Adventures in solving INSA Toulouse challenge

Julien Marchand <jmarchan@etud.insa-toulouse.fr>
Benoît Morgan <morgan@etud.insa-toulouse.fr>

INSA de Toulouse

30 avril 2021

Plan

Presentation

- Challenge proposed to the students attending to 5IR security training.
- Proposed by :
  - Éric Alata
  - Fernand Lone Sang
  - Vicent Nicomette

Objectives

- Guessing the name of a famous city

Guessing the name of a famous city

How ?

Download file to study wget, Chrome, IE6

Analyse the file file

Analyse network traces wireshark

Unix debugging tools objdump, gdb

Text editors vim

Neat hexa editor vim + xxd

a C compiler gcc

desassembler & decompiler IDA

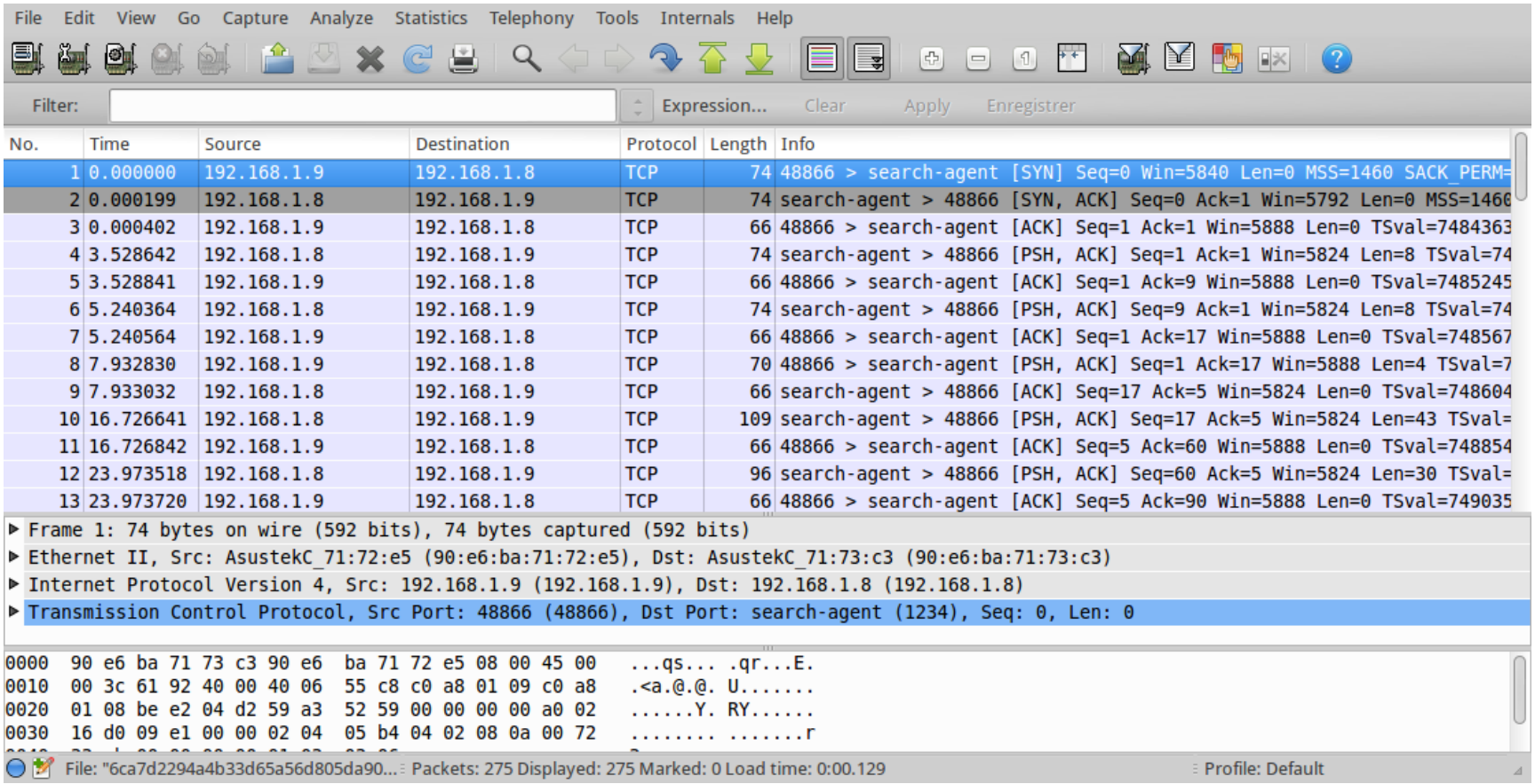
a scripting language which helps a bit PHP

Analysing mystery file

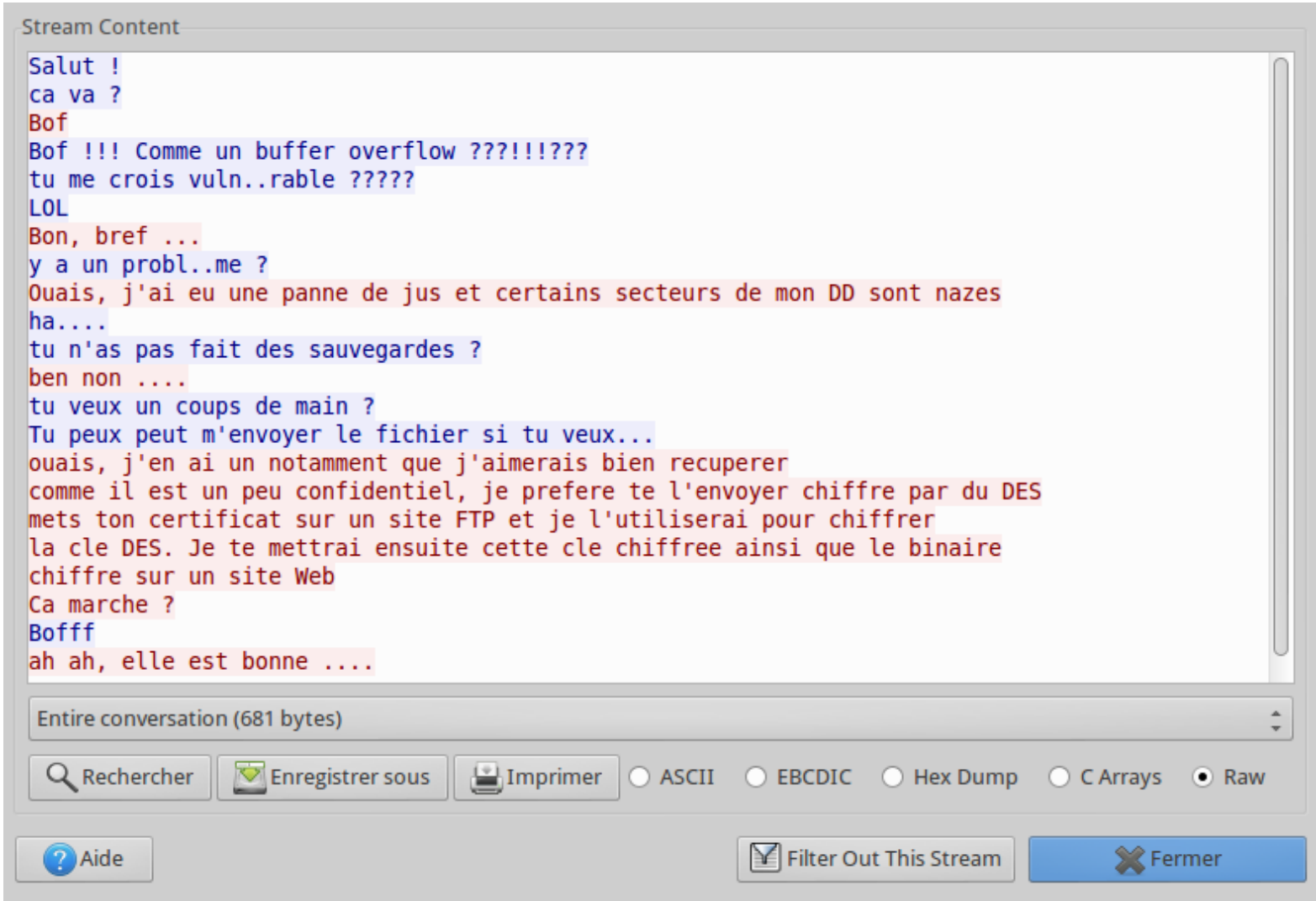
How about asking file ? (/etc/magic)

```
1 $ file 6ca7d2294a4b33d65a56d805da903297
2 6ca7d2294a4b33d65a56d805da903297: extended
    tcpdump capture file (little-endian) -
    version 2.4 (Ethernet, capture length 65535)
```

