

T2'10 Challenge



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
255 255 255 255 255 255 255 255
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 0 0 0 0 255 255 255
255 0 0 0 0 255 255 255
255 255 255 255 255 0 0 0
0 0 255 255 255 255 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
0 0 0 0 255 255 255 255
255 255 255 255 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)

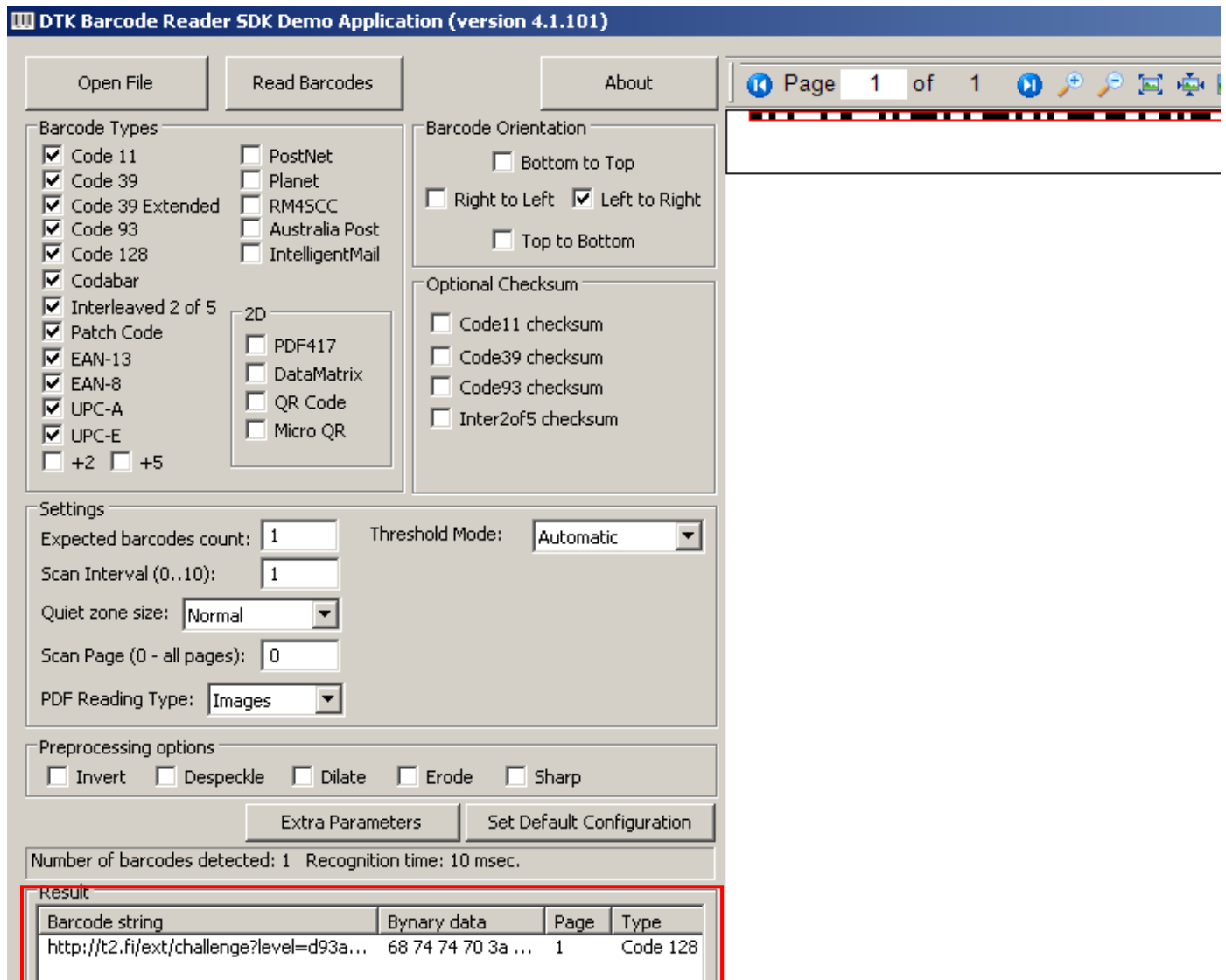
0255 255... : pixels of the image

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>)

