T2’10 Challenge

Mathieu GASPARD

Prerequisite

What I used to complete the challenge:

- Python 2.5
- Python Crypto (http://www.dlitz.net/software/pycrypto/)
- Wireshark (http://www.wireshark.org/)
- DTK Barcode Reader demo (http://www.dtksoft.com/index.php)
- VirtualBox (http://www.virtualbox.org/)
- IDA disassembler (http://www.hex-rays.com/)

Level 1

The file for level 1 (t210-challenge-level1.pgm) is, as its extension states, a PGM image file (http://en.wikipedia.org/wiki/Netpbm_format).

A PGM file is actually a text file which can be opened with any text editor and thus easily modified.

Each value in purple represents a single pixel, with its value ranging from 0 to 255

```
P2
2886 7
255
0255 255 255 255 255 255 255 255
 255 255 255 255 255 255   0   0
   0   0   0   0   0   0 255 255
 255 255   0   0   0   0 255 255
 255 255 255 255 255   0   0   0
   0 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255
 255   0   0   0   0 255 255 255
 255 255 255 255 255   0   0   0
   0   0   0   0 255 255 255 255
 255 255 255 255 255 255 255 255
 255 255 255 255   0   0   0   0
 255 255 255 255   0   0   0   0
 255 255 255 255 255 255 255   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0 255
 255 255 255   0   0   0   0 255
 255 255 255 255 255 255 255   0
   0   0   0 255 255 255 255 255
 255 255   0   0   0   0   0 00
   0   0   0   0   0   0   0   0
   0   0 255 255 255 255   0   0
   0   0 255 255 255 255 255 255
 255   0   0   0   0 255 255 255
 255   0   0   0   0 255 255 255
 255 255 255 255 255   0   0   0
   0   0   0   0 00    0   0   0
   0   0   0   0   0 255 255 255
 255 255 255 255   0   0   0   0
   0   0   0   0   0   0   0   0
 255 255 255 255 255 255 255 255
   0   0   0   0 255 255 255 255
 255 255 255 255   0   0   0   0
   0   0   0 255 255 255 255   0
   0   0   0 255 255 255 255   0
   0   0   0   0   0   0   0   0
   0   0   0 255 255 255 255 255
 255 255   0   0   0   0   0   0
   0   0 255 255 255 255 255 255
 255 255   0   0   0   0 0255 255
 255 255   0   0   0   0   0   0
   0   0   0   0   0   0 255 255
 255 255 255 255 255   0   0   0
```

P2 : magic that indicates a PGM file

2886 7 : Width Height of the image (7 lines and 2886 columns)

255 : Maximum value of each pixel (so 0 to 255)

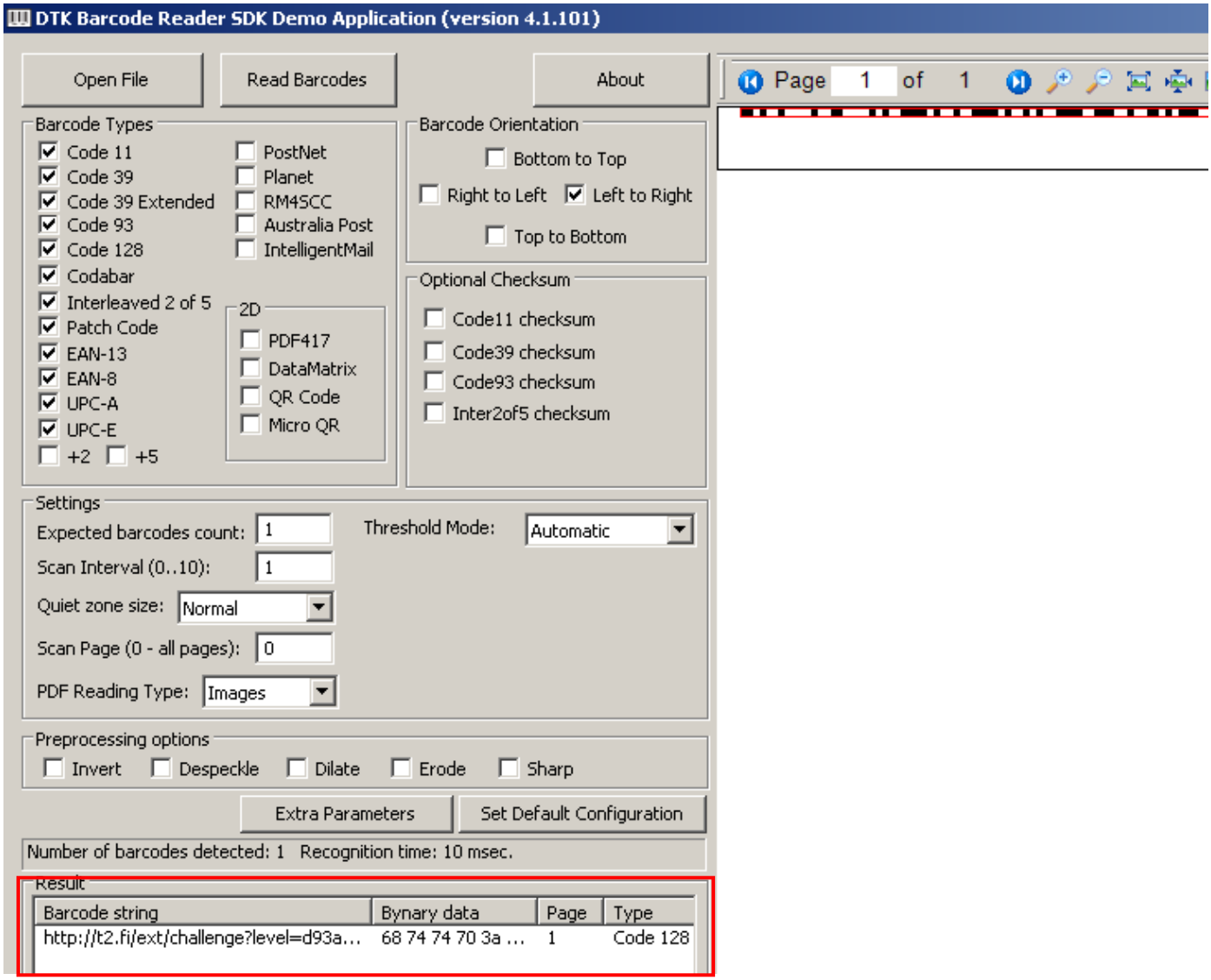
0255 255… : pixels of the image

First, I believed there was some kind of steganography in the image (Least Significant Byte method for example, or bytes with a leading 0 in them: 00 and 0255) but those trails ended up being dead ends.

By looking at the image, one can see that it looks like a barcode.

I first thought that barcodes could represent only figures, but they can also represent text, with the “code 128” for example. (see http://www.adams1.com/128code.html)

I used “DTK Barcode Reader SDK demo” (see “Prerequisites”) to decode the barcode:



The software reveals the challenge URL along with the type of the barcode (Code 128).