Wireshark


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Enregistrer
No. Time Source Destination Protocol Length Info
1 0.000000 192.168.1.9 192.168.1.8 TCP 74 48866 > search-agent [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=
2 0.000199 192.168.1.8 192.168.1.9 TCP 74 search-agent > 48866 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460
3 0.000402 192.168.1.9 192.168.1.8 TCP 66 48866 > search-agent [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSval=7484363
4 3.528642 192.168.1.8 192.168.1.9 TCP 74 search-agent > 48866 [PSH, ACK] Seq=1 Ack=1 Win=5824 Len=8 TSval=74
5 3.528841 192.168.1.9 192.168.1.8 TCP 66 48866 > search-agent [ACK] Seq=1 Ack=9 Win=5888 Len=0 TSval=7485245
6 5.240364 192.168.1.8 192.168.1.9 TCP 74 search-agent > 48866 [PSH, ACK] Seq=9 Ack=1 Win=5824 Len=8 TSval=74
7 5.240564 192.168.1.9 192.168.1.8 TCP 66 48866 > search-agent [ACK] Seq=1 Ack=17 Win=5888 Len=0 TSval=748567
8 7.932830 192.168.1.9 192.168.1.8 TCP 70 48866 > search-agent [PSH, ACK] Seq=1 Ack=17 Win=5888 Len=4 TSval=7
9 7.933032 192.168.1.8 192.168.1.9 TCP 66 search-agent > 48866 [ACK] Seq=17 Ack=5 Win=5824 Len=0 TSval=748604
10 16.726641 192.168.1.8 192.168.1.9 TCP 109 search-agent > 48866 [PSH, ACK] Seq=17 Ack=5 Win=5824 Len=43 TSval=
11 16.726842 192.168.1.9 192.168.1.8 TCP 66 48866 > search-agent [ACK] Seq=5 Ack=60 Win=5888 Len=0 TSval=748854
12 23.973518 192.168.1.8 192.168.1.9 TCP 96 search-agent > 48866 [PSH, ACK] Seq=60 Ack=5 Win=5824 Len=30 TSval=
13 23.973720 192.168.1.9 192.168.1.8 TCP 66 48866 > search-agent [ACK] Seq=5 Ack=90 Win=5888 Len=0 TSval=749035
Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: AsustekC_71:72:e5 (90:e6:ba:71:72:e5), Dst: AsustekC_71:73:c3 (90:e6:ba:71:73:c3)
Internet Protocol Version 4, Src: 192.168.1.9 (192.168.1.9), Dst: 192.168.1.8 (192.168.1.8)
Transmission Control Protocol, Src Port: 48866 (48866), Dst Port: search-agent (1234), Seq: 0, Len: 0
0000 90 e6 ba 71 73 c3 90 e6 ba 71 72 e5 08 00 45 00 ...qs... .qr...E.
0010 00 3c 61 92 40 00 40 06 55 c8 c0 a8 01 09 c0 a8 .<a.@.@. U.....
0020 01 08 be e2 04 d2 59 a3 52 59 00 00 00 00 a0 02 ......Y. RY.....
0030 16 d0 09 e1 00 00 02 04 05 b4 04 02 08 0a 00 72 ....r
File: "6ca7d2294a4b33d65a56d805da90... Packets: 275 Displayed: 275 Marked: 0 Load time: 0:00.129
Profile: Default

