Challenge: Vault breaker

Challenge Description :

Money maker, Big Boy Bonnie has a crew of his own to do his dirty jobs. In a tiny little planet a few lightyears away, a custom-made vault has been found by his crew. Something is hidden inside it, can you find out the way it works and bring it to Bonnie?

Context :

- You are given a compiled binary file that we need to decompile, during decompiling the file you need to analyze the binary code and find an exploit towards this file, so you can exploit the remote version.
- To solve the challenge, you need to exploit the fact that DataStore can be interpreted both as a string and as an integer, and they share the same memory..

Flag :

- When we run the program, it gives us two options. The first option lets us create a new encryption key using random data from /dev/urandom.

```
void main() {
  long option;

  setup();
  banner();
  key_gen();
  fprintf(stdout, "%s\n[+] Random secure encryption key has been generated!\n%s", &DAT_00103142, &DAT_001012f8);
  fflush(stdout);

  while (true) {
    while (true) {
      printf(&DAT_00105160, &DAT_001012f8);
      option = read_num();

      if (option != 1) break;

      new_key_gen();
    }

    if (option != 2) break;

    secure_password();
  }

  printf("%s\n[-] Invalid option, exiting..\n", &DAT_00101300);
  exit(0x45);
}
```

- However, there's a big mistake in how the new key is handled. The program uses strcpy to copy the new key into random_key, but strcpy treats the key as a string that ends with a null byte.

```
void new_key_gen() {
  // Initialization of variables
  int fd;
  FILE *__stream;
  long in_FS_OFFSET;
  ulong i;
  ulong length;
  char new_key[40];
  long canary;

  canary = *(long *) (in_FS_OFFSET + 0x28);
  i = 0;
  length = 0x22;
  __stream = fopen("/dev/urandom", "rb");

  // Error handling if fopen fails
  if (__stream == (FILE *) 0x0) {
    fprintf(stdout, "\n%sError opening /dev/urandom, exiting..\n", &DAT_00101300);
    exit(0x15);
  }

  // Loop to ensure length is within valid bounds
  while (0x1f < length) {
    printf("\n[*] Length of new password (0-%d): ", 31);
    length = read_num();
  }

  memset(new_key, 0, 32);  // Initialize new_key buffer with null bytes
  fd = fileno(__stream);
  read(fd, new_key, length);

  // Copy new_key into random_key using strcpy
  strcpy(random_key, new_key);

  fclose(__stream);
  printf("\n%s[+] New key has been generated successfully!\n%s", &DAT_00103142, &DAT_001012f8);

  if (canary != *(long *) (in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }

  return;
}
```

- If the new key has any null bytes, it can cut off the rest of the key. This mistake can lead to random_key being filled with null bytes if the new key is shorter than the old one.

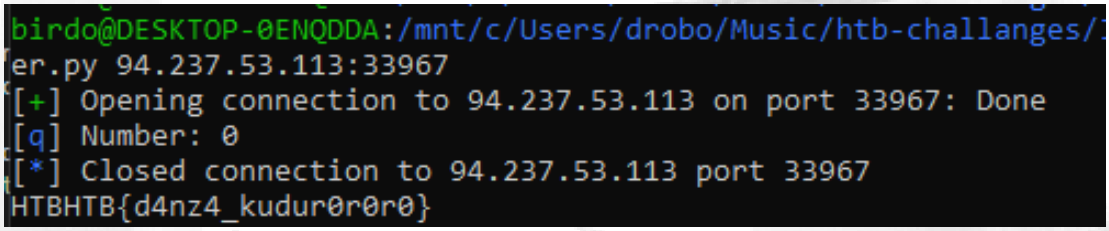
- If we choose the second option, "Secure the Vault," the program reads a password from flag.txt and encrypts it using a simple XOR with random_key.

```
void secure_password() {
  char *__buf;
  int __fd;
  ulong uVar1;
  size_t sVar2;
  long in_FS_OFFSET;
  char acStack136[24];
  undefined8 uStack112;
  int i;
  int local_64;
  char *local_60;
  undefined8 local_58;
  char *flag;
  FILE *fp;
  undefined8 canary;

  canary = *(undefined8 *) (in_FS_OFFSET + 0x28);
  uStack112 = 0x100c26;
  puts("\x1b[1;34m");
  uStack112 = 0x100c4c;
  printf(&DAT_00101308, &DAT_001012f8, &DAT_00101300, &DAT_001012f8);
  local_60 = &DAT_00101330;
  local_64 = 0x17;
  local_58 = 0x16;
  flag = acStack136;
  memset(acStack136, 0, 0x17);
  fp = fopen("flag.txt", "rb");
  __buf = flag;

  if (fp == (FILE *) 0x0) {
    fprintf(stderr, "\n%s[-] Error opening flag.txt, contact an Administrator..\n", &DAT_00101300);
    exit(0x15);
  }

  sVar2 = (size_t) local_64;
  __fd = fileno(fp);
  read(__fd, __buf, sVar2);
  fclose(fp);
  puts(local_60);
  fwrite("\nMaster password for Vault: ", 1, 0x1c, stdout);

  // XOR encryption of the flag with random_key
  i = 0;
  while (true) {
    uVar1 = (ulong) i;
    sVar2 = strlen(flag);

    if (sVar2 <= uVar1) break;

    putchar((int) (char) (random_key[i] ^ flag[i]));
    i = i + 1;
  }

  puts("\n");
  exit(0x1b39);
}
```

- Because of the bug in new_key_gen, an attacker can change random_key to be all null bytes, making the encryption useless.
- This allows the attacker to get the plain text password from flag.txt without needing to decrypt it using the flaw in how strcpy works and the XOR method.

- Basically the new_key_gen uses strcpy to copy new_key to random_key.If new_key contains null bytes, strcpy will stop copying prematurely, potentially leaving random_key filled with null bytes. Providing a length to a new_key_gen that starts at 31 characters and decreases.
- Eventually, when new_key has a length of 0, random_key will be completely nullified due to strcpy stopping at the first null byte encountered.
- The python script to complete this will be linked, as it's on my github, you can either run it locally side by side with the compiled Binary file as well as sending the payload to the hosted instance of the file.

  [Link]



- The Flag given is : HTB{d4nz4_kudur0r0r0}