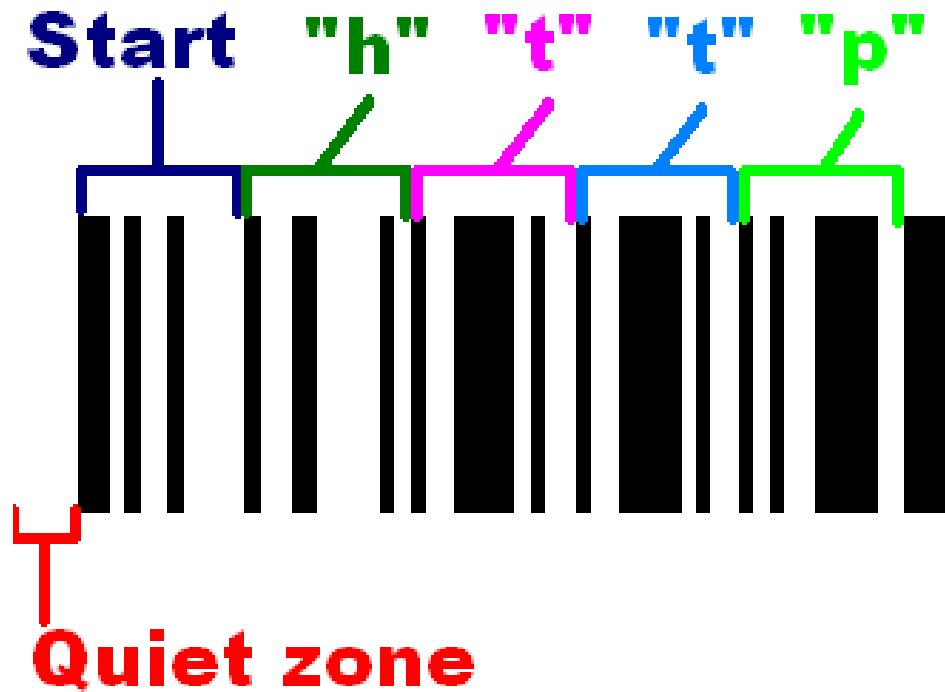
Characters encoding are the following (let's concentrate on the « h », « t » and « p » of « http »)

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 ao0 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):

Search for: PDF as Text Find Next Find Previous

Signed 8 bit: 13 Signed 32 bit: 218785134 Hexadecimal: 0D 0A 65 6E

Unsigned 8 bit: 13 Unsigned 32 bit: 218785134 Decimal: 013 010 101 110

Signed 16 bit: 3338 Float 32 bit: 4,264662E-31 Octal: 015 012 145 156

Unsigned 16 bit: 3338 Float 64 bit: 7,55049144561858E-246 Binary: 00001101 00001010 01100101 01101110

ASCII Text: PDF

Show little endian decoding Show unsigned as hexadecimal

Offset: 246989 / 44041215 Selection: 246694 to 246988 (295 bytes) INS

And the URL we found before:

```

0003c6ab 37 37 31 2E 34 5D 20 2F 42 53 3C 3C 2F 57 20 30 3E 3E 2F 46 20 34 2F 41 3C 3C 2F 54 79 70 65 2F 41 771.4] /BS<</W 0>>/F 4/A<</Type/A
0003c6cc 63 74 69 6F 6E 2F 53 2F 55 52 49 2F 55 52 49 28 68 74 74 70 3A 2F 2F 74 32 2E 66 69 2F 65 78 74 2F ction/S/URI/URI (http://t2.fi/ext/
0003c6ed 63 68 61 6C 6C 65 6E 67 65 3F 6C 65 76 65 6C 3D 39 35 66 30 35 61 32 32 62 39 36 39 34 65 64 66 32 challenge?level=95f05a22b9694edf2
0003c70e 30 66 64 35 62 66 35 64 64 63 63 38 65 39 66 29 20 3E 3E 3E 3E 0D 0A 65 6E 64 6F 62 6A 0D 0A 38 20 0fd5bf5ddcc8e9f) >>>>.endobj..8
0003c72f 30 20 6F 62 6A 0D 0A 3C 3C 2F 50 72 6F 64 75 63 65 72 28 FE FF 00 4D 00 69 00 63 00 72 00 6F 00 73 0 obj.<</Producer(.M.i.c.r.o.s