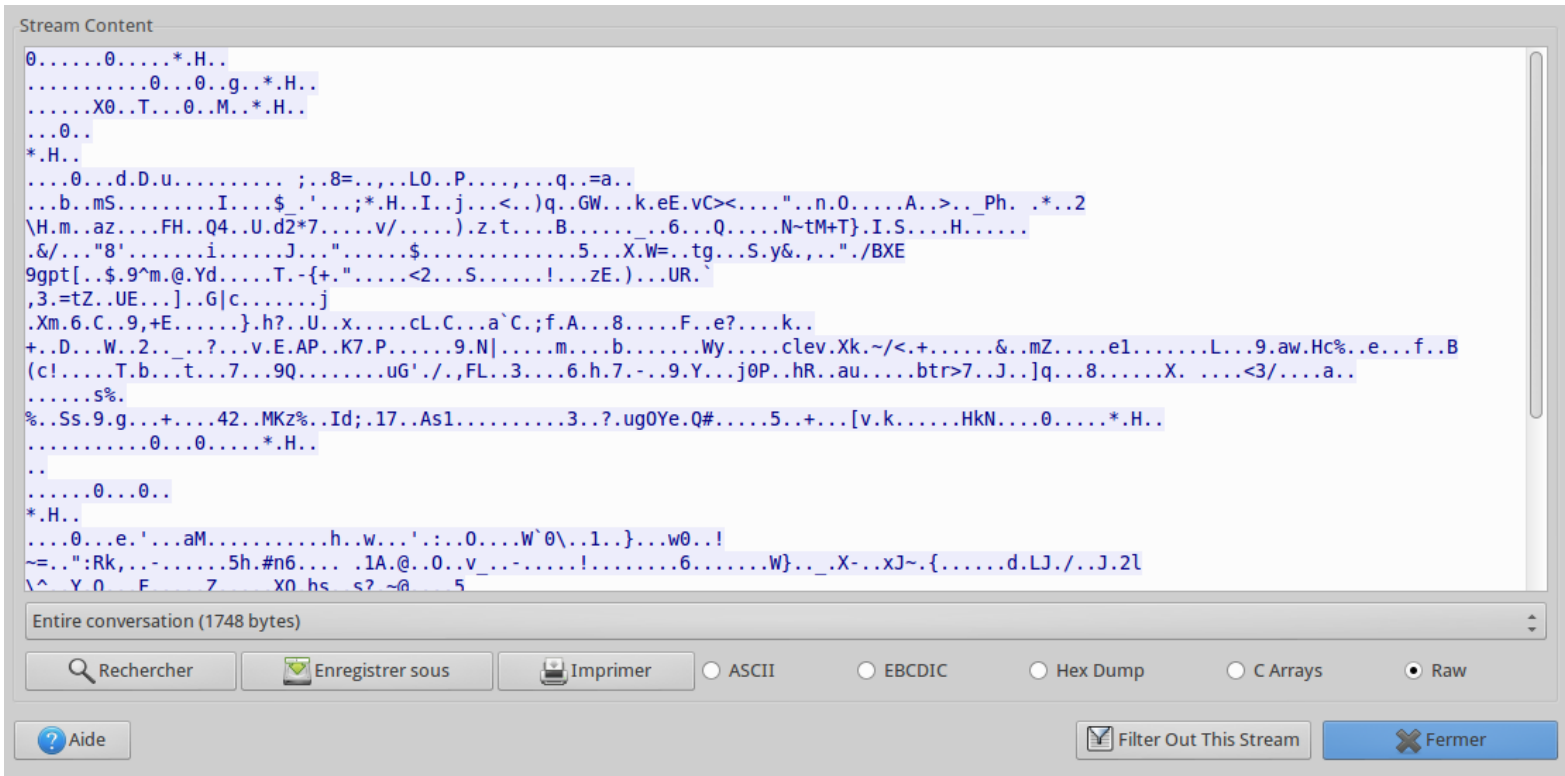
Conversation



Certificate

Protocol

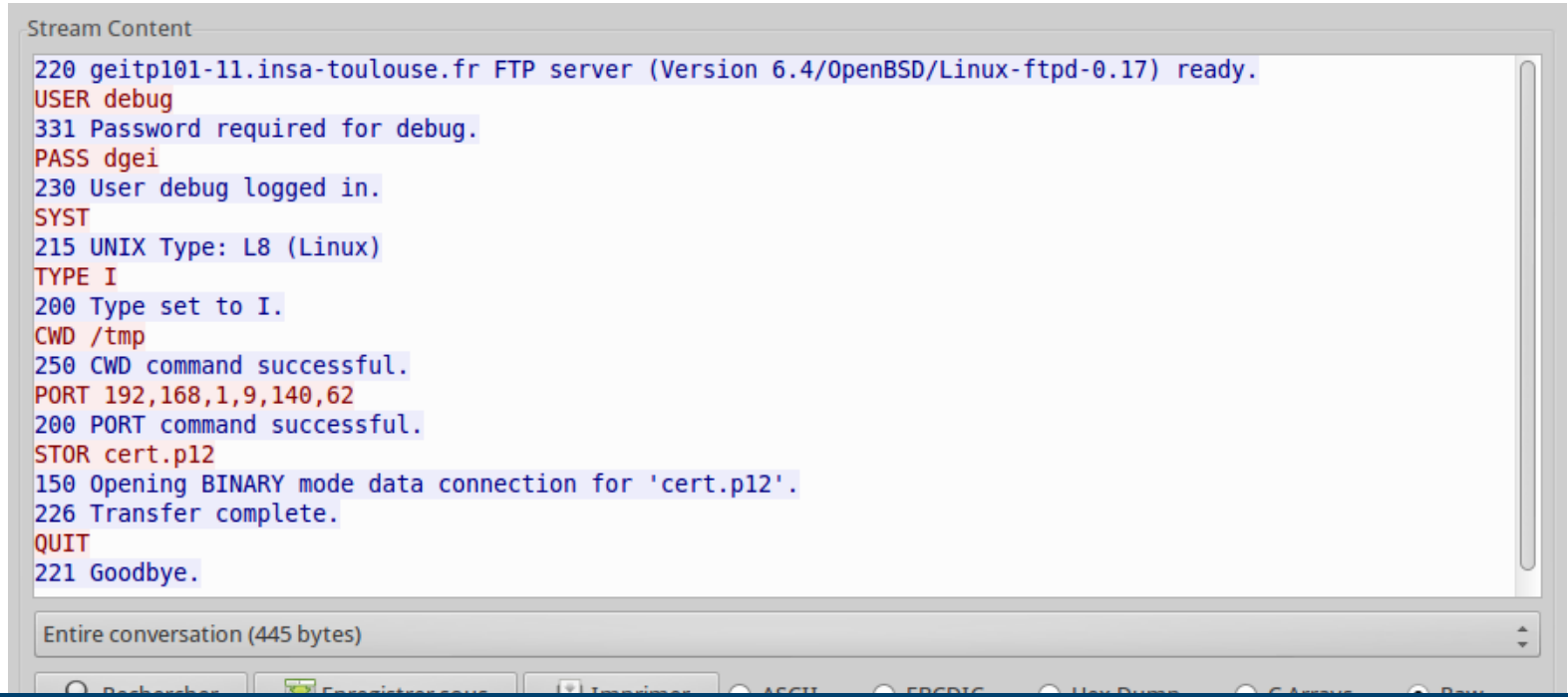
- File transfert via FTP in binary mode (TYPE I)



Certificate

Name of the file sent

- named `cert.p12`.
- p12... critical for the next part.



Encrypted target key and file

HTTP

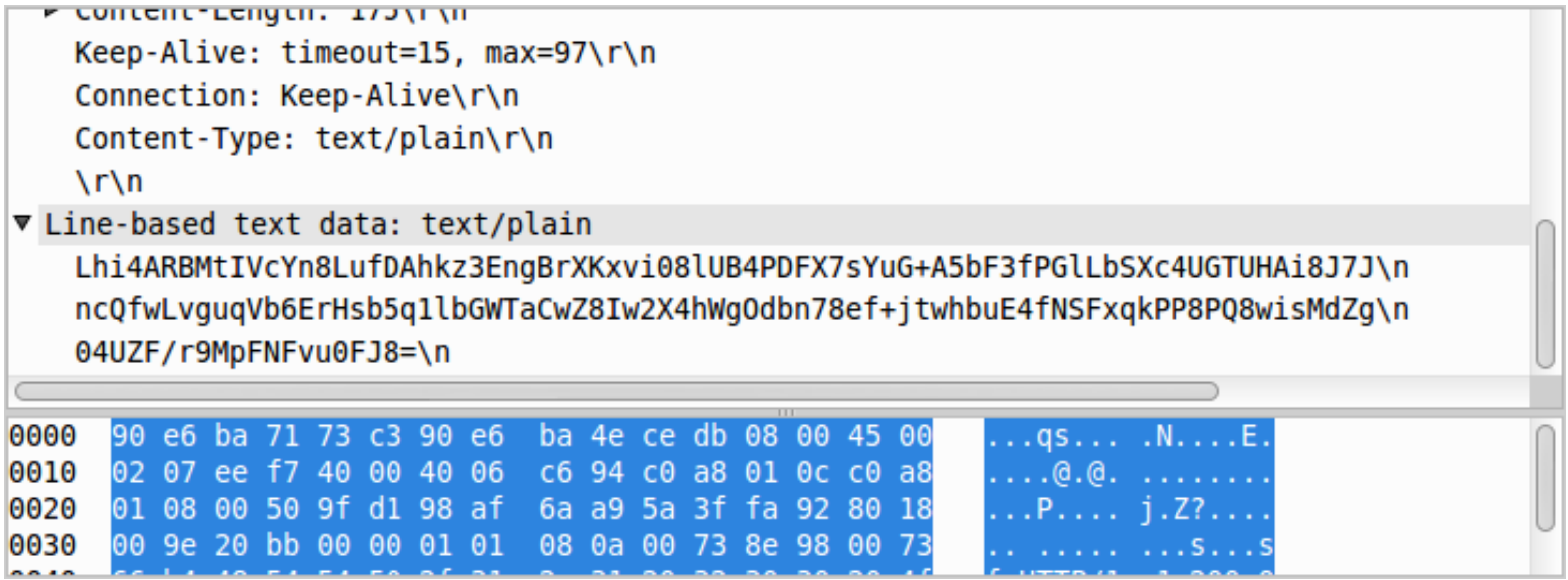
- GET / on 192.168.1.12

```
1 <html><head><meta charset="utf-8" /></head>
2 <body>
3 <P><b><A href=./cle_des.chiffree>La cle chiffree
    </A>
4 <P><b><A href=./fichier.chiffre>Le fichier
    chiffre</A>
5 </body></html>
```

