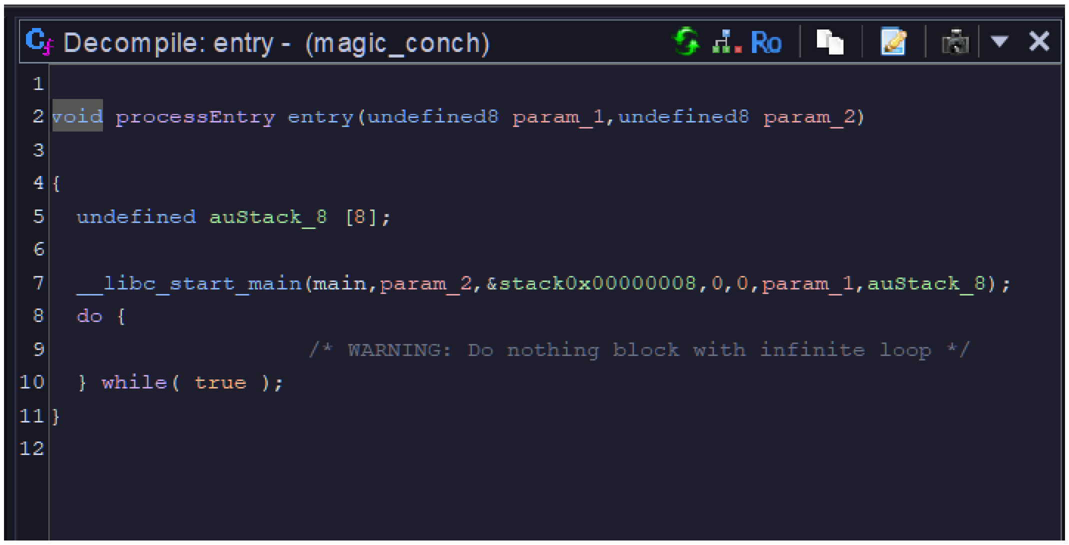
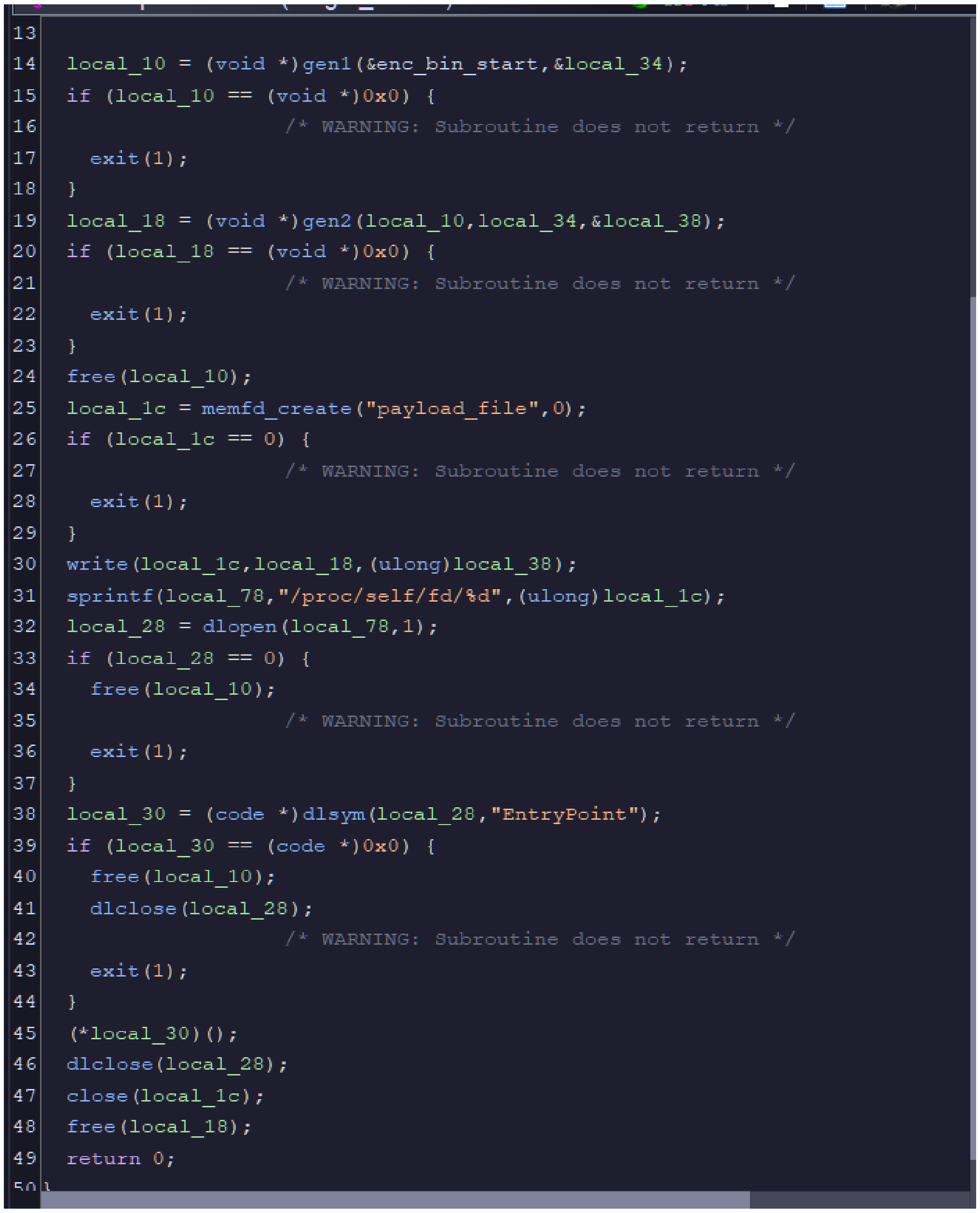
Magical Conch (rev)

