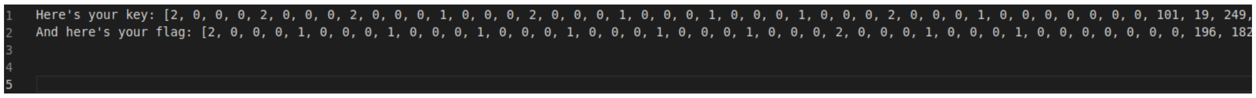
HTB Reverse Engineering: Nuts and Bolts

Tools: Ghidra

This challenge was pretty cool and looked a lot different than I had previously seen. Also there were only about 50 completions during the CTF.

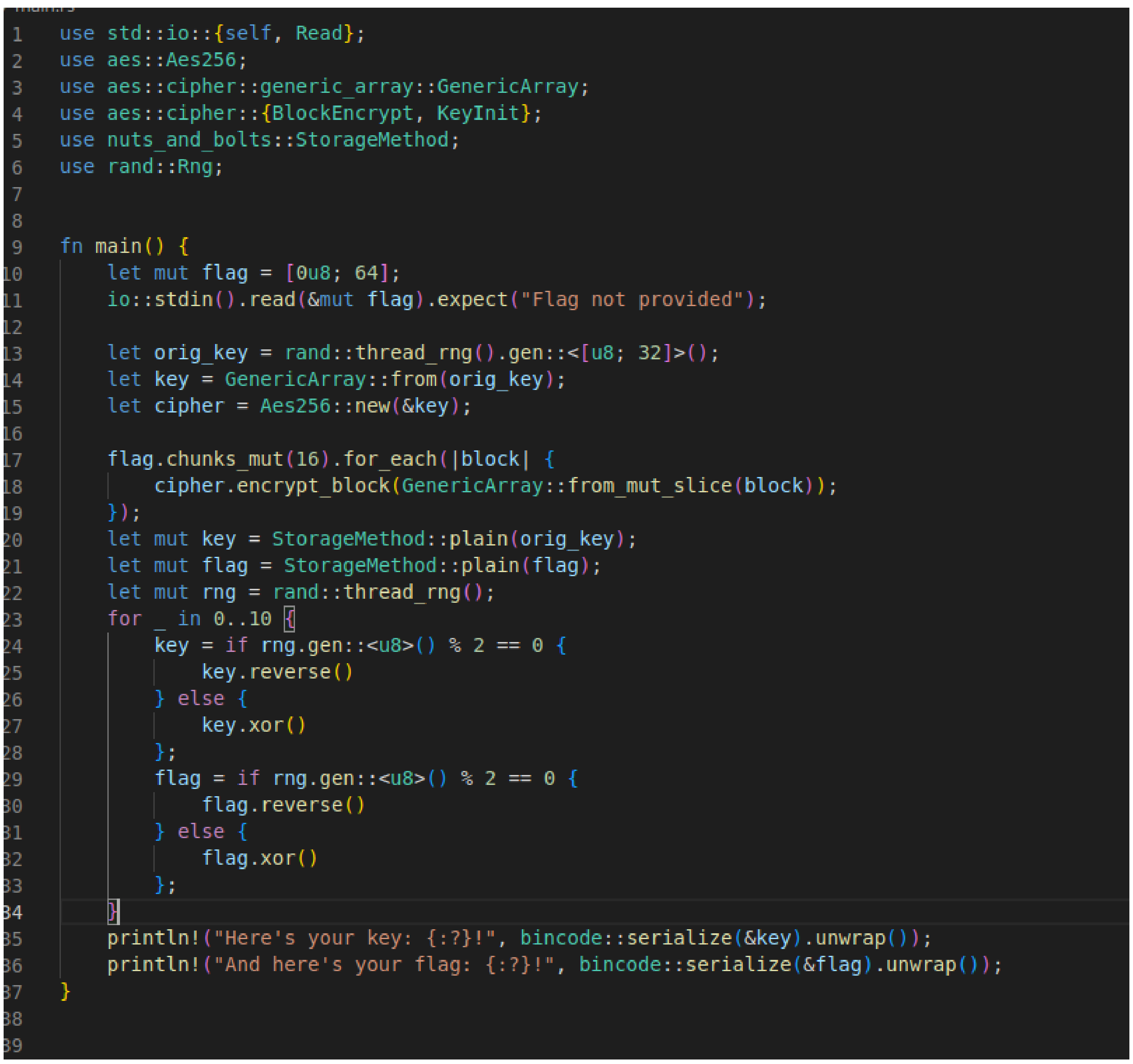
Starting off we are given a binary, a main.rs file and an output.txt file.