```


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):

```

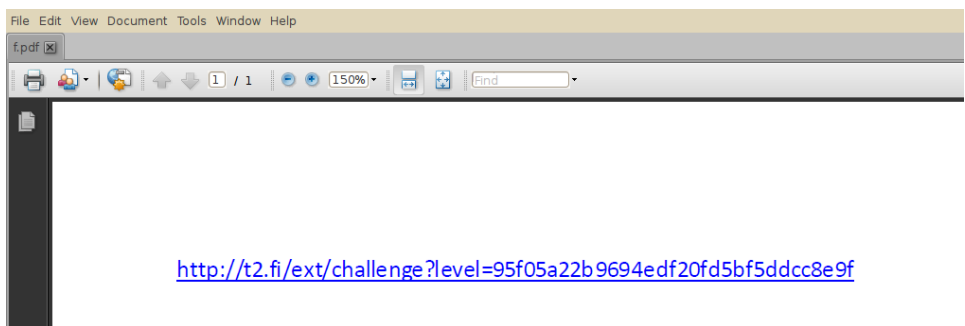
0004445a A5 2D 2A 6D 51 69 8B 4A 5B 54 DA A2 D2 16 95 B6 A8 B4 45 A5 2D 2A 6D 51 69 8B 4A 5B 54 DA A2 D2 16 ...
0004447b 95 B6 A8 B4 45 A5 2D 2A 6D 51 69 8B 4A 5B 54 DA A2 D2 B6 31 F4 8E BD FA 2B DE 45 09 56 61 8D B7 AF ...
0004449c 45 29 D6 61 7D 72 A3 94 ED 0D 9E 96 B0 B8 84 C5 25 2C 2E 61 71 09 8B 4B 58 5C C2 E2 12 16 97 B0 B8 ...
000444bd 84 C5 25 2C CE B1 1B 39 76 23 C7 6E E4 CD 08 6F 46 79 33 CA 9B 51 DE 8C F2 66 34 78 22 59 C1 9D 6B ...
000444de B8 73 0D 77 AE E1 CE 35 DC B9 46 5A E2 D2 12 97 96 B8 B4 C4 83 A7 93 07 83 67 F0 2C 9E C3 1F 31 06 ...
000444ff 7F C2 58 3C BF 17 50 97 3C A0 BA 0F A8 EE 03 AA BB 56 75 37 A8 EE 06 D5 DD A0 BA 1B 54 77 83 EA AE ...
00044520 55 DD 55 AA BB 4A 75 57 A9 EE 2A D5 5D D5 F2 D3 D3 9D 70 5E EA E7 A4 F1 13 FC 14 A9 9F A4 BE 10 9D ...
00044541 F1 33 5C 84 9F E3 62 5C 82 4B 71 19 52 3F 6B FD 0B 74 45 37 74 C7 2F 71 39 AE 40 0F 5C 09 AB F0 2B ...
00044562 F4 C4 D5 B8 06 A9 9F D0 BE 16 BF 41 3A 52 3F AB 7D 3D 6E C0 8D E8 85 C9 98 82 6C 4C 43 0E A6 63 06 ...
00044583 66 22 17 79 98 85 7C CC C6 1C CC 45 01 E6 E1 75 FC 05 F3 B1 00 0B F1 06 16 A1 10 8B 53 3F 0F 8D D5 ...
000445a4 C9 46 55 10 57 05 71 55 10 57 05 71 55 10 4F BB 35 B9 87 7F F7 70 EF 9E E0 CA E0 5B 4E 5F E7 58 D1 ...
000445c5 73 D1 09 E7 E1 7C 74 0D 3A E8 C8 1D 52 FF 5E 28 6F 36 F0 66 03 6F 36 F0 66 03 6F 36 F0 66 43 EA 5F ...
000445e6 CA E4 CD 06 DE 6C E0 CD 86 E0 7E 67 A2 DF A3 3F C6 F3 D2 04 4C C4 24 BC 8C 57 54 48 41 4E 4B 20 59 ...
00044607 4F 55 20 4D 41 52 49 4F 21 0D 0A 0D 0A 42 55 54 20 4F 55 52 20 50 52 49 4E 43 45 53 53 20 49 53 20 ...
00044628 49 4E 0D 0A 41 4E 4F 54 48 45 52 20 43 41 53 54 4C 45 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
    
```