A binary is given to me. Opening it in ghidra,



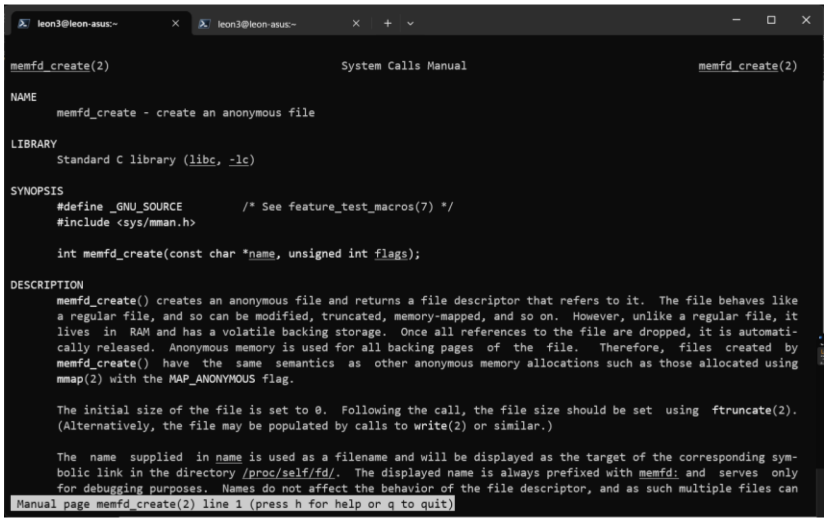
```

void processEntry entry(undefined8 param_1,undefined8 param_2)

{
  undefined auStack_8 [8];
  
  __libc_start_main(main,param_2,&stack0x00000008,0,0,param_1,auStack_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}

```

There is an entry point, which then we can use to locate the main function.



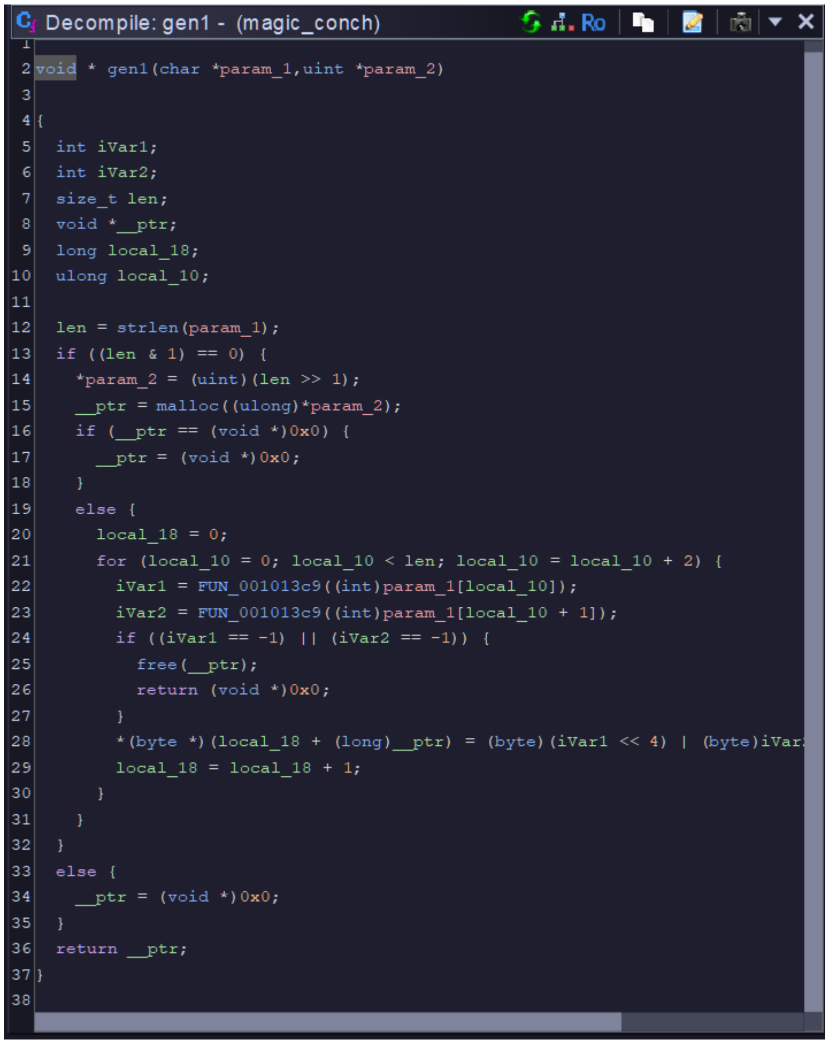
```
local_10 = (void *)gen1(&enc_bin_start,&local_34);
if (local_10 == (void *)0x0) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_18 = (void *)gen2(local_10,local_34,&local_38);
if (local_18 == (void *)0x0) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
free(local_10);
local_1c = memfd_create("payload_file",0);
if (local_1c == 0) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
write(local_1c,local_18,(ulong)local_38);
sprintf(local_78,"/proc/self/fd/%d",(ulong)local_1c);
local_28 = dlopen(local_78,1);
if (local_28 == 0) {
  free(local_10);
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_30 = (code *)dlsym(local_28,"EntryPoint");
if (local_30 == (code *)0x0) {
  free(local_10);
  dlclose(local_28);
                  /* WARNING: Subroutine does not return */
  exit(1);
}
(*local_30)();
dlclose(local_28);
close(local_1c);
free(local_18);
return 0;
}
```

Taking a initial look, it appears that the main function is loading some sort of shared library because of the dlopen call. I also noticed that there's a `write` call, which is writing to a fd created by memfd_create. Looking at manpage for memfd_create:



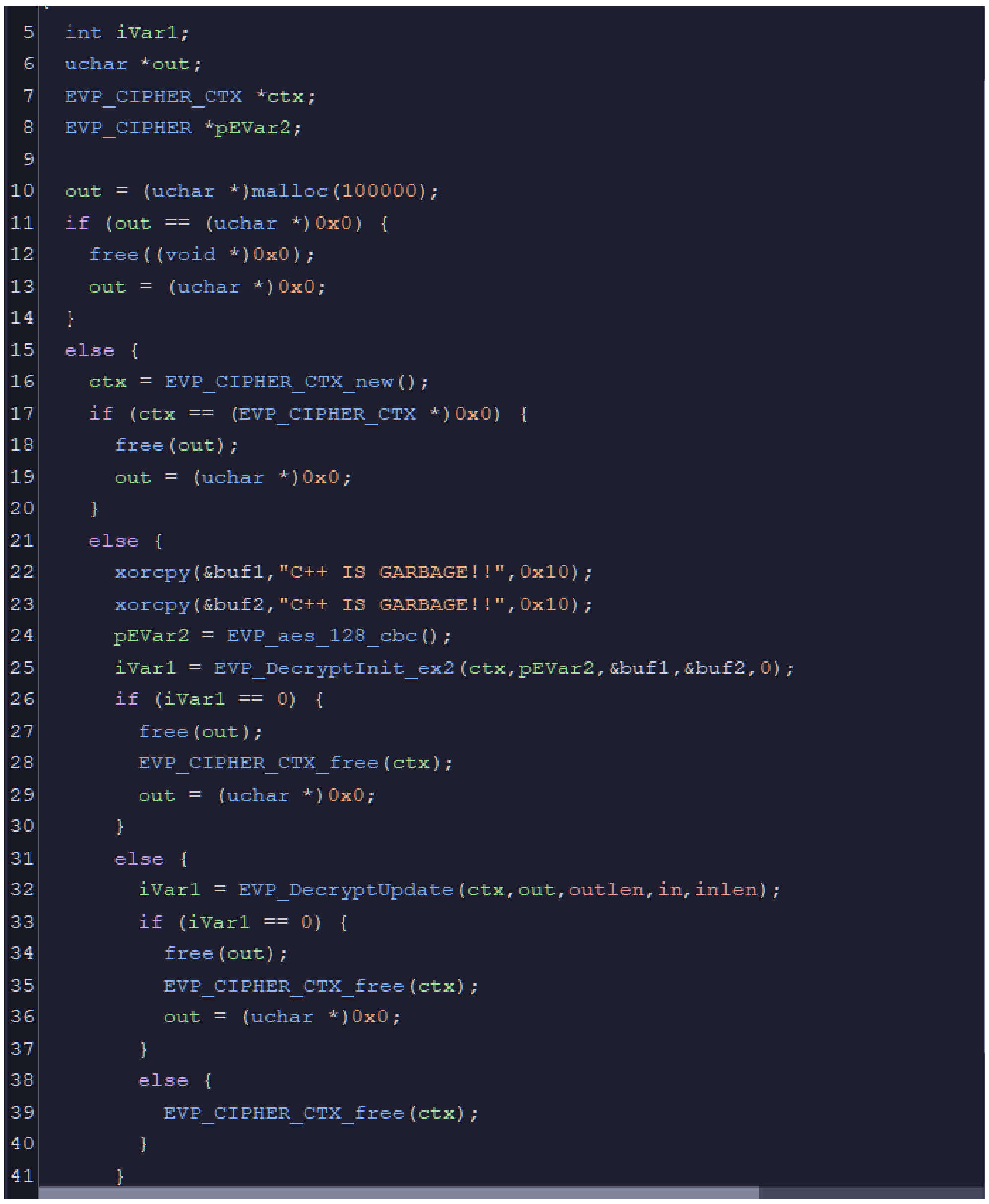
```
memfd_create(2)                     System Calls Manual                     memfd_create(2)

NAME
       memfd_create - create an anonymous file

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #define _GNU_SOURCE         /* See feature_test_macros(7) */
       #include <sys/mman.h>

       int memfd_create(const char *name, unsigned int flags);

DESCRIPTION
       memfd_create() creates an anonymous file and returns a file descriptor that refers to it.  The file behaves like
       a regular file, and so can be modified, truncated, memory-mapped, and so on.  However, unlike a regular file, it
       lives  in  RAM and has a volatile backing storage.  Once all references to the file are dropped, it is automati-
       cally released.  Anonymous memory is used for all backing pages  of  the  file.  Therefore,  files  created  by
       memfd_create()  have  the  same  semantics  as  other anonymous memory allocations such as those allocated using
       mmap(2) with the MAP_ANONYMOUS flag.

       The initial size of the file is set to 0.  Following the call, the file size should be set  using  ftruncate(2).
       (Alternatively, the file may be populated by calls to write(2) or similar.)

       The  name  supplied  in name is used as a filename and will be displayed as the target of the corresponding sym-
       bolic link in the directory /proc/self/fd/.  The displayed name is always prefixed with memfd: and  serves  only
       for debugging purposes.  Names do not affect the behavior of the file descriptor, and as such multiple files can
 Manual page memfd_create(2) line 1 (press h for help or q to quit)
```

It appears to make a file that is in memory. This is important later, now lets see what other functions are doing.



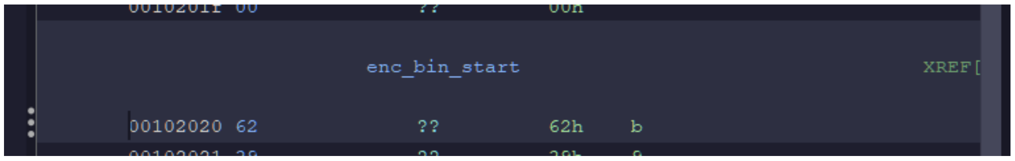
```
void * gen1(char *param_1,uint *param_2)

{
  int iVar1;
  int iVar2;
  size_t len;
  void *__ptr;
  long local_18;
  ulong local_10;

  len = strlen(param_1);
  if ((len & 1) == 0) {
    *param_2 = (uint)(len >> 1);
    __ptr = malloc((ulong)*param_2);
    if (__ptr == (void *)0x0) {
      __ptr = (void *)0x0;
    }
    else {
      local_18 = 0;
      for (local_10 = 0; local_10 < len; local_10 = local_10 + 2) {
        iVar1 = FUN_001013c9((int)param_1[local_10]);
        iVar2 = FUN_001013c9((int)param_1[local_10 + 1]);
        if ((iVar1 == -1) || (iVar2 == -1)) {
          free(__ptr);
          return (void *)0x0;
        }
        *(byte *)(local_18 + (long)__ptr) = (byte)(iVar1 << 4) | (byte)iVar
        local_18 = local_18 + 1;
      }
    }
  }
  else {
    __ptr = (void *)0x0;
  }
  return __ptr;
}
```

