# KiNS CTF 2023 writeup

## -- Kategori Nettverkstrafikk

**Dump**
Åpner vi fila i Wireshark, så ser vi at den inneholder fire ICMP packets. Det er ikke mye å jobbe med. Hvis vi sjekker file comments (Wireshark – Statistics – Capture File Properties) så ser vi en base64 enkodet streng:

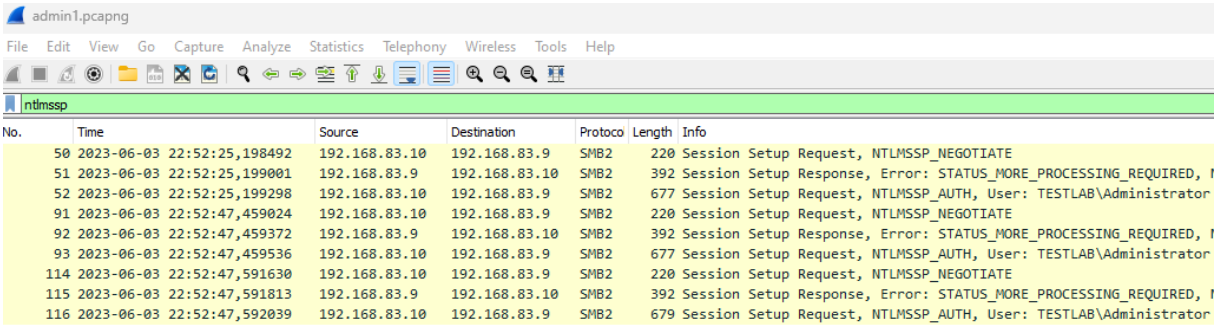

Dekoder vi strengen så finner vi flagget.

Rett svar er ctf(kommentar).

**Korrupt**
I denne oppgaven skal vi fikse en korrupt capturefil og avdekke flagget. Filstørrelsen indikerer at capturefilen har innhold, men dersom vi åpner filen i Wireshark ser det ut til at den er tom. Hvis vi laster opp filen i onlinetjenesten pcapfix, så retter den feilen, og vi kan laste ned en fil som lar seg åpne i Wireshark. Flagget er delt opp i flere chuncks, og kan settes sammen ved å se over de enkelte framene i fila.

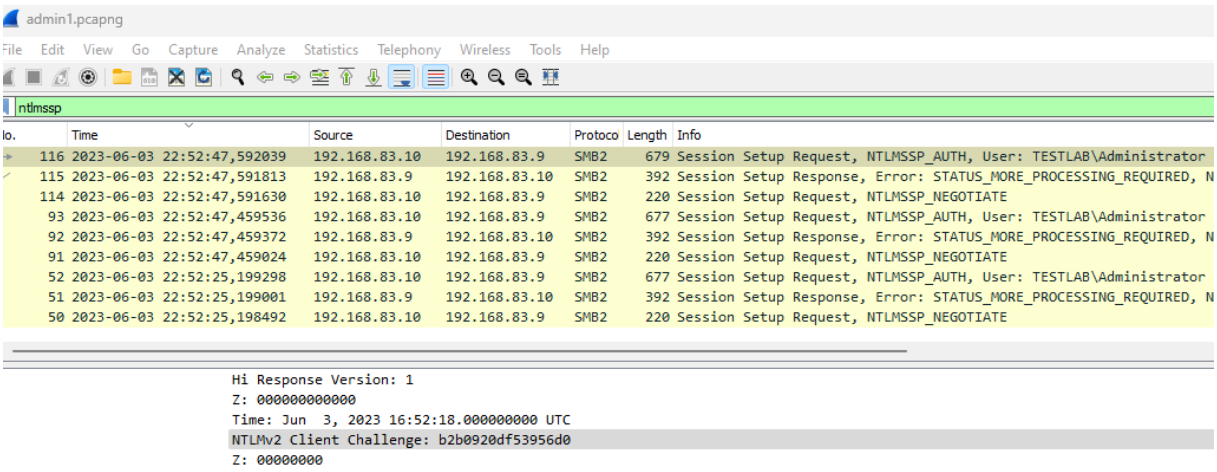
Rett svar er ctf(chopping!).

**Admin 1**
I denne oppgaven skal vi finne autentiseringsinformasjonen for en administrator konto, og avdekke passordet til kontoen. La oss starte Wireshark og filtrere på ntlmssp:




Her ser vi en del autentiseringsforsøk for kontoen TESTLAB\Administrator.

Utforsker vi den første pakken, så ser vi en NTLMv2 Client Challenge:




Domenet bruker altså NTLMv2 som autentiseringsprotokoll.

Vi må dermed finne alle relevante hasher og sette de sammen i et format som vi kan mate inn i Hashcat. Korrekt format for NTLMv2 er:

> username::domain:ntlm server challenge:ntlm response

Challenge verdien finner vi i pakke nr 51
Response verdien finner vi i pakke nr 52

Setter vi alle verdiene sammen, så får vi følgende tekststreng:

> Administrator::TESTLAB: 437b82285f05d53c:
> 9638b67abf1d806add524d4b922c331f01010000000000000000da2c13b96d901abe68c9b044c71a5000
> 00000200080037004f004700560001001e00570049004e002d004b00530054003300460033004600510
> 0470042004d0004003400570049004e002d004b00530054003300460033004600510047004200 4d0
> 02e0037004f00470056002e004c004f00430041004c000300140037004f00470056002e004c004f00430
> 041004c000500140037004f00470056002e004c004f00430041004c0007000800000da2c13b96d90106
> 0004000200000008003000300000000000000000000000000300000b0ef5345727a1c379cc18beefdd10
> ac98afe1a952cadcf8459bc0ce9e261ed020a001000000000000000000000000000000000009000c006
> 3006900660073002f00310000000000000000000000000

