Ten dokument zawiera SPOILERY!

Bardzo zachęcamy do próby rozwiązania zadań samodzielnie przed zapoznaniem się z rozwiązaniami :)

Dostęp do zadań

Dostęp do platformy cwiczenia.hackArcana.pl:

1. Dostęp wymaga posiadania lub utworzenia konta na cwiczenia.hackArcana.pl:
   Rejestracja: https://cwiczenia.hackarcana.pl/register
   Logowanie: https://cwiczenia.hackarcana.pl/login

2. Po zalogowaniu, kliknij przycisk **Menu** (prawa górna strona ekranu) → **Realizacja kodu**, i użyj kodu **7c3f01372982b97a**

3. Kurs "**Arkana Pythona**" pojawi Ci się w zakładce **Szkolenia** w sekcji **Posiadane szkolenia**.

Rozwiązania

Dis Track

W zadaniu otrzymujemy funkcję `mixer`, która jest w postaci wyjścia z Pythonowego deasemblera (`dis.dis()`). Pierwszym krokiem jest "podniesienie" funkcji do kodu Pythona, tj. przeanalizowanie "asemblera" (tekstowej reprezentacji kodu bajtowego) cPythona i przepisanie go na wysokopoziomowy kod Pythona.

Przydatna w tym miejscu może być lista instrukcji cPythona, dostępna m.in. w dokumentacji modułu dis: https://docs.python.org/3/library/dis.html#:~:text=General%20instructions

Poniżej znajduje się funkcja mixer po przetłumaczeniu jej na kod Pythona, wraz z odpowiednimi instrukcjami tekstowej reprezentacji kodu bajtowego Pythona dla każdej z jej linii.

```
def mixer(p):
  # 3           0 RESUME                   0
  # Można zignorować.

  # 4           2 LOAD_GLOBAL              1 (NULL + list)
  #            12 LOAD_FAST                0 (p)
  #            14 CALL                     1
  #            22 STORE_FAST               0 (p)
  p = list(p)

  # 5          24 BUILD_LIST               0
  #            26 STORE_FAST               1 (m)
  m = []

  # 6          28 LOAD_CONST               1 (947)
  #            30 STORE_FAST               2 (s)
  s = 947

  # 7          32 LOAD_GLOBAL              3 (NULL + range)
  #            42 LOAD_GLOBAL              5 (NULL + len)
  #            52 LOAD_FAST                0 (p)
  #            54 CALL                     1
  #            62 CALL                     1
  #            70 GET_ITER
  #       >>   72 FOR_ITER                57 (to 190)
  #            76 STORE_FAST               3 (_)
  for _ in range(len(p)):
```

```
    # 8           78 LOAD_FAST                2 (s)
    #               80 LOAD_CONST               2 (751)
    #               82 BINARY_OP                5 (*)
    #               86 LOAD_CONST               3 (1439)
    #               88 BINARY_OP                0 (+)
    #               92 LOAD_CONST               4 (683)
    #               94 BINARY_OP                6 (%)
    #               98 STORE_FAST               2 (s)
    s = (s * 751 + 1439) % 683

    # 9         100 LOAD_FAST                 1 (m)
    #           102 LOAD_ATTR                 7 (NULL|self + append)
    #           122 LOAD_FAST                 0 (p)
    #           124 LOAD_ATTR                 9 (NULL|self + pop)
    #           144 LOAD_FAST                 2 (s)
    #           146 LOAD_GLOBAL               5 (NULL + len)
    #           156 LOAD_FAST                 0 (p)
    #           158 CALL                      1
    #           166 BINARY_OP                 6 (%)
    #           170 CALL                      1
    #           178 CALL                      1
    #           186 POP_TOP
    #           188 JUMP_BACKWARD            59 (to 72)
    m.append(p.pop(s % len(p)))
    # 7     >>  190 END_FOR

  # 10        192 LOAD_CONST                5 ('')
  #           194 LOAD_ATTR                11 (NULL|self + join)
  #           214 LOAD_FAST                 1 (m)
  #           216 CALL                      1
  #           224 RETURN_VALUE
  return ''.join(m)
```