Let’s verify it manually:

A Code 128 barcode looks like this:

(source http://www.adams1.com/128code.html)

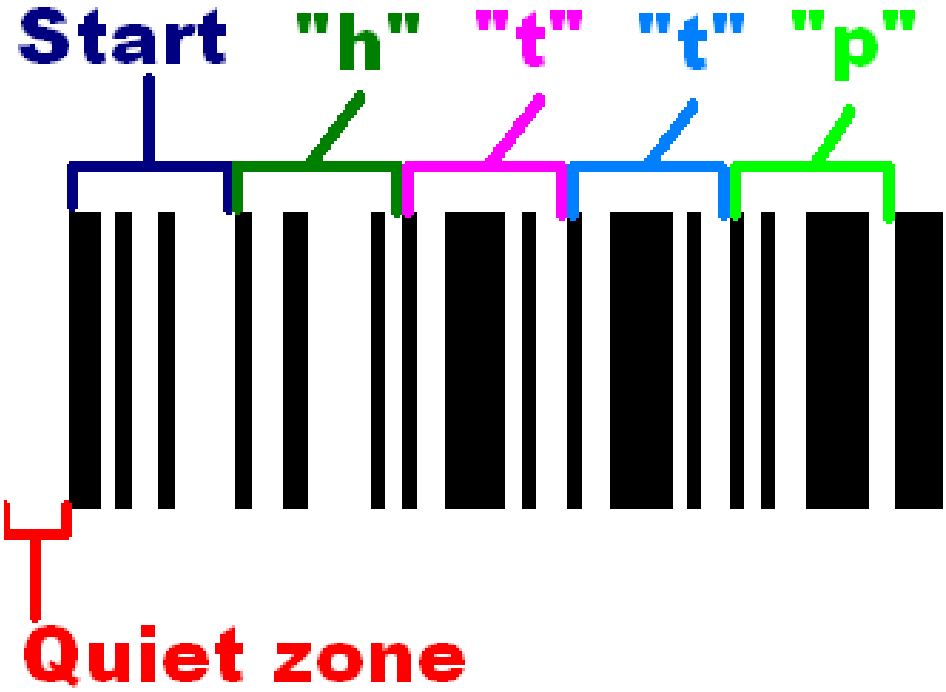
Characters encoding are the following (let’s concentrate on the « h », « t » and « p » of « http »)

The letter “h” is encoded as: 1 Black, 2 spaces, 2 Black, 4 spaces, 1 Black and 1 space

Value	Code A	Code B	Code C	Pattern B S B S B S	
70	ACK	f	70	1 1 2 4 1 2	f (ASCII 102)
71	BEL	g	71	1 2 2 1 1 4	g (ASCII 103)
72	BS	h	72	1 2 2 4 1 1	h (ASCII 104)
73	HT	i	73	1 4 2 1 1 2	i (ASCII 105)
74	LF	j	74	1 4 2 2 1 1	j (ASCII 106)
75	VT	k	75	2 4 1 2 1 1	k (ASCII 107)
76	FF	l	76	2 2 1 1 1 4	l (ASCII 108)
77	CR	m	77	4 1 3 1 1 1	m (ASCII 109)
78	SO	n	78	2 4 1 1 1 2	n (ASCII 110)
79	SI	o	79	1 3 4 1 1 1	o (ASCII 111)
Value	**Code A**	**Code B**	**Code C**	**Pattern B S B S B S**	
80	DLE	p	80	1 1 1 2 4 2	p (ASCII 112)
81	DC1	q	81	1 2 1 1 4 2	q (ASCII 113)
82	DC2	r	82	1 2 1 2 4 1	r (ASCII 114)
83	DC3	s	83	1 1 4 2 1 2	s (ASCII 115)
84	DC4	t	84	1 2 4 1 1 2	t (ASCII 116)

(source http://www.adams1.com/128table.html)

When applied to the challenge image, it gives the following:



Level 2

The file for level 2 is a VirtualBox disk image.

```
# strings t210-level2.vdi |head
<<< Sun VirtualBox Disk Image >>>
sQOtN2
t+a`j
Invalid partition table
Error loading operating system
Missing operating system
MSDOS5.0
NO NAME    FAT16   3
```

One can very easily pass this level with the following command:

```
# strings t210-level2.vdi |grep -i "http"

<</Subtype/Link/Rect[ 54.45 745.96 370.03 771.4] /BS<</W 0>>/F
4/A<</Type/Action/S/URI/URI(http://t2.fi/ext/challenge?level=95f05a22b
9694edf20fd5bf5ddcc8e9f) >>>>
```

But let’s get a little bit deeper.

If one mounts the image in a virtual machine, only 1 file is visible: t210_level2.txt and it doesn’t contain anything useful.

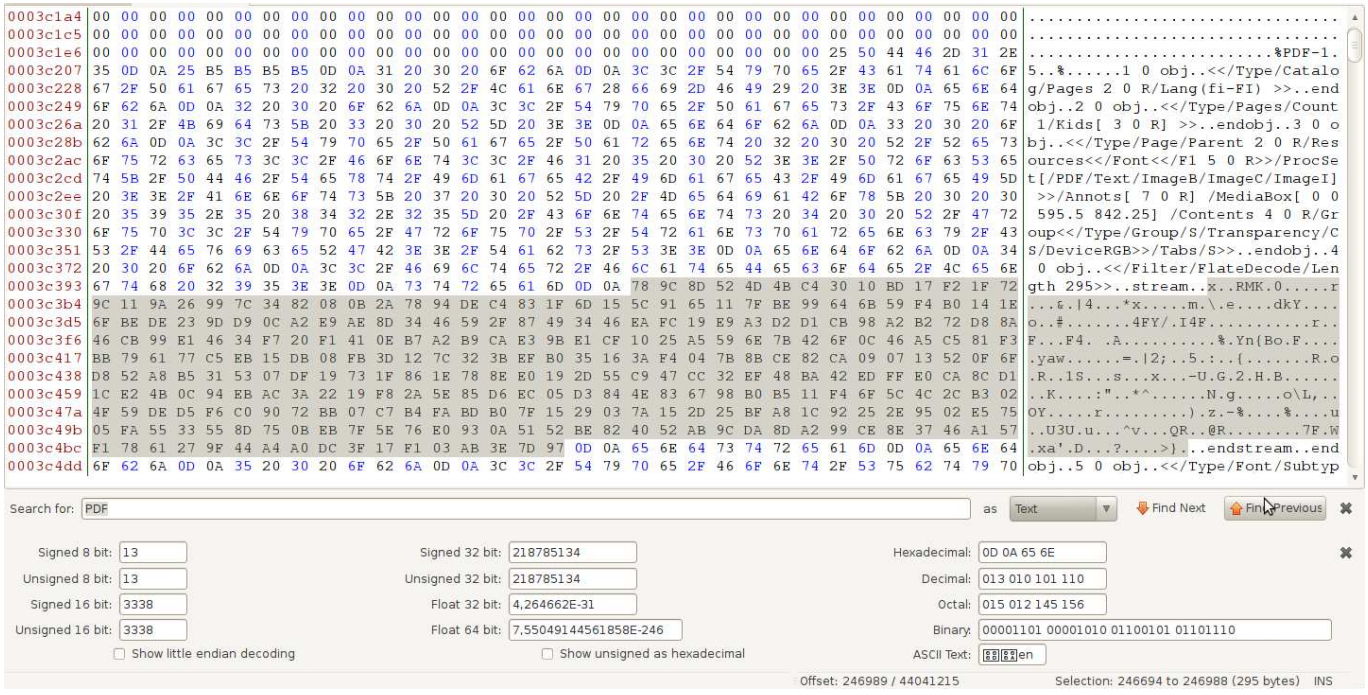
```
debian:~# mount /dev/sdb1 /tmp/tmp/
debian:~# ll -a /tmp/tmp/
total 21
drwxr-xr-x 2 root root 16384 jan  1  1970 .
drwxrwxrwt 5 root root  4096 aoû 31 09:32 ..
-rwxr-xr-x 1 root root    59 mai 16 21:01 t210_level2.txt
debian:~# cat /tmp/tmp/t210_level2.txt
THANK YOU MARIO!

BUT OUR PRINCESS IS IN
ANOTHER CASTLE!debian:~# _
```

