

# Výprava do chrámu

## Zip zip zip

Dostaneme vstup - `input.txt`.

Po otevření v textovém editoru ale nevypadá úplně v pořádku.

```
PK^C^D^T^@^@^@H^@Q=eVq)İı~u
^@äu
^@^O^@^@\^@FIKS_zadani.pdfUT ^@^CÚ9^DdÚ9^Ddux^K^@^A^D! ^@^@^@^D! ^@^@^@^@<84>·uTTQİ
<9d>&^Fó[<9a>ÿ^Er<87>@±^G^E`<99><-s8f0wŞ^T^DlŞÉ1<97>õÜé<85>ç^Aj<9d>^Qpă^W<97>İ
Ó*{äU^K®<8d>ß«<85>LB, 0Àû=Ê"<81>à<9f>À&İα~E²<9f>álÁi^P^TI<83>tİ±^Q<87>^]Á^@ KÜ:
Ö1^U<8b>ö½^Læ^a<82>Ýc0ê%! ÍÖß^Y^Au<83>AÑnG0^HfÁ]^?Rö^B3^A^Aú`ë<97>0A^K0øÅô^QÖıhg
w<93>½8o·<99>Ô\X^o^Gİ`^@İ(øL^N#<8b>^F<80>!ðK1^^pyÇ^\ù^Rr »^A^DgaσÝ;9<82>y^_Ñê(ç
<98>^C@İ<8d>[`ÆÑPL.^Z
İXâ<80>úYÀ ^Hα^[^H>»^@^S^C^PhA{ÑÜ5İÛa^GđĚ^](V^Z>è<87>^D@Îø^[<9c>^A^°ê£=^@¨ãèô^İ
```

Není to textový soubor, je to zip. Napoví nám například hlavička souboru začínající `PK`, poměrně unikátní identifikátor právě pro ZIP archivy.

```
> file input.txt
input.txt: Zip archive data, at least v2.0 to extract, compression method=deflate
```

Nebo třeba linuxový příkaz `file`, který právě podle hlavičky umí soubor identifikovat.

```
> mv input.txt input.zip
```

Po rozbalení získáme tyto soubory:

```
├─ checker.py
├─ feedback.txt
├─ FIKS_zadani.pdf
└─ README.txt
```

Přečteme si `README.txt`, koukneme na `feedback.txt` a `checker.py` a zjistíme co hledáme a v jakém formátu odevzdáváme.

Nejvíce nás ale zajímá `FIKS_zadani.pdf`. K naší smůle ho ale žádný PDF viewer nedokáže otevřít. Co to?

```
> file FIKS_zadani.pdf
FIKS_zadani.pdf: bzip2 compressed data, block size = 900k
```

Je to další archiv, tentokrát BZIP2. Zákeřní orgové nám určitě naschvál popletli přípony!

```
> mv FIKS_zadani.pdf FIKS_zadani.bz2
> bunzip2 FIKS_zadani.bz2
```

Rozbalíme tedy archiv, a v něm... další!

```
> file FIKS_zadani
FIKS_zadani: Zip archive data, at least v2.0 to extract, compression method=deflate
> mv FIKS_zadani FIKS_zadani.zip
> unzip FIKS_zadani.zip
```

A další...

```
> file kapela_startup_nahravka.tar
kapela_startup_nahravka.tar: POSIX tar archive (GNU)
> tar xf kapela_startup_nahravka.tar
```

Začínáme tu tušit určitý vzorec. Mimochodem, kapela Startup je reálná a hrajou v ní někteří naši orgové (see <https://www.instagram.com/startupcz>).

Jen ta nahrávka tu chybí, protože teprve doděláváme první album :D

```
> file cute_kitten.jpeg
cute_kitten.jpeg: 7-zip archive data, version 0.4
> mv cute_kitten.jpeg cute_kitten.7z
> 7z e cute_kitten.7z
```

Pokračujeme v boji s archivy

```
> file not_a_virus.exe
not_a_virus.exe: gzip compressed data, was "fiks{turtl3s_all_th3_way_d0wn_955854}"
```

A máme první vlajku: `fiks{turtl3s_all_th3_way_d0wn_955854}`!

```
> mv not_a_virus.exe not_a_virus.gz
> gunzip not_a_virus.gz
```

```
> file not_a_virus
not_a_virus: Python script
> mv not_a_virus fiks{turtl3s_all_th3_way_d0wn_955854}.py
```

## Dropper

---

Máme před sebou soubor `fiks{turtl3s_all_th3_way_d0wn_955854}.py`, tentokrát už je v něm doopravdy to co bychom čekali, tedy kód v Pythonu.