Encrypted target key and file

HTTP

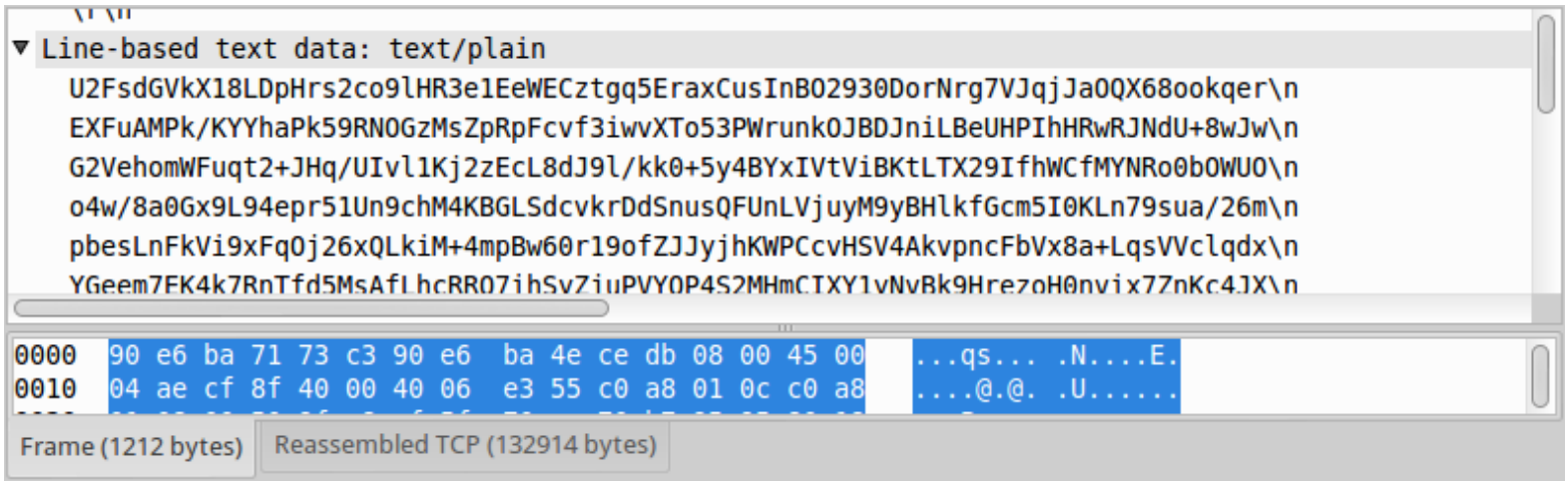
- GET /cle_des.chiffree on 192.168.1.12



Encrypted target key and file

HTTP

- GET /fichier.chiffre on 192.168.1.12



Pizza break

Certificate

PKCS12 container

- Store public keys...
- but also private ones
- convert PKCS12 to PEM

```
1 $ openssl pkcs12 -clcerts -in cert.p12 -out cert
    .pem
2 Enter Import Password:
3 MAC verified OK
4 Enter PEM pass phrase:
5 Verifying - Enter PEM pass phrase:
```

Decrypting DES secret key

cle_des.chiffree file

- Encrypted using RSA
- Base 64 encoded
- Base 64 decoding then RSA decryption using private key from cert.pem

```
1 $ base64 -d clef_chiffree > cle_bin
2 $ openssl rsautl -decrypt -inkey cert.pem -in
    cle_bin -out cle
3 Enter pass phrase for cert.pem:
4 $ cat cle
5 ce1scdes
```

Decrypting fichier chiffré

fichier.chiffre file

- Encrypted using DES
- base 64 encoded
- base 64 decoding then DES decryption using DES (ce1scdes)

```
1 $ base64 -d fichier > fichier_bin
2 $ openssl des-cbc -d -in fichier_bin -out bin
3 $ file bin
4 bin: ELF 32-bit LSB executable, Intel 80386,
    [...] dynamically linked, [...] for GNU/Linux
     2.6.15, [...] not stripped
```

Pizza break

Bad instruction

Assembly analysis

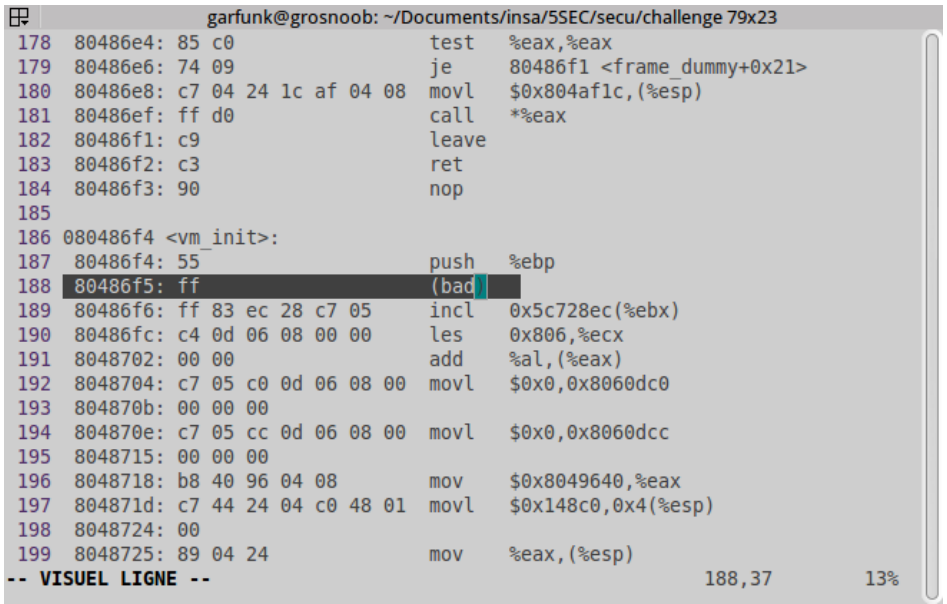
- Enter height hexadecimal characters
- Bad instruction ???

```
1 $ ./bin F00BAABA
2 Welcome in StUp12.9
3 Instruction non permise (core dumped)
```