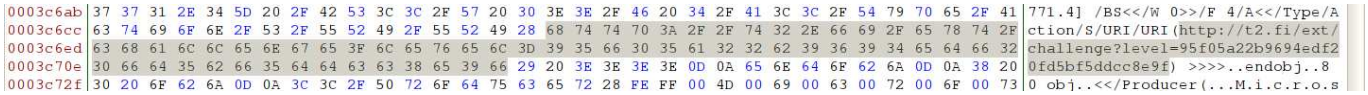
A “strings” on the binary shows strings relative to PDF files:

```
11 0 obj
<</Type/XRef/Size 11/W[ 1 4 2] /Root 1 0 R/Info 8 0 R/ID[<02730414E9C9A0409E3B1F4012A84C1E><02730414E9C9A0409E3B1F4012A84C1E>] /Filter/FlateDecode/Length 50>>
stream
endstream
endobj
xref
0 12
0000000000 65535 f
0000000017 00000 n
0000000078 00000 n
0000000134 00000 n
0000000369 00000 n
0000000738 00000 n
0000000905 00000 n
0000001144 00000 n
0000001325 00000 n
0000001540 00000 n
0000001796 00000 n
0000064761 00000 n
trailer
<</Size 12/Root 1 0 R/Info 8 0 R/ID[<02730414E9C9A0409E3B1F4012A84C1E><02730414E9C9A0409E3B1F4012A84C1E>] >>
startxref
65010
%%EOF
xref
trailer
<</Size 12/Root 1 0 R/Info 8 0 R/ID[<02730414E9C9A0409E3B1F4012A84C1E><02730414E9C9A0409E3B1F4012A84C1E>] /Prev 65010/XRefStm 64761>>
startxref
65406
%%EOF
%PDF-1.5
1 0 obj
<</Type/Catalog/Pages 2 0 R/Lang(fi-FI) >>
endobj
2 0 obj
<</Type/Pages/Count 1/Kids[ 3 0 R] >>
endobj
3 0 obj
<</Type/Page/Parent 2 0 R/Resources<</Font<</F1 5 0 R>>/ProcSet[/PDF/Text/ImageB/ImageC/ImageI] >>/Annots[ 7 0 R] /MediaBox[ 0 0 595.5 842.25] /Contents 4 0 R/
GB>>/Tabs/S>>
endobj
```

Using a hex editor, one can easily find the PDF file (the highlighted part is some compressed text, I’ll come back to this later):

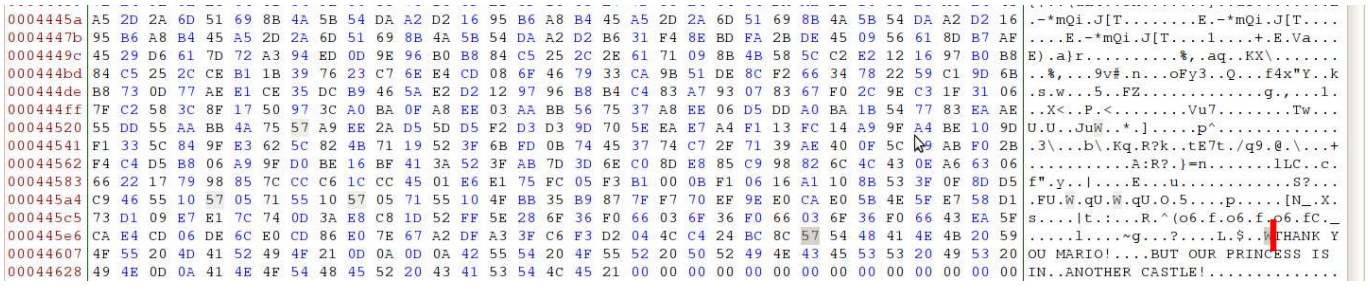


And the URL we found before:

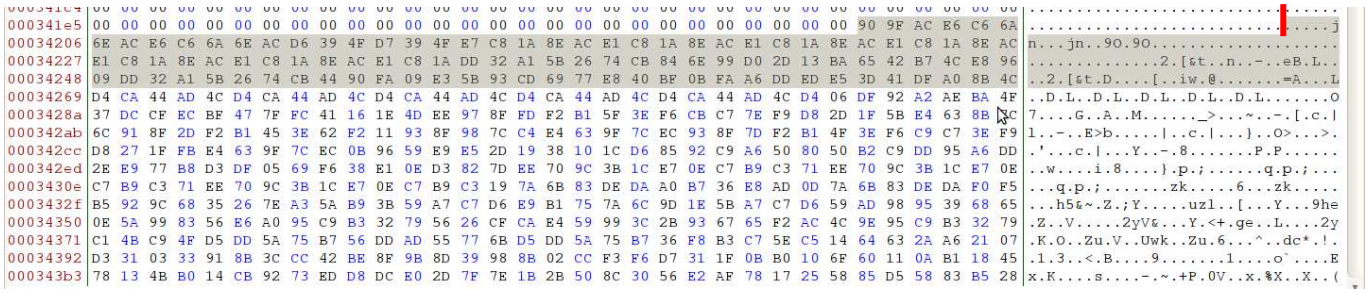


The PDF file is split into 2 parts into the image (file was probably fragmented when written on the FAT partition):

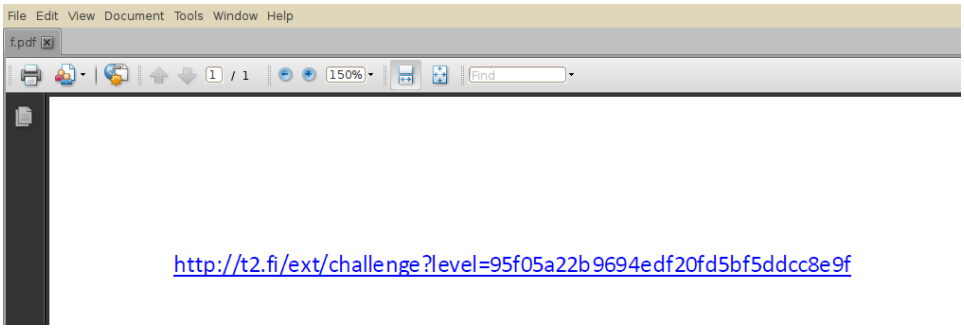
End of part 1 (red line, just before the content of the text file):



Beginning of part 2 (red line)



We can reconstruct the full PDF and open it:



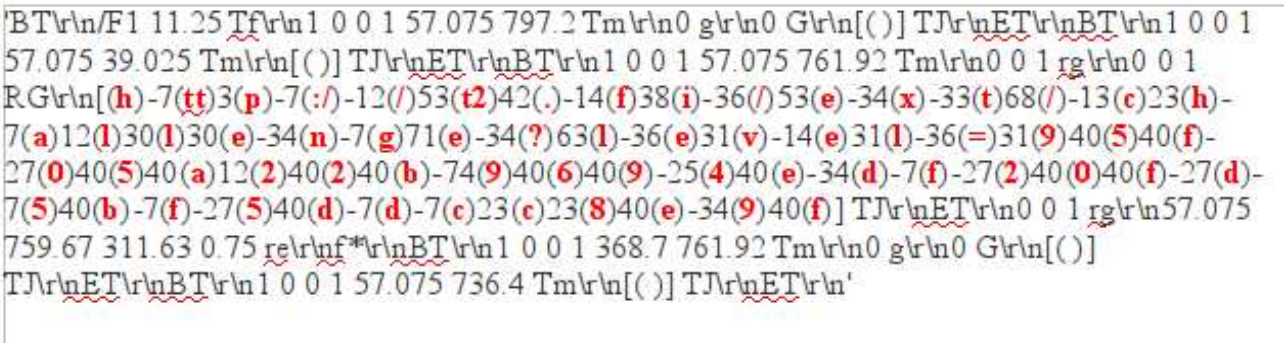
If the PDF could not have been reconstructed or if the URL had not been in cleartext, the level could still be validated by decompressing a compressed string which contained the URL (the printed text actually).