Když soubor zkusíme spustit, selže.

```
File "fiks{turtles_all_the_way_down}.py", line 36
    global out
        ^
TabError: inconsistent use of tabs and spaces in indentation
```

Když se podíváme dovnitř, zjistíme proč. Kód je rozbitý a musíme ho opravit!

Odkomentujeme řádek 40: `#exec(''.join(chr(i) for i in ex), globals(), globals())`. Komentář sice tvrdí, že ho nepotřebujeme, ale nám je po přečtení kódu jasné, že komentářům se v tomto případě nedá věřit.

Array `mex` prochází na řádce 37 XORem s neznámým klíčem. Na řádcích 28-33 se nám nějaké klíče objevují, tak je postupně zkusíme. A hned první klíč, `0xBE`, nám po XORu dá něco co vypadá jako payload - další python kód.

```
nex = [0x29, 0x29, 0x22, 0x74, 0x69, 0x66, 0x22, 0x28, 0x64, 0x65, 0x66, 0x20]
def AaAaAaAa(a, b):
    d = ""
    c = (a ^ b)
    if (c >= 256):
        d = chr(c % 256)
    else:
        d = chr(c)
    return d
def AaAaAaAaA():
    global result
    a = ""
```

```

ii = ''.join(chr(i) for i in nex)
exec(ii, globals(), globals())
b = result
c = ""
d = ""
e = "e"
d = ""
if (b != ""):
c = b[1]
if (c != ""):
d = b[0]
if (d != ""):
if (e == ""):
return c
elif (not AaAAaAaaA(0xCC, key1)):
return d
else:
d = (str(AaAAaAaaA(0xCC, key1)) + str(d), str(c) + str(AaAAaAaaA(0xCC, key1)))
return d
else:
return b
return a
def AaAAaAaaaA(a, b, c):
return "{aAaaAA}t{aAaaaA}{aAaaAa}{aAAaaA}e".format(aAaaAa = b, aAaaaAA = c, aAa
out = AaAAaAaaaA(0xE1, AaAAaAaaA()[1], AaAAaAaaA()[0])

```

Tenhle kód ale taky dostal trochu na frak. Chybí mu indentace!

Po troše rozmýšlení se nám podaří kód zaindentovat a zároveň z něj získáme array jménem `nex`, obsahující druhý payload - taktéž python kód.

```

))"tif"(def ,)"t" ,"u" ,"v" ,"c"(uwu( = tuser
r nruter
]0[])"(" + f(lave + ))48x0(owo(rhc = r
))(tsegidxeh.))(edocne."tiftuvc"(5dm.bilhsah(rts = f
bilhsah tropmi
:)g(def fed
]0[e + e nruter
]1[])(e27a2c065ab27fbf856936b2ff3abb2b + ))08x0(owo(rhc = e
:)d ,c ,b ,a(uwu fed
2yek ^ k nruter
:)k(owo fed
)]1-[]1-::[]s(rts ,]3[]1-::[]s(rts( nruter
]3[]0[])(kcats.tcepsni = s
tcepsni tropmi

```

```
:(e27a2c065ab27fbf856936b2ff3abb2b fed
```

Tenhle kód je ještě divnější než ty předchozí. Že by byl zašifrovaný? Ne, jen napsaný pozpátku!