Wersja bez komentarzy:

```
def mixer(p):
  p = list(p)
  m = []
  s = 947
  for _ in range(len(p)):
    s = (s * 751 + 1439) % 683
    m.append(p.pop(s % len(p)))
  return ''.join(m)
```

Kolejnym krokiem jest przeanalizowanie i zrozumienie zrewersowanej funkcji. W tym przypadku okazuje się, że funkcja `mixer` tworzy nowy ciąg tekstowy znak po znaku z podanego stringu. Co istotne, każdy kolejny znak jest wybierany pseudolosowo, za pomocą prostej funkcji matematycznej[1], która mówi który znak wybrać jako kolejny. Wybrany znak jest równocześnie usuwany z ciągu źródłowego. Tj. funkcja `mixer` po prostu "przetasowuje" kolejność znaków w podanym ciągu.

```
pwd = input("Enter the password (flag): ").strip()
if mixer(pwd) == "noHiprgodgxotii}udn{yreuAhsutmerhrn":
  print("Correct password ")
else:
  print("Nope, wrong!")
```

Ponieważ zadanie ma klasyczną formę enc(input) == $flag_{enc}$ (patrz np. szkolenie "Jak wygrywać CTFy"), w ostatnim kroku musimy stworzyć funkcję odwrotną do funkcji mixer (tj. funkcję dekodującą), tak, aby z posiadanego zakodowanego ciągu otrzymać flagę.

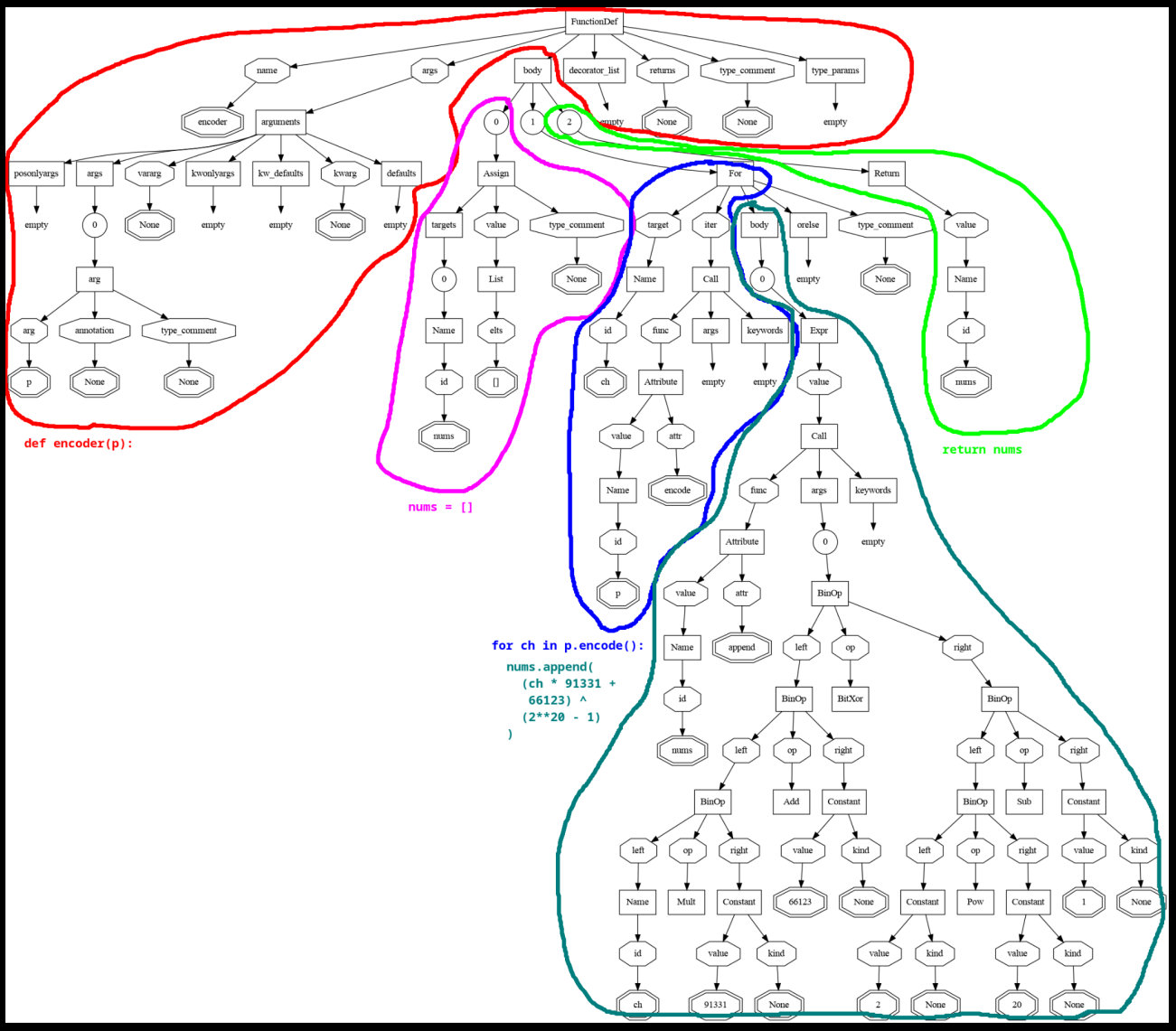
W przypadku funkcji tasującej znaki najprościej będzie podać jej na wejściu tablicę "indeksów", tj. kolejnych liczb od 0 do 34 (wielkości flagi minus 1 włącznie), przetasować tę tablicę, a następnie zobaczyć gdzie który indeks "wylądował", co pozwoli na łatwe przywrócenie posiadanego.

```
def unmixer(p):
  idxs = list(range(len(p)))
  m = []
  s = 947
  for _ in range(len(p)):
    s = (s * 751 + 1439) % 683
    m.append(idxs.pop(s % len(idxs)))
  print(m)

  # Przywrócenie oryginalnego ciągu na bazie tablicy indeksów.
  o = []
  for i in range(len(p)):
    o.append(p[m.index(i)])
  return ''.join(o)

print(unmixer("noHiprgodgxotii}udn{yreuAhsutmerhrn"))
```