Bad instruction

Assembly analysis

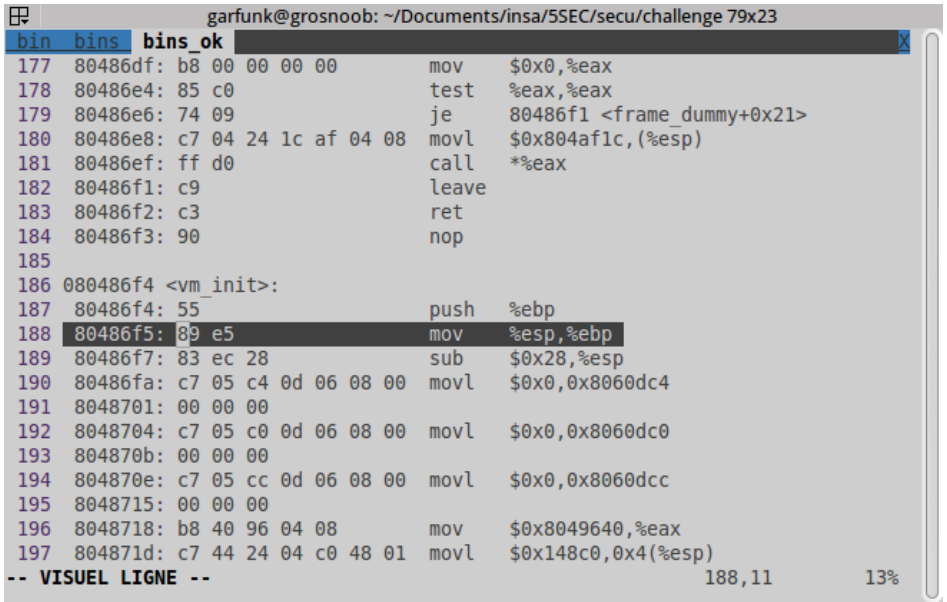
- Reading assembly code
- Is somebody bitflipping around :D



Bad instruction

Correcting defaults

- Stack frame setup is missing
- Replacing faulty instruction using `mov %esp,%ebp`



Page faults !

Segmentation WTF fault

- We still get another page fault

```
1 $ ./bin F00BAABA
2 Welcome in StUp12.9
3 rom size: 84160 bytes
4 ram size: 4194304 bytes
5 Erreur de segmentation (core dumped)
```

Page faults, page faults !

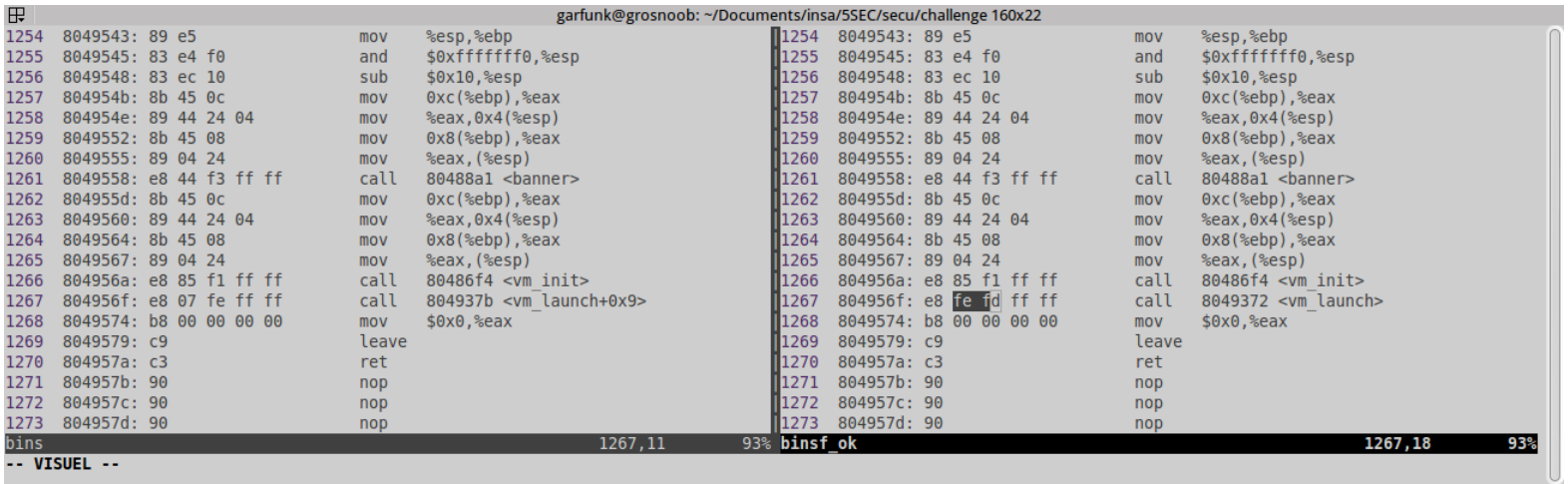
Assembly analysis

- gdb insights

Page faults, page faults, page faults !

Correction faulty instructions

- Function call to some faulty address
- Remplacing to a right branch address
- Those faulty hdd sectors :) ...



A virtual machine...

IDA disassembly

- Analysing essential functions
- vm_launch, get_op_a, get_op_b.
- A virtual machine executes a program using a custom ISA

vm_launch

```
.text:08049372                  public vm_launch
.text:08049372 vm_launch        proc near               ; CODE XREF: main+2D↓p
.text:08049372
.text:08049372 var_C            = dword ptr -0Ch
.text:08049372
.text:08049372                  push    ebp
.text:08049373                  mov     ebp, esp
.text:08049375                  sub     esp, 28h
.text:08049378                  jmp     loc_8049530
.text:0804937D ; ---------------------------------------------------------------------------
.text:0804937D
.text:0804937D loc_804937D:                             ; CODE XREF: vm_launch+1C8↓j
.text:0804937D                  mov     eax, ds:dword_8060DC0
.text:08049382                  mov     eax, dword ptr rom[eax*4]
.text:08049389                  shr     eax, 18h
.text:0804938C                  mov     [ebp+var_C], eax
.text:0804938F                  cmp     [ebp+var_C], 3
.text:08049393                  jnz     short loc_804939F
.text:08049395                  call    vm_exec_mov
.text:0804939A                  jmp     loc_8049523
.text:0804939F ; ---------------------------------------------------------------------------
.text:0804939F
.text:0804939F loc_804939F:                             ; CODE XREF: vm_launch+21↑j
.text:0804939F                  cmp     [ebp+var_C], 0Fh
.text:080493A3                  jnz     short loc_80493AF
.text:080493A5                  call    vm_exec_add
.text:080493AA                  jmp     loc_8049523
.text:080493AF ; ---------------------------------------------------------------------------
.text:080493AF
.text:080493AF loc_80493AF:                             ; CODE XREF: vm_launch+31↑j
```

vm_launch

Description

- Running over the ROM.
- Decoding opcodes.

get_op_a

```
.text:08048932                  public get_opa
.text:08048932 get_opa          proc near               ; CODE XREF: vm_exec_mov+E↓p
.text:08048932                                          ; vm_exec_afc+E↓p ...
.text:08048932
.text:08048932 var_18           = dword ptr -18h
.text:08048932 var_14           = dword ptr -14h
.text:08048932 var_10           = dword ptr -10h
.text:08048932 var_C            = dword ptr -0Ch
.text:08048932
.text:08048932                  push    ebp
.text:08048933                  mov     ebp, esp
.text:08048935                  push    ebx
.text:08048936                  sub     esp, 24h
.text:08048939                  mov     [ebp+var_C], 0
.text:08048940                  mov     eax, ds:dword_8060DC0
.text:08048945                  mov     eax, dword ptr rom[eax*4]
.text:0804894C                  and     eax, 3
.text:0804894F                  mov     [ebp+var_10], eax
.text:08048952                  mov     [ebp+var_14], 0
.text:08048959                  jmp     loc_8048A5A
.text:0804895E ; ---------------------------------------------------------------------------
.text:0804895E
.text:0804895E loc_804895E:                             ; CODE XREF: get_opa+12E↓j
.text:0804895E                  mov     eax, ds:dword_8060DC0
.text:08048963                  mov     edx, dword ptr rom[eax*4]
.text:0804896A                  mov     eax, [ebp+var_14]
.text:0804896D                  add     eax, 1
.text:08048970                  add     eax, eax
.text:08048972                  mov     ebx, edx
.text:08048974                  mov     ecx, eax
```

get_op_a

Description

- Operand decoding first operand from current instruction.
- Identical to get_op_b.

