Reverzné inžinierstvo

Bezpečnosť informačných systémov z pohľadu praxe

Peter Švec

Everything is open source if you try hard enough.

0x01

>Motivácia

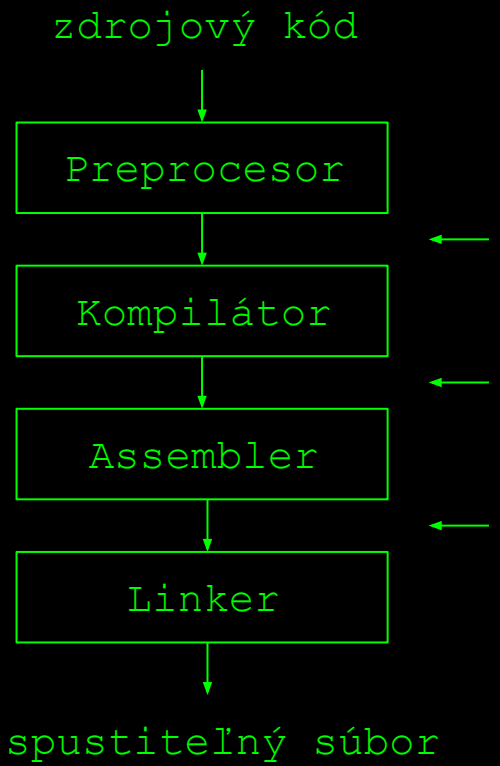
>Schopnosť pochopiť program aj bez zdrojového kódu
>Vývoj exploitov
>Analýza malvéru
>Cracking, patching
>Prečo assembler?
    >Je všade
    >Znalosť základov pomáha aj pri programovaní v jazykoch vyššej úrovne
    >Programovanie HW, ovládačov

>Štandardný proces

```
#include <stdio.h>
int main()
{
    printf("BISPP");
    return 0;
}
```

↓

```
00 0f ff 48 22 01 55
42 12 69 12 00 48 12
13 22 f5 a5 …
```



zdrojový kód (makrá, komentáre, hlavičkové súbory)

zdrojový kód (jazyk sym. inštrukcií)

objektový kód

>Zostavovací process

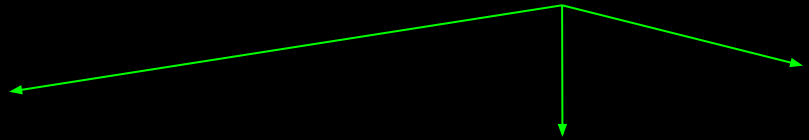
>Počas zostavovania programu strácame množstvo informácií:
- >Názvy premenných
- >Názvy funkcií, tried,...
- >Komentáre
- >Štruktúry
- >Optimalizácie prekladača (inlining, loop unrolling,...)