W efekcie wykonania kodu otrzymamy prawidłową flagę!

[1] Technicznie jest to tzw. LCG, tj. liniowy generator kongruencyjny (ang. *linear congruential generator*) – jest to prosta i popularna funkcja do generowania liczb pseudolosowych. O ile nie nadaje się ona do zastosowań kryptograficznych, to sprawdza się w wielu innych zastosowaniach, gdzie jakość losowości i nieprzewidywalność mają drugorzędne znaczenie. Więcej o LCG można znaleźć na https://en.wikipedia.org/wiki/Linear_congruential_generator

I like trees



(wersja w wysokiej rozdzielczości:
https://cdn-hackarcana.net/pyarcana-5e57b9c9e23f9de127494c36c7526bd8/i-like-trees-annots.png)

Zadanie to jest bardzo zbliżone do zadania Dis Track, z tą różnicą, że funkcję kodującą wejście otrzymujemy w postaci AST (ang. *Abstract Syntax Tree*, patrz również https://en.wikipedia.org/wiki/Abstract_syntax_tree).

O ile na pierwszy rzut oka drzewo może wyglądać na dość duże, to w praktyce okazuje się, że jest ekwiwalentne jedynie trzem liniom kodu oraz nagłówkowi funkcji. Co więcej, jest ono zaskakująco czytelne i powinno być w pełni zrozumiałe po jedynie krótszym zastanowieniu się. W razie czego dokumentację różnych rodzajów węzłów można znaleźć w oficjalnej dokumentacji modułu ast, tj. https://docs.python.org/3/library/ast.html

Po przepisaniu drzewa z powrotem na kod Pythona otrzymujemy:

```
def encoder(p):
  nums = []
  for ch in p.encode():
    nums.append((ch * 91331 + 66123) ^ (2**20 - 1))
  return nums
```