Pizza break

Studying ISA

vm_launch() main switch

- Instructions 4 double words sized (4×4 bytes)
- least significant byte from 1st double word : opcode
- 24 different opcodes :
  - MOV, AFC
  - JMP, JXX, CALL, RET, ZRET
  - ADD, SUB, MUL, DIV, MOD
  - NOT, AND, OR, XOR, SHIFTL, SHIFTR
  - LT
  - PRI, DUMP, OUT, DBG, ERROR

Studying ISA

```
1 void vm_launch() {
2   for (eip = 0; eip < 0x5230; eip += 4) {
3     int opcode = rom[eip] >> 24;
4     switch (opcode) {
5       case  0: vm_exec_pri();      break;
6       case  1: vm_exec_dump();     break;
7       case  2: vm_exec_out();      break;
8       case  3: vm_exec_mov();      break;
9       case  4: vm_exec_afc();      break;
10      case  5: vm_exec_jmp();      break;
11      case  6: vm_exec_jxx();      break;
12      case  7: vm_exec_ret();      break;
13      case  8: vm_exec_zret();     break;
14      case  9: vm_exec_call();     break;
15      case 10: vm_exec_shiftl();   break;
16      (...)
17      case 23: vm_exec_error();    break;
18      default:
19        fprintf(stderr, "Error vm_launch: opcode = %x\n", opcode);
20        exit(1);
21    }
22  }
23 }
```

Studying ISA

get_opa(), get_opb()

- Instructions can use up to 2 operands.
- Each operand is uses 1 to 3 double words (composition using sum).
- An operand can be up to 3 double words big.
- Memory access : direct or indirect
- 2 memory addressing modes :
  - Absolute
  - Relative to esp

Studying ISA

get_opa(), get_opb()

- The number of double words used by each operand and the addressing modes are encoded in the least significant bits from the first double word :
  - 2 bits : number of double words used by opA
  - For each double words composing opA, 2 bits : addressing mode used by this par of opA
  - 2 bits : number of double words used by opB
  - For each double words composing opB, 2 bits : addressing mode used by this par of opB

Studying ISA

```
 1 int get_opa() {
 2   int nbMotsOpA = rom[eip] & 3;
 3   int opA = 0;
 4
 5   for (int i = 0; i < nbMotsOpA; i++) {
 6     int tag = (rom[eip] >> 2 * (i + 1)) & 3;
 7     int partOpA = rom[eip + i + 1];
 8     switch (tag) {
 9       case 0: opA += esp + partOpA; break;
10       case 1: opA += partOpA; break;
11       case 2: opA += ram[esp + partOpA]; break;
12       case 3: opA += ram[partOpA]; break;
13       default:
14         fprintf(stderr, "Error vm_exec_mov: tag(a) = %x\n", tag);
15         exit(1);
16     }
17   }
18
19   return opA;
20 }
```

ROM disassembly

Creating a disassembler

- Reading ROM and decoding the instructions
- Displaying using mnemonics...
  - `JXX esp + 0x0000000c 5079`
- ... and pseudo-code
  - `if ram[esp + 0xc] then eip <- 0x4f5c (L001C)`
- Solving jump symbols and function calls
- Computing necessary elements to start reverse !

ROM disassembly

```
 1 AFC esp + 0x00000004 0x81847082                | ram[esp + 0x00000004] <- 0x81847082
 2 AFC esp + 0x00000005 0x11bb243b                | ram[esp + 0x00000005] <- 0x11bb243b
 3 AFC esp + 0x00000006 0xaa72594e                | ram[esp + 0x00000006] <- 0xaa72594e
 4 AFC esp + 0x00000007 0x21fe5662                | ram[esp + 0x00000007] <- 0x21fe5662
 5 AFC esp + 0x00000008 0x67452301                | ram[esp + 0x00000008] <- 0x67452301
 6 AFC esp + 0x00000009 0xefcdab89                | ram[esp + 0x00000009] <- 0xefcdab89
 7 AFC esp + 0x0000000a 0x98badcfe                | ram[esp + 0x0000000a] <- 0x98badcfe
 8 AFC esp + 0x0000000b 0x10325476                | ram[esp + 0x0000000b] <- 0x10325476
 9 AFC esp + 0x0000000c + 0x00000000 0x351ff482   | ram[esp + 0x0000000c + 0x00000000] <- 0x351ff482
10 ...
11 AFC esp + 0x00000f64 + 0x0000003f 0xeb86d391   | ram[esp + 0x00000f64 + 0x0000003f] <- 0xeb86d391
12 JMP 4025                                       | eip <- 0x3ee4 (L0000)
13 Lleftrotate:
14 AFC esp + 0x00000003 0x00000000                | ram[esp + 0x00000003] <- 0x00000000
15 AFC esp + 0x00000004 0x00000000                | ram[esp + 0x00000004] <- 0x00000000
16 MOV esp + 0x00000005 esp + 0xfffffffe          | ram[esp + 0x00000005] <- ram[esp + 0xfffffffe]
17 MOV esp + 0x00000000 esp + 0x00000005          | ram[esp + 0x00000000] <- ram[esp + 0x00000005]
18 MOV esp + 0x00000005 esp + 0xffffffff          | ram[esp + 0x00000005] <- ram[esp + 0xffffffff]
19 MOV esp + 0x00000001 esp + 0x00000005          | ram[esp + 0x00000001] <- ram[esp + 0x00000005]
20 MOV esp + 0x00000005 esp + 0x00000000          | ram[esp + 0x00000005] <- ram[esp + 0x00000000]
21 MOV esp + 0x00000006 esp + 0x00000001          | ram[esp + 0x00000006] <- ram[esp + 0x00000001]
22 SHL esp + 0x00000005 esp + 0x00000006          | ram[esp + 0x00000005] <<= ram[esp + 0x00000006]
23 MOV esp + 0x00000003 esp + 0x00000005          | ram[esp + 0x00000003] <- ram[esp + 0x00000005]
24 AFC esp + 0x00000005 0x00000020                | ram[esp + 0x00000005] <- 0x00000020
25 MOV esp + 0x00000006 esp + 0x00000001          | ram[esp + 0x00000006] <- ram[esp + 0x00000001]
26 SUB esp + 0x00000005 esp + 0x00000006          | ram[esp + 0x00000005] -= ram[esp + 0x00000006]
27 MOV esp + 0x00000002 esp + 0x00000005          | ram[esp + 0x00000002] <- ram[esp + 0x00000005]
28 MOV esp + 0x00000005 esp + 0x00000000          | ram[esp + 0x00000005] <- ram[esp + 0x00000000]
29 MOV esp + 0x00000006 esp + 0x00000002          | ram[esp + 0x00000006] <- ram[esp + 0x00000002]
30 SHR esp + 0x00000005 esp + 0x00000006          | ram[esp + 0x00000005] >>= ram[esp + 0x00000006]
31 MOV esp + 0x00000004 esp + 0x00000005          | ram[esp + 0x00000004] <- ram[esp + 0x00000005]
32 MOV esp + 0x00000005 esp + 0x00000003          | ram[esp + 0x00000005] <- ram[esp + 0x00000003]
```

JUMP ! JUMP ! JUMP ! JUMP !