>Reverzné inžinierstvo

>Opačný proces

spustiteľný súbor

analytický proces



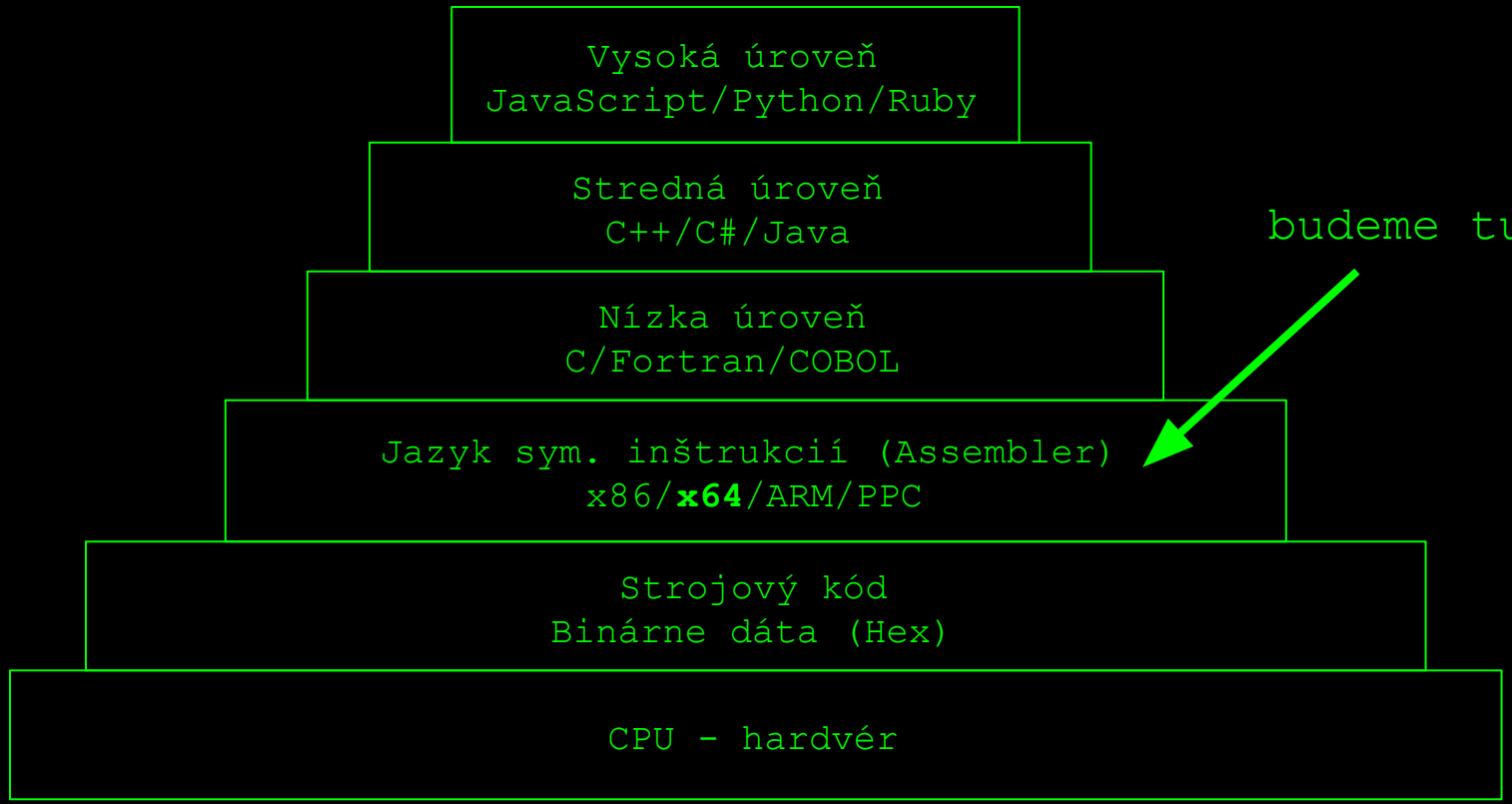
disassembler
(to čo pozná CPU počas vykonávania)

dekompilátor
(to čo programátor poznal počas kompilácie)

pochopenie functionality

ladenie

>Abstrakcia


Vysoká úroveň
JavaScript/Python/Ruby
Stredná úroveň
C++/C#/Java
Nízka úroveň
C/Fortran/COBOL
Jazyk sym. inštrukcií (Assembler)
x86/**x64**/ARM/PPC
Strojový kód
Binárne dáta (Hex)
CPU - hardvér
budeme tu

>Spustiteľný súbor

>ELF formát (**E**xecutable and **L**inkable **F**ormat)

0x20 0x75 0x6c → AND BYTE PTR[RBP+0x6c], DH

0010 0000 0111 0101 0110 1100

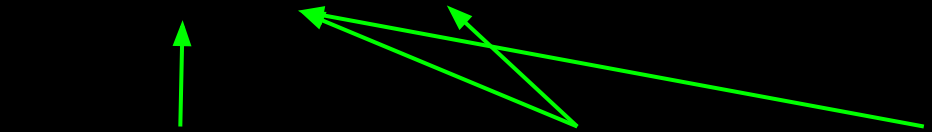
>Jazyk symbolických inštrukcií

>Tri základné koncepty:
- >**Inštrukcie**
- >**Registre**
- >**Pamäť** (samotné inštrukcie/zásobník/…)