I named this function gen1, but I'm not sure what it does. I'm going to move on to another function gen2.



```
  int iVar1;
  uchar *out;
  EVP_CIPHER_CTX *ctx;
  EVP_CIPHER *pEVar2;

  out = (uchar *)malloc(100000);
  if (out == (uchar *)0x0) {
    free((void *)0x0);
    out = (uchar *)0x0;
  }
  else {
    ctx = EVP_CIPHER_CTX_new();
    if (ctx == (EVP_CIPHER_CTX *)0x0) {
      free(out);
      out = (uchar *)0x0;
    }
    else {
      xorcpy(&buf1,"C++ IS GARBAGE!!",0x10);
      xorcpy(&buf2,"C++ IS GARBAGE!!",0x10);
      pEVar2 = EVP_aes_128_cbc();
      iVar1 = EVP_DecryptInit_ex2(ctx,pEVar2,&buf1,&buf2,0);
      if (iVar1 == 0) {
        free(out);
        EVP_CIPHER_CTX_free(ctx);
        out = (uchar *)0x0;
      }
      else {
        iVar1 = EVP_DecryptUpdate(ctx,out,outlen,in,inlen);
        if (iVar1 == 0) {
          free(out);
          EVP_CIPHER_CTX_free(ctx);
          out = (uchar *)0x0;
        }
        else {
          EVP_CIPHER_CTX_free(ctx);
        }
      }
```

This function is pretty interesting: there are openssl crypto functions, that's interesting. Does this mean there are decryption involved? So, I started thinking about decrypting. First I was able to locate a large chunk of data in the data section of the binary.



```
                         enc_bin_start                                  XREF[
     00102020 62              ??          62h     b
```

Which I named `enc_bin_start`, this data is about 20kb long. I dumped out this data and attempted to decrypt.

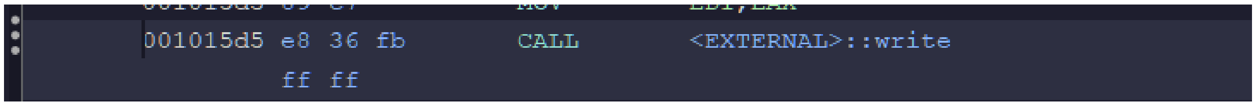
However, this was not the greatest idea:
- I needed to figure out the key, although that didn't seem to be super hard, as I discovered a function to XOR the string "C++ IS GARBAGE!!" with some predefined data.

- I spent an about an hour trying to figure out how to decrypt data using EVP and AES

This was, in fact, not the greatest idea, recall earlier that there is the write call and the memfd_create. `write(memfd,dec,(ulong)len);` This call actually writes the decrypted data to the memory region created by memfd_create. Realizing that I quickly booted up pwntools.

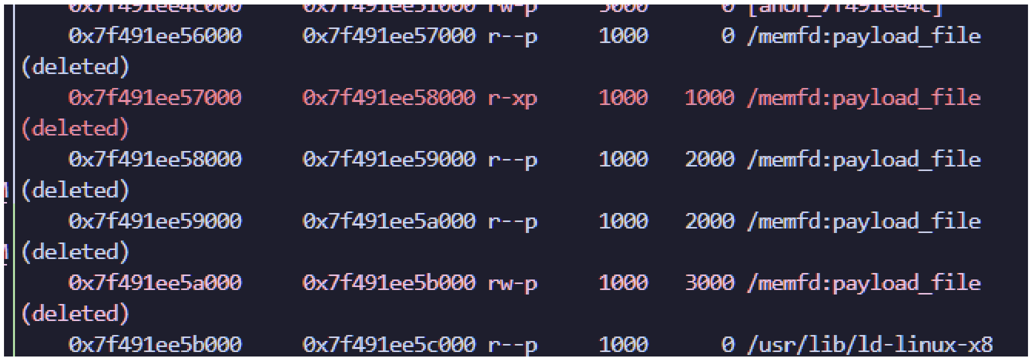
```python
from pwn import *
e = context.binary = ELF("./magic_conch")
context.terminal = ["tmux", "splitw", "-h"]

p = gdb.debug(
    context.binary.path,
    env={"FLAG": "FLAG{WIN}", "PORT": "9090"},
    gdbscript="""
              continue
              brva 0x15d5
              brva 0x1676
             """,
)
p.interactive()
```

Using this script and the `brva` command, I was able to break at the write syscall.



```
        001015d5 e8 36 fb        CALL        <EXTERNAL>::write
                 ff ff
```

Doing `vmmap` in pwndbg



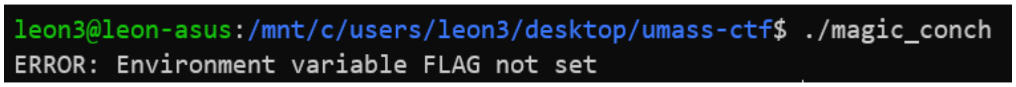
```
    0x7f491ee56000     0x7f491ee57000 r--p     1000      0 /memfd:payload_file
(deleted)
    0x7f491ee57000     0x7f491ee58000 r-xp     1000   1000 /memfd:payload_file
(deleted)
    0x7f491ee58000     0x7f491ee59000 r--p     1000   2000 /memfd:payload_file
(deleted)
    0x7f491ee59000     0x7f491ee5a000 r--p     1000   2000 /memfd:payload_file
(deleted)
    0x7f491ee5a000     0x7f491ee5b000 rw-p     1000   3000 /memfd:payload_file
(deleted)
    0x7f491ee5b000     0x7f491ee5c000 r--p     1000      0 /usr/lib/ld-linux-x8
```

I can indeed see a memory region labeled as memfd:payload_file.
I used `dump memory` gdb command to dump this decrypted shared library so I can take a closer look at it in ghidra.