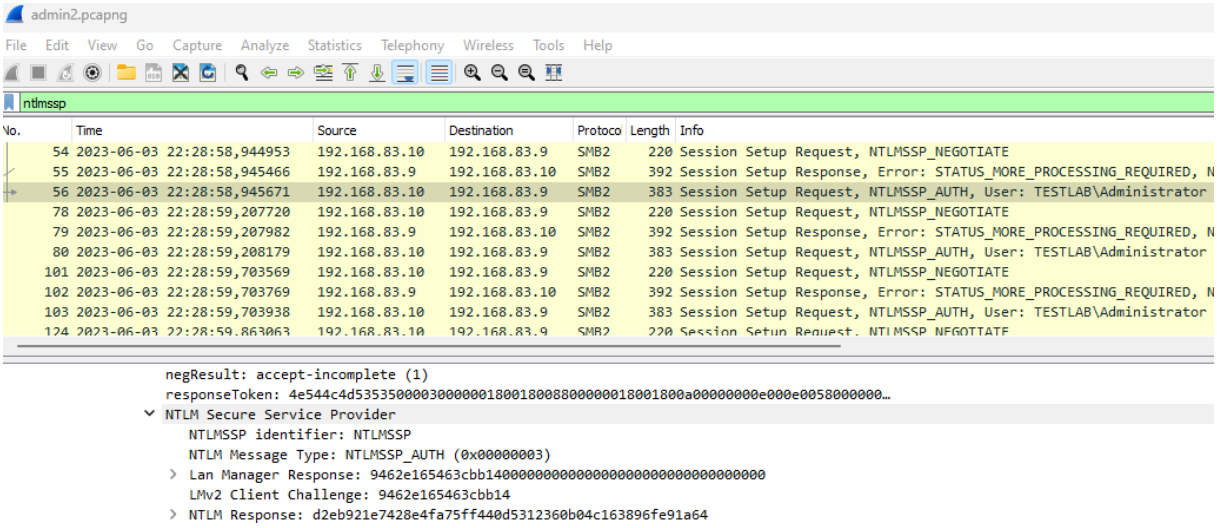
Legger vi tekststrengen i en tekstfil, så kan vi cracke hashen med Hashcat.
Bruk hash mode -m 5600. Suksess avhenger av ordlisten og det regelsettet du bruker. Det bør holde med regelsettet best64.rule.

Rett svar er ctf(Korona!)

**Admin 2**

Denne oppgaven ser ganske lik ut som Admin 1, men her får vi et hint om domenet bruker gamle autentiseringsprotokoller. Vi kan filtrere på ntlmssp, og her ser vi autentiseringsforespørsler som tilsynelatende ser lik ut som for Admin 1.

Hvis vi utforsker disse pakkene, så finner vi Lan Manager Response verdier, noe som indikerer at autentiseringsprotokollen er NetNTLMv1:




Vi må finne alle relevante hasher og sette de sammen i et format som vi kan mate inn i Hashcat.

Korrekt format for NTLMv1 er:

> username::domain:LM Response:NTLM Response:NTLM Server Challenge

LM Response finner vi i pakke nr 56
NTLM Response finner vi i pakke nr 56
NTLM Server Challenge finner vi i pakke nr 55

Setter vi dette sammen, så får vi følgende tekststreng:

> ADMINISTRATOR::TESTLAB:9462E165463CBB140000000000000000000000000000000:D2EB921E7428E4FA75FF440D5312360B04C163896FE91A64:429DCFA3392A3FB9

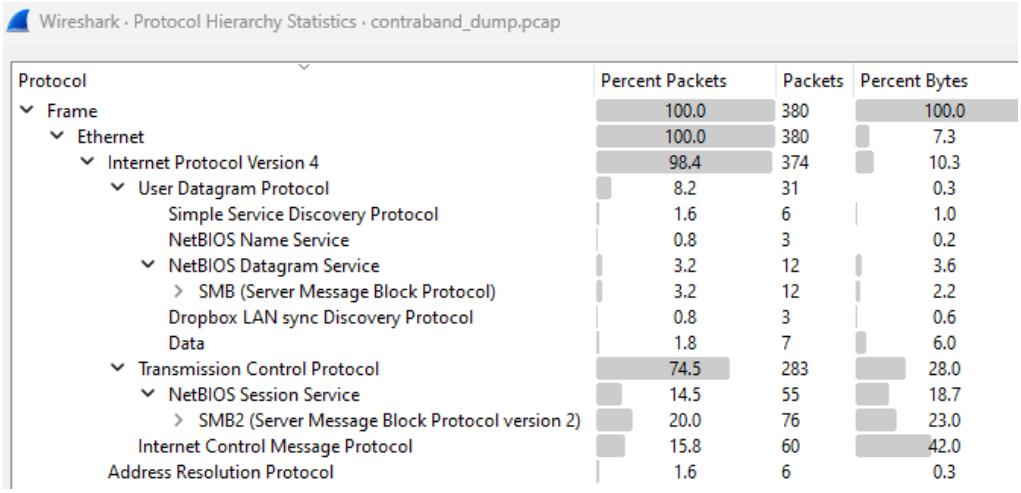
Legger vi tekststrengen i en tekstfil, så kan vi cracke hashen med Hashcat.
Bruk hash mode -m 5500. Suksess avhenger av ordlisten og det regelsettet du bruker. Det bør holde med regelsettet best64.rule.

NB! Her er det en del oppskrifter på Internett som blant annet handler om å konvertere til DES. Det er som vi ser ikke nødvendig.

Rett svar er ctf(Bitcoin!).

**Contraband**

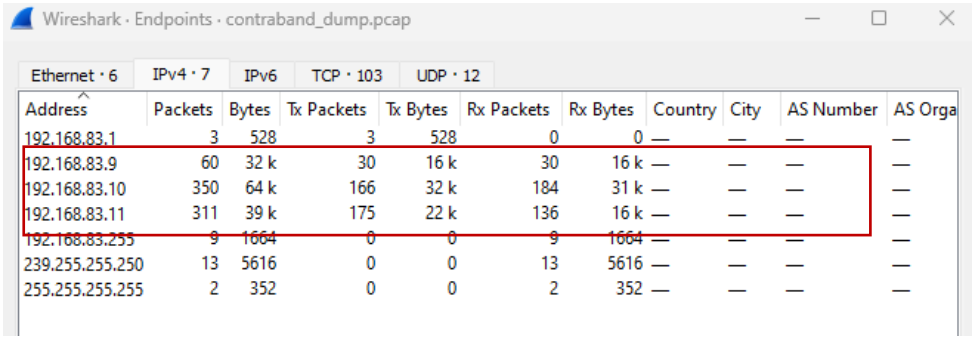
I denne oppgaven skal vi analysere en dumpfil med nettverkstrafikk. Vi kan først åpne filen i Wireshark og se på Statistics – Protocol Hierarchy.




| Protocol | Percent Packets | Packets | Percent Bytes |
|---|---|---|---|
| Frame | 100.0 | 380 | 100.0 |
| Ethernet | 100.0 | 380 | 7.3 |
| Internet Protocol Version 4 | 98.4 | 374 | 10.3 |
| User Datagram Protocol | 8.2 | 31 | 0.3 |
| Simple Service Discovery Protocol | 1.6 | 6 | 1.0 |
| NetBIOS Name Service | 0.8 | 3 | 0.2 |
| NetBIOS Datagram Service | 3.2 | 12 | 3.6 |
| SMB (Server Message Block Protocol) | 3.2 | 12 | 2.2 |
| Dropbox LAN sync Discovery Protocol | 0.8 | 3 | 0.6 |
| Data | 1.8 | 7 | 6.0 |
| Transmission Control Protocol | 74.5 | 283 | 28.0 |
| NetBIOS Session Service | 14.5 | 55 | 18.7 |
| SMB2 (Server Message Block Protocol version 2) | 20.0 | 76 | 23.0 |
| Internet Control Message Protocol | 15.8 | 60 | 42.0 |
| Address Resolution Protocol | 1.6 | 6 | 0.3 |

Her ser vi en rekke protokoller i bruk, blant annet SMB, SMB2, ICMP og ARP.