Analizując otrzymana funkcję widzimy, że mamy do czynienia z prostą odwracalną operacją matematyczną ((ch * 91331 + 66123) ^ (2**20 - 1)) zaaplikowaną na każdym bajcie (UTF-8) ciągu wejściowego.

```
inp = input("Enter the flag (or your favorite tree, I don't mind): ")
nums = encoder(inp)
if nums == [6989532, 8535237, 10994252, 5531697, 10720259, 7990712,
10358396, 8535237, 9445086, 8535237, 3711999, 11450907, 6350215, 8900561,
10084403, 5531697, 9810410, 8626568, 5440366, 10267065, 9445086, 8717899,
10358396, 5531697, 9810410, 8626568, 5531697, 11450907, 10358396,
5531697, 9810410, 8626568, 6532877, 8900561, 9627748, 9993072, 8535237,
5531697, 9810410, 8626568, 5169834, 5169834, 5169834, 10537597]:
  print("Correct! That is the flag!")
else:
  print("Nope. Spruces are cool though!")
```

Ponieważ jest to kolejne zadanie w formie enc(input) == $flag_{enc}$, to wystarczy wziąć listę stałych ($flag_{end}$, tj. [6989532, 8535237, ...]) i na każdej liczbie wykonać odwrotne operacje matematyczne w odwrotnej kolejności względem funkcji `encoder`. Będzie to więc najpierw operacja XOR, następnie odejmowanie (jako przeciwieństwo dodawania), a potem dzielenie całkowite (jako przeciwieństwo mnożenia):

```
def decoder(nums):
  p = []
  for num in nums:
    p.append(((num ^ (2**20 - 1)) - 66123) // 91331)
  return bytes(p).decode()

print(decoder([6989532, 8535237, 10994252, 5531697, 10720259, 7990712,
10358396, 8535237, 9445086, 8535237, 3711999, 11450907, 6350215, 8900561,
10084403, 5531697, 9810410, 8626568, 5440366, 10267065, 9445086, 8717899,
10358396, 5531697, 9810410, 8626568, 5531697, 11450907, 10358396,
5531697, 9810410, 8626568, 6532877, 8900561, 9627748, 9993072, 8535237,
5531697, 9810410, 8626568, 5169834, 5169834, 5169834, 10537597]))
```

Po uruchomieniu powyższej funkcji otrzymujemy flagę!

Image of a Flag

Jak można się domyślić bazując na treści szkolenia, otrzymany plik BMP jest również poprawnym plikiem ZIP[2]. Zmieniając rozszerzenie z BMP na ZIP możemy go w łatwy sposób rozpakować, otrzymując tym sposobem plik `__main__.py`.

```
$ cp image-of-a-flag.bmp image-of-a-flag.zip
$ unzip image-of-a-flag.zip
Archive:  image-of-a-flag.zip
warning [image-of-a-flag.zip]:  286322 extra bytes at beginning or within
zipfile
  (attempting to process anyway)
  inflating: __main__.py
```

Plik ten zawiera następujący kod:

```
FLAG = [1376940440376, 1931541451083, 2294900733960, 1243071230895,
2352273252309, 1243071230895, 1529933822640, 2008038142215,
1893293105517, 2218404042828, 2237528215611, 2180155697262,
1931541451083, 1396064613159, 2199279870045, 1663803032121,
2122783178913, 2180155697262, 2218404042828, 1988913969432,
1587306340989, 2218404042828, 2008038142215, 2065410660564,
2065410660564, 1415188785942, 2237528215611, 2199279870045,
2218404042828, 1510809649857, 2103659006130, 1931541451083,
1338692094810, 2065410660564, 1855044759951, 1969789796649,
2390521597875]
t = input("What do you think the flag is? ").strip()
```

[2] Jest to tzw. binarny plik poliglotyczny, a więc plik zgodny z więcej niż jedną specyfikacją różnych formatów plików binarnych.

```
if len(t) != len(FLAG):
  print("Nah. Wrong length.")
  exit()
t = [ord(ch) * 19124172783 for ch in t]
if t != FLAG:
  print("Nah. Wrong flag.")
  exit()
print("Correct! That is the flag!")
```