Beginning of part 2 (red line)

```

000341c4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
000341e5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
00034206 6E AC E6 C6 6A 6E AC D6 39 4F D7 39 4F E7 C8 1A 8E AC E1 C8 1A 8E AC E1 C8 1A 8E AC E1 C8 1A 8E AC ...
00034227 E1 C8 1A 8E AC E1 C8 1A 8E AC E1 C8 1A DD 32 A1 5B 26 74 CB 84 6E 99 D0 2D 13 BA 65 42 B7 4C E8 96 ...
00034248 09 DD 32 A1 5B 26 74 CB 44 90 FA 09 E3 5B 93 CD 69 77 E8 40 BF 0B FA A6 DD ED E5 3D 41 DF A0 8B 4C ...
00034269 D4 CA 44 AD 4C D4 CA 44 AD 4C D4 CA 44 AD 4C D4 CA 44 AD 4C D4 CA 44 AD 4C D4 CA 44 AD 4C D4 CA 44 AD ...
0003428a 37 DC CF EC BF 47 7F FC 41 16 1E 4D EE 97 8F FD F2 B1 5F 3E F6 CB C7 7E F9 D8 2D 1F 5B E4 63 8B C3 ...
000342ab 6C 91 8F 2D F2 B1 45 3E 62 F2 11 93 8F 98 7C C4 EA 63 9F 7C EC 93 BF 7D F2 B1 4F 3E F6 C9 C7 3E F9 ...
000342cc D8 27 1F FB E4 63 9F 7C EC 0B 96 59 E9 E5 2D 19 38 10 1C D6 85 92 C9 A6 50 80 50 B2 C9 DD 95 A6 DD ...
000342ed 2E E9 77 B8 D3 DF 05 69 3E 81 0E D3 82 7D EE 70 9C 3B 1C E7 0E C7 B9 C3 71 EE 70 9C 3B 1C E7 0E ...
0003430e C7 B9 C3 71 EE 70 9C 3B 1C E7 0E C7 B9 C3 19 7A 6B 83 DE DA A0 B7 36 88 AD 0D 7A 6B 83 DE DA F0 E5 ...
0003432f B5 92 9C 68 35 26 7E A3 5A B9 3B 59 A7 C7 D6 E9 B1 75 7A 6C 9D 1E 5B A7 C7 D6 E9 5D 98 95 39 68 65 ...
00034350 0E 5A 99 83 56 E6 A0 95 C9 B3 32 79 56 26 CF CA E4 59 99 3C 2B 93 67 65 F2 AC 4C 9E 95 C9 B3 32 79 ...
00034371 C1 4B C9 4F D5 DD 5A 75 B7 56 DD AD 55 77 6B D5 DD 5A 75 B7 36 F8 B3 C7 5E C5 14 64 63 2A A6 21 07 ...
00034392 D3 31 03 33 91 8B 3C CC 42 BE 8F 9B 8D 39 98 8B 02 CC F3 F6 D7 31 1F 0B B0 10 6F 60 11 0A B1 18 45 ...
000343b3 78 13 4B B0 14 CB 92 73 ED D8 DC E0 2D 7F 7E 1B 2B 50 8C 30 56 E2 AF 78 17 25 58 85 D5 58 83 B5 28 ...
    
```