Fortsetter vi med å se på Statistics – Endpoints, så ser vi de aktive maskinene. Her er det tre maskiner som vi bør utforske:




| Address | Packets | Bytes | Tx Packets | Tx Bytes | Rx Packets | Rx Bytes | Country | City | AS Number | AS Orga |
|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.83.1 | 3 | 528 | 3 | 528 | 0 | 0 | — | — | — | — |
| 192.168.83.9 | 60 | 32 k | 30 | 16 k | 30 | 16 k | — | — | — | — |
| 192.168.83.10 | 350 | 64 k | 166 | 32 k | 184 | 31 k | — | — | — | — |
| 192.168.83.11 | 311 | 39 k | 175 | 22 k | 136 | 16 k | — | — | — | — |
| 192.168.83.255 | 9 | 1664 | 0 | 0 | 9 | 1664 | — | — | — | — |
| 239.255.255.250 | 13 | 5616 | 0 | 0 | 13 | 5616 | — | — | — | — |
| 255.255.255.255 | 2 | 352 | 0 | 0 | 2 | 352 | — | — | — | — |

I denne oppgaven er mistanken at en trusselaktør har infisert domenekontrolleren, og er i ferd med å overføre filer til domenekontrolleren for fremtidig eksfiltrasjon. Vi trenger følgelig å vite hvilken av disse maskinene som er domenekontrolleren. Det kan vi finne ut ved for eksempel sjekke SMB-trafikk og lete etter host names. Vi kan også sjekke etter hvilke maskiner som kommuniserer gjennom port 389/tcp, da den porten brukes av domenekontrollere for å svare på LDAP-forespørsler.

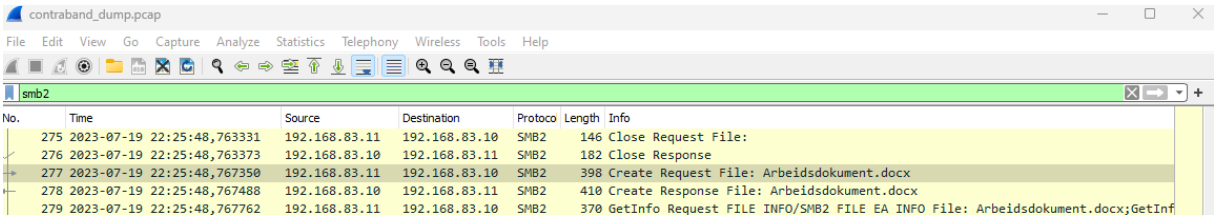
Etter litt utforsking får vi oversikt:

| IP-adresse | Host name |
|---|---|
| 192.168.83.9 | ? |
| 192.168.83.10 | DC01 |
| 192.168.83.11 | KLIENT1 |

192.168.83.9 sender bare ICMP-pakker til DC01, så vi vet ikke host name for den maskinen.

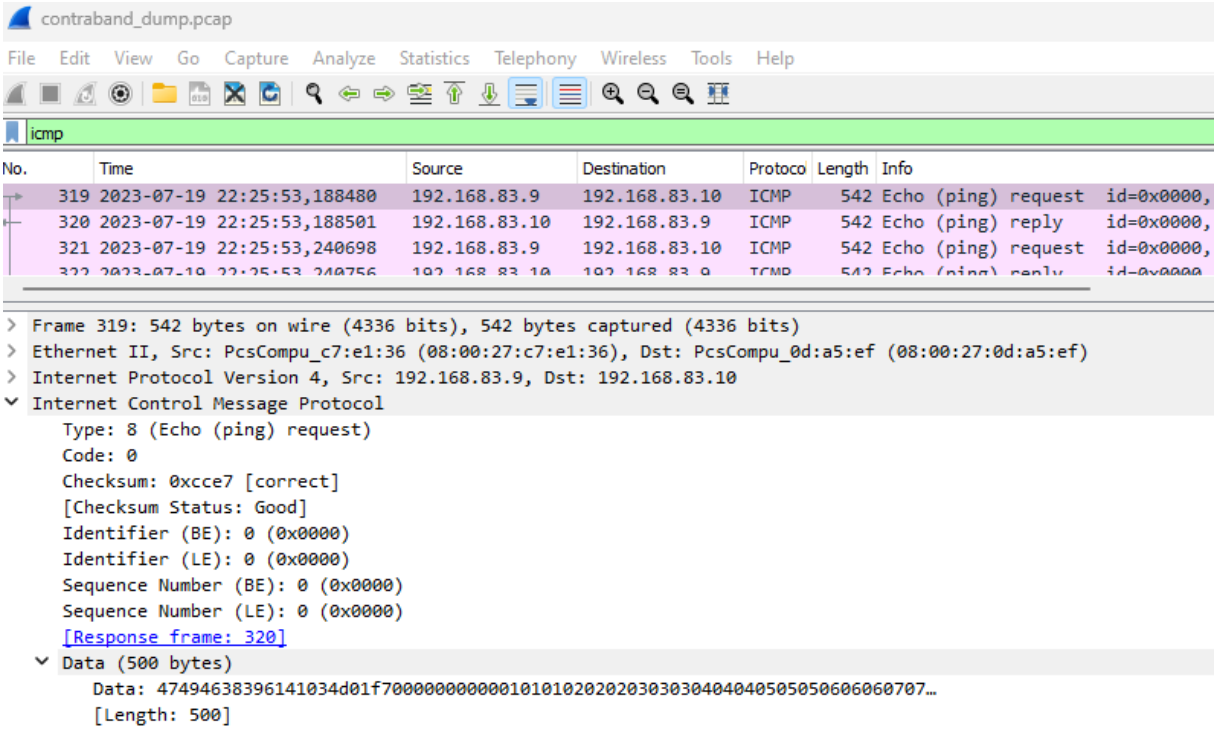
Etter mer analyse kan vi dedusere at det er to interessante hendelser i dumpfilen.

Den første er overføring av et arbeidsdokument.




Men her ser vi at det er KLIENT1 som requester filen på DC01, så DC01 har sannsynligvis et file share med dokumenter. Det er altså ikke filer som overføres *til* DC01, men som hentes *fra*. Dermed er denne hendelsen sannsynligvis et feilspor.

Den andre hendelsen er et sett med ICMP pakker som sendes fra 192.168.83.9 til DC01. Hvis vi utforsker en ICMP pakke, så ser vi at den inneholder 500 bytes med data. Det gjør faktisk alle ICMP pakkene. Det er ikke normalt med så store payloads.