>Inštrukcie

>**Inštrukcie** hovoria CPU akú operáciu ideme vykonať
(matematické operácie, logické operácie, systémové volania…)
>**Operandy** hovoria nad akými dátami ideme vykonávať operáciu
>Každá inštrukcia ma svoj opkód

XOR AL, 0x2e -> 0x34 0x2e



inštrukcia operandy register

Všeobecná forma: **INŠTRUKCIA OPERAND, OPERAND, …**

>Operandy

>Nad akými dátami ideme vykonávať operáciu
>Tri metódy adresovania:
    >Priame adresovanie:

```
MOV RAX, 0x42
```

    >Registre:

```
MOV RAX, RBX
```

    >Pamäť:

```
MOV RAX, [RBX]
```

>Registre

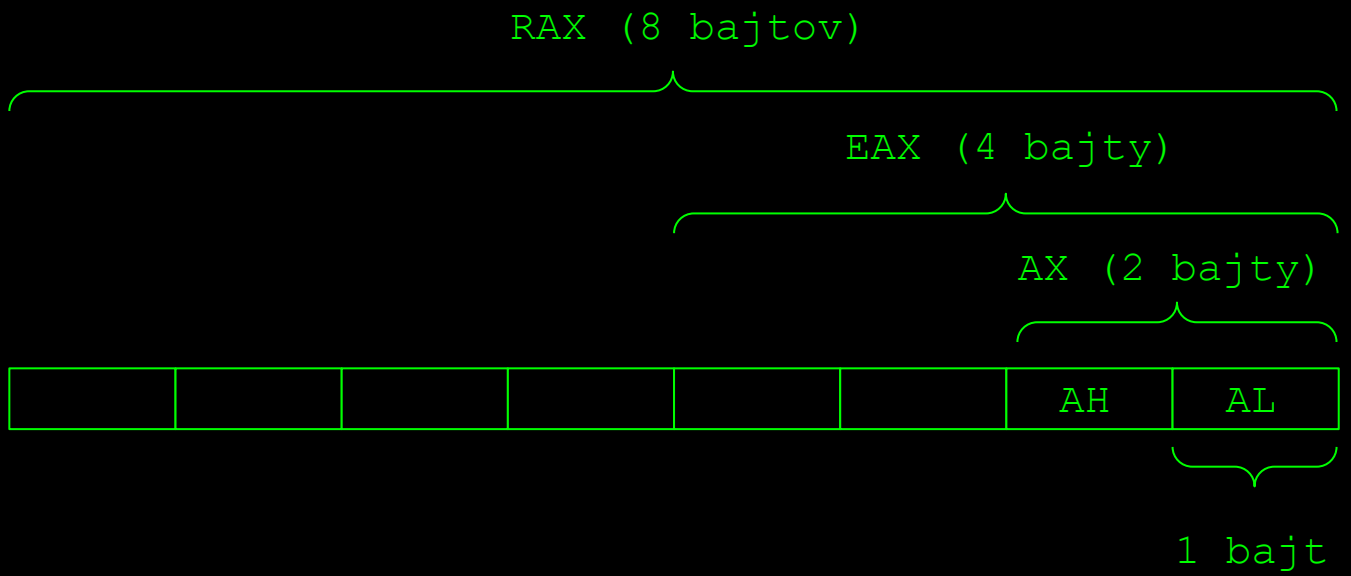
>Malý a rýchly úložný priestor (**8 bajtov** na x64)

>Všeobecné registre:
- >8086: AX, BX, CX, DX, **SP**, **BP**, SI, DI
- >x86: EAX, EBX, ECX, EDX, **ESP**, **EBP**, ESI, EDI
- >x64: RAX, RBX, RCX, RDX, **RSP**, **RBP**, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15