Ve finále tedy z druhého payloadu dostáváme:

```
def b2bba3ff2b639658fbf72ba560c2a72e():
    import inspect
    s = inspect.stack()[0][3]
    return (str(s)[::-1][3], str(s)[::-1][-1])
def owo(k):
    return k ^ key2
def uwu(a, b, c, d):
    e = chr(owo(0x80)) + b2bba3ff2b639658fbf72ba560c2a72e()[1]
    return e + e[0]
def fed(g):
    import hashlib
    f = str(hashlib.md5("cvutfit".encode()).hexdigest())
    r = chr(owo(0x84)) + eval(f + "()")[0]
    return r
result = (uwu("c", "v", "u", "t"), fed("fit"))
```

Můžeme buď přemýšlet co tenhle kód dělá, nebo ho můžeme vložit do prvního payloadu. A když si první payload oddělíme do souboru, který pak zkusíme zpustit, zjistíme, že už běží, jen nám chybí nějaké klíče.

```
Traceback (most recent call last):
  File "p1.py", line 57, in <module>
    out = AaAAaaAaaaA(0xE1, AaAAaaAaaA()[1], AaAAaaAaaA()[0])
  File "p1.py", line 30, in AaAAaaAaaA
    result = (uwu("c", "v", "u", "t"), fed("fit"))
  File "p1.py", line 22, in uwu
    e = chr(owo(0x80)) + b2bba3ff2b639658fbf72ba560c2a72e()[1]
  File "p1.py", line 20, in owo
    return k ^ key2
NameError: name 'key2' is not defined
```

Klíče, klíče, kde bychom je našli? Aha, v hlavním souboru nějaké byly!

Tak je vezmeme, vložíme do programu, přidáme pár printů na data která nám vznikají a zkusíme to znovu. Program teď vypadá nějak takto:

```

nex = [0x29, 0x29, 0x22, 0x74, 0x69, 0x66, 0x22, 0x28, 0x64, 0x65, 0x66, 0x20,

bytearr = [0xBE, 0xEF]
key1 = bytearray[0]
key2 = bytearray[1]

def AaAAaAaaA(a, b):
    d = ""
    c = (a ^ b)
    if (c >= 256):
        d = chr(c % 256)
    else:
        d = chr(c)
    return d

def b2bba3ff2b639658fbf72ba560c2a72e():
    import inspect
    s = inspect.stack()[0][3]
    return (str(s)[: -1][3], str(s)[: -1][ -1])

def owo(k):
    return k ^ key2

def uwu(a, b, c, d):
    e = chr(owo(0x80)) + b2bba3ff2b639658fbf72ba560c2a72e()[1]
    return e + e[0]

def fed(g):
    import hashlib
    f = str(hashlib.md5("cvutfit".encode()).hexdigest())
    r = chr(owo(0x84)) + eval(f + "()")[0]
    return r

def AaAAaAaaA():
    global result
    a = ""
    result = (uwu("c", "v", "u", "t"), fed("fit"))
    print(result)

    b = result
    c = ""
    d = ""
    e = "e"
    d = ""
    if (b != ""):
        c = b[1]
        if (c != ""):

```

```

        d = b[0]
        if (d != ""):
            if (e == ""):
                return c
            elif (not AaAaaAaaaA(0xCC, key1)):
                return d
            else:
                d = (str(AaAaaAaaaA(0xCC, key1)) + str(d), str(c) + str(AaAaaAaaaA(0xE1, AaAaaAaaaA()[1], AaAaaAaaaA()[0])))
                return d
        else:
            return b
    return a

def AaAaaAaaaA(a, b, c):
    return "{aAaaAA}t{aAaaaA}{aAaaAa}{aAaaaA}e".format(aAaaAa = b, aAaaaA = c, aAaaAA = a)

out = AaAaaAaaaA(0xE1, AaAaaAaaaA()[1], AaAaaAaaaA()[0])
print(out)

```

A tady už dostáváme výstup: `robot_karel`. Teď nám stačí tento výstup dosadit do původního programu a vidíme, že vlajka by měla být `fiks{robot_karel}`. Hurá!

No a kam dál?

Všimneme si, že program stále ještě neběží. Chybí klíč číslo tři, který se používá při ověřování hashe vlajky a dešifrování další úlohy. Bez něj program nepoběží a nedá nám další úlohu dokud nebude správný a neoznámí, že byla úloha úspěšně vyřešena.