The main.rs file is actually the main function for the Nuts and Bolts binary. The output.txt file gives us our flag and key, but is encoded.



```
Here's your key: [2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 101, 19, 249,
And here's your flag: [2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 196, 182
```

From the main.rs files we can see how the main function runs. We can see that the key and flag are both being encrypted by AES-256. (AES-256 also uses a 32 byte length for its key!) Towards the bottom of the function we can see that the flag and key are also being if/else to be reversed or xored. So now we almost have what we need to decipher this code. But now we need the XOR key.



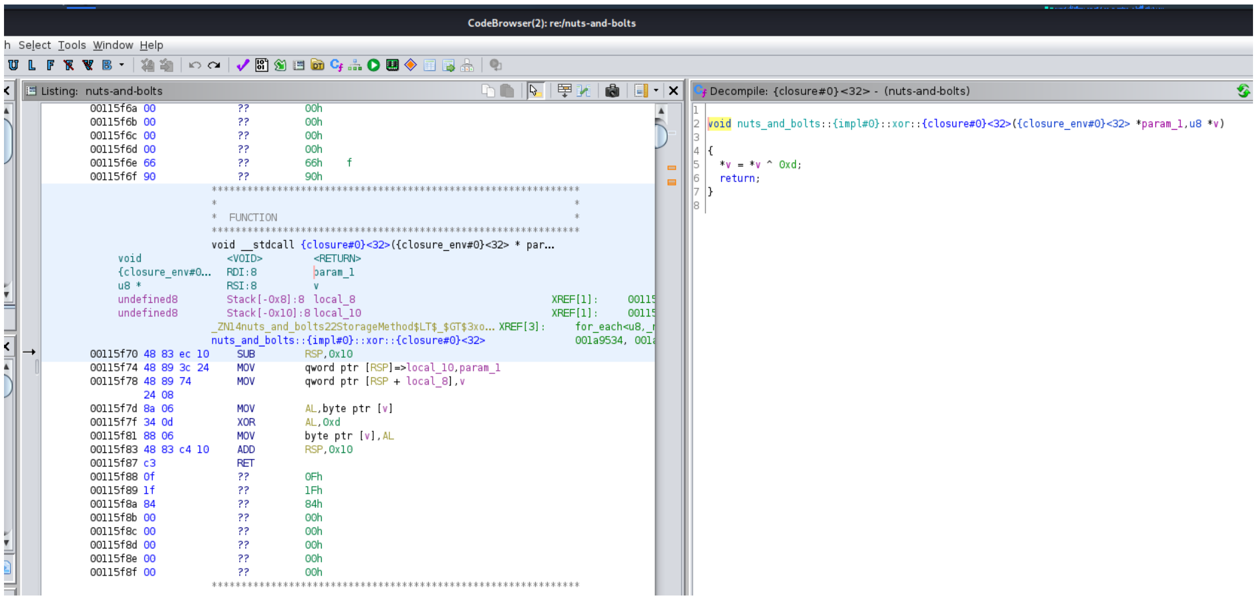
```rust
use std::io::{self, Read};
use aes::Aes256;
use aes::cipher::generic_array::GenericArray;
use aes::cipher::{BlockEncrypt, KeyInit};
use nuts_and_bolts::StorageMethod;
use rand::Rng;


fn main() {
    let mut flag = [0u8; 64];
    io::stdin().read(&mut flag).expect("Flag not provided");

    let orig_key = rand::thread_rng().gen::<[u8; 32]>();
    let key = GenericArray::from(orig_key);
    let cipher = Aes256::new(&key);

    flag.chunks_mut(16).for_each(|block| {
        cipher.encrypt_block(GenericArray::from_mut_slice(block));
    });
    let mut key = StorageMethod::plain(orig_key);
    let mut flag = StorageMethod::plain(flag);
    let mut rng = rand::thread_rng();
    for _ in 0..10 {
        key = if rng.gen::<u8>() % 2 == 0 {
            key.reverse()
        } else {
            key.xor()
        };
        flag = if rng.gen::<u8>() % 2 == 0 {
            flag.reverse()
        } else {
            flag.xor()
        };
    }
    println!("Here's your key: {:?}!", bincode::serialize(&key).unwrap());
    println!("And here's your flag: {:?}!", bincode::serialize(&flag).unwrap());
}
```

After some digging I found what might be the XOR key in the decompile on the right. “^ 0xd” So let's get to cracking.