>Adresa nasledujúcej inštrukcie:
- > **IP**(8086), **EIP**(x86), **RIP**(x64)

>Špeciálny register kde sa nastavujú flagy podľa operácií:
- > **EFLAGS**(x86), **RFLAGS**(x64)

>Registre




```
>RAX = 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
>AH  = 0xcc
>RAX = 0xff 0xff 0xff 0xff 0xff 0xff 0xcc 0xff
>EAX = 0xaa 0xaa 0xaa 0xaa
>RAX = 0x00 0x00 0x00 0x00 0xaa 0xaa 0xaa 0xaa
```

>RFLAGS register

>Register ma 64 bitov (ako inak)
>Každý bit reprezentuje určitý príznak (1: nastavený 0: nenastavený)
>Príznaky sa používajú na kontrolu CPU operácií alebo výsledky CPU operácií (napr. pri matematických operáciách, kontrole toku programu a pod.)
>Najdôležitejšie príznaky:
- >**ZF** - **Z**ero **F**lag (výsledok op. je nula)
- >**OF** - **O**verflow **F**lag (výsledok op. je pretečenie)
- >**SF** - **S**ign **F**lag (výsledok op. je negatívna hodnota)
- >**CF** - **C**arry **F**lag (výsledok op. je väčší ako cieľova dst)

>Inštrukcia MOV

>Inštrukcia MOV presúva dáta (nie úplne presné…)
>Formát:

```
MOV dst, src
```

>Príklady:

```
MOV RAX, 0x42
MOV RAX, [RBX + 8]
```

>Inštrukcia LEA

>Inštrukcia načítava adresu do registra (**L**oad **E**ffective **A**ddress)
>K pamäťovému miestu nepristupuje
>Formát:

```
LEA dst, src
```

>Príklady:

```
LEA RAX, [RBX + 8]
```

v RAX budeme mať uloženú adresu RBX + 8

>Aritmetické inštrukcie

>Súčet:

```
ADD dst, value      ADD RBX, 0x5
```

>Odčítanie:

```
SUB dst, value      SUB RBX, 0x5
```

>Násobenie:

```
MUL value           MUL 0x5
```

>násobí hodnotu v registri RAX, výsledok uloží do RDX:RAX

>Delenie:

```
DIV value           DIV 0x5
```

>delí hodnotu v RAX, výsledok uloží do RAX, zvyšok do RDX

>Ďalšie aritmetické inštrukcie

>Logické operácie (OR, AND, XOR)

```
AND dst, value          AND RAX, 0x5
XOR RAX, RAX <- rýchle nastavenie RAX na nula
```

>Bitový posun (SHR, SHL)

```
SHR dst, count          SHR RAX, 0x5
```

>Bitová rotácia (ROR, ROL)

```
ROR dst, count          ROR RAX, 0x5
```

>Pomocné direktívy

```
MOV [0x1337000], 1
```

>0x01?
>0x00 0x01?
>0x00 0x00 0x00 0x01??

```
MOV BYTE PTR [0x1337000], 1
MOV WORD PTR [0x1337000], 1
MOV DWORD PTR [0x1337000], 1
MOV QWORD PTR [0x1337000], 1
```

BYTE: 8 bitov (1 bajt)
WORD: 16 bitov (2 bajty)
DWORD: 32 bitov (4 bajty)
QWORD: 64 bitov (8 bajtov)

>Testovanie podmienok

>Inštrukcia **TEST**
- >Jedná sa o rovnakú inšt. ako **AND**, avšak nemodifikuje op.
- >**TEST value, value**
- >**TEST RAX, RAX** -> nastaví **ZF** na 1

>Inštrukcia **CMP**
- >Jedná sa o rovnakú inšt. ako **SUB**, avšak nemodifikuje op.
- >**CMP dst, src**
- >Môže meniť **ZF** a **CF**

```
dst == src    ZF: 1 CF: 0
dst  < src    ZF: 0 CF: 1
dst >  src    ZF: 0 CF: 0
```