Here comes the real deal.

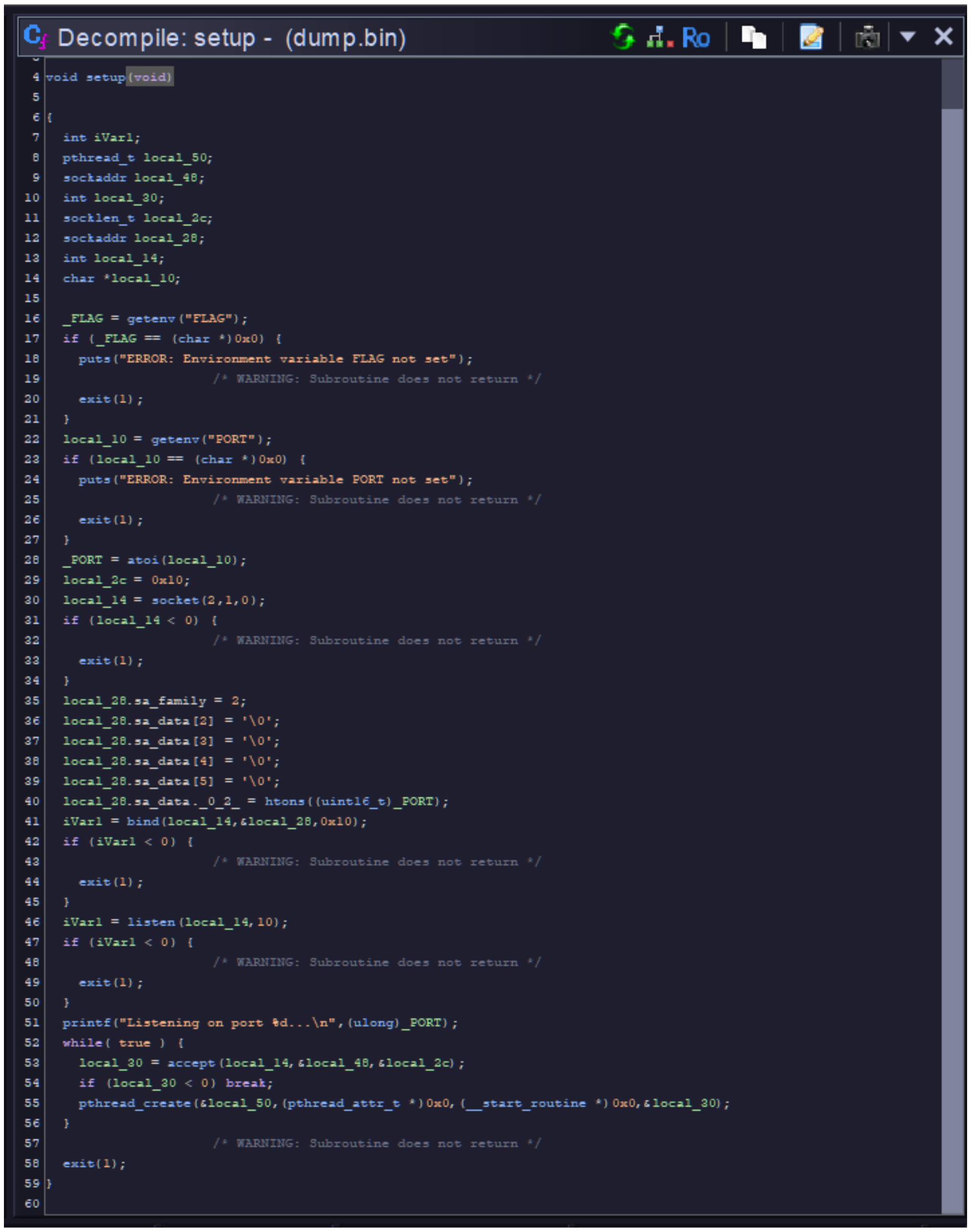
Opening the dumped ELF file in ghidra, it’s not immediately clear what is the entry point.

However, looking through the different functions, there are serval clues:



```
leon3@leon-asus:/mnt/c/users/leon3/desktop/umass-ctf$ ./magic_conch
ERROR: Environment variable FLAG not set
```