CodeBrowser(2): re:/nuts-and-bolts

Listing: nuts-and-bolts

```
        00115f6a 00           ??          00h
        00115f6b 00           ??          00h
        00115f6c 00           ??          00h
        00115f6d 00           ??          00h
        00115f6e 66           ??          66h    f
        00115f6f 90           ??          90h
                            **************************************************************
                            *                                                            *
                            *  FUNCTION                                                  *
                            **************************************************************
                            void __stdcall {closure#0}<32>({closure_env#0}<32> * par…
            void              <VOID>         <RETURN>
            {closure_env#0…   RDI:8          param_1
            u8 *              RSI:8          v
            undefined8        Stack[-0x8]:8  local_8                              XREF[1]:     00115
            undefined8        Stack[-0x10]:8 local_10                             XREF[1]:     00115
                            _ZN14nuts_and_bolts22StorageMethod$LT$_$GT$3xo… XREF[3]:     for_each<u8,_
                            nuts_and_bolts::{impl#0}::xor::{closure#0}<32>            001a9534, 001a
        00115f70 48 83 ec 10    SUB         RSP,0x10
        00115f74 48 89 3c 24    MOV         qword ptr [RSP]=>local_10,param_1
        00115f78 48 89 74       MOV         qword ptr [RSP + local_8],v
                 24 08
        00115f7d 8a 06          MOV         AL,byte ptr [v]
        00115f7f 34 0d          XOR         AL,0xd
        00115f81 88 06          MOV         byte ptr [v],AL
        00115f83 48 83 c4 10    ADD         RSP,0x10
        00115f87 c3             RET
        00115f88 0f           ??          0Fh
        00115f89 1f           ??          1Fh
        00115f8a 84           ??          84h
        00115f8b 00           ??          00h
        00115f8c 00           ??          00h
        00115f8d 00           ??          00h
        00115f8e 00           ??          00h
        00115f8f 00           ??          00h
                            **************************************************************
```

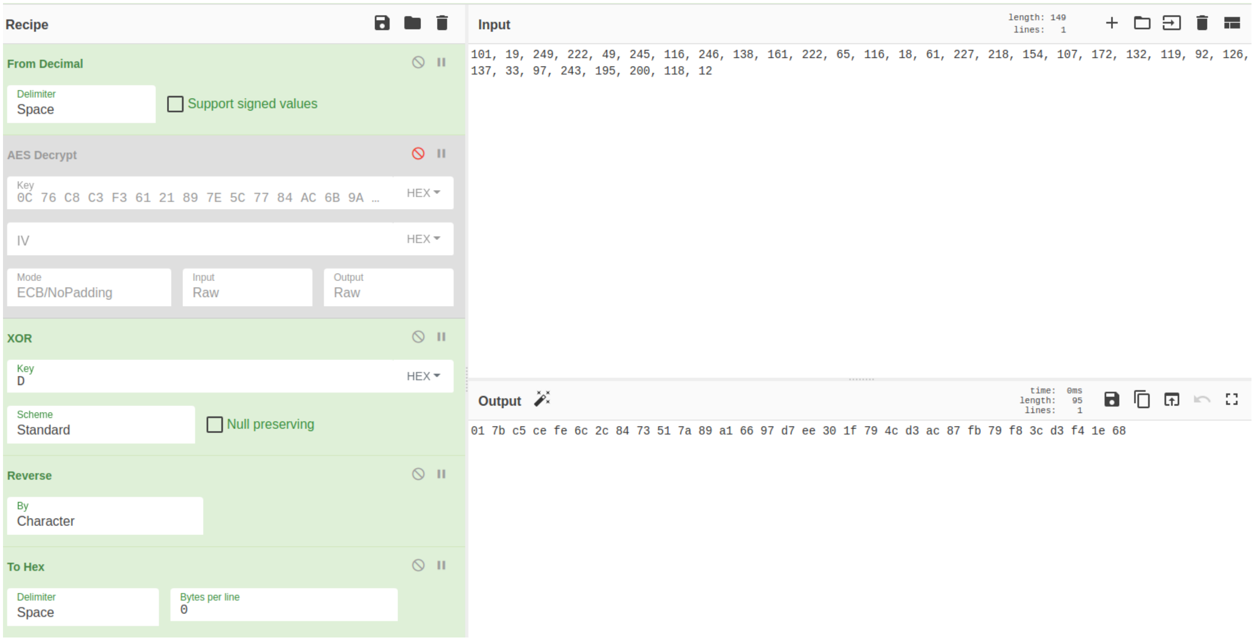
Decompile: {closure#0}<32> - (nuts-and-bolts)

```
void nuts_and_bolts::{impl#0}::xor::{closure#0}<32>({closure_env#0}<32> *param_1,u8 *v)

{
  *v = *v ^ 0xd;
  return;
}
```

A friend I made put me on a new tool called cyberchef and it is the best I have used so far for decrypting.

To begin I set up what I needed to decrypt the key. I used the first 32 bytes of the key given in the output.txt file (if you remember what I said earlier our key is 32 bytes since it is AES-256) and placed them in the input section. Also adding my XOR key “D” and adding in reversing to reverse the output.