At ICMP pakkene sendes fra en maskin i nettverket som kun sender ICMP trafikk er også mistenkelig, så her er vi sannsynligvis inne på en mulig filoverføring.

Vi bør forsøke å ekstrahere alle ICMP payloads og sette de sammen til en fil.

Her er et python skript som kan brukes:

```
from scapy.all import *

pcap_file = 'contraband_dump.pcap'

# Open pcap file
packets = rdpcap(pcap_file)

# Filter ICMP packets
icmp_packets = [pkt for pkt in packets if ICMP in pkt]

# Sort ICMP packets by sequence number
icmp_packets.sort(key=lambda pkt: pkt[ICMP].seq)

# Extract data from ICMP packets
data = b''
for pkt in icmp_packets:
    # Get the payload data directly from the Raw layer
    data += bytes(pkt[Raw])

# Write data to file
with open('extracted_flag.gif', 'wb') as f:
    f.write(data)
```

Skriptet setter sammen alle payloads til en fil, og gir den navn extracted_flag.gif. Filen kan åpnes og flagget vises. Trusselaktøren har altså overført filen som ICMP payloads.

Rett svar er ctf(covert_operation_23).

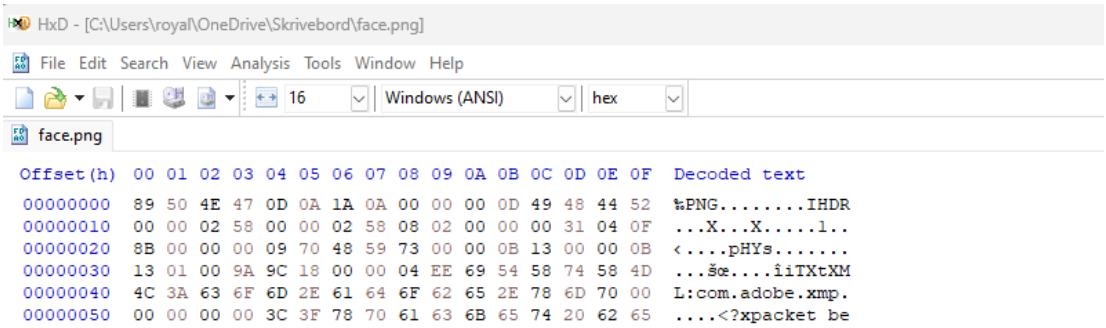
## -- Kategori Forensic

### Stego

Bildet er et portrettbilde (rendert av MidJourney btw) på 600 x 600 pixler (385 kB). Binwalk finneret par zlib filer i bildet. Det er som forventet når vi kjører binwalk på et png bilde, siden png bruker zlib compression for å redusere bildestørrelsen.

```
└─$ binwalk -e face.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 600 x 600, 8-bit/color RGB, non-interlaced
1336          0x538           Zlib compressed data, default compression
391422        0x5F8FE         Zlib compressed data, default compression
```

Hvis vi åpner filen i en hexeditor, så ser vi file headeren til png filen:




File headeren til png er 89 50 4E 47 0D 0A 1A 0A, og denne ser vi ovenfor.

File footeren til png er 49 45 4E 44 AE 42 60 82 etterfulgt av checksummen AE 42 60 82. La oss søke etter file footeren:

```
0005F1D0  F0 FF 01 9C A7 45 DD 86 B7 3B D6 00 00 00 00 49  ðÿ.œ§EÝ†·;Ö....I
0005F1E0  45 4E 44 AE 42 60 82 00 00 00 00 0D 0A 1A 0A 00  END®B`,.........
0005F1F0  00 00 0D 49 48 44 52 00 00 03 41 00 00 01 4D 08  ...IHDR...A...M.
0005F200  00 00 00 00 E3 CB 3B 57 00 00 00 09 70 48 59 73  ....ãË;W....pHYs
0005F210  00 00 2E 23 00 00 2E 23 01 78 A5 3F 76 00 00 06  ...#...#.x¥?v...
0005F220  CD 69 54 58 74 58 4D 4C 3A 63 6F 6D 2E 61 64 6F  ÍiTXtXML:com.ado
0005F230  62 65 2E 78 6D 70 00 00 00 00 00 3C 3F 78 70 61  be.xmp.....<?xpa
```

Her ser vi file footeren, men så ser vi at verdiene etter file footeren er nullet ut. Det kan være file headeren til en embedded fil. Hvis vi går til slutten av filen så ser vi en til png file footer signatur, så her er det sannsynligvis png file headeren som mangler. La oss legge inn signaturen:

```
0005F1C0  DF 1A 77 1D 9A A4 BC 8E 3F 93 1E 78 26 84 B2 05  ß.w.š¤¼Ž?".x&„².
0005F1D0  F0 FF 01 9C A7 45 DD 86 B7 3B D6 00 00 00 00 49  ðÿ.œ§EÝ†·;Ö....I
0005F1E0  45 4E 44 AE 42 60 82 89 50 4E 47 0D 0A 1A 0A 00  END®B`‰PNG.....
0005F1F0  00 00 0D 49 48 44 52 00 00 03 41 00 00 01 4D 08  ...IHDR...A...M.
0005F200  00 00 00 00 E3 CB 3B 57 00 00 00 09 70 48 59 73  ....ãË;W....pHYs
0005F210  00 00 2E 23 00 00 2E 23 01 78 A5 3F 76 00 00 06  ...#...#.x¥?v...
0005F220  CD 69 54 58 74 58 4D 4C 3A 63 6F 6D 2E 61 64 6F  ÍiTXtXML:com.ado
0005F230  62 65 2E 78 6D 70 00 00 00 00 00 3C 3F 78 70 61  be.xmp.....<?xpa
0005F240  63 6B 65 74 20 62 65 67 69 6E 3D 22 EF BB BF 22  cket begin="ï»¿"
```

Nå klarer binwalk å ekstrahere det innebygde png bildet:

```
└─$ binwalk -e face.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 600 x 600, 8-bit/color RGB, non-interlaced
1336          0x538           Zlib compressed data, default compression
389607        0x5F1E7         PNG image, 833 x 333, 8-bit grayscale, non-interlaced
391422        0x5F8FE         Zlib compressed data, default compression
```

Vi kan selvfølgelig bruke andre verktøy som for eksempel CyberChef til å ekstrahere bildet.

Hvis vi åpner bildet så ser det helt svart ut. Hvis vi justerer litt på bildet i bildebehandleren, så får vi frem flagget i bildet.

Rett svar er ctf(magic_bytes).

**Black Box 1**

I denne oppgaven skal vi gjennomføre en minneanalyse og finne et flagg som er skrevet inn i en av applikasjonene. Oppgaven kan løses med Volatility 3.