This compressed string is the one highlighted a few blocks before. It is manually copied into “/tmp/stream.bin” and then decompressed using python.

```
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Current history file (/home/mga/.py_history) size: 41414 bytes, 965 lines.

>>> a=open("/tmp/stream.bin","r").read()
>>> import zlib
>>> zlib.decompress(a)
'BT\r\n/F1 11.25 Tf\r\n1 0 0 1 57.075 797.2 Tm\r\n0 g\r\n0 G\r\n[( )] TJ\r\nET\r\nBT\r\n1 0 0 1 57.075 39
(p)-7(:/)-12(/)53(t2)42(.)-14(f)38(i)-36(/)53(e)-34(x)-33(t)68(/)-13(c)23(h)-7(a)12(l)30(l)30(e)-34(n)-7(
40(6)40(9)-25(4)40(e)-34(d)-7(f)-27(2)40(0)40(f)-27(d)-7(5)40(b)-7(f)-27(5)40(d)-7(d)-7(c)23(c)23(8)40(e)
.92 Tm\r\n0 g\r\n0 G\r\n[( )] TJ\r\nET\r\nBT\r\n1 0 0 1 57.075 736.4 Tm\r\n[( )] TJ\r\nET\r\n'
>>>
```

One can clearly see the challenge URL (in red)

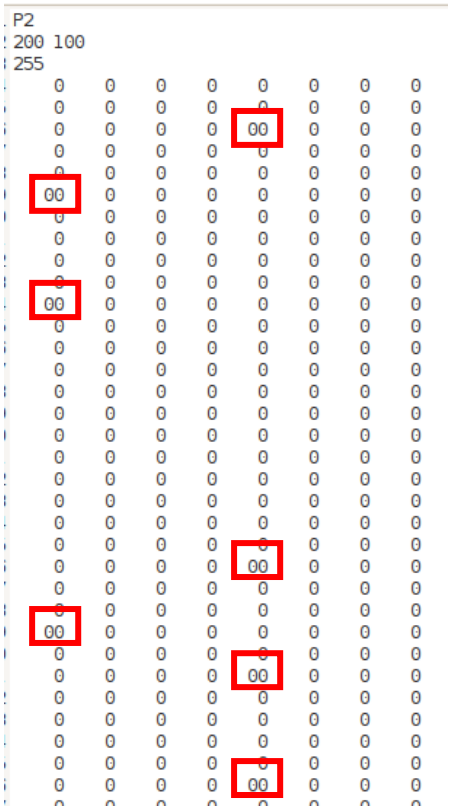


Level 3

Once again, this level is about PGM files.

This time, it’s not an obvious barcode, but a classic image (kudo to Mario again!)

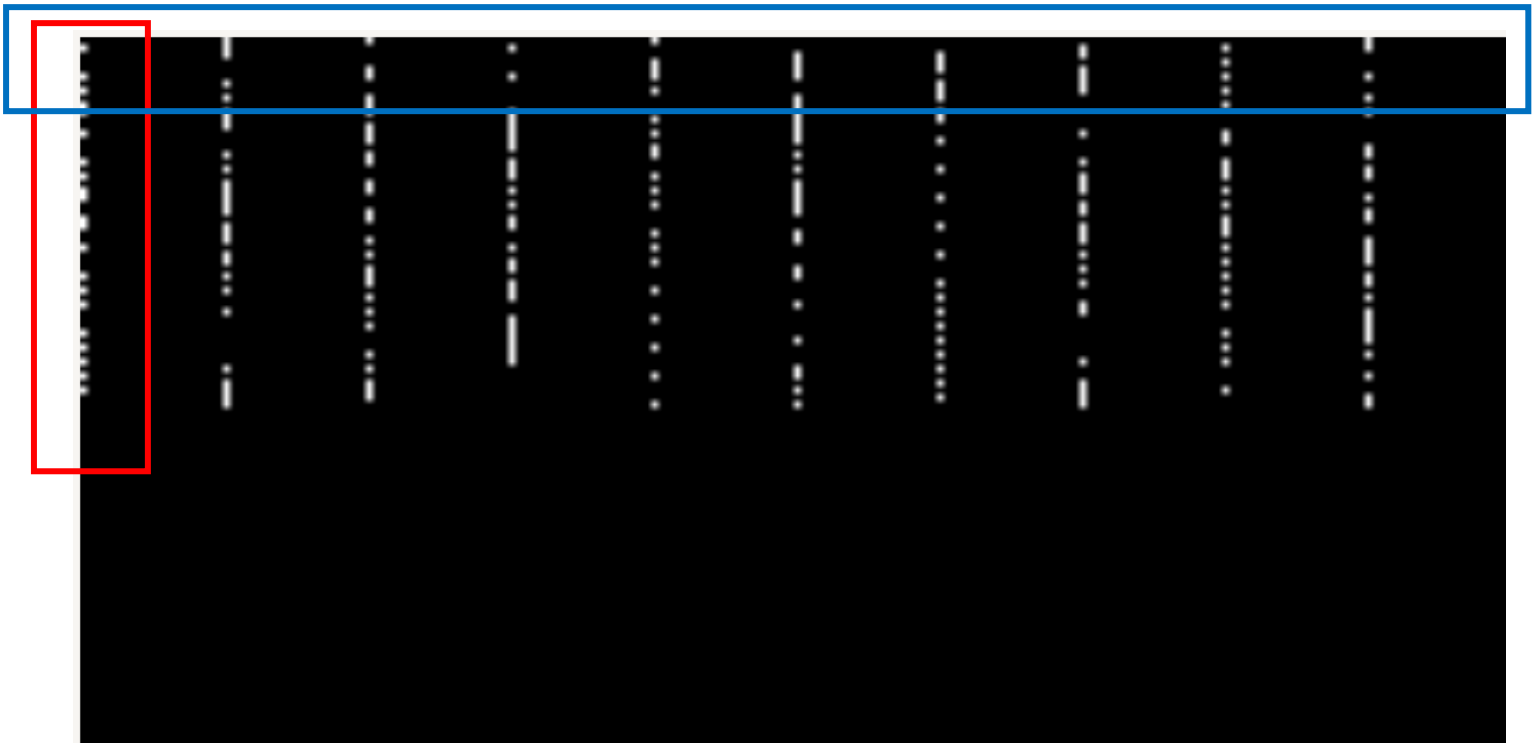
In the first image (level1), I had already seen “strange bytes” with a leading 0 (“00” and “0255” instead of plain “0” and “255”).



This time, to make them “visible”, I changed the bytes “00” to “0255” (actually black pixel to white pixel), and it gives the following:

Wow, again it looks like barcodes!

Let’s remove the text that hinders us: replace “255” by “0” (white pixel to black pixel)



First, I thought it was again a barcode and I tried to assemble the different columns into a line (in a new PGM file) and make it recognized by the barcode reader software, but I soon discovered that there was no “quiet zone” and “start zone” in those lines, so it couldn’t be a barcode.

This time again, I tried to search for steganography in the image with those ”strange” bytes.