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\nl 0 0 1 57.075 797.2 Tm\r\n0 g\r\n0 G\r\n[( ) TJ\r\nET\r\nBT\r\nl 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\nl 0 0 1 57.075 736.4 Tm\r\n[( ) TJ\r\nET\r\n'
>>> □
```

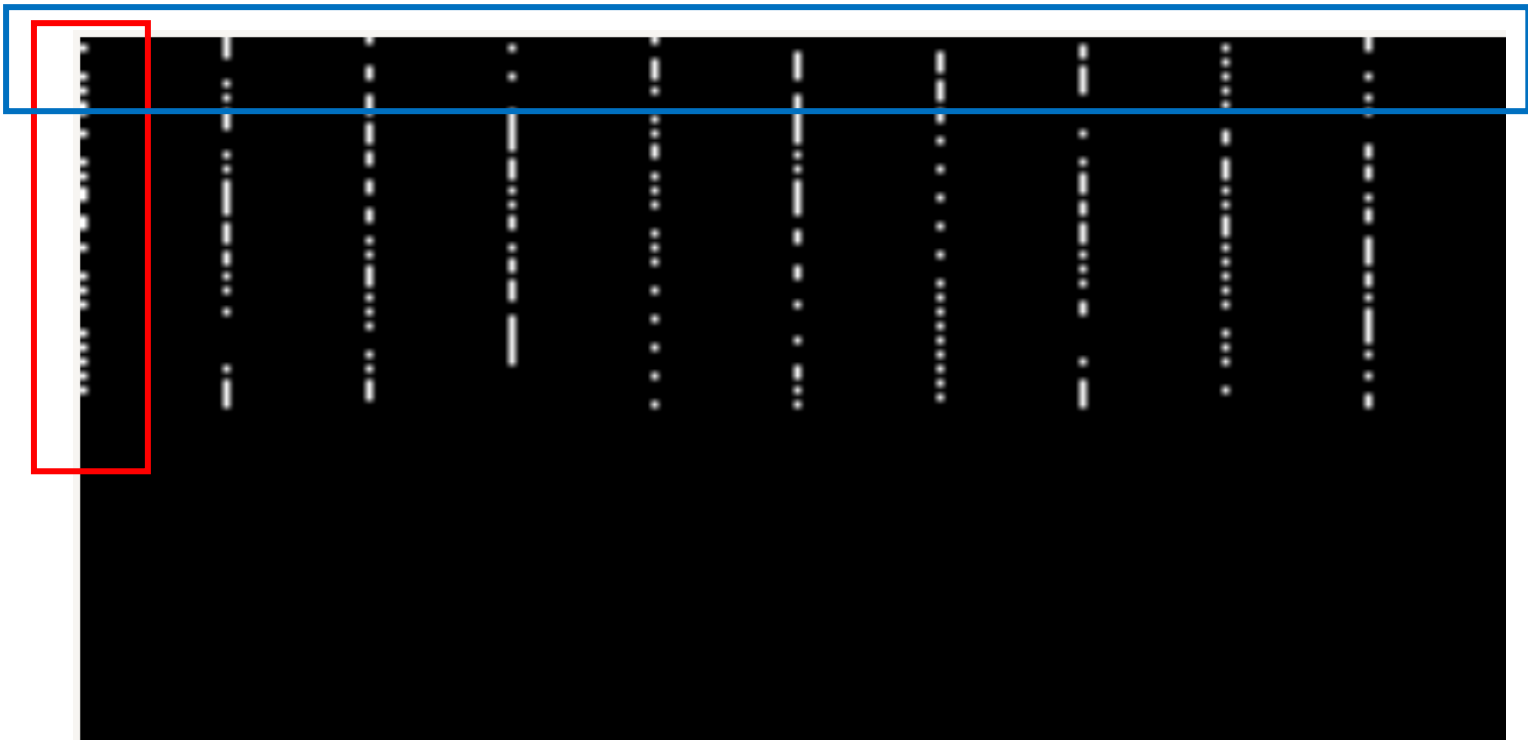
One can clearly see the challenge URL (in red)

```
BT\r\n/F1 11.25 Tf\r\nl 0 0 1 57.075 797.2 Tm\r\n0 g\r\n0 G\r\n[( ) TJ\r\nET\r\nBT\r\nl 0 0 1
57.075 39.025 Tm\r\n[( ) TJ\r\nET\r\nBT\r\nl 0 0 1 57.075 761.92 Tm\r\n0 0 1 rg\r\n0 0 1
RG\r\n[(h)-7(tt)3(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(g)71(e)-34(?)63(l)-36(e)31(v)-14(e)31(l)-36(=)31(9)40(5)40(f)-
27(0)40(5)40(a)12(2)40(2)40(b)-74(9)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)-34(9)40(f)] TJ\r\nET\r\n0 0 1 rg\r\n57.075
759.67 311.63 0.75 re\r\nf*\r\nBT\r\nl 0 0 1 368.7 761.92 Tm\r\n0 g\r\n0 G\r\n[( )
TJ\r\nET\r\nBT\r\nl 0 0 1 57.075 736.4 Tm\r\n[( ) TJ\r\nET\r\n'
```




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'
>>>

```

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\x
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.

