When we download the binary for random junks, we get an x86_64 executable that seems reasonably straightforward.
Essentially, the program is taking some sort of operation at the top, which we can ignore, reversing and "encrypting" it, and then printing the resulting string. We can see this when we run the program.

`?em wonk uoy oD !!!olleH !sdnatsrednu eno on taht yrots gib a evah I dnA !RACIE si eman yM !edoc ym si siht dnA`

This is followed by two extremely long pieces of text, the second of which looks more like a target for a flag than anything else. More on that later. So, we want to get to the initial string so that we can (probably!) get the flag from that initial string. There are two operations we need to stop from happening: the string reverse and the encryption. The string reverse block is simple. Use your decompiler of choice (I'm partial to Binary Ninja) to replace the strrev() call with a no-operation and do this for all four calls.

```
00401c49  call    std::allocator<char>::allocator
00401c4e  lea     rax, [rbp-0xd9f0 {var_d9f8}]
00401c55  mov     rdi, rax
00401c58  nop
00401c59  nop
00401c5a  nop
00401c5b  nop
00401c5c  nop
00401c5d  mov     rcx, rax
00401c60  lea     rdx, [rbp-0x1a841 {var_1a849}]
00401c67  lea     rax, [rbp-0x1a700 {var_1a708}]
00401c6e  mov     rsi, rcx
00401c71  mov     rdi, rax
00401c74  call    std::__cxx11::basic_stri... std::allocator<char> >::basic_string
00401c79  lea     rax, [rbp-0x1a6e0 {var_1a6e8}]
00401c80  lea     rdx, [rbp-0x1a700 {var_1a708}]
00401c87  mov     rsi, rdx
00401c8a  mov     rdi, rax
00401c8d  call    encryptDecryptb
```

Once this is done, we get a slightly nicer output, but the final print still looks wrong, and we get nothing out of it in terms of converting to another binary or file type, nor do we get a flag string. So, we need to reverse the encryption. Looking at the top, we can see that the so-called encryption isn't really encryption at all. It's an XOR operation with some number, which is different for each encrypt call. This is very easy to stop without using a no-operation patch (because there's a chance that it could mess something else up, so best to take the lowest-impact approach), because an XOR with zero will simply result in an output of the original input.

```
encryptDecrypt:
00401189  push    rbp
0040118a  mov     rbp, rsp
0040118d  push    rbx
0040118e  sub     rsp, 0x28
00401192  mov     qword [rbp-0x28], rdi
00401196  mov     qword [rbp-0x30], rsi
0040119a  mov     rax, qword fs:[0x28]
004011a3  mov     qword [rbp-0x18], rax
004011a7  xor     eax, eax
004011a9  mov     byte [rbp-0x1d {var_25}], 0x0
004011ad  mov     rdx, qword [rbp-0x30]
004011b1  mov     rax, qword [rbp-0x28]
004011b5  mov     rsi, rdx
004011b8  mov     rdi, rax
004011bb  call    std::__cxx11::basic_stri... std::allocator<char> >::basic_string
004011c0  mov     dword [rbp-0x1c], 0x0
```

Once this is done, we're ready to run our patched binary. On a hunch, I decoded the file using `xxd -p -r <copiedtext></copiedtext> <targetfile></targetfile>` and used `file` on the output, expecting to see another binary. However, `file h h: PNG image data, 862 x 171, 8-bit/color RGB, non-interlaced` tells us otherwise. It's an image, and viewing the image normally shows that it contains the flag.