With the following python code, I retrieve the data for the 10 “special” columns.(like the one squared in red)

```
a=open("t210-level3-modified2.pgm","r").read().split('\n')
width = int(a[1].split(" ")[0])
height = int(a[1].split(" ")[1])
data = [i for i in "".join(a[3:]).strip().split(' ') if i != ""]
print "w : %d, h : %d"%(width, height)
columns = []
# get data per column
for i in range(width):
    columns.append([])
    for j in range(height):
        columns[i].append(data[width*j + i])

# keep only 10 columns with "special" bytes (i.e : with "00" or "0255"
in it)
good_columns = [i for i in columns if "00" in i or "0255" in i]
```

Now I can test if each pixel represents a bit of a character

Good_columns[0] is the column squared in red in the previous screenshot. “0” will represent a 0 bit, “0255” a 1 bit.

```
>>> good_columns[0]

['0', '0255', '0', '0', '0', '0255', '0', '0255', '0', '0255', '0255',
'0', '0', '0255', '0', '0', '0', '0255', '0', '0255', '0', '0255',
'0255', '0', '0', '0255', '0255', '0', '0', '0255', '0', '0', '0',
'0255', '0', '0255', '0', '0255', '0', '0', '0', '0255', '0', '0255',
'0', '0255', '0', '0255', '0', '0255', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0']

>>> chr(int("01000101",2))

'E'

>>>
```

Apparently, going with the columns is not the good answer.

Why not try this but with the lines instead (squared in blue), but with the values of only the “good columns”: There are 10 good columns and 100 lines, hence 1000 bytes of data, which can represent 1000 bit.

```
# now get the lines instead of the columns
# put all lines in 1 list (1000 elements long, because 10 columns of
100 elements each)
lines = []
for j in range(len(good_columns[0])):
    for i in range(len(good_columns)):
        lines.append(good_columns[i][j])
```

Let’s test now (we test only the first 2 characters):

```
>>> lines[:16]
['0', '0255', '0255', '0', '0255', '0', '0', '0', '0', '0255', '0255',
'0255', '0', '0255', '0', '0']
>>> chr(int("01101000",2))
'h'
>>> chr(int("01110100",2))
't'
>>>
```

The hypothesis seems to be confirmed (first 2 characters == “ht” as in “http”). Let’s print the full hidden string:

```
# decode characters (8 bits per char)
char = ""
s = ""
for i in range(len(lines)):
    tmp = int(lines[i])
    if tmp == 255:
        char += "1"
    elif tmp == 0:
        char += "0"
    if len(char) == 8:
        s += chr(int(char,2))
        char = ""


>> s
'http://t2.fi/ext/challenge?level=96ef65c5937b2af4a2d1fb2dfb1c9f55\x00
\x00\x00\x00\x00\x00….\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00'
>>>
```

BINGO !

Level 4

This level is about decrypting a message:

```
I can't find the good old decryption tool on my new computer! Can you help, please? I need you to decrypt the below message for me ASAP! I'm pretty sure I've used the same encryption key (my birthdate) for the last five years...

b6 22 4a 16 5e 60 dd 79 0f c5 cd cb 8a fc 48 cd 87 00 61 cb e0 1a e1 e1 dd b8 b5 f8 67 12 d8 7c 25 cd ab f2 f1 2b 83 b0 45 b5 18 c1 45 1b 60 6c ee 1a bc b8 f9 39 c2 fd 3d b0 ff 51 8d 41 6e 01 3e 4c 3d c3 44 34 17 2a
```

The encrypted message is 72 bytes long. I first thought of block ciphers and ruled out stream ciphers because the encrypted message probably contains the challenge URL, which is 65 bytes long (without any trailing character).

Furthermore, it is stated in the mail that the encryption key is the user birthdate.

It probably is in the format “MMDDYYYY” (or DDMMYYYY or YYYYDDMM or YYYYMMDD) so most probably 8 bytes long (considering ASCII encoding, no UTF-8 encoding).

In this case, AES and 3DES are also ruled out because of the size of the key.

In the obvious ciphers left, DES and Blowfish were the most probable candidates.

If the birthdate had been encoded in UTF-8, the encryption key would have been 16 bytes long, and AES-128 would have been a possible match.

If there had been “garbage” at the end of the encrypted URL (to pad it to 72 bytes), a stream cipher would have been possible too.

In the bruteforce, I had to test the 4 types of date format described above. (date, date2, date3, date4).

To test if the decryption was successful, I simply searched for “http” in the decrypted string.

I tested Blowfish first and got lucky quickly:

```
#!/usr/bin/env python

from Crypto.Cipher import *

enc =
"\xb6\x22\x4a\x16\x5e\x60\xdd\x79\x0f\xc5\xcd\xcb\x8a\xfc\x48\xcd\x87\x00\x61
\xcb\xe0\x1a\xe1\xe1\xdd\xb8\xb5\xf8\x67\x12\xd8\x7c\x25\xcd\xab\xf2\xf1\x2b\
x83\xb0\x45\xb5\x18\xc1\x45\x1b\x60\x6c\xee\x1a\xbc\xb8\xf9\x39\xc2\xfd\x3d\x
b0\xff\x51\x8d\x41\x6e\x01\x3e\x4c\x3d\xc3\x44\x34\x17\x2a"

day = 1
month = 1
year = 1940

for day in range(1,31):
  for month in range(1,12):
    for year in range(1940,1980):
        date = "%02d%02d%s"%(day,month,year)
        date2 = "%02d%02d%s"%(month,day,year)
        date3 = "%s%02d%02d"%(year,day,month)
        date4 = "%s%02d%02d"%(year,month,day)
        for the_date in [date,date2,date3,date4]:
            for mode in [Blowfish.MODE_CBC, Blowfish.MODE_CFB,
Blowfish.MODE_CTR, Blowfish.MODE_ECB, Blowfish.MODE_OFB, Blowfish.MODE_PGP]:
                a = Blowfish.new(the_date)
                b = a.decrypt(enc)
                if "http" in b:
                    print "%s (date : %s)"%(b,the_date)
```

```
# time ./decrypt.py
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
http://t2.fi/ext/challenge?level=8b57c5946ee283fea01e8646e7c1ebf9 (date : 19630115)
^CTraceback (most recent call last):
  File "./decrypt.py", line 23, in <module>
    a = Blowfish.new(the_date)
KeyboardInterrupt

real  0m9.118s
user  0m9.113s
sys   0m0.000s
```

Key found in less than 10 seconds of bruteforce ☺ (The user is born on 15th January 1963)

Level 5

On the last level, a pcap file is provided.