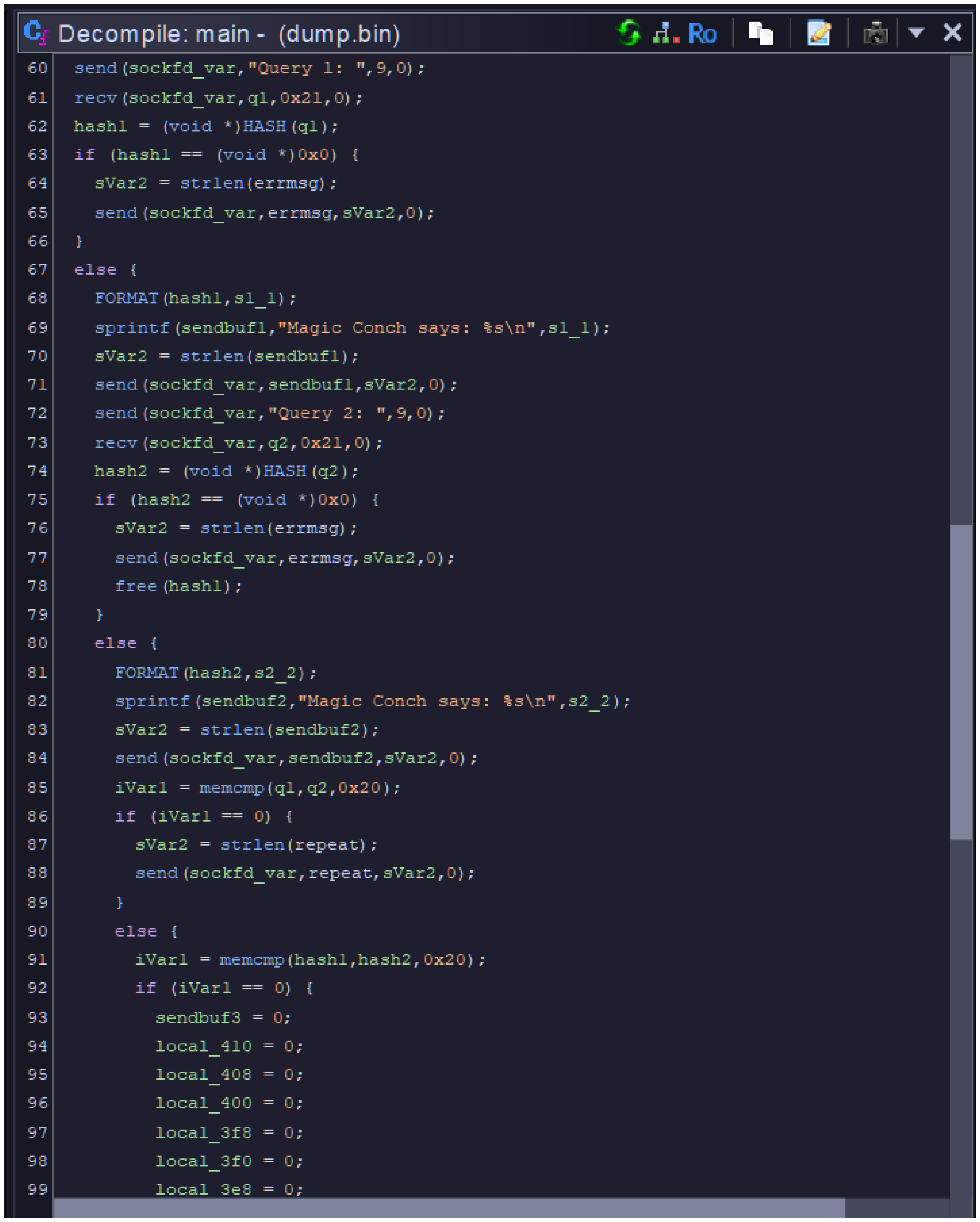
If you run the binary directly, it says that environment variables not set. Then looking through all the functions, there is one that I named `setup` which reads the environment variables and setup the sockets.



Decompile: setup - (dump.bin)

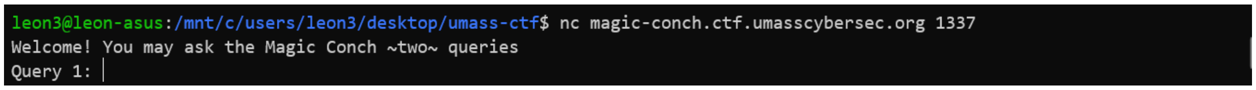
```c
void setup(void)

{
  int iVar1;
  pthread_t local_50;
  sockaddr local_48;
  int local_30;
  socklen_t local_2c;
  sockaddr local_28;
  int local_14;
  char *local_10;

  _FLAG = getenv("FLAG");
  if (_FLAG == (char *)0x0) {
    puts("ERROR: Environment variable FLAG not set");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  local_10 = getenv("PORT");
  if (local_10 == (char *)0x0) {
    puts("ERROR: Environment variable PORT not set");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  _PORT = atoi(local_10);
  local_2c = 0x10;
  local_14 = socket(2,1,0);
  if (local_14 < 0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  local_28.sa_family = 2;
  local_28.sa_data[2] = '\0';
  local_28.sa_data[3] = '\0';
  local_28.sa_data[4] = '\0';
  local_28.sa_data[5] = '\0';
  local_28.sa_data._0_2_ = htons((uint16_t)_PORT);
  iVar1 = bind(local_14,&local_28,0x10);
  if (iVar1 < 0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  iVar1 = listen(local_14,10);
  if (iVar1 < 0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  printf("Listening on port %d...\n",(ulong)_PORT);
  while( true ) {
    local_30 = accept(local_14,&local_48,&local_2c);
    if (local_30 < 0) break;
    pthread_create(&local_50,(pthread_attr_t *)0x0,(__start_routine *)0x0,&local_30);
  }
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

This function isn’t super interesting tho, let’s look at the other functions.



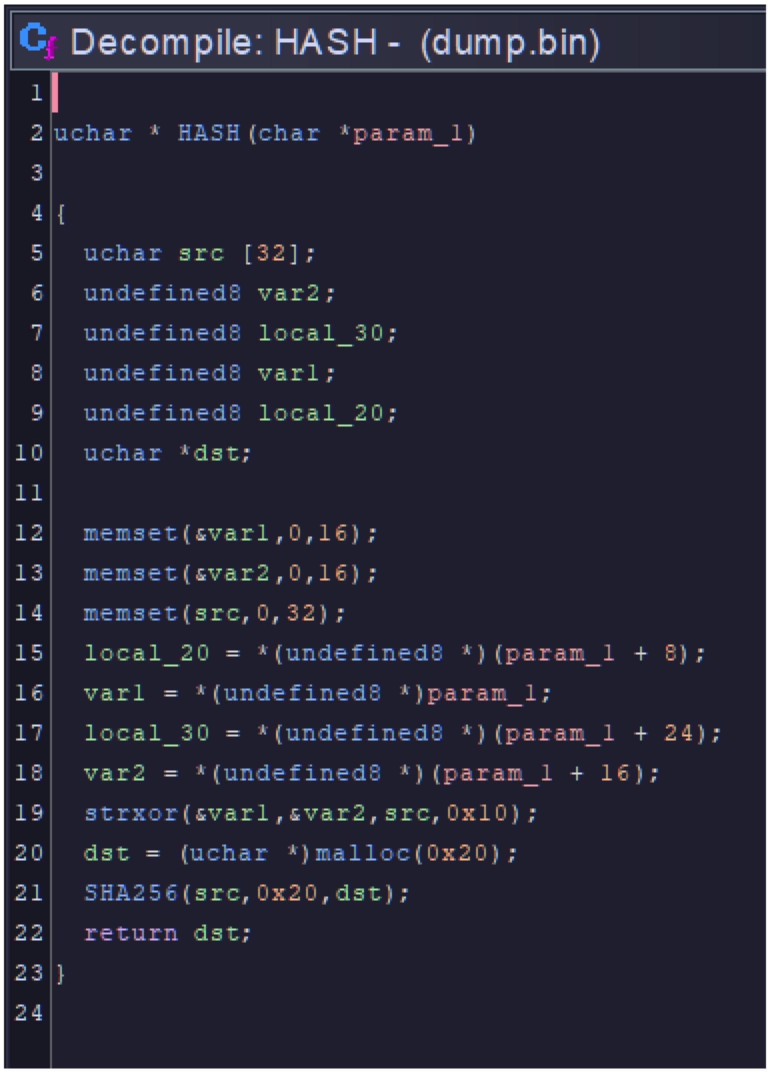
```c
send(sockfd_var,"Query 1: ",9,0);
recv(sockfd_var,q1,0x21,0);
hash1 = (void *)HASH(q1);
if (hash1 == (void *)0x0) {
  sVar2 = strlen(errmsg);
  send(sockfd_var,errmsg,sVar2,0);
}
else {
  FORMAT(hash1,s1_1);
  sprintf(sendbuf1,"Magic Conch says: %s\n",s1_1);
  sVar2 = strlen(sendbuf1);
  send(sockfd_var,sendbuf1,sVar2,0);
  send(sockfd_var,"Query 2: ",9,0);
  recv(sockfd_var,q2,0x21,0);
  hash2 = (void *)HASH(q2);
  if (hash2 == (void *)0x0) {
    sVar2 = strlen(errmsg);
    send(sockfd_var,errmsg,sVar2,0);
    free(hash1);
  }
  else {
    FORMAT(hash2,s2_2);
    sprintf(sendbuf2,"Magic Conch says: %s\n",s2_2);
    sVar2 = strlen(sendbuf2);
    send(sockfd_var,sendbuf2,sVar2,0);
    iVar1 = memcmp(q1,q2,0x20);
    if (iVar1 == 0) {
      sVar2 = strlen(repeat);
      send(sockfd_var,repeat,sVar2,0);
    }
    else {
      iVar1 = memcmp(hash1,hash2,0x20);
      if (iVar1 == 0) {
        sendbuf3 = 0;
        local_410 = 0;
        local_408 = 0;
        local_400 = 0;
        local_3f8 = 0;
        local_3f0 = 0;
        local_3e8 = 0;
```

This function I called main contains code of interest, this is because when I connect to the url, it shows exactly the output:



```
leon3@leon-asus:/mnt/c/users/leon3/desktop/umass-ctf$ nc magic-conch.ctf.umasscybersec.org 1337
Welcome! You may ask the Magic Conch ~two~ queries
Query 1:
```

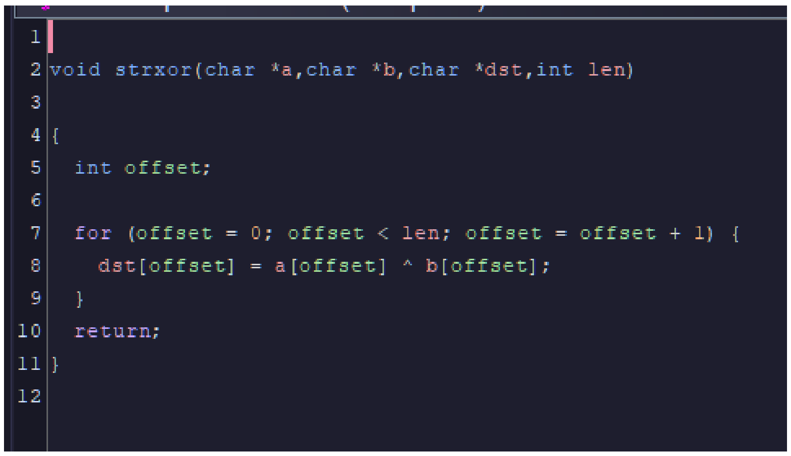
I have named the two queries q1 and q2 respectively. As seen in the code, it puts the queries into a function I called `hash`.



```c
uchar * HASH(char *param_1)

{
  uchar src [32];
  undefined8 var2;
  undefined8 local_30;
  undefined8 var1;
  undefined8 local_20;
  uchar *dst;

  memset(&var1,0,16);
  memset(&var2,0,16);
  memset(src,0,32);
  local_20 = *(undefined8 *)(param_1 + 8);
  var1 = *(undefined8 *)param_1;
  local_30 = *(undefined8 *)(param_1 + 24);
  var2 = *(undefined8 *)(param_1 + 16);
  strxor(&var1,&var2,src,0x10);
  dst = (uchar *)malloc(0x20);
  SHA256(src,0x20,dst);
  return dst;
}
```

The hash function looks like so, I have renamed variables accordingly to make it clearer.

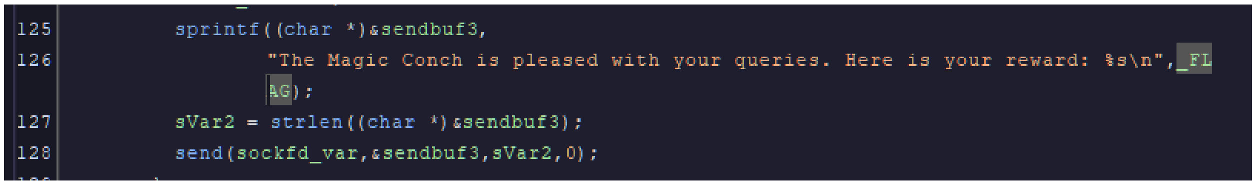
I also named this function ` strxor` , let’s look at this function:



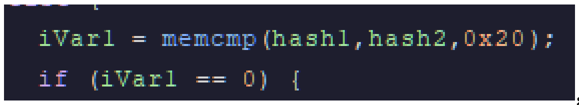
```c
void strxor(char *a,char *b,char *dst,int len)