La oss starte med å liste ut alle prosessene:

```
┌──(kali㉿kali)-[~/volatility3]
└─$ python vol.py -f memdump1.mem windows.psscan.PsScan
```

Oppgaveteksten hinter om at flagget kan være tekst skrevet inn i en tekstbehandler. Den eneste tekstbehandleren ser ut å være notepad med pid 4484:

```
5672    768     CalculatorApp.  0xe58ffb144080  18      -       1       False   2023-07-07 17:38:58.000000
4252    648     SearchIndexer.  0xe58ffb15c080  20      -       0       False   2023-07-07 17:38:45.000000
4484    3276    notepad.exe     0xe58ffb1710c0  3       -       1       False   2023-07-07 17:38:51.000000
2572    608     userinit.exe    0xe58ffb33d080  0       -       1       False   2023-07-07 17:38:43.000000
sabled
400     3276    chrome.exe      0xe58ffbba8080  0       -       1       False   2023-07-07 17:38:54.000000
sabled
1440    768     StartMenuExper  0xe58ffbc3d080  41      -       1       False   2023-07-07 17:38:45.000000
832     768     RuntimeBroker.  0xe58ffbc43340  11      -       1       False   2023-07-07 17:38:45.000000
4920    648     SecurityHealth  0xe58ffbce3080  15      -       0       False   2023-07-07 17:38:55.000000
2584    3276    SecurityHealth  0xe58ffbcea080  7       -       1       False   2023-07-07 17:38:55.000000
```

Vi kan dumpe minneområdet som brukes av denne prosessen:

```
┌──(kali㉿kali)-[~/volatility3]
└─$ python vol.py -f memdump1.mem windows.memmap.Memmap --pid 4484 --dump
```

Så kan vi søke etter flagget med strings:

```
┌──(kali㉿kali)-[~/volatility3]
└─$ strings -e l /home/kali/volatility3/pid.4484.dmp | grep "ctf"
ext-ms-win-tsf-msctf-l1-1-3
msctf.dll
er ctf(super_duper_secr
Flagget er ctf(super_duper_secret)
 ctf(super_duper_secret
 ctf(super_dup
 ctf(super_duper_se
 ctf(super_duper_s
 ctf(super_duper_sec
```

Rett svar er ctf(super_duper_secret).

**Black Box 2**

I denne oppgaven skal vi gjennomføre en minneanalyse og finne et flagg som er tegnet inn i en av applikasjonene. Vi starter med å liste ut alle prosessene:

```
┌──(kali㉿kali)-[~/volatility3]
└─$ python vol.py -f memdump2.mem windows.psscan.PsScan
```

Oppgaveteksten hinter om at flagget kan være et bilde. Den eneste bildebehandleren vi ser er mspaint med pid 8172:

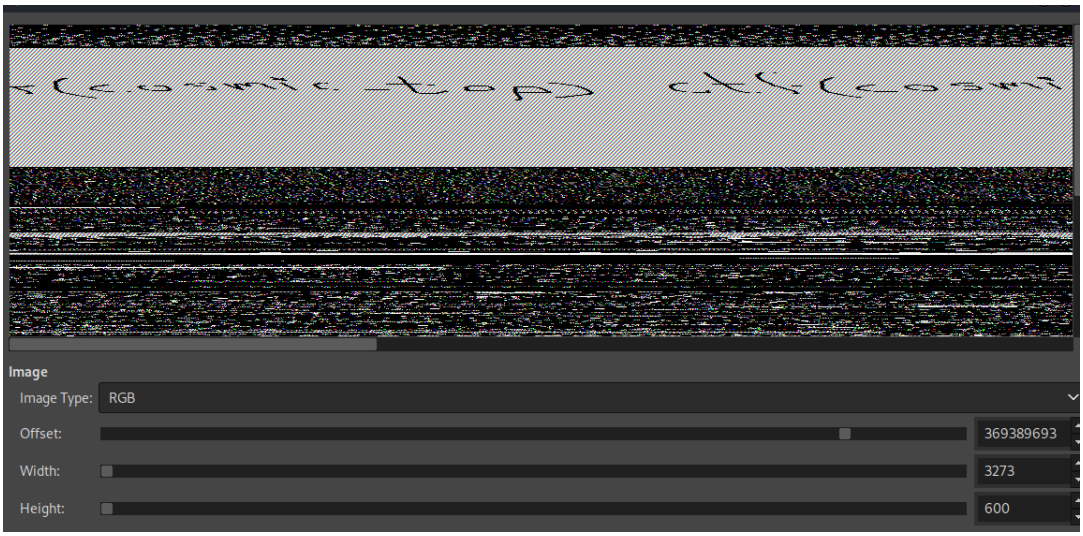
```
7312    772     RuntimeBroker.  0×bd0e3208c340  11      -       1       False
7264    640     svchost.exe     0×bd0e321a6080  8       -       0       False
8172    3848    mspaint.exe     0×bd0e321d5080  6       -       1       False
4604    772     ShellExperienc  0×bd0e32207300  16      -       1       False
7800    772     RuntimeBroker.  0×bd0e322bd0c0  7       -       1       False
7136    772     dllhost.exe     0×bd0e323020c0  10      -       1       False
4280    5876    conhost.exe     0×bd0e323350c0  3       -       1       False
```

Vi kan på samme måte som for Black Box 1 dumpe minneområdet til denne prosessen:

```
┌──(kali㉿kali)-[~/volatility3]
└─$ python vol.py -f memdump2.mem windows.memmap.Memmap --pid 8172 --dump
```

Vi kan ikke bruke strings, siden flagget er tegnet inn i paint. Hvis vi researcher litt på Internett, så finner vi en fremgangsmåte som handler om å endre dumpfilen til .data og så lete etter bitmap grafikken i GIMP.

Her må vi manuelt parse hele filen med inkrementelle offset verdier. Flagget fremkommer som vi ser her:

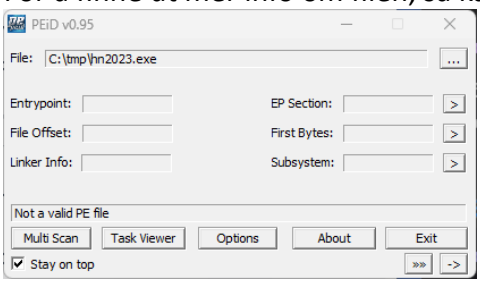



Rett svar er ctf(cosmic_top).

**Password checker**

Password checker (hn2023.exe) er en windows executable som prompter etter et passord. Vi må finne en måte å avdekke passordet på. Pestudio avdekker ikke noe spesielt, som for eksempel interessante strings.