Drugi krok jest praktycznie identyczny jak w zadaniu I like trees, z tą różnicą, że operacja matematyczna jest jeszcze prostsza (tylko jedno mnożenie). Wystarczy zatem zaaplikować odwrotną operację na posiadanej tablicy stałych (FLAG = [1376940440376, 1931541451083, ...]) by otrzymać flagę:

```
FLAG = [1376940440376, 1931541451083, 2294900733960, 1243071230895,
2352273252309, 1243071230895, 1529933822640, 2008038142215,
1893293105517, 2218404042828, 2237528215611, 2180155697262,
1931541451083, 1396064613159, 2199279870045, 1663803032121,
2122783178913, 2180155697262, 2218404042828, 1988913969432,
1587306340989, 2218404042828, 2008038142215, 2065410660564,
2065410660564, 1415188785942, 2237528215611, 2199279870045,
2218404042828, 1510809649857, 2103659006130, 1931541451083,
1338692094810, 2065410660564, 1855044759951, 1969789796649,
2390521597875]
print(''.join([chr(v // 19124172783) for v in FLAG]))
```

Irreplaceable

Kod zadania Irreplaceable składa się w zasadzie z trzech części:

1. Zestawu funkcji `mix_*`, które wykonują proste odwracalne transformacje na podanej tablicy bajtów i które stanowią podstawowy element budulcowy dla funkcji `transform_*`.
2. Zestawu Funkcje `transform_*`, które używają 5 wybranych podstawowych transformacji (`mix_*`) na podanym ciągu bajtów.
3. Oraz delikatnie ukrytego kodu, który modyfikuje to które faktycznie funkcje `mix_*` są wywoływane w funkcjach `transform_*`.

Warto zacząć od analizy ukrytego kod – widocznego od razu jeśli nasz edytor ma włączony "*word wrapping*" (w innym wypadku trzeba zauważyć np. brak dwukropka w nagłówku funkcji `main`):

```
def main()
-> any(globals().__setitem__(k,
type(mix_ror)(v.__code__.replace(co_names=v.__code__.co_names[::-1]),
globals(), k)) for k, v in dict(globals()).items() if
zlib.crc32(k[:-1].encode()) == 2177289788): # Nothing to see here, move
along.
```

Przepisując kod na trochę czytelniejszy pseudokod otrzymujemy:

1. Dla wszystkich funkcji zadeklarowanych w przestrzenii globalnej, kórych nazwa bez ostatniego znaku ma CRC32 równe 2177289788:
    a. Podmień tę funkcję na identyczną funkcją, z zapisaną od tyłu tablicą `co_names` (odwołania do nazw globalnych).

Ustalenie które funkcje mają odpowiednie CRC32(nazwa_bez_ostatniego_znaku) jest bardzo proste – nie ma w końcu zbyt wielu możliwości:

```
>>> import zlib
>>> zlib.crc32(b"mix_")
3007102966
>>> zlib.crc32(b"transform_")
2177289788
```

Warto w tym momencie popatrzeć na tablicę `co_names` obiektu `code` jednej z tych funkcji i pomyśleć jak "przepisanie tej tablicy na wspak" zmieni działanie tej funkcji.

Przykładowo, `transform_0` wygląda następująco:

```
def transform_0(b):
  mix_xor(b) or mix_add(b) or mix_add(b) or mix_sub(b) or mix_ror(b)
```

```
>>> transform_0.__code__.co_names
('mix_xor', 'mix_add', 'mix_sub', 'mix_ror')
>>> transform_0.__code__.co_names[::-1]
('mix_ror', 'mix_sub', 'mix_add', 'mix_xor')
```

Co za tym idzie, po modyfikacji wywołanie

- `mix_xor` zmienia się na `mix_ror`,
- `mix_add` na `mix_sub`,
- `mix_sub` na `mix_add`,
- i `mix_ror` na `mix_xor`.