>Tok programu

>Inštrukcia **JMP**:
- >Nepodmienený skok
- >**JMP navestie**
- >vždy skočí na blok kódu, ktorý je ozn. návestím
- >návestie = označenie bloku inštrukcií menom

>Podmienené skoky (skáču podľa príznakov, nastavených **TEST**, **CMP**)
- > **JG** - Jump if greater
- > **JL** - Jump if lower
- > **JNE** - Jump if not equal
- > **JLE** - Jump if less or equal
- > **JGE** - Jump if greater or equal
- > …

>Systémové volania

>Interakcia s operačným systémom
>**open**, **read**, **write**, **fork**, **exec**,...
>Inštrukcia **SYSCALL** (iba na x64)
>Príklad:
>>Chceme zavolať systémové volanie **exit** s hodnotou 42
>>Číslo systémového volania je 60[1]

```
MOV RAX, 60
MOV RDI, 42
SYSCALL          ; navratová hodnota v RAX
```

[1]https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

>Zásobník

>Zásobníkový rámec pre volanie funkcií
>Obsahuje:
    >Kde začína zásobníkovy rámec predchádzajúceho volania
    >Lokálne premenné pre funkciu
    >Návratová adresa (návrat z funkcie - inštrukcia **RET**)
>Registre ovladajúce zásobník:
    >**RSP** -> vrch zásobníka (**S**tack **P**ointer)
    >**RBP** -> spodok zásobníka (**B**ase **P**ointer)
>Práca so zásobnikom: **PUSH, POP**

```
PUSH RAX        POP RCX
PUSH RAX        POP [RCX]
```

>Ďalšie inštrukcie

>Volanie funkcie **CALL**
- >Ukladá na zásobník adresu nasledujúcej inšt. (pre návrat)
- >Skáče na návestie

```
CALL navestie
```

>Žiadna operácia:
- >**NOP** (No Operation)

>Endianita

>Dáta na x86 architektúrach sú uložené v opačnom poradí
>Little Endian
>**LSB**(**L**east **S**ignificant **B**it) je uložený na najnižšej adrese

>Nástroje

>Statická analýza(bez spustenia)
- >Binary Ninja[1] (cloud verzia)
- >IDA Free 7.6
- >Ghidra
- >Radare2

>Dynamická analýza (so spustením)
- >strace
- >GDB

[1]https://cloud.binary.ninja/

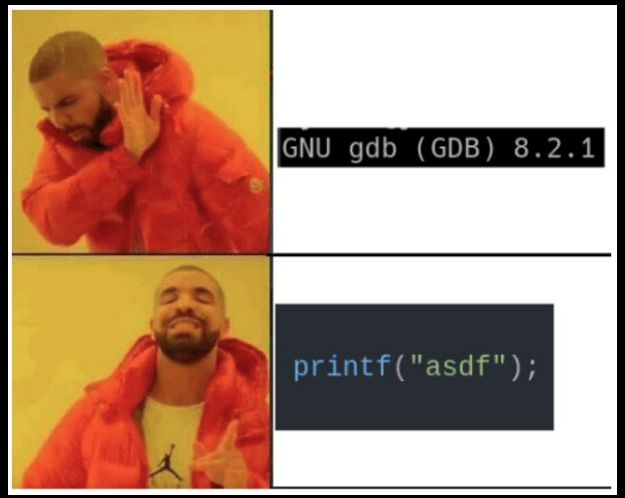
>Dynamická analýza

>Kontrola systémových volaní

```
strace ./level_1.0
```

>GDB (pwndbg plugin)

```
echo source /opt/pwndbg/gdbinit.py >> ~/.gdbinit
```



>GDB

>si (**S**tep **I**nstruction) -> ďalšia inštrukcia (vnorenie do **CALL**)
>ni (**N**ext **I**nstruction) -> ďalšia inštrukcia (preskočenie **CALL**)
>Prehliadanie registrov:
    x/gx $rsp
    x/8b $rax
    x/20i $rip
>Breakpointy:
    >Manuálne inštrukcia int3 (0xcc) v kóde
    >Návestia/mená funkcií v kóde (break návestie)
    >break *adresa (break *0x1337000)
>Ďalšie príkazy[1]