For å finne ut mer info om filen, så kan vi åpne den i PEiD:




PEiD gir tilbakemelding om at filen ikke er en valid PE file (Portable Executable). Kanskje filen er kompilert med pyinstaller?

Vi kan forsøke å pakke ut filen med pyinstaller extractor:

```
C:\Users\royal\pyinstxtractor>python3 pyinstxtractor.py ..\hn2023.exe
[+] Processing ..\hn2023.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.8
[+] Length of package: 11756732 bytes
[+] Found 1014 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth__tkinter.pyc
[+] Possible entry point: pyi_rth_certifi.pyc
[+] Possible entry point: ff1.pyc
[!] Warning: This script is running in a different Python version than the one used to build the executable.
[!] Please run this script in Python 3.8 to prevent extraction errors during unmarshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: ..\hn2023.exe

You can now use a python decompiler on the pyc files within the extracted directory
```

Det gikk bra! Merk at pyinstaller informerer om at python versjonen på analysemaskinen bør være samme versjon som ble brukt for å bygge exe-filen (python 3.8). I dette tilfellet er ikke det nødvendig.

Her ser vi en del interessante entry points tilgjengelige som egne pyc-filer. Vi kan dekompilere disse filene med for eksempel pycdc, eller vi kan laste de opp til https://www.toolnb.com/tools-lang-en/pyc.html

Dekomplierer vi filen ff1.pyc, så finner vi koden som sjekker passordet. Her kan vi se at koden initialiseres ved å gjøre et dns oppslag mot et domene, og henter ut txt recorden til domenet. Deretter kalkuleres md5-hashen av txt-recorden (correct_input), før hashen til slutt speilvendes. Når brukeren skriver inn et passord, så sjekkes verdien i input-feltet med correct_input.

Riktig passord er altså en speilvendt md5-hash av innholdet i txt-recorden. Når vi skriver inn dette passordet, så blir vi presentert med flagget.

## -- Kategori passordhasher

**Hash 1**
I denne oppgaven skal vi avdekke passordet til en NTLM hash. Til det trenger vi en ordliste av ordene på kins.no. For å gjøre det kan vi bruke cewl. Med en ordlengde på minimum 8 og en dybde på 3, så resulterer det i en ordliste på ca 500 ord. Vi kan deretter kjøre Hashcat med ordlisten og legge til et passende regelsett.

Rett svar er ctf(p3rsonv3rnkons3kv3nsutr3dning).

**Hash 2**
I denne oppgaven skal vi avdekke passordet til en NTLM hash. Passordet består delvis av et norsk fornavn og/eller etternavn. Vi kan for eksempel ta utgangspunkt i en liste på Wikipedia: https://no.wikipedia.org/wiki/Liste_over_norske_etternavn

Hvis vi ekstraherer navnene i tabellen til en egen passordliste, så har vi navnet. Vi trenger bare å legge til et passende regelsett.

Rett svar er ctf(Ødegård123!).

**Hash 3**
I denne oppgaven skal vi avdekke passordet til en NTLM hash. Til det trenger vi en ordliste med navn fra Star Wars. Her kan vi bruke en annen liste på Wikipedia: https://en.wikipedia.org/wiki/List_of_Star_Wars_characters

Hvis vi ekstraherer titlene i listen, så har vi navnet. Husk å få med hele tittelen (fornavn og etternavn) før du bryter over til ny linje. Vi trenger deretter å legge til et passende regelsett.

Rett svar er ctf(Zev Senesca).

**Hash 4**
I denne oppgaven skal vi avdekke passordet til en ukjent hash med tilhørende salt. For å identifisere hashalgoritmen kan vi for eksempel bruke https://www.tunnelsup.com/hash-analyzer/.

Vi finner ut at dette er en SHA-256 hash, og vi trenger bare å definere en tekststreng med hashen og saltverdien så den kan mates inn i Hashcat med hash mode -m 1410. En god ordliste med tilhørende regelsett (for eksempel leetspeak.rule) fikser biffen.

Rett svar er ctf(Gr@tul3r3r1).

## -- Kategori Lett blanding

**Gatenavn 1**
I denne oppgaven skal vi finne navnet til en gate som går inn til venstre på bildet. Åpner vi bildet ser vi at det er et skjermdump av Street View on Google Maps. Bildet viser flere clues, blant annet ser vi Storgata, og flere butikker (Intersport, fristør og en butikklogo med bokstaven M). Oppgaven kan løses på mange måter, for eksempel ved å søke etter Intersport butikker som holder til i Storgata. Rett by er Lillehammer. Hvis vi åpnet Street View i Storgata, så kan vi følge veien til vi kjenner oss igjen, og da finner vi også navnet på gaten til venstre.

Rett svar er ctf(Gamlevegen).

**Gatenavn 2**
I denne oppgaven skal vi finne navnet til et fiskevær det står en statue som vi ser på bildet. Dersom vi cropper statuen i et eget bilde, og gjennomfører Google image reverse søk så finner vi fort ut hvor statuen står.

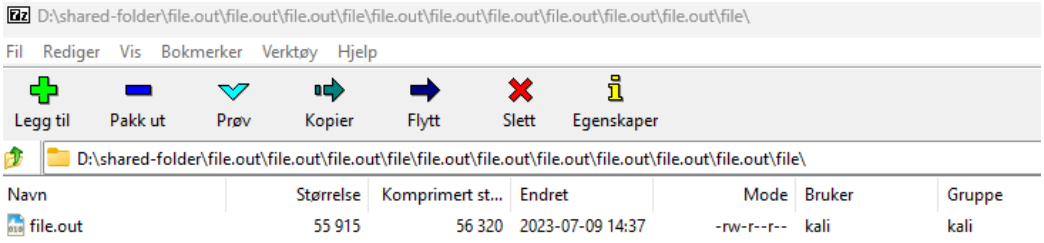
Rett svar er ctf(Veiholmen).

**File.out**

I denne oppgaven skal vi finne ut hva som ligger i filen file.out. Hvis vi kjører file kommandoen i linux, så ser vi at dette er en zip-fil:

```
└─$ file file.out
file.out: Zip archive data, at least v2.0 to extract, compression method=deflate
```

Hvis vi åpner filen i 7-zip, kan vi forsøke å klikke på filen file.out. Det resulterer bare i at det dukker opp en ny fil med samme navn. Dette er altså en nested zip-fil:




Vi kan manuelt forsøke å neste oss ut, men etter en stund mister vi tålmodigheten. Her er det mye nesting, så vi bør skripte oss frem til en løsning. Vi vet altså at nestingen bruker navnet file.out, så vi kan lage et skript som parser en nested zip-fil og søker etter et annet navn enn file.out. Sannsynligvis vil det være dokumentet vi er interessert i.