Ale je to jeden bajt (jak nám říká řádek 30 - tři klíče jsou tři bajty a dva už máme), takže těch 256 možných hodnot ověříme jednoduchým cyklem proti hashi v programu.

Hledaný klíč nám vyjde 0x69, nice.

Dosadíme ho na své místo a hurá, program běží!

```

> ./fiks{turtles_all_the_way_down}.py

```

```

You did it! UwU :3
Your flag is: fiks{robot_karel}

```

Zároveň se nám ve složce objevil soubor `z7.vihcra`, který tam předtím nebyl. Černá magie!

Pozpátku to je `archive.7z`. Aha!

```
> mv z7.vihcra archive.7z
> 7z e archive.7z
Enter password (will not be echoed):
```

Chce to po nás heslo. Ale my přece žádné neznáme.

...

Nebo snad ano?

V původním python souboru se na řádce 25 nachází nápověda: “Your new flag will help you find the way”

Ať už jí najdeme nebo ne, nakonec nás napadne vyzkoušet jako heslo `fiks{robot_karel}`, které nám otevře cestu k posledním třem úkolům.

```
├─ index.html
├─ obrazek.png
└─ win32.exe
```

## Web

---

V souboru `index.html` se mimo jiné nachází

```
<p id='flag' hidden> Zmlrc3tqYXY0c2NyMxB0X2lzM3QzcnIxYmwzXzY1NTE0MX0= </p>
```

Což je base64 string.

Base64 jde typicky poznat podle znaků `=` na konci, ale nemusí tomu tak vždy být.

```
> echo -n 'Zmlrc3tqYXY0c2NyMxB0X2lzM3QzcnIxYmwzXzY1NTE0MX0=' | base64 -d
fiks{jav4scr1pt_is_t3rr1bl3_655141}
```

## Obrázek

---

### Řešení

Text je zakódovaný do bajtů barev jednotlivých pixelů obrázku.

Tedy každý znak je jedna složka pixelu v pořadí RGB.

RGBRGBRGBRGBRGBRGBR  
Nejaky zakodovany text

Ukázkový skript na převod do textu.

```
#!/usr/bin/env python3

from PIL import Image

def generate_string(filename: str) -> str:
    # Open the image file
    image = Image.open(filename)

    # Get the pixel values from the image
    pixel_values = list(image.getdata())

    # Flatten the list of pixel values and convert it to a bytes object
    input_bytes = bytes([i for sublist in pixel_values for i in sublist])

    # Remove any padding bytes (zeros) from the end of the bytes object
    input_bytes = input_bytes.rstrip(b'\x00')

    # Convert the bytes object to a string and return it
    return input_bytes.decode()

output_string = generate_string("obrazek.png")
print(output_string)
```

Výsledkem je base64 string

CnBuY2doZXIgdZ3VyIHN5bnQgKHBncykqdmEgcGJ6Y2hncmUgZnJwaGV2Z2wgdmYgbmEgcmtYzXB2Zr

Zde base64 zrovna na `=` nekončí, ale jde poznat ze znaků, ze kterých se skládá.

Pod dekódování na nás vykoukne zdánlivě nesmyslný text, který ale už alespoň vzdáleně připomíná normální text.

```
pncgher gur synt (pgs) va pbzchgre frphevgl vf na rkrepvfr va juvpu "syntf" ne
...
vba ng qrspba, gurer unir orra bgure pgs pbzcrvgvbf ubfgrq vapyhqvat pfnj p
```

Jedná se o primitivní substituční šifru - Césarova šifra - nebo také rot13.

Po dešifrování získáme výsledný text včetně vlajky `fiks{w4ve_th3_flag_553287}`

## Easter egg

První nápovědou že v souboru je něco schované je velikost souboru.

Obrázek má 712 pixelů.

I kdyby nebyl nijak zkomprimovaný, tak to je  $712 \times 3 = 2136$  bajtů, ne ~587000 kterých soubor má.

Soubor můžeme otevřít v nějakém hex editoru.

Začíná `.PNG` ("magic" bajty na začátku souboru) označující, že se jedná o png obrázek.

Sekce dat obrázku začíná `IDAT` a končí `IEND`.