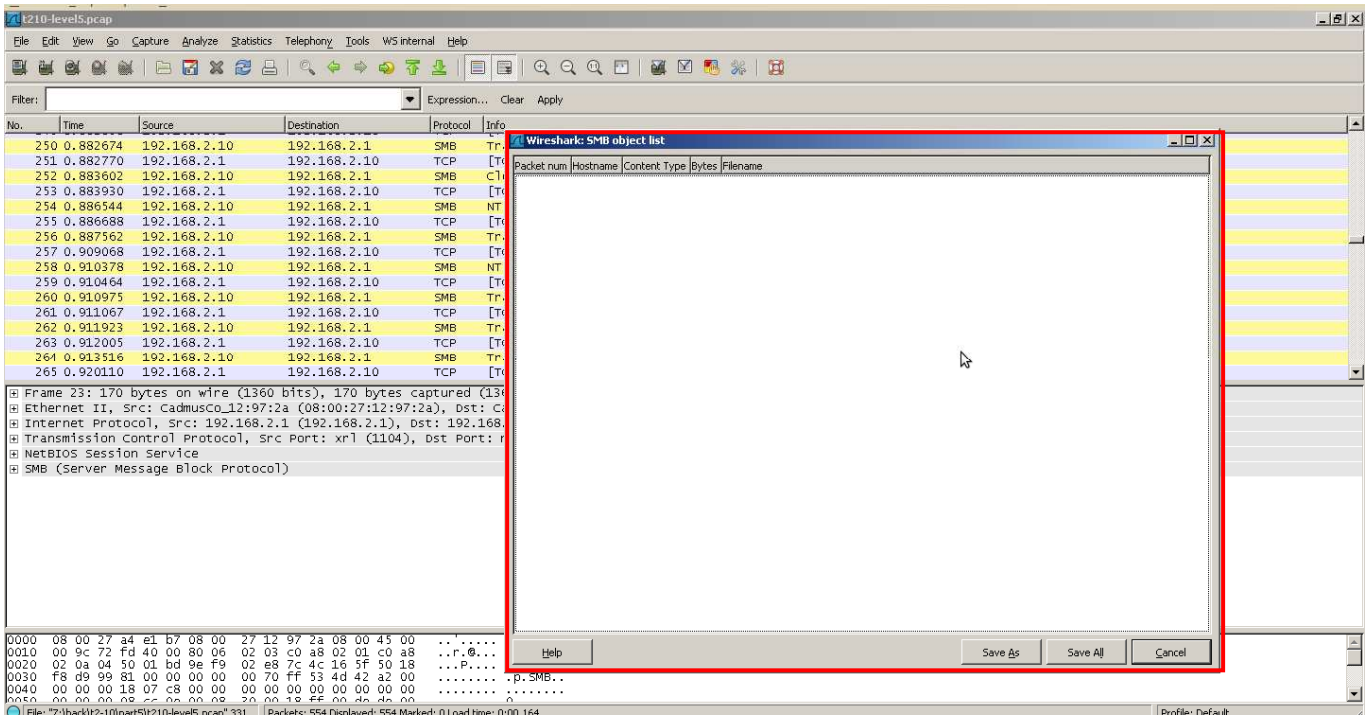
After investigation, it appears that the machine 192.168.2.1 used psexec (http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx) to launch binary “t210test.exe” (located remotely in \System32\) on remote machine 192.168.2.10 (which runs Windows Small Business Server).

Psexec permits the execution of binary remotely (using the CLI) and get the output of the command on the local computer.

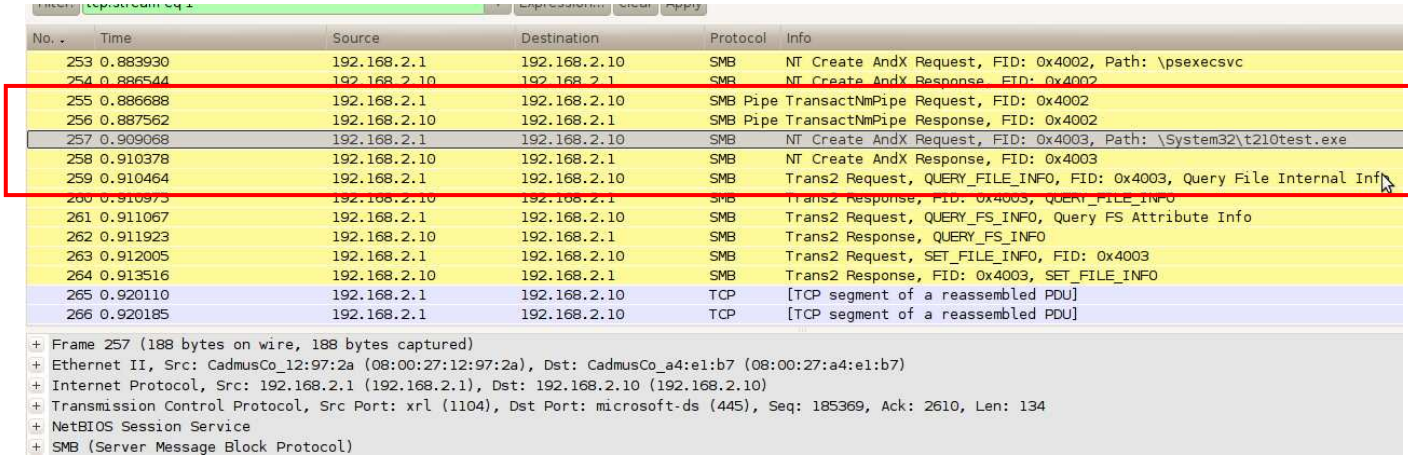
To be able to launch the binary on the remote server, the binary has to be copied on it, so it is transferred using SMB.

This is how we will be able to retrieve it:

First, I thought of using a development version of Wireshark (1.5 svn) which permits the retrieval of files transmitted through SMB (see http://blog.taddong.com/2010/05/capturing-smb-files-with-wireshark.html) but it didn’t work (Wireshark didn’t see any file transferred):



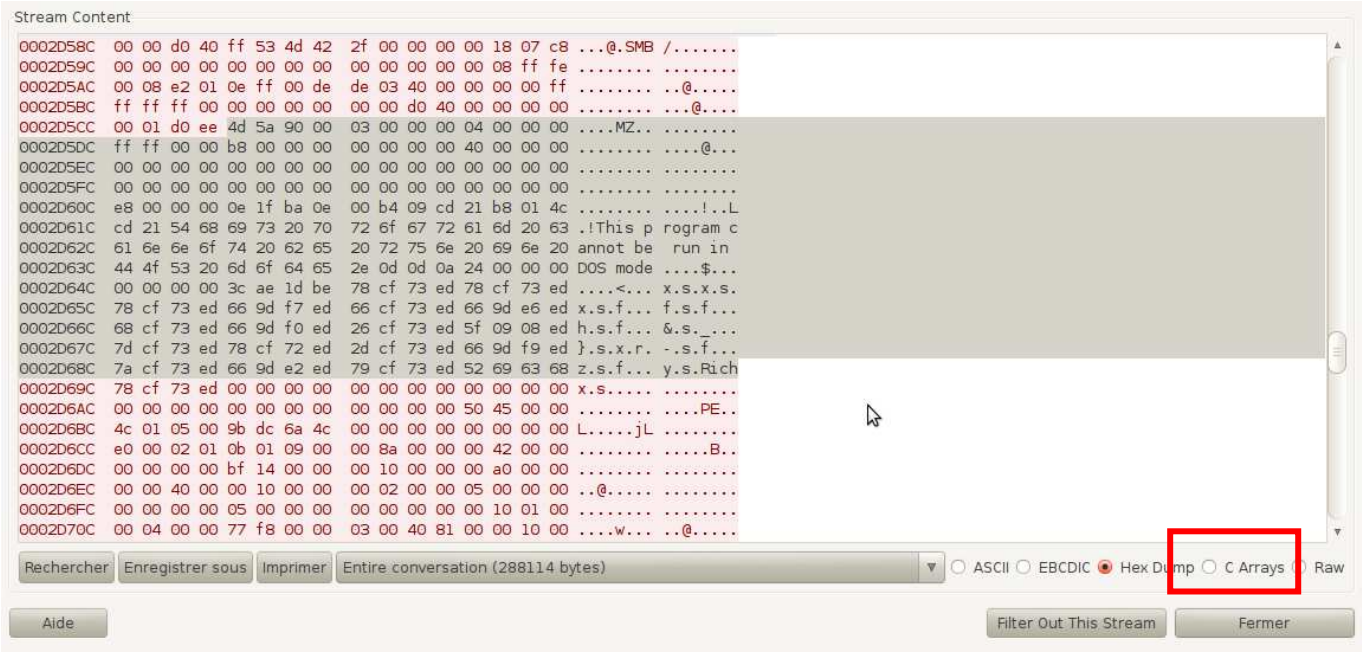
I had to revert to a more “manual” method and copy manually the bytes into a file:



No.	Time	Source	Destination	Protocol	Info
253	0.883930	192.168.2.1	192.168.2.10	SMB	NT Create AndX Request, FID: 0x4002, Path: \psexecsvc
254	0.886544	192.168.2.10	192.168.2.1	SMB	NT Create AndX Response, FID: 0x4002
255	0.886688	192.168.2.1	192.168.2.10	SMB Pipe	TransactNmPipe Request, FID: 0x4002
256	0.887562	192.168.2.10	192.168.2.1	SMB Pipe	TransactNmPipe Response, FID: 0x4002
257	0.909068	192.168.2.1	192.168.2.10	SMB	NT Create AndX Request, FID: 0x4003, Path: \System32\t210test.exe
258	0.910378	192.168.2.10	192.168.2.1	SMB	NT Create AndX Response, FID: 0x4003
259	0.910464	192.168.2.1	192.168.2.10	SMB	Trans2 Request, QUERY_FILE_INFO, FID: 0x4003, Query File Internal Info
260	0.910973	192.168.2.10	192.168.2.1	SMB	Trans2 Response, FID: 0x4003, QUERY_FILE_INFO
261	0.911067	192.168.2.1	192.168.2.10	SMB	Trans2 Request, QUERY_FS_INFO, Query FS Attribute Info
262	0.911923	192.168.2.10	192.168.2.1	SMB	Trans2 Response, QUERY_FS_INFO
263	0.912005	192.168.2.1	192.168.2.10	SMB	Trans2 Request, SET_FILE_INFO, FID: 0x4003
264	0.913516	192.168.2.10	192.168.2.1	SMB	Trans2 Response, FID: 0x4003, SET_FILE_INFO
265	0.920110	192.168.2.1	192.168.2.10	TCP	[TCP segment of a reassembled PDU]
266	0.920185	192.168.2.1	192.168.2.10	TCP	[TCP segment of a reassembled PDU]

+ Frame 257 (188 bytes on wire, 188 bytes captured)
+ Ethernet II, Src: CadmusCo_12:97:2a (08:00:27:12:97:2a), Dst: CadmusCo_a4:e1:b7 (08:00:27:a4:e1:b7)
+ Internet Protocol, Src: 192.168.2.1 (192.168.2.1), Dst: 192.168.2.10 (192.168.2.10)
+ Transmission Control Protocol, Src Port: xrl (1104), Dst Port: microsoft-ds (445), Seq: 185369, Ack: 2610, Len: 134
+ NetBIOS Session Service
+ SMB (Server Message Block Protocol)

Then “follow TCP stream”, identify the EXE by the PE header (magic MZ).There are actually 2 binaries in the stream: psexec (1st) and t210test (2nd), so retrieve only the 2nd one.



Copy the bytes (using “C arrays” in Wireshark) in a text file

```
aaa =[ 0x4d, 0x5a, 0x90, 0x00,
0x03, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
0xff, 0xff, 0x00, 0x00, 0xb8, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xe8, 0x00, 0x00, 0x00, 0x0e, 0x1f, 0xba, 0x0e,
0x00, 0xb4, 0x09, 0xcd, 0x21, 0xb8, 0x01, 0x4c,
0xcd, 0x21, 0x54, 0x68, 0x69, 0x73, 0x20, 0x70,
0x72, 0x6f, 0x67, 0x72, 0x61, 0x6d, 0x20, 0x63,
0x61, 0x6e, 0x6e, 0x6f, 0x74, 0x20, 0x62, 0x65,
0x20, 0x72, 0x75, 0x6e, 0x20, 0x69, 0x6e, 0x20,
0x44, 0x4f, 0x53, 0x20, 0x6d, 0x6f, 0x64, 0x65,
0x2e, 0x0d, 0x0d, 0x0a, 0x24, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x3c, 0xae, 0x1d, 0xbe,
0x78, 0xcf, 0x73, 0xed, 0x78, 0xcf, 0x73, 0xed,
0x78, 0xcf, 0x73, 0xed, 0x66, 0x9d, 0xf7, 0xed,
0x66, 0xcf, 0x73, 0xed, 0x66, 0x9d, 0xe6, 0xed,
0x68, 0xcf, 0x73, 0xed, 0x66, 0x9d, 0xf0, 0xed,
```

And write them in a file (here “/tmp/binary.exe”)

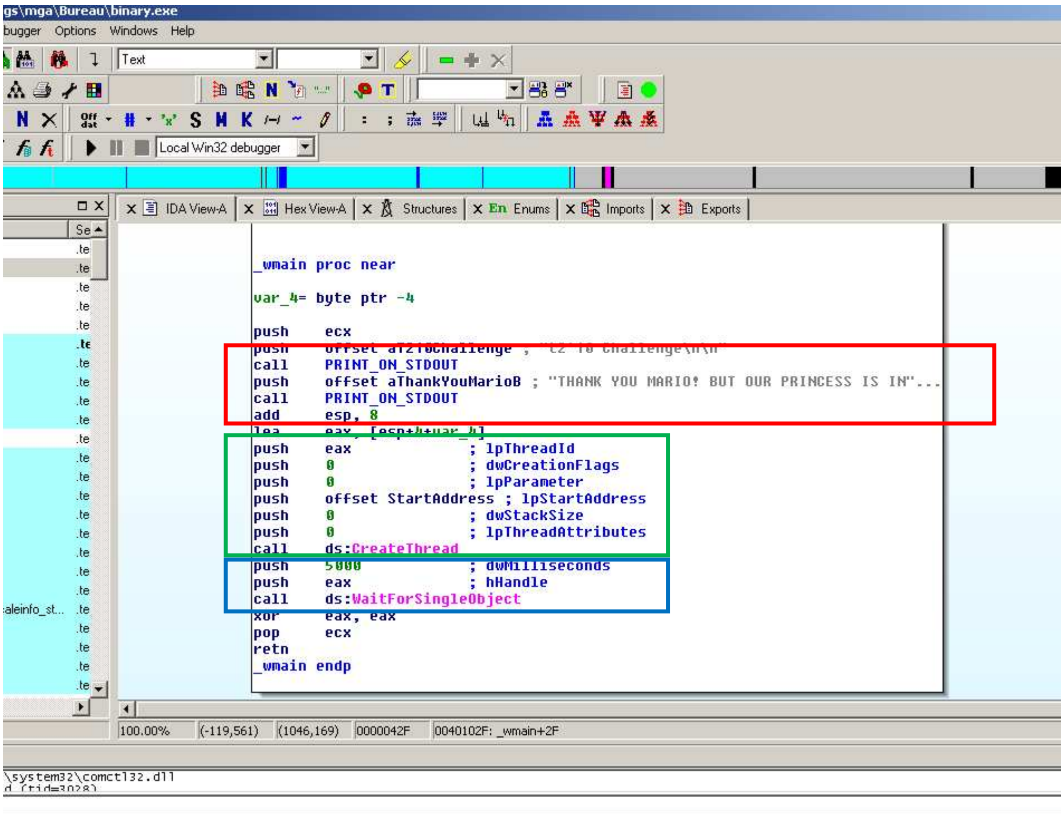
```
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00 ]
```

```
b = open("/tmp/binary.exe","w")
b.write("".join([chr(i) for i in aaa]))
b.close()
```

We can now begin analyzing the binary. The interesting part (the “_wmain” function) is the following:

2 strings are printed on stdout (using function PRINT_ON_STDOUT):