Faktycznie wykonana funkcja ma zatem postać:

```
def transform_0(b):
  mix_ror(b) or mix_sub(b) or mix_sub(b) or mix_add(b) or mix_xor(b)
```

Pierwszym krokiem będzie zatem przepisanie wszystkich funkcji `transform_*` na ich zmodyfikowaną postać.

```
# >>> transform_0.__code__.co_names
# ('mix_xor', 'mix_add', 'mix_sub', 'mix_ror')
# >>> transform_0.__code__.co_names[::-1]
# ('mix_ror', 'mix_sub', 'mix_add', 'mix_xor')
def transform_0(b):
  mix_ror(b) or mix_sub(b) or mix_sub(b) or mix_add(b) or mix_xor(b)

# >>> transform_1.__code__.co_names
# ('mix_add', 'mix_ror', 'mix_xor')
# >>> transform_1.__code__.co_names[::-1]
# ('mix_xor', 'mix_ror', 'mix_add')
def transform_1(b):
  mix_xor(b) or mix_ror(b) or mix_ror(b) or mix_ror(b) or mix_add(b)

# >>> transform_2.__code__.co_names
# ('mix_sub', 'mix_xor')
# >>> transform_2.__code__.co_names[::-1]
# ('mix_xor', 'mix_sub')
def transform_2(b):
  mix_xor(b) or mix_sub(b) or mix_xor(b) or mix_sub(b) or mix_xor(b)

# >>> transform_3.__code__.co_names
# ('mix_rol', 'mix_add', 'mix_xor')
# >>> transform_3.__code__.co_names[::-1]
# ('mix_xor', 'mix_add', 'mix_rol')
def transform_3(b):
  mix_xor(b) or mix_add(b) or mix_xor(b) or mix_xor(b) or mix_rol(b)

# >>> transform_4.__code__.co_names
# ('mix_add', 'mix_xor', 'mix_sub')
# >>> transform_4.__code__.co_names[::-1]
# ('mix_sub', 'mix_xor', 'mix_add')
def transform_4(b):
  mix_sub(b) or mix_sub(b) or mix_sub(b) or mix_xor(b) or mix_add(b)
```

Teraz możemy usunąć "ukryty" kod i przeanalizować pozostałe zadanie, które, zwyczajowo ma postać enc(input) == flag_{enc}. Co za tym idzie, naszym celem będzie wykonanie odwrotnych operacji

w odwrotnej kolejności aby zdekodować flagę z jej posiadanej wersji zakodowanej ($flag_{enc}$, tj. `b'K\xe4\xf9...'`).

Zacznijmy od przepisania funkcji `mix_*` na ich odwrotne ekwiwalenty, tj. zamienimy dodawanie na odejmowanie, obrót bitowy w prawo na obrót bitory w lewo, i tak dalej (w zasadzie wystarczy zmienić 4 znaki zaznaczone poniżej):

```
def mix_xor(b):
  for i in range(len(b)):
    b[i] ^= 0b10100101

def mix_add(b):
  for i in range(len(b)):
    b[i] = (b[i] - 0b11001100) & 0xff

def mix_sub(b):
  for i in range(len(b)):
    b[i] = (b[i] + 0b00110011) & 0xff

def mix_ror(b):
  for i in range(len(b)):
    b[i] = ((b[i] << 1) | (b[i] >> 7)) & 0xff

def mix_rol(b):
  for i in range(len(b)):
    b[i] = ((b[i] >> 1) | (b[i] << 7)) & 0xff
```

Drugim krokiem będzie odwrócenie zmodyfikowanych funkcji `transform_*`, czyli po prostu wywołanie "poprawionych" funkcji `mix_*`, ale w odwrotnej kolejności:

```
def transform_0(b):
  mix_xor(b)
  mix_add(b)
  mix_sub(b)
  mix_sub(b)
  mix_ror(b)

def transform_1(b):
  mix_add(b)
  mix_ror(b)
  mix_ror(b)
  mix_ror(b)
  mix_xor(b)

def transform_2(b):
```