Po `IEND` ale následuje znovu `.PNG`.

Buď to můžeme udělat v hex editoru, nebo použijeme jiný nástroj, např. `foremost`

```
> foremost obrazek.png
Processing: obrazek.png
|*|
```

```
> tree output
output
├── audit.txt
├── png
│   ├── 00000000.png
│   └── 00000003.png
```

```
> ls -l output/png
total 576
-rwxr----- 1 david david 1796 10. dub 15.34 00000000.png
-rwxr----- 1 david david 584758 10. dub 15.34 00000003.png
```

První obrázek (který má 1796 bajtů) je obrázek, který se zobrazí při otevření a ve kterém je schovaná vlajka.

Druhý obrázek je easter egg.

## Win32

---

Perlička na konec byla reverse engineering Win32 executable souboru.

Na začátek dostaneme soubor `win32.exe`, který nám řekne že chce znát heslo a dodá pár užitečných rad.

Cílem úlohy je prozkoumat fungování programu a zjistit, jaké heslo je potřeba zadat, aby nám vydal flag.

```
V tomto programu se schovává tajemství. K jeho získání musíš zadat správné heslo.  
Reverse engineering! Dobrý začátek je prozkoumat program hezkými instrukcemi po ir  
Pomoci ti mohou nástroje jako OllyDBG, Ida Free, Radare2 nebo Ghidra.
```

```
-----  
Zadej heslo: abcdefgh
```

```
Spatné heslo!
```

Program umí zobrazit dvě chybové hlášky - "Špatná délka hesla" a "Špatné heslo". Ještě než se vůbec pustíme do reverzování, zjistíme díky tomu, že hledáme heslo s osmi znaky.

Koho napadl bruteforce přístup, má docela smůlu - programu trvá kontrola každého hesla zhruba půl vteřiny (obsahuje volání `Sleep`), takže vyzkoušet všech cca  $127^8$  kombinací není moc realistické.

Program nám radí použít nějaký z volně dostupných disassemblerů, tzn. programů které nám umí ukázat zdrojový kód programu ve formě assemblerových instrukcí.

Může to vypadat například nějak takto:

```
lea     eax, [ebp+Buffer]  
push    eax  
call    sub_4012C8  
movzx   eax, ax  
add     eax, 0FFFFFFF8h  
pop     ecx  
cmp     ax, 2  
ja      loc_40152C  
mov     al, [ebp+var_F8]  
test    al, al  
jz      short loc_401476  
cmp     al, 0Dh  
jnz     loc_401525  
movsx   edi, [ebp+var_FE]  
push    offset a117  
mov     [ebp+var_F8], 0  
add     edi, edi  
call    sub_401334
```

```

mov     dl, [ebp+var_FC]
pop     ecx
movsx   ecx, [ebp+Buffer]
movzx   ebx, ax
movsx   eax, dl
add     ecx, eax
cmp     ecx, 0ACh
jnz     short loc_401525
cmp     dl, 3Bh
jle     short loc_401525
mov     al, [ebp+var_FF]
cmp     al, [ebp+var_F9]
jnz     short loc_401525
push    0FAh
call    ds:__imp_Sleep
cmp     [ebp+var_FD], 69h
jnz     short loc_401525
movsx   ecx, [ebp+var_FA]
movsx   eax, [ebp+var_F9]
sub     eax, ecx
cmp     eax, 17h
jnz     short loc_401525
movsx   eax, [ebp+var_FE]
sub     eax, edi
add     eax, ecx
jnz     short loc_401525
movsx   ecx, [ebp+var_FB]
cmp     ecx, ebx
jnz     short loc_401525
mov     al, [ebp+var_FF]
xor     al, 69h
cmp     al, 10h
jnz     short loc_401525
cmp     [ebp+var_FC], 5Bh
jge     short loc_401525

```

Před námi se otevřely (alespoň) dvě cesty, kterými se lze vydat.

První možností je studovat tento assemblerový kód a z něj zjistit co program dělá.

Druhá možnost je využít funkcionality, kterou mají některé z uváděných nástrojů (např. Ida a Ghidra, nebo třeba ChatGPT) a assembler si nechat převést na (nefunkční, ale ke čtení postačující) Cčkový kód.