[1]https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

>Disassembler

```
f3 0f 1e fa e9 77 ff ff  ff f3 0f 1e fa 55 48 89
e5 48 8d 3d ac 0e 00 00  b8 00 00 00 00 e8 ee fe
ff ff b8 00 00 00 00 5d  c3 0f 1f 80 00 00 00 00
f3 0f 1e fa 41 57 4c 8d  3d 3b 2c 00 00 41 56 49
89 d6 41 55 49 89 f5 41  54 41 89 fc 55 48 8d 2d
2c 2c 00 00 53 4c 29 fd  48 83 ec 08 e8 5f fe ff
```

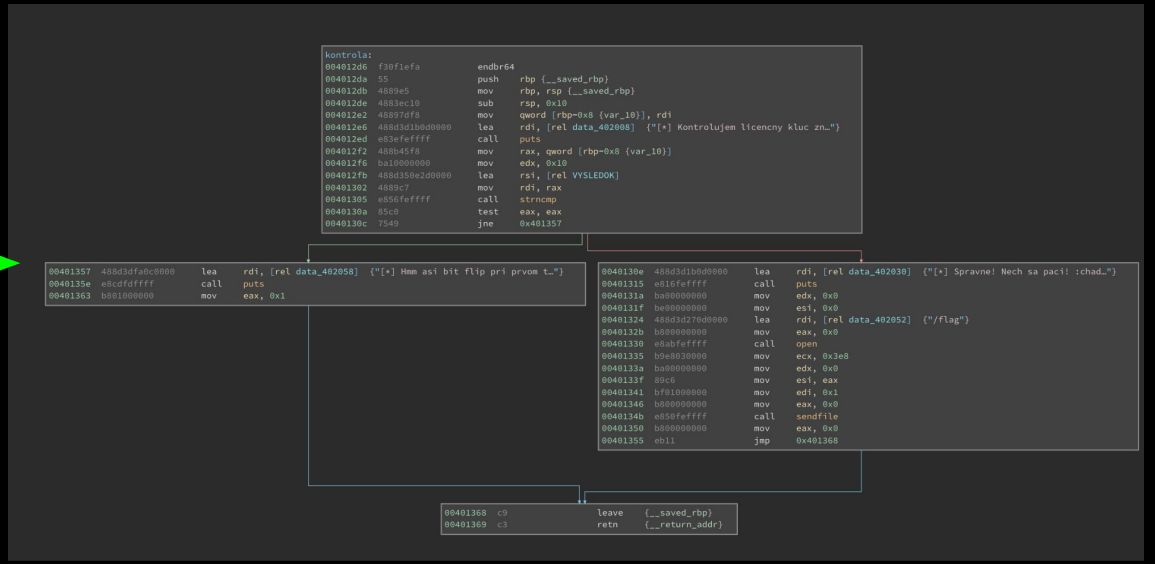
```
endbr64
push   rbp
mov    rbp,rsp
lea    rdi,[rip+0xeac]
mov    eax,0x0
call   1050 <printf@plt>
mov    eax,0x0
pop    rbp
ret
```