{
  int offset;

  for (offset = 0; offset < len; offset = offset + 1) {
    dst[offset] = a[offset] ^ b[offset];
  }
  return;
}
```

Ok, looks it stores the result of XOR of the two string parts into dst. Variables renamed accordingly.

For us to reach the flag, we need to hit this sprint line:

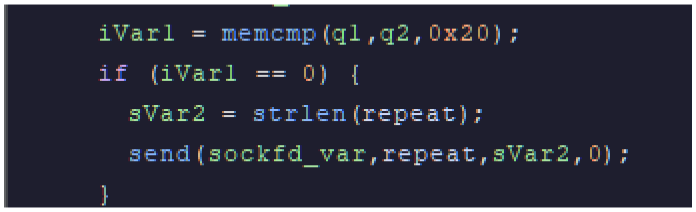


```
        sprintf((char *)&sendbuf3,
                "The Magic Conch is pleased with your queries. Here is your reward: %s\n",_FL
                AG);
        sVar2 = strlen((char *)&sendbuf3);
        send(sockfd_var,&sendbuf3,sVar2,0);
```

Which is allowed by the condition 

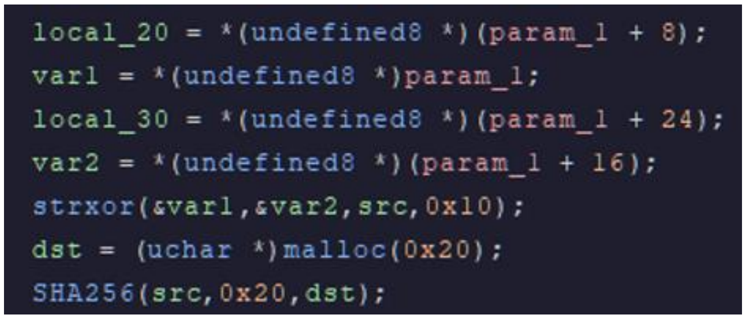
```
  iVar1 = memcmp(hash1,hash2,0x20);
  if (iVar1 == 0) {
```

,, meaning that two of our hash must be the same, but the two inputs must be different, by this comparison here.



```
    iVar1 = memcmp(q1,q2,0x20);
    if (iVar1 == 0) {
      sVar2 = strlen(repeat);
      send(sockfd_var,repeat,sVar2,0);
    }
```

Using this logic, this means that we must be able to put in two queries that have the same resulting hash, hash collision? Practically impossible with SHA256. But is there a specific way input, which allows it to be modified by the `strxor` function, so that to the SHA256 hash function, it appears the same? Okay, maybe.



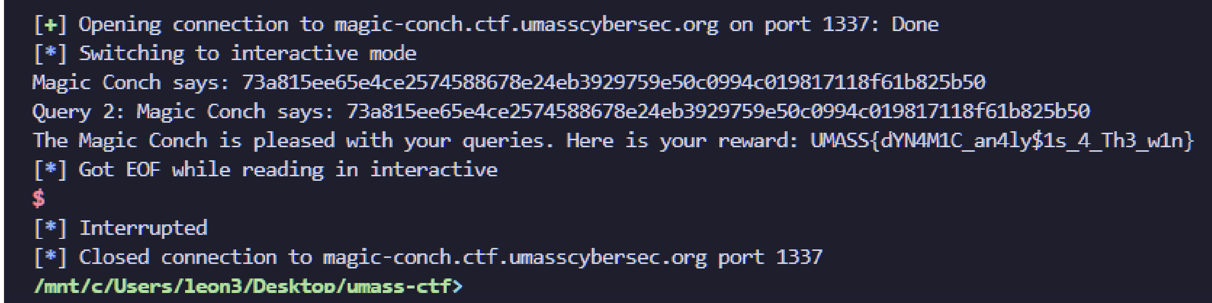
```
  local_20 = *(undefined8 *)(param_1 + 8);
  var1 = *(undefined8 *)param_1;
  local_30 = *(undefined8 *)(param_1 + 24);
  var2 = *(undefined8 *)(param_1 + 16);
  strxor(&var1,&var2,src,0x10);
  dst = (uchar *)malloc(0x20);
  SHA256(src,0x20,dst);
```

Looking closer at this, the two 16 bytes part of the 32 bytes input is xor'd together, so we can craft a payload like

16 0xFF and then 16 0x00

16 0x00 and then 16 0xFF
These two queries, when XOR'd will just produce 16 FF bytes, for the 32 byte input buffer of

SHA256. However, this should allow us to bypass the memcmp check that checks for the same inputs, but resulting in the same hash value.



```
[+] Opening connection to magic-conch.ctf.umasscybersec.org on port 1337: Done
[*] Switching to interactive mode
Magic Conch says: 73a815ee65e4ce2574588678e24eb3929759e50c0994c019817118f61b825b50
Query 2: Magic Conch says: 73a815ee65e4ce2574588678e24eb3929759e50c0994c019817118f61b825b50
The Magic Conch is pleased with your queries. Here is your reward: UMASS{dYN4M1C_an4ly$1s_4_Th3_w1n}
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to magic-conch.ctf.umasscybersec.org port 1337
/mnt/c/Users/leon3/Desktop/umass-ctf>
```

Win!

Complete socket solve script:

```python
from pwn import *

r = remote('magic-conch.ctf.umasscybersec.org',1337)
r.recvuntil(b"Query 1")
r.recvuntil(b" ")

r.send(b"\x00"*16+b"\xFF"*16)
r.send(b"\xFF"*16+b"\x00"*16)

r.interactive()
```