- ROM starts by 4000 AFC (initialising memory)...
- Puis : `JMP 4025 | eip <- 0x3ee4 (L0000)`
- L0000 : `JMP 4224 | eip <- 0x4200 (L0001)`
- L0001 : `JMP 4280 | eip <- 0x42e0 (L000B)`
- L000B : `JMP 4321 | eip <- 0x4384 (L000C)`
- L000C : `JMP 4621 | eip <- 0x4834 (L000D)` (after 256 AFC)
- L000D : `JMP 5098 | eip <- 0x4fa8 (L0010)` (after 256 AFC)
- L0010 : `JMP 5129 | eip <- 0x5024 (L001D)`
- L001D : `JMP 5175 | eip <- 0x50dc (L0020)`
- L0020 : `JMP 5200 | eip <- 0x5140 (L0021)`

Identifying main procedure

```
 1 L0024:
 2 AFC esp + 0x000011a4 0x00000000
 3 MOV esp + 0x000011a5 esp + 0x00000000
 4 MOV ram[esp + 0x000011a4] + esp + 0x00000f12 esp + 0x000011a5
 5 AFC esp + 0x000011a5 0x00000001
 6 MOV esp + 0x000011a4 esp + 0x000011a5
 7 MOV esp + 0x000011a5 esp + 0x00000001
 8 MOV ram[esp + 0x000011a4] + esp + 0x00000f12 esp + 0x000011a5
 9 CALL 4026 4517
10 MOV esp + 0x000011a4 esp + 0x000011a5
11 CALL 4281 4517
12 MOV esp + 0x000011a4 esp + 0x000011a5
13 CALL 4578 4517
14 MOV esp + 0x000011a4 esp + 0x000011a5
15 CALL 4878 4517
16 MOV esp + 0x000011a4 esp + 0x000011a5
17 MOV esp + 0x000011a5 esp + 0x00000002
18 CALL 5130 4518
19 MOV esp + 0x000011a4 esp + 0x000011a6
20 MOV esp + 0x000011a5 esp + 0x000011a4
21 MOV esp + 0x0000000c + 0x00000001 esp + 0x000011a5
22 MOV esp + 0x000011a5 esp + 0x00000003
23 CALL 5130 4518
24 MOV esp + 0x000011a4 esp + 0x000011a6
25 MOV esp + 0x000011a5 esp + 0x000011a4
26 MOV esp + 0x0000000c + 0x00000002 esp + 0x000011a5
27 CALL 5099 4517
28 MOV esp + 0x000011a4 esp + 0x000011a5
29 CALL 5201 4517
30 MOV esp + 0x000011a4 esp + 0x000011a5
```

Identifying main procedure

```
 1 L0024:
 2 ram[esp + 0x000011a4] <- 0x00000000
 3 ram[esp + 0x000011a5] <- ram[esp + 0x00000000]
 4 ram[ram[esp + 0x000011a4] + esp + 0x00000f12] <- ram[esp + 0x000011a5]
 5 ram[esp + 0x000011a5] <- 0x00000001
 6 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
 7 ram[esp + 0x000011a5] <- ram[esp + 0x00000001]
 8 ram[ram[esp + 0x000011a4] + esp + 0x00000f12] <- ram[esp + 0x000011a5]
 9 func: Lmd5, eip <- 0x3ee8, esp <- esp + 0x11a5
10 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
11 func: Loutmd5, eip <- 0x42e4, esp <- esp + 0x11a5
12 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
13 func: Lrc4init, eip <- 0x4788, esp <- esp + 0x11a5
14 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
15 func: Lrc4enc, eip <- 0x4c38, esp <- esp + 0x11a5
16 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
17 ram[esp + 0x000011a5] <- ram[esp + 0x00000002]
18 func: Lint2str, eip <- 0x5028, esp <- esp + 0x11a6
19 ram[esp + 0x000011a4] <- ram[esp + 0x000011a6]
20 ram[esp + 0x000011a5] <- ram[esp + 0x000011a4]
21 ram[esp + 0x0000000c + 0x00000001] <- ram[esp + 0x000011a5]
22 ram[esp + 0x000011a5] <- ram[esp + 0x00000003]
23 func: Lint2str, eip <- 0x5028, esp <- esp + 0x11a6
24 ram[esp + 0x000011a4] <- ram[esp + 0x000011a6]
25 ram[esp + 0x000011a5] <- ram[esp + 0x000011a4]
26 ram[esp + 0x0000000c + 0x00000002] <- ram[esp + 0x000011a5]
27 func: Loutdata, eip <- 0x4fac, esp <- esp + 0x11a5
28 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
29 func: Ltestmd5, eip <- 0x5144, esp <- esp + 0x11a5
30 ram[esp + 0x000011a4] <- ram[esp + 0x000011a5]
```

Identifying main procedure

Translation in C-vilized language

```
1  void main() {
2    TAB_0xf12[0] = VAL_0x0;
3    TAB_0xf12[1] = VAL_0x1;
4
5    md5();
6    outmd5();
7    rc4init();
8    rc4enc();
9    VAL_0xd = int2str(VAL_0x2);
10   VAL_0xe = int2str(VAL_0x3);
11   outdata();
12   testmd5();
13 }
```

Reverse ALL the functions !

Reverse des fonctions

- Example : *int2str*
- LIVE Demo !
- Eg `listings/int2str.asm`, `listings/int2str_1.c`, `listings/int2str_2.c`, `listings/int2str_3.c`, `listings/int2str_4.c`.

Analysing main procedure

WTF is this ?

- The key k is 4 bytes long : 0xaabbccdd.
- It is partitioned in two parts.
- The first part (aabb) is a secret key which has been used to cipher a file using RC4 primitive.
- This file is decrypted using `rc4init()` and `rc4enc()` functions. The decrypted result is written is 0xbeef file by `outdata()` function.
- If the key is incorrect, the VM program displays the following message : "Error while executing VM : bad key !".

Analysing main procedure

Why an hash function for ?

- A quasi identical to MD5 hash function is used to verify integrity of decoded file. It is done by verifying the secret key a using a sort of key derivation function.
- The ROM file contains an expected value h computed as : $h = \mathcal{H}(k[1:0]||p[14 << 2 : 2 << 2])$ (byte unit)
- Padding value p is stored in the binary.
- Decryption is verified by comparing expected h to the computed one using user given key.

Analysing main procedure

What about the remaining part of the key ?

- The second part of the key is converted to a decimal ASCII string representation.
- Then it is written in the second and the third position of the decrypted file. To an expected position as in a template.
- We guess that it will be used to decrypt once again some data in the decrypted file in a next step...

Looking for the first part of the key...

How to get the first part of the key ?