>CFG

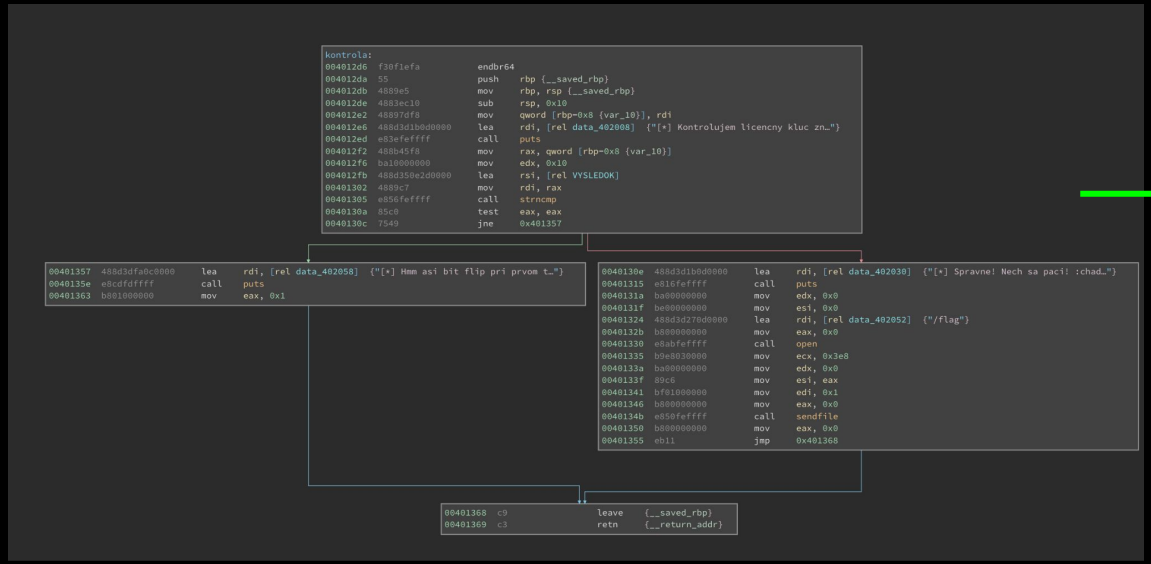
>Control Flow Graph

```
004012d6  endbr64
004012da  push    rbp {__saved_rbp}
004012db  mov     rbp, rsp {__saved_rbp}
004012de  sub     rsp, 0x10
004012e2  mov     qword [rbp-0x8 {var_10}], rdi
004012e6  lea     rdi, [rel data_402008]  {"[*] Kontrolujem licencny kluc zn…"}
004012ed  call    puts
004012f2  mov     rax, qword [rbp-0x8 {var_10}]
004012f6  mov     edx, 0x10
004012fb  lea     rsi, [rel VYSLEDOK]
00401302  mov     rdi, rax
00401305  call    strncmp
0040130a  test    eax, eax
0040130c  jne     0x401357

0040130e  lea     rdi, [rel data_402030]  {"[*] Spravne! Nech sa paci! :chad…"}
00401315  call    puts
0040131a  mov     edx, 0x0
0040131f  mov     esi, 0x0
00401324  lea     rdi, [rel data_402052]  {"/flag"}
0040132b  mov     eax, 0x0
00401330  call    open
00401335  mov     ecx, 0x3e8
0040133a  mov     edx, 0x0
0040133f  mov     esi, eax
00401341  mov     edi, 0x1
00401346  mov     eax, 0x0
0040134b  call    sendfile
00401350  mov     eax, 0x0
00401355  jmp     0x401368
```



>Dekompilátor




```
{
    puts("[*] Kontrolujem licencny kluc zn…");

    if (strncmp(arg1, &VYSLEDOK, 0x10))
    {
        puts("[*] Hmm asi bit flip pri prvom t…");
        return 1;
    }

    puts("[*] Spravne! Nech sa paci! :chad…");
    sendfile(1, open("/flag", 0, 0), 0, 0x3e8);
    return 0;
}
```

0x1e

>Úlohy

>Cieľom úloh bude zreverzovat binárky a nájsť licenčný kľúč
    >Nájsť komunikačný kanál
    >Zreverzovat transformácie a formát
>Analytické úlohy
    >Žiadne programovanie, t.j. odovzdať stručnú dokumentáciu

```
scp -i ./key -O hacker@feictf.xyz:/challenge/level1.0 .
```

>Veľa šťastia

deadline: 7.3.2025 13:37