Her er et bash skript som kan brukes:

```
#!/bin/bash

while (true)
do
 if [[ $(file file.out | awk '{print $2}') == "Zip" ]]
 then
  unzip -o file.out
 fi

 for file in *; do
  if [[ $file != "file.out" ]]; then
   echo "Flagget er funnet i fil: $file"
   exit 0
  fi
 done
done
```

Pass på at filen file.out ligger i samme mappe som skriptet. Når vi kjører skriptet, så får vi en ny fil med navn flag.txt.

Rett svar er ctf(zippety-zip).

**Holy PDF!**

I denne oppgaven skal vi analysere en PDF-fil og avdekke flagget. Åpner vi filen så ser vi at den inneholder vedtektene til KiNS. Metadata viser ingenting relevant.

Vi vet at PDFer kan inneholde mange forskjellige objekter, så vi bør utforske filen med relevante verktøy. To slike verktøy er pdf-parser og peepdf. Disse verktøyene er nyttige for å avdekke embedded innhold i pdf-filer, som for eksempel malware droppere.

Hvis vi åpner pdf-filen med peepdf, så ser vi flere interessante elementer:

```
Objects with JS code (1): [390]
Suspicious elements:
        /Names (2): [389, 412]
        /JS (2): [392, 413]
        /JavaScript (3): [392, 411, 413]
```

Vi ser at objekt 390 inneholder javascript kode, så la oss utforske det objektet:

```
PPDF> object 390

<< /Length 444
/Filter [ /FlateDecode ] >>
stream
function function1() {
    var numbers = [3, 1, 4, 1, 5, 9];
    for(var i=0; i<numbers.length; i++){
        numbers[i] += 2;
    }
    console.log(numbers);
}

function function2() {
    var str = "Hello, world!";
    var reversed = str.split("").reverse().join("");
    console.log(reversed);
}

function function3() {
    var part1 = shiftCharacters("fwi", -3);
    var part2 = shiftCharacters("wkh_vrqv_ri_vrud", -3);
    var test = part1 + "(" + part2 + ")";

    console.log(test);
}

function shiftCharacters(input, shift) {
    var output = '';
    for (var i = 0; i < input.length; i ++) {
        var c = input.charCodeAt(i);
        if (c ⩾ 97 && c ⩽ 122) {
            output += String.fromCharCode((c - 97 + shift + 26) % 26 + 97);
        } else if (c ⩾ 65 && c ⩽ 90) {
            output += String.fromCharCode((c - 65 + shift + 26) % 26 + 65);
        } else {
            output += input.charAt(i);
        }
    }
    return output;
}

// Call the functions
function1();
function2();
function3();

endstream
```

Her ser vi en stream som inneholder javascript. Vi kan analysere funksjonene i skriptet, eller vi kan forsøke å kjøre koden. Javascript kode kan kjøres direkte i en nettleser.

Start nettleseren du bruker (som forhåpentligvis er Brave), velg Flere verktøy og Utviklerverktøy. Åpne en ny fane i nettleseren, velg Console og lim inn koden som ligger mellom stream og endstream. Trykk Enter og vi ser resultatet av skriptet nederst i konsollet. Her ser vi flagget. Det er kun den tredje funksjonen som er relatert til flagget.

Rett svar er ctf(the_sons_of_sora).

## -- Kategori Kryptografi

**RSA e**

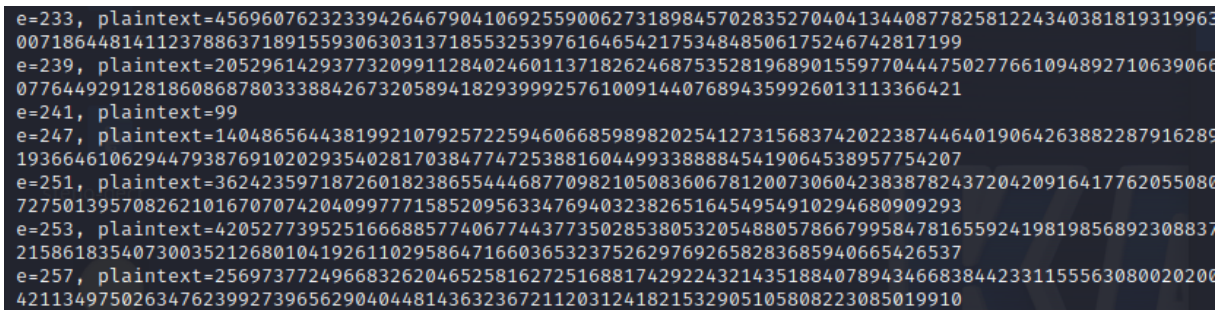
I denne oppgaven får vi oppgitt primtallene p og q, og den krypterte sifferteksten. Vi skal finne e, som er den offentlige eksponenten (det vil si en del av den offentlige nøkkelen). Det er ikke mulig å dekryptere sifferteksten uten e.

e kan teknisk sett variere mellom $3 \leq e < \phi(n)$, men i praksis brukes ofte ranget 3 - 65537. Vi kan altså forsøke et brute force angrep hvor vi forsøker å dekryptere et av tegnene i sifferteksten med hele dette ranget.

Her er et python skript som kan brukes:

```
import gmpy2

# Known values
p =
710836779884488350410420595177274900138474066100082030463150994006603734359228469640522394280595107 1
q =
776348914766351709072347974028806932313758072652182962424035055117035442834924889739714295578877270 1
ciphertext =
229516704880753274993874498256315652118388141219354787312160212084347467627364922804508476410199031728195155822160004761228087125045353549506716240408047341206664481287680894519289450174140632446731 58

# Brute force e
for e in range(3, 65537):
    # Calculate modulus and totient
    n = p * q
    phi = (p - 1) * (q - 1)

    # Calculate private exponent
    try:
        d = gmpy2.invert(e, phi)
    except ZeroDivisionError:
        # skip this value of e if gcd(e, phi) != 1
        continue

    # Decrypt ciphertext
    plaintext = pow(ciphertext, d, n)

    # Print plaintext for each possible e value
    print(f"e={e}, plaintext={plaintext}")
```

Kjører vi dette skriptet, så ser vi at en bestemt verdi (e=241) gir et annet resultat enn de øvrige verdiene:




e=241 gir plaintext=99, som i ASCII tabellen tilsvarer c. Hvis vi antar at ctf er en crib, så kan vi være trygge på at 241 er rett eksponentverdi.

Rett svar er ctf(241).

**RSA dekryptering**

Nå som vi har p, q og e, så kan vi bare fortsette å dekryptere sifferteksten. Vi har allerede dekryptert første tegn i sifferteksten. Hvert tegn i sifferteksten er oppgitt som en egen streng. Dekrypteringen kan gjøres manuelt, med RsaCtfTool, dcode.fr eller skriptes.

Rett svar er ctf(byron).