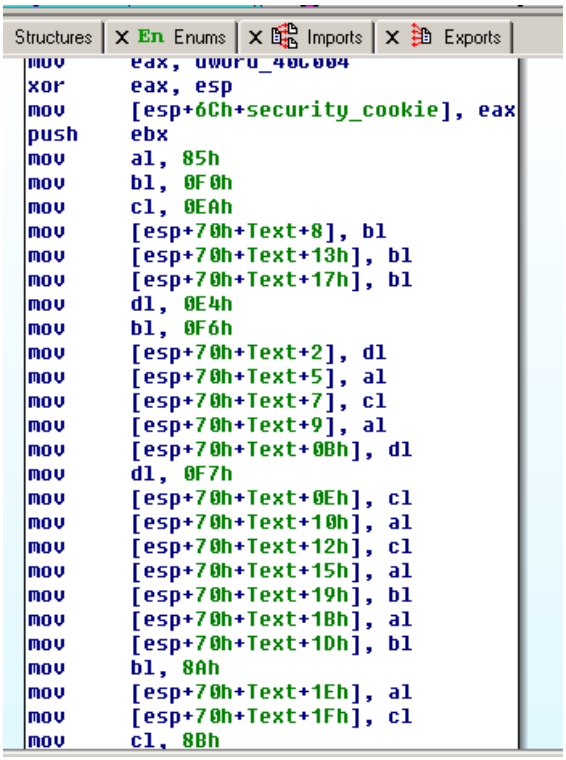
- “t2’10 Challenge”
- “THANK YOU MARIO! ….”



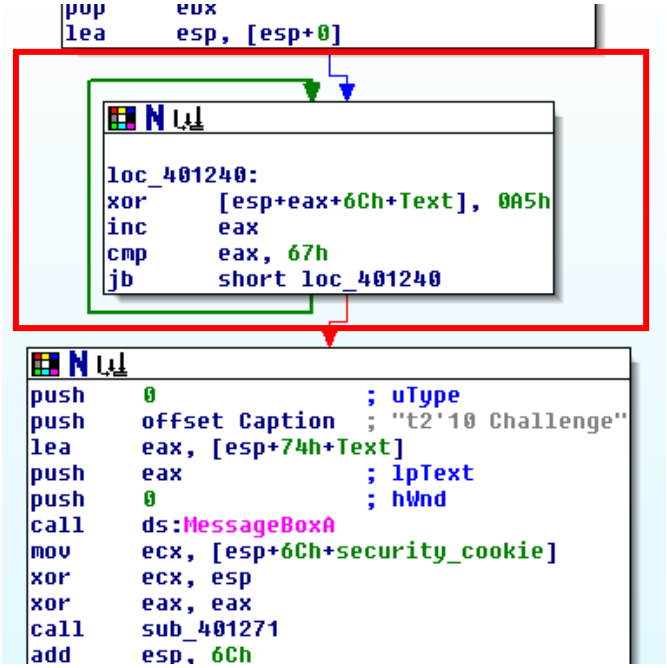
Then a thread is created (function CreateThread) and launches function StartAddress.

This thread is “killed” after 5 seconds (timeout of WaitForSingleObject) so better be quick to copy the URL in the messagebox ;)

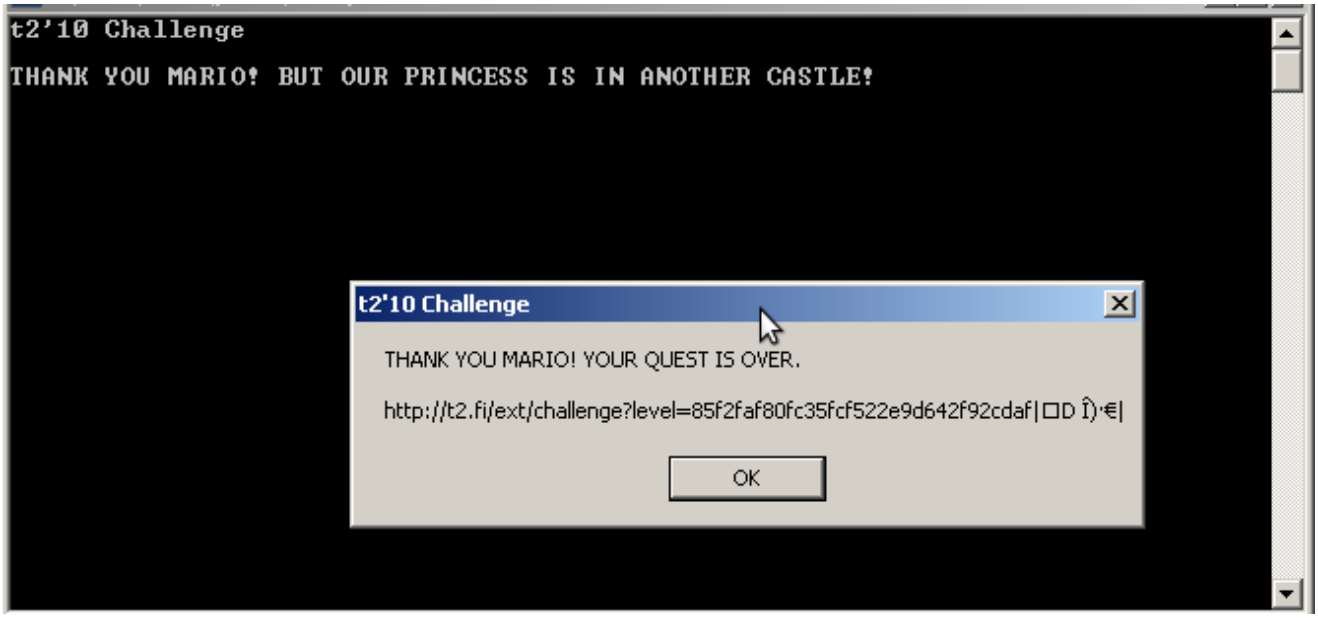
The main part of the program is the function StartAddress which contains the encrypted URL.
First, the encrypted string is copied into the “Text” variable.



Then the string is decrypted (simple XOR encryption with byte 0xA5) and shown to the user in a MessageBox.



We can finally get the last URL of the challenge by simply running the binary:



It was really compulsory to retrieve the binary in the pcap capture because only “stdout” and “stderr” of the binary were retrieved when it was launched using “psexec”, and the URL was stored encrypted.

So only the 2 strings (“t2’10 Challenge”, “THANK YOU MARIO!..”) could be seen in the network capture:

```
\.p.s.e.x.e.c.s.v.c.-.T.E.S.T.-.E.D.3.1.1.4.7.6.9.A.-.3.7.8.8.-.s.t.d.e.r.r.....8.SMB%.....
........8.....8.........SMB.................................L.................................(
\.p.s.e.x.e.c.s.v.c.-.T.E.S.T.-.E.D.3.1.1.4.7.6.9.A.-.3.7.8.8.-.s.t.d.e.r.r.....SMB.....
@.....................................................................................
@                    ; SMB                                  ....@.....................SMB.....
<...........M..t2'10 Challenge

THANK YOU MARIO! BUT OUR PRINCESS IS IN ANOTHER CASTLE!
...;.SMB.....................................@.....................#.SMB.....................
B.....................................).SMB.....................@.....#.SMB.....
B.....................@.....#.SMB.....................).SMB.....
B.....................;.SMB.....................!.....@.....XJ.....
B.....................!.....<.....XJ.....T.E.S.T.-.E.D.3.1.1.4.7.6.9.
A.....................................................................................
t.2.1.0.t.e.s.t...e.x.
e.....................................................................................
SMB.....l.....@.....9.....SMB.....l.....
```

Conclusion

I'd like to thank the T2 committee for this very fun challenge with varying levels which covered different domains (cryptography, steanography…)

Please keep making this type of fun challenge in the coming years :)