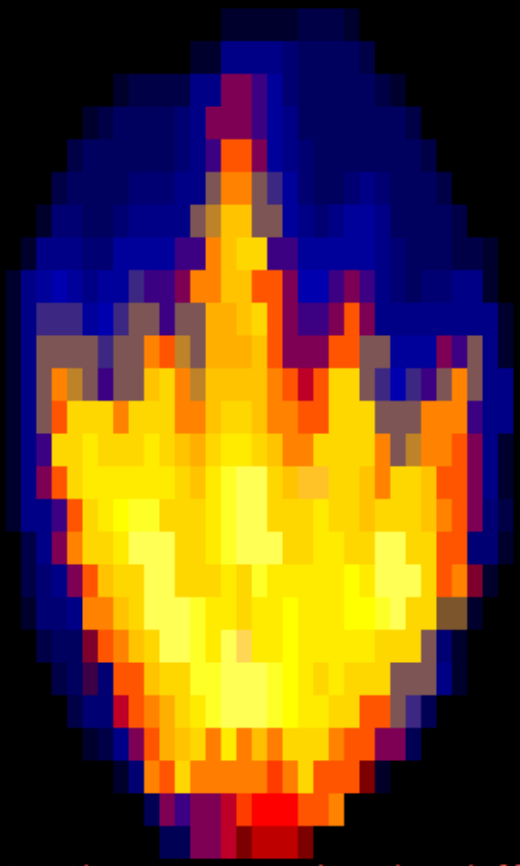
```
  mix_xor(b)
  mix_sub(b)
  mix_xor(b)
  mix_sub(b)
  mix_xor(b)

def transform_3(b):
  mix_rol(b)
  mix_xor(b)
  mix_xor(b)
  mix_add(b)
  mix_xor(b)

def transform_4(b):
  mix_add(b)
  mix_xor(b)
  mix_sub(b)
  mix_sub(b)
  mix_sub(b)
```

I na samym końcu wystarczy odwrócić kolejność wywołań samych funkcji `transform_*` w funkcji `main()` i wywołać je na posiadanej zakodowanej fladze.

```
def main():
  b =
bytearray(b'K\xe4\xf9\xb1Nhn\x82\xbd\xa4\x88\x99\xa4n\xb1\xa8d\xe7D\xc7\x
e4\xebd\xf9BD\x8e')

  transform_4(b)
  transform_3(b)
  transform_2(b)
  transform_1(b)
  transform_0(b)

  print(b)

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

Po uruchomieniu tak przygotowanego kodu otrzymamy flagę!

PyRE



Próba otwarcia pliku PyRE.py szybko pokazuje, że mamy do czynienia z plikiem binarnym – konkretniej z plikiem ZIP.

```
$ file PyRE.py
PyRE.py: Zip archive data, at least v2.0 to extract, compression
method=deflate
$ unzip PyRE.py
Archive:  PyRE.py
  inflating: __main__.pyc
```

Po rozpakowaniu ZIPa otrzymujemy plik `__main__.pyc`, a więc skrypt cPythona w postaci skompilowanej.

Co za tym idzie, musimy teraz odzyskać kod źródłowy, co najłatwiej jest uczynić korzystając z dowolnego działającego dekompilatora[3] Python 3.12.3. Jeśli byśmy korzystali z serwisów dekompilujących online, warto pamiętać, że te mogą sobie zachować zarówno skompilowany, jak i odzyskany kod źródłowy naszego programu. Na szczęście w przypadku tego ćwiczenia nie ma to znaczenia, więc możemy skorzystać np. z PyLingual.io.

Odzyskany kod (z pominiętym ANSI-artem) wygląda następująco:

```
# Decompiled with PyLingual (https://pylingual.io)
# Internal filename: pyre.py
# Bytecode version: 3.12.0rc2 (3531)
# Source timestamp: 2025-08-29 13:10:32 UTC (1756473032)

import sys
import base64
if sys.platform == 'win32':
    import ctypes
    handle = ctypes.windll.kernel32
    mode = ctypes.c_ulong()
    kernel32.GetConsoleMode(handle, ctypes.byref(mode))
    kernel32.SetConsoleMode(handle, mode.value | 4)
print(b''.fromhex('20' + "...dużo bajtów..."*0 + '0a').decode())
flag = input('Enter the flag or face the PyRE: ')
flag =
base64.b32encode(base64.b64encode(flag.encode()))[::-1].hex()[::-1]
if flag ==
'b4e44465d4e43425c4242364149545e405142365f465a5a57464944554254565b4655454
55334454e465c484b4a53464a4a59465d465b423f42425752365c494d4a59445230594d3'
:
    print('Correct flag!')
else:
    print('Nope')
```

Ignorując kod, który wykonuje się tylko pod Windowsem (który włącza na konsoli kody ANSI na potrzeby wyświetlenia kolorowego ANSI artu), dostajemy bardzo prostą funkcję transformującą wejście i porównującą wynik ze stałą. Czyli, jak zwykle, mamy do czynienia z enc(input) == flag_{enc}.

Aby odzyskać flagę wystarczy wykonać odwrotne operacje w odwrotnej kolejności na posiadanej zakodowanej fladze, tj.:

```
import base64
encoded =
"b4e44465d4e43425c4242364149545e405142365f465a5a57464944554254565b4655454553
34454e465c484b4a53464a4a59465d465b423f42425752365c494d4a59445230594d3"
```

[3] Wersja trudna dla chętnych: zamiast korzystać z dekompilatora, ręcznie zerwersuj to zadanie.

```
flag = b''.fromhex(encoded[::-1])[::-1]
flag = base64.b64decode(base64.b32decode(flag)).decode()
print(flag)
```

Uruchomienie tego kodu wyświetli naszą flagę!

Run Me If You Can

Tym razem dostajemy funkcję – a w zasadzie jej kod bajtowy – która od razu wypiszę nam flagę. Wiemy nawet, jak ta funkcja jest wywołana, tj. wiemy ile jest parametrów i jakie to parametry.

Co za tym idzie, wystarczy stworzyć jakikolwiek (hint: jak najprostszy) kompatybilny obiekt `code` oraz `function`, i uruchomić tak uzyskaną funkcje:

```
import types
c = types.CodeType(
  8, # argcount, ← możemy policzyć
  0, # posonlyargcount, ← pewnie 0
  0, # kwonlyargcount, ← widzimy po wywołaniu, że 0
  8, # nlocals, ← co najmniej tyle co argumentów
  20, # stacksize, ← możemy strzelać
  3, # flags, ← podpatrujemy w jakiejś innej funkcji

b''.fromhex("970002007c0402007c0702007c0502007c067c007c01ab02000000000000
44008f088f096303670063025d1000005c0200007d087d097c087c097a0c00007c027a0a0
0007c037a06000091028c12040063037d097d08ab01000000000000ab01000000000000ab
0100000000000053006302010063037d097d087700"),# codestring,
  (None,), # constants, ← None zawsze warto dać
  (), # names, ← pewnie nic, bo wszystko idzie w argumentach
  ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'), # varnames,
  "whatever.py", # filename, ← mało istotne
  "print_flag",# name, ← mało istotne
  "print_flag",# qualname, ← mało istotne
  1, # firstlineno, ← mało istotne
  b'', # linetable, ← meh
  b'', # exceptiontable ← meh
)
print_flag = types.FunctionType(c, globals())
print_flag(
b''.fromhex("a91dc5dff5f65bfe7fcccabf87a9b0dbcedcd5be9b5d35a5486edbaf2277
1bce0acb5a9c1f3f6b5f79beb1a47690"),
b''.fromhex("d6816aa747933e9be76951341835344367787315 11fd9e0f30cb40218bef
bc648a60da39988fedc6d8222b0fdc24"),
  55, 256, print, bytes, zip, bytes.decode
)
```

ReRe

Omówienie rozwiązania bazowej wersji ReRe można znaleźć na poniższym archiwalnym nagraniu z livestreama:
https://youtu.be/JExnV1-GNxk?t=1159 (00:19:18 - 01:54:22)

W razie czego kod źródłowy oryginalnej wersji można znaleźć na moim blogu:
https://gynvael.coldwind.pl/?id=602