Recipe

From Decimal
Delimiter: Space
Support signed values

AES Decrypt
Key: 0C 76 C8 C3 F3 61 21 89 7E 5C 77 84 AC 6B 9A … HEX
IV: HEX
Mode: ECB/NoPadding
Input: Raw
Output: Raw

XOR
Key: D HEX
Scheme: Standard
Null preserving

Reverse
By: Character

To Hex
Delimiter: Space
Bytes per line: 0

Input (length: 149, lines: 1)

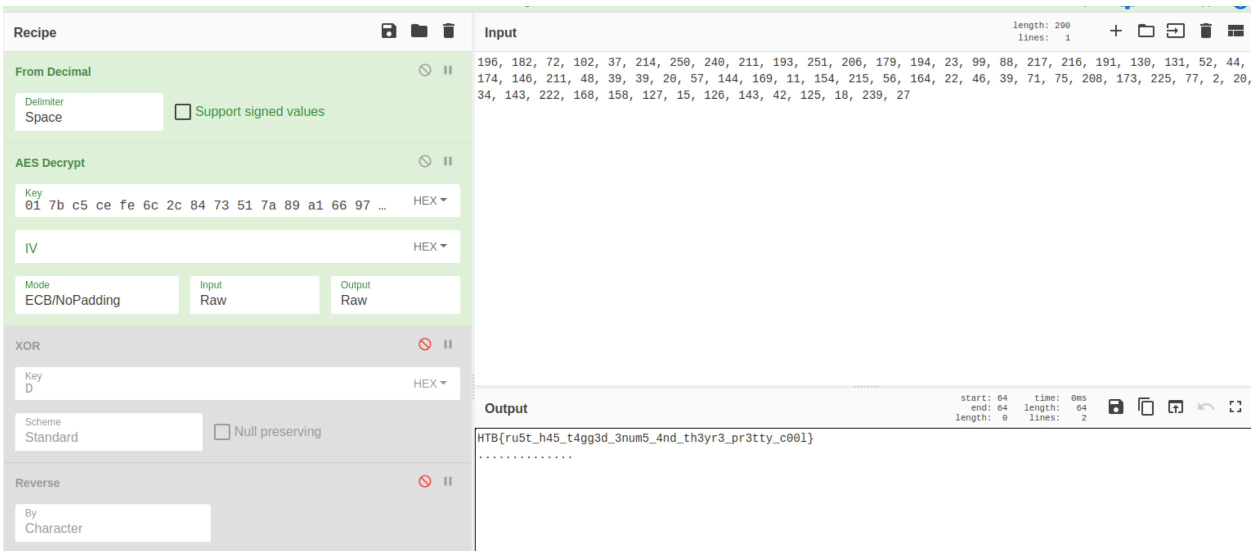
```
101, 19, 249, 222, 49, 245, 116, 246, 138, 161, 222, 65, 116, 18, 61, 227, 218, 154, 107, 172, 132, 119, 92, 126,
137, 33, 97, 243, 195, 200, 118, 12
```

Output (time: 0ms, length: 95, lines: 1)

```
01 7b c5 ce fe 6c 2c 84 73 51 7a 89 a1 66 97 d7 ee 30 1f 79 4c d3 ac 87 fb 79 f8 3c d3 f4 1e 68
```

Next I set up another recipe with cyberchef and input the 64 bytes from the end of the flag portion of the output.txt file. I put the previously decrypted key in the AES decrypt and started cooking.

Presto we have our key!



Recipe

From Decimal
Delimiter: Space
Support signed values

AES Decrypt
Key: 01 7b c5 ce fe 6c 2c 84 73 51 7a 89 a1 66 97 … HEX
IV HEX
Mode: ECB/NoPadding
Input: Raw
Output: Raw

XOR
Key: D HEX
Scheme: Standard
Null preserving

Reverse
By: Character

Input

```
length: 290
 lines:   1
```

```
196, 182, 72, 102, 37, 214, 250, 240, 211, 193, 251, 206, 179, 194, 23, 99, 88, 217, 216, 191, 130, 131, 52, 44,
174, 146, 211, 48, 39, 39, 20, 57, 144, 169, 11, 154, 215, 56, 164, 22, 46, 39, 71, 75, 208, 173, 225, 77, 2, 20,
34, 143, 222, 168, 158, 127, 15, 126, 143, 42, 125, 18, 239, 27
```

Output

```
 start: 64      time:  0ms
   end: 64    length:   64
length:  0     lines:    2
```

```
HTB{ru5t_h45_t4gg3d_3num5_4nd_th3yr3_pr3tty_c00l}
..............
```

Summary: This challenge was not as easy as it might appear. I did a lot of different combinations of the output.txt file until I got a key. The big part missing was the XOR key. You have to have a XOR key in order to decrypt XOR, but in some of these challenges they are not as easy to find. There is also another method that was briefly explained to me that involved using the bytes ahead of the flag and key “1,0,0,0,2,0,0,0” and it is supposed to be a faster/proper method.