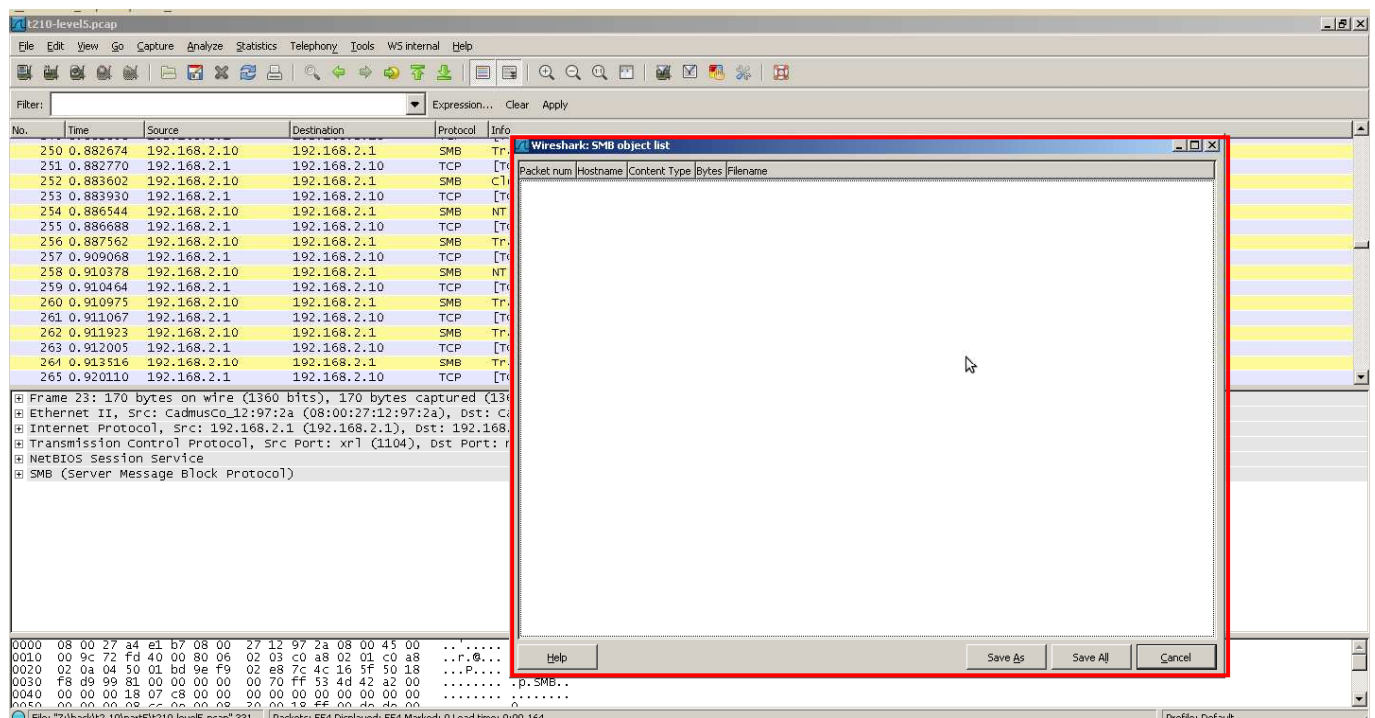
After investigation, it appears that the machine 192.168.2.1 used psexec (<http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx>) to launch binary "t210test.exe" (located remotely in \System32\) on remote machine 192.168.2.10 (which runs Windows Small Business Server).

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:

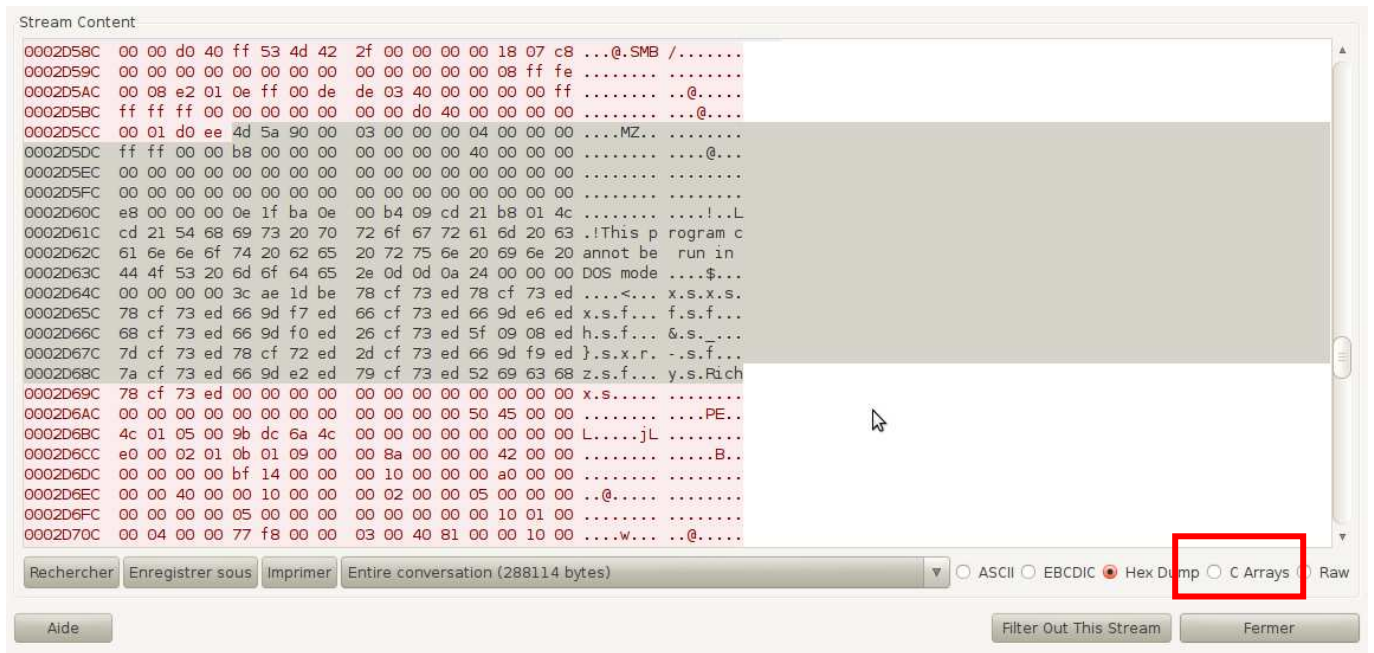
First, I thought of using a development version of Wireshark (1.5 svn) which permits the retrieval of files transmitted through SMB (see <http://blog.taddong.com/2010/05/capturing-smb-files-with-wireshark.html>) but it didn't work (Wireshark didn't see any file transferred):



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, QUERY_FILE_INFO, 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]

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,
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")

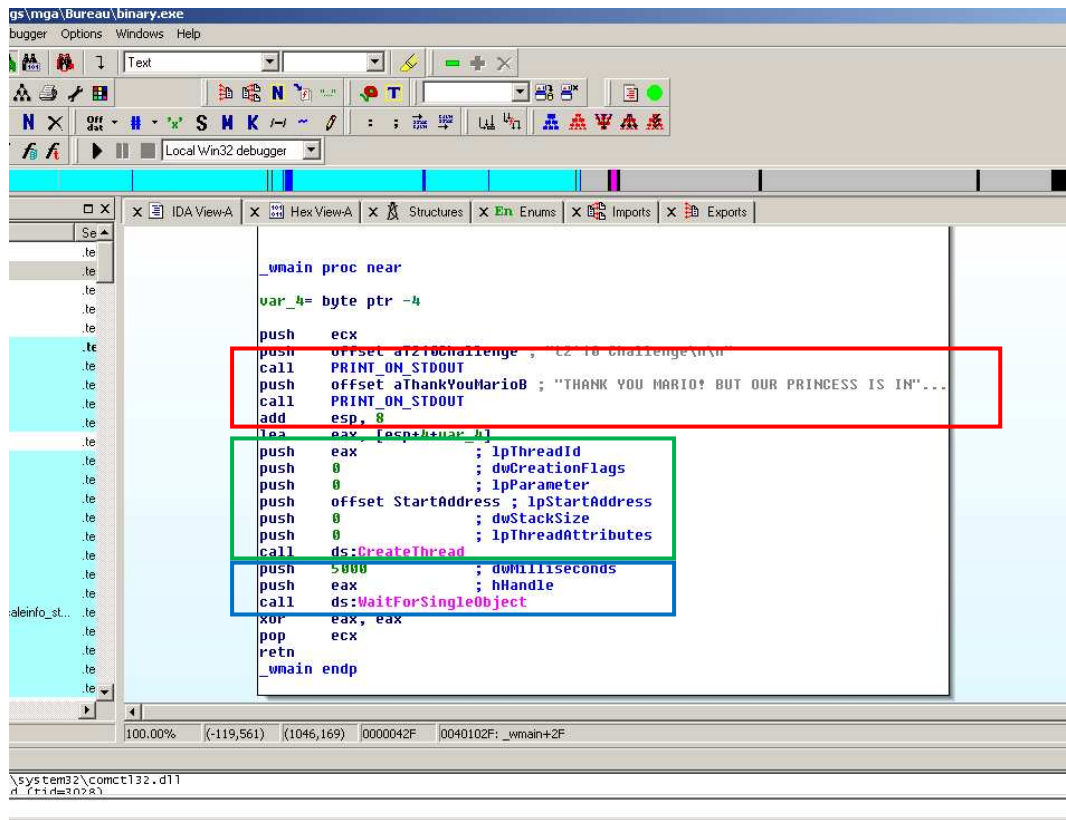
```
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,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