- 2 bytes = 65536 possibilities.
- A brute force seem now reasonable !
- But execution in the VM is pretty (too) slow (1 minute per attempt).
- That waitable but we want to win :D
- Solution : extract the reversed md5 key check and brute force it on hardware.

Getting help from PHP !

```
1  <?php
2  include 'fake_md5.php';
3
4  $message = array(
5    0x31313131, 0x32323232, 0x33333333, 0x34343434,
6    0x35353535, 0x36363636, 0x37373737, 0x38383838,
7    0x39393939, 0x30303030, 0x31313131, 0x32323232,
8    0x33333333, 0x80343434, 0x000001b8, 0x00000000
9  );
10
11 $hash = sprintf("%08x", reorder(0x81847082)) .
12         sprintf("%08x", reorder(0x11bb243b)) .
13         sprintf("%08x", reorder(0xaa72594e)) .
14         sprintf("%08x", reorder(0x21fe5662));
15
16 for ($i = 0; $i <= 255; $i++) {
17   for ($j = 0; $j <= 255; $j++) {
18     $message[0] = $i;
19     $message[1] = $j;
20     if (fake_md5($message) == $hash) {
21       printf("%02x%02x", $i, $j);
22       break 2;
23     }
24   }
25 }
```

Looking for the first part of the key...

Banzaï !

```
1 $ php bruteforce_1.php
2 3320
3 $ ./challenge.bin 3320ffff
4 Welcome in StUp12.9
5 [...]
6 Executing Ltestvalue
7 $ file 0xbeef
8 0xbeef: ASCII text, with very long lines
```

Pizza break

0xbeef file... what's in there ?

```
 1 /K [ 255 255 ] def % deuxieme partie de la cle
 2 /D [
 3   % code PS a dechiffrer avec RC4
 4 ] def
 5 /makestring {dup length string dup /NullEncode
       filter 3 -1 roll {1 index
 6 exch write} forall flushfile } def
 7 /E { S j S i get S i S j get put put } def
 8 /init { /S 0 1 255 {} for 256 array astore def /
       j 0 def 0 1 255 { /i exch def
 9 /j j S i get add K i K length mod get add 256
       mod def E } for } def
10 /rcfour {
11 /i 0 def /j 0 def [ D { /i i 1 add 256 mod def /
       j j S i get add 256 mod def E S
12 S i get S j get add 256 mod get xor } forall D
       length array astore /D exch def
13 pop } def
14 init rcfour D makestring cvx exec
```

Looking for the second half of the key...

A key for ?

- We meet again with the decimal representation of the second part of the user entered secret key from the last step..
- It is used to decrypt Postscript code from within `/D` variable.
- Obviously without the right key, file execution fails (`exec`).
- 65536 possibilities of small file RC4 decryption, one can brute force efficiently.

Looking for the second half of the key...

Decryption oracle :

```
1 <?php
2 $ret = 0;
3 $ps = file_get_contents('0xbeef');
4
5 for ($i = 255; $i >= 0; $i--) {
6   for ($j = 255; $j >= 0; $j--) {
7     $ps = preg_replace('#/K \[ [0-9]+ [0-9]+ \] def#', "/K [ $i $j ] def", $ps);
8     file_put_contents("0xbeef.ps", $ps);
9     passthru("ps2pdf 0xbeef.ps >/dev/null 2>&1", $ret);
10    if ($ret == 0) {
11      printf("%02x%02x", $i, $j);
12      exit;
13    }
14  }
15 }
```

But fails to allow extraction of cleartext

Stealing cleartext

"Ma préférée, c'est PHP !"

- Once the second half of the key is broken, (**0xbb2b**) using our oracle, we can extract the content from the Postsript file (`/D`) and decrypt it once again using... PHP
- We rely on PHP *mcrypt* library to decrypt RC4 ciphertext...
- to get another Postscript file to move on to the next step :)

Pizza break

The secret lies into this image

Open image using evince

Postscript file structure of the image

- image.ps

```
1 %!PS
2 126 16 scale
3 126 16 8 [ 126 0 0 -16 0 16 ]
4 { <
5 %DATA%
6 > }
```

Postscript file structure of the image

Informations

- Width : 126 pixels.
- Height : 16 pixels.
- Image type : 8 bits grey scale.

First intuition

Steganography...

LSB

- LSB : Least Significant Bit.
- LSB from height consecutive pixels are forming a byte containing information to extract.
- As you can see on the former image, human eye cannot distinguish from two blacks : 0x01 and 0x00.

Image secret data

Aide

- The first gray pixel (**0x38**) is likely giving the size of the data to be excracted in bytes.
- We then have to consider **LSB** from **0x1c0** bytes.

C extraction

- img.h

```
1 char lol[] = {0x38, 0xfe, 0xfe, 0xff, 0xff, 0xfe
    , 0xfe, 0xff, 0xfe, 0xff, 0xfe, 0xff, 0xfe, 0
    xfe, 0x01, 0xff, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
    , 0xfe, 0xff, 0xfe, 0xfe, 0xff, 0xff, 0xfe, 0
    xfe, 0xff, 0xff, 0xff, 0xfe, 0xff, 0xfe, 0xff
    , 0xfe, 0xfe, 0xff, 0xff, 0xfe, 0xff, 0xff, 0
    xfe, 0xfe, 0xfe, 0xff, 0xff, 0xfe, 0xfe, 0xff
    , 0xfe, 0xfe, 0xff, 0xff, 0xff, 0xfe, 0xff, 0
    xfe /*...*/};
```

C extraction

- img.c

```
1 #include <stdio.h>
2 #include "img.h"
3
4 void lsb() {
5   unsigned char *img = lol;
6   unsigned char i, n = *(img++);
7   unsigned int cpt = 0;
8   for (i = 0; i < n; i++) {
9     unsigned char j, c = 0;
10    for (j = 0; j < 8; j++) {
11      c |= (*(img++) & 1) << j;
12      cpt++;
13    }
14    printf("%c", c);
15  }
16  printf("total : %u chars.\n", cpt);
17 }
18
19 void main() {
20  lsb();
21 }
```

C extraction

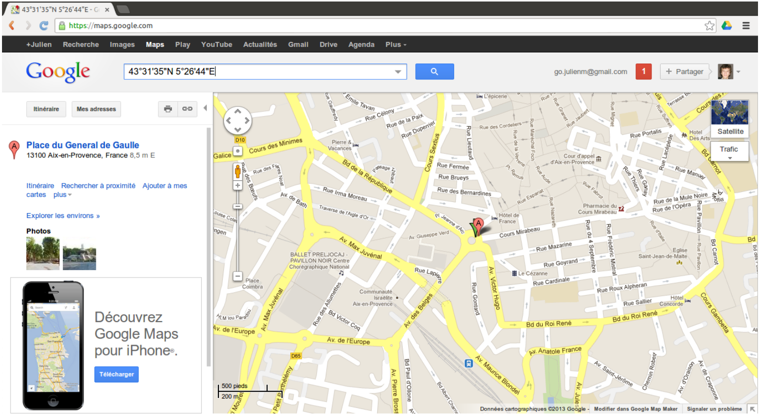
- Out

```
1 $ gcc img.c && ./a.out
2 Le secret est ce qui se trouve en 43o31'35"N 5
    o26'44"E.
3 total : 448 chars.
```

Last pizza break

Conclusion

Mr. Alata dreams to move to...



Conlusion

- That was fun !

Conlusion

- That was fun !
- But a bit too easy :)