V případě Idy může výstup vypadat nějak takhle:

```

if ( sub_4012DD(&Buffer) && (Sleep(0xFAu), (unsigned __int16)(sub_4012C8(&Buf
{
    if ( !v11 || v11 == 13 )
    {
        v11 = 0;
        v0 = 2 * v5;
        v1 = (unsigned __int16)sub_401334("117");
        if ( v7 + Buffer == 172 && v7 > 59 && v4 == v10 )
        {
            Sleep(0xFAu);
            if ( v6 == 105 && v10 - v9 == 23 && !(v9 + v5 - v0) && v8 == v1 && (v
                return sub_401372(&Buffer);
            }
        }
    }
    sub_401304("\nSpatne heslo!\n");
}

```

Z toho už nám je výrazně jasnější, jaké podmínky musí heslo (buffer) splňovat.

Jsou to:

- $\text{buffer}[0] + \text{buffer}[4] == 0xAC$
- $\text{buffer}[1] == \text{buffer}[7]$
- $\text{buffer}[4] > 59 \ \&\& \ \text{buffer}[4] < 0x5B$  (z toho nám jasně plyne že  $\text{buffer}[4]$  je  $0x5A$  a tedy  $\text{buffer}[0]$  je  $0x52$ )
- $\text{buffer}[3] == 'i'$
- $\text{buffer}[7] - \text{buffer}[6] == 0x17$
- $\text{buffer}[2] + \text{buffer}[6] == 3 * \text{buffer}[2] - \text{buffer}[2]$  (jinými slovy,  $\text{buffer}[2] == \text{buffer}[6]$ )
- $\text{buffer}[5] == \text{atoi}("117")$  (neboli  $\text{buffer}[5] == 117$ )
- $\text{buffer}[3] \wedge \text{buffer}[1] == 0x10$  (a jelikož víme, že  $\text{buffer}[3]$  je  $0x69$ ,  $\text{buffer}[1]$  musí být  $0x69 \wedge 0x10 = 0x79$ , a víme že stejnou hodnotu má i  $\text{buffer}[7]$ . A když jsme zjistili že  $\text{buffer}[7]$  je  $0x79$ , můžeme zjistit hodnotu  $\text{buffer}[6]$  (a  $\text{buffer}[2]$ ), která je  $0x79 - 0x17 = 0x62$ )

Z toho nám celkem jednoduše vyplyne, když si zjištěné hodnoty převedeme na ASCII charaktery, že heslo je `RybiZuby`.

Jiný přístup může být požadavky formalizovat a použít nějaký nástroj na řešení soustavy rovnic, například Z3 solver:

```
#!/usr/bin/env python3
from z3 import *

v1 = 117

Buffer = [ Int(f'buffer[{i}]') for i in range(9)]

s = Solver()

for i in range(9):
    s.add(Or(Buffer[i] == 0, And(Buffer[i] >= 0x20, Buffer[i] <= 0x7E)))

s.add(Or(Buffer[8] == 0, Buffer[8] == 13))

v0 = 2 * Buffer[2]

s.add(Buffer[4] + Buffer[0] == 172)
s.add(Buffer[4] > 59)
s.add(Buffer[1] == Buffer[7])

s.add(Buffer[3] == 105)
s.add(Buffer[7] - Buffer[6] == 23)
s.add(Buffer[6] + Buffer[2] - (2 * Buffer[2]) == 0)
s.add(Buffer[5] == v1)

# s.add((Buffer[1] ^ 0x69) == 16) # z3 neumi bit xor
s.add(Buffer[1] == 0x79) # hex(0x69 ^ 16): 0x79

s.add(Buffer[4] < 91)

print(s.check())
m = s.model()

heslo = []

for i in range(9):
    heslo.append(int(str(m.evaluate(Buffer[i]))))

print(bytes(heslo))
```

Tím jsme získali heslo. Po jeho zadání se nám dešifruje flag, uložený v programu a zašifrovaný pomocí XORu s hledaným heslem.

Flag: `fiks{rev3rz0vani_j3_pr0st3_pr1ma}`