**RSA d**

I denne oppgaven skal vi skal finne d, som er den private eksponenten. Siden vi allerede har p, q og e, så er det forholdsvis enkelt å utlede d. Vi KAN bruke RsaCtfTool for å utlede den private nøkkelen, men verktøyet gir ikke d i desimalform. Vi må i så fall bruke OpenSSL for å finne d i desimalform. Det enkleste er å bare lage et skript som løser oppgaven direkte.

Her er et python skript som kan brukes:

```
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, y, x = extended_gcd(b % a, a)
        return g, x - (b // a) * y, y

def modinv(e, phi):
    g, x, _ = extended_gcd(e, phi)
    if g != 1:
        raise Exception('modular inverse eksisterer ikke')
    else:
        return x % phi
```

```
# RSA Values
p =
710836779884488350410420595177274900138474066100082030463150994006603734359
228469640522394280595107 1
q =
776348914766351709072347974028806932313758072652182962424035055117035442834
924889739714295578877270 1
e = 241

phi = (p-1) * (q-1)
d = modinv(e, phi)

print('Den private eksponenten d er:', d)
```

**OTP**

Mens RSA er en asymmetrisk krypteringsalgoritme som bruker store primtall, så er One-Time Pad (OTP) en symmetrisk krypteringsalgoritme som bruker en tilfeldig nøkkel som er like lang som meldingen som skal krypteres. OTP krypterer originalmeldingen ved å anvende XOR-operasjon på hvert tegn (eller bit) med det tilsvarende tegnet i nøkkelen.

I denne oppgaven har vi fått oppgitt OTP-nøkkelen og den krypterte teksten, så da kan vi dekryptere teksten for å avdekke originalmeldingen.

Her er et python skript som kan brukes:

```
cipher_text = [
  0b00011011, 0b00001110, 0b00011010, 0b00010100,
  0b00000111, 0b00001101, 0b00000101, 0b00001110,
  0b00010011, 0b00010111, 0b00011110, 0b00000011,
  0b00000101, 0b00010010, 0b00001111, 0b00010100
]

otp_key = [
  0b01101101, 0b01100111, 0b01101100, 0b01110101,
  0b01101011, 0b01101100, 0b01100001, 0b01100111,
  0b01100001, 0b01100011, 0b01110010, 0b01100110,
  0b01100100, 0b01110101, 0b01111010, 0b01110001
]

plain_text = [c ^ k for c, k in zip(cipher_text, otp_key)]
plain_text_ascii = ''.join(chr(c) for c in plain_text)

print(plain_text_ascii)
```

Rett svar er ctf(vivaladirtleague).

## -- Kategori Enkoding og obfuskering

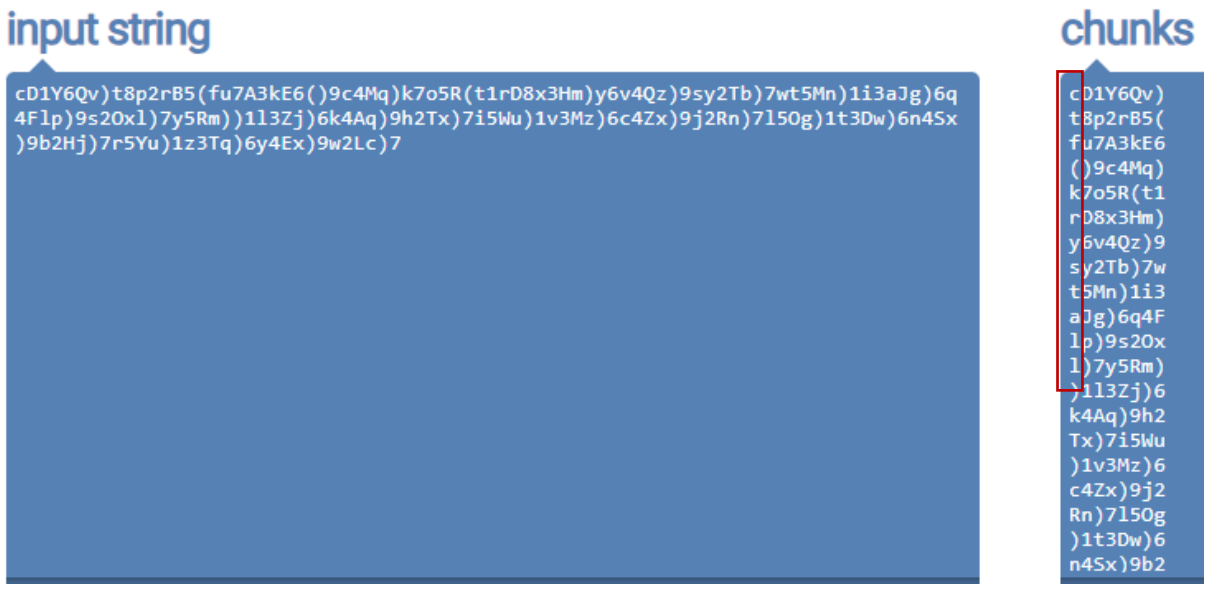
**Enkoding**
I denne oppgaven skal vi dekode en enkodet tekst. Her må vi teste ut noen forskjellige algoritmer, eller for eksempel bruke Magic funksjonen i CyberChef. Teksten er enkodet med base85.

Rett svar er ctf(happymonday).

**Obfuskering 1**
I denne oppgaven skal vi utforske en tallrekke med heksadesimale verdier. Vi ser umiddelbart at enkelte verdier er gjentakene (AB CD EF) og representerer sannsynligvis støy. Hvis vi fjerner disse verdiene og dekoder resten til ASCII så får vi en tekstrekke som ser litt merkelig ut. Men hvis vi speilvender teksten så får vi frem flagget.

Rett svar er ctf(heavy_shit).

**Obfuskering 2**
I denne oppgaven skal vi utforske en tekststreng og avdekke flagget. Tekststrengen ser litt merkelig ut, med små bokstaver, store bokstaver, tall og spesialtegnene (). Det kan hinte om at teksten inneholder flagget i formatet ctf(svar). Hvis vi bryter teksten opp i blokker med 8 tegn, så ser vi flagget:




Her har vi brukt verktøyet onlinestringstools.com.

Rett svar er ctf(krystall).

**Obuskering 3**
I denne oppgaven skal vi finne flagget i et skript med navn obfuscation. Hvis vi åpner filen i eks notepad, så ser vi at skriptet sannsynligvis er powershell. Vi kan følgelig endre etternavn til .ps1.

Vi ser i skriptet at $SecretFlag består av $flgStart + en kodet melding + $flgEnd. Oppgaven kan løses ved å analysere og deobfuskere koden, men det er også mulig å bare hoppe over hele analysen og bare legge til Write-Host $SecretFlag i slutten av skriptet.

Rett svar er ctf(gizmo).