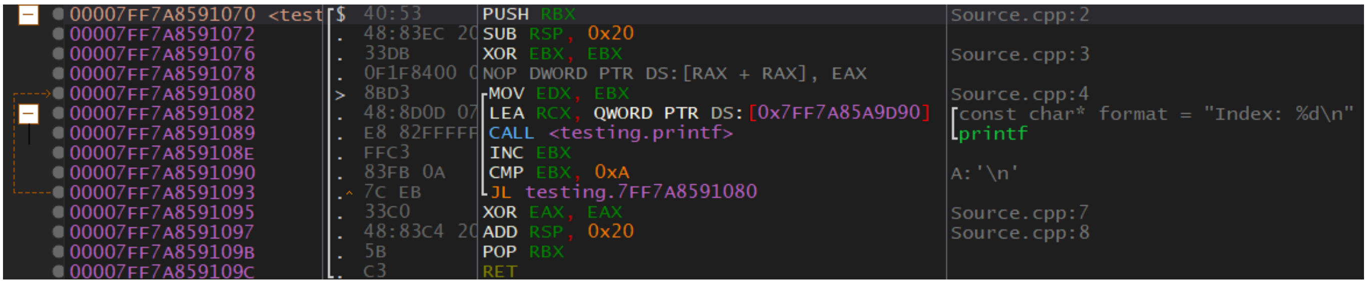
Loops

Let's take a look at a loop. Loops are pretty simple, but they are easy to miss if you are just skimming over the disassembly. When reversing a binary you won't see keywords like "while" or "for" which are indicators of a loop. This is the main reason why they can be easy to miss.

For Loop



```
00007FF7A8591070 <test $ 40:53          PUSH RBX                                       Source.cpp:2
00007FF7A8591072       . 48:83EC 20     SUB RSP, 0x20
00007FF7A8591076       . 33DB           XOR EBX, EBX                                   Source.cpp:3
00007FF7A8591078       . 0F1F8400 0     NOP DWORD PTR DS:[RAX + RAX], EAX
00007FF7A8591080       > 8BD3           MOV EDX, EBX                                   Source.cpp:4
00007FF7A8591082       . 48:8D0D 07     LEA RCX, QWORD PTR DS:[0x7FF7A85A9D90]         const char* format = "Index: %d\n"
00007FF7A8591089       . E8 82FFFFF     CALL <testing.printf>                          printf
00007FF7A859108E       . FFC3           INC EBX
00007FF7A8591090       . 83FB 0A        CMP EBX, 0xA                                   A:'\n'
00007FF7A8591093       .^ 7C EB         JL testing.7FF7A8591080
00007FF7A8591095       . 33C0           XOR EAX, EAX                                   Source.cpp:7
00007FF7A8591097       . 48:83C4 20     ADD RSP, 0x20                                  Source.cpp:8
00007FF7A859109B       . 5B             POP RBX
00007FF7A859109C       . C3             RET
```

The quickest way to identify a loop is to find the index/iteration counter. In the example above we can see that EBX is being zeroed out with `XOR EBX, EBX`. This is nothing special, but if we go down a little we can see that EBX is being incremented with `INC EBX`. x64dbg also uses arrows (such as orange one on the far left) that hint at a possible loop. Alright, enough skimming, let's actually take a real look at the loop.

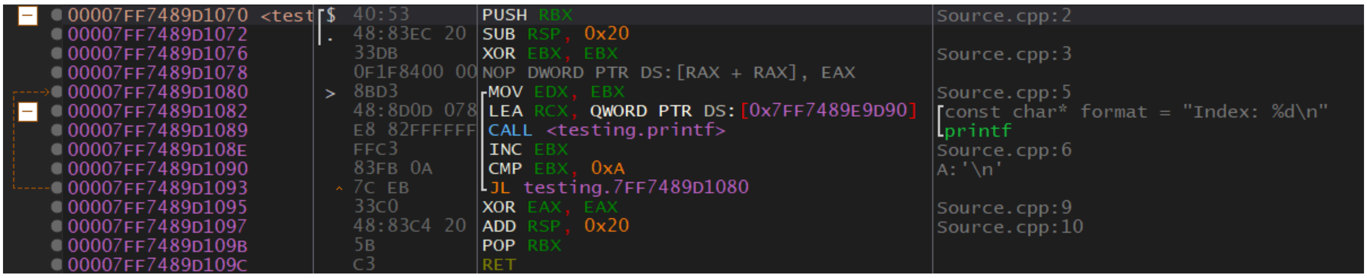
- First EBX is zeroed. We now know that this loop is going to start with a loop of zero. Let's refer to EBX as the loop index.
- EDX is set to the loop index.
- A string is moved into RCX.
- `printf()` is called. RCX (the first parameter passed to `printf()` contains the string "Index: %d\n". EDX (the second parameter passed to `printf()`) contains the loop index. We can assume the `printf()` call looks something like:
  `printf("Index: %d\n", index);`
- The loop index is then incremented by one.
- The index is compared to 10 (0xA).
- If the index is less than 10, it will repeat the loop. The start of the loop is `MOV EDX, EBX`.

So this loop will print "Index: %d\n" 10 times. Here is what it looks like written in C/C++:

```
for (int i = 0; i < 10; i++) {
    printf("Index: %d\n", i);
}
```

While Loops

Here is a program that is functionally the same as the previous one, except this one is a while loop.



```
00007FF7489D1070 <test $ 40:53          PUSH RBX                                     Source.cpp:2
00007FF7489D1072       . 48:83EC 20     SUB RSP, 0x20
00007FF7489D1076         33DB           XOR EBX, EBX                                 Source.cpp:3
00007FF7489D1078         0F1F8400 00    NOP DWORD PTR DS:[RAX + RAX], EAX
00007FF7489D1080       > 8BD3           MOV EDX, EBX                                 Source.cpp:5
00007FF7489D1082         48:8D0D 078    LEA RCX, QWORD PTR DS:[0x7FF7489E9D90]       const char* format = "Index: %d\n"
00007FF7489D1089         E8 82FFFFFF    CALL <testing.printf>                        printf
00007FF7489D108E         FFC3           INC EBX                                      Source.cpp:6
00007FF7489D1090         83FB 0A        CMP EBX, 0xA                                 A:'\n'
00007FF7489D1093       ^ 7C EB          JL testing.7FF7489D1080
00007FF7489D1095         33C0           XOR EAX, EAX                                 Source.cpp:9
00007FF7489D1097         48:83C4 20     ADD RSP, 0x20                                Source.cpp:10
00007FF7489D109B         5B             POP RBX
00007FF7489D109C         C3             RET
```

As you can see, the while loop is the same as the for loop.

Do While Loops

Do while loops are not used very often by most developers, but compilers generate them constantly. Do while loops are while loops except for the code inside of the while loop is run at least once. Most developers make loops that do exactly this even though they may not define them as such. Both the while loop and for loop examples given previously could be done with do while loops. In an attempt to improve performance many compilers will use do while loops instead of for loops or while loops. This is because they can eliminate one comparison and jump from the loop. If you ever use a decompiler you will mostly see them showing loops as do while loops, this is why.