```
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.

```
Structures | X En Enums | X Imports | X Exports
mov     eax, uword_40c004
xor     eax, esp
mov     [esp+6Ch+security_cookie], eax
push   ebx
mov     al, 85h
mov     bl, 0F0h
mov     cl, 0EAh
mov     [esp+70h+Text+8], bl
mov     [esp+70h+Text+13h], bl
mov     [esp+70h+Text+17h], bl
mov     dl, 0E4h
mov     bl, 0F6h
mov     [esp+70h+Text+2], dl
mov     [esp+70h+Text+5], al
mov     [esp+70h+Text+7], cl
mov     [esp+70h+Text+9], al
mov     [esp+70h+Text+0Bh], dl
mov     dl, 0F7h
mov     [esp+70h+Text+0Eh], cl
mov     [esp+70h+Text+10h], al
mov     [esp+70h+Text+12h], cl
mov     [esp+70h+Text+15h], al
mov     [esp+70h+Text+19h], bl
mov     [esp+70h+Text+1Bh], al
mov     [esp+70h+Text+1Dh], bl
mov     bl, 8Ah
mov     [esp+70h+Text+1Eh], al
mov     [esp+70h+Text+1Fh], cl
mov     cl, 8Bh
```

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

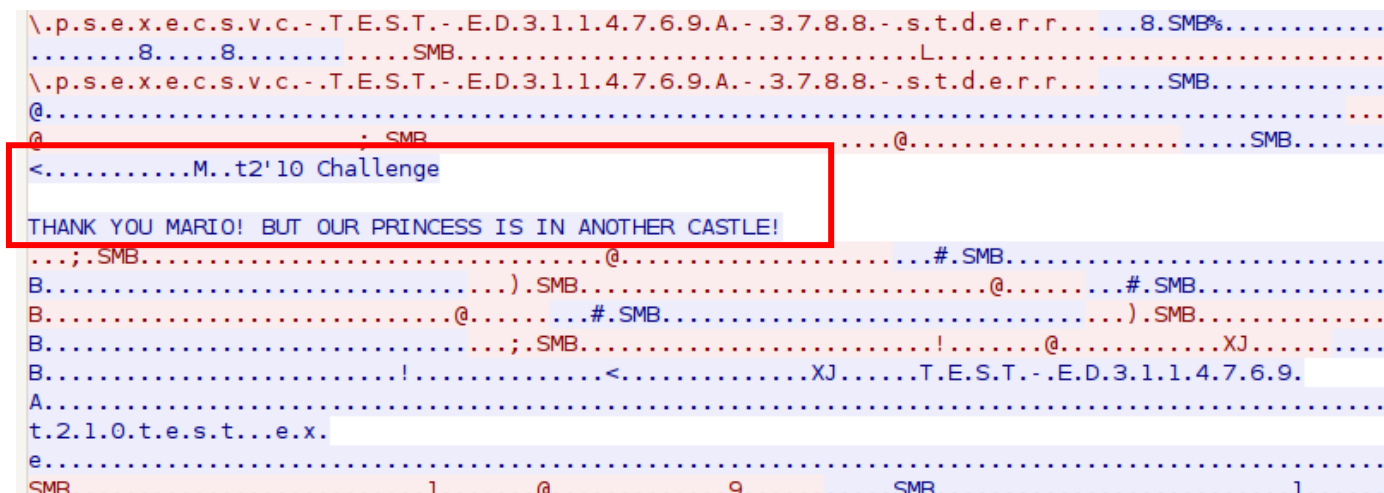
```
pop     eax
lea     esp, [esp+0]
loc_401240:
xor     [esp+eax+6Ch+Text], 0A5h
inc     eax
cmp     eax, 67h
jb     short loc_401240
push   0 ; uType
push   offset Caption ; "t2'10 Challenge"
lea   eax, [esp+74h+Text]
push   eax ; lpText
push   0 ; hWnd
call   ds:MessageBoxA
mov   ecx, [esp+6Ch+security_cookie]
xor   ecx, esp
xor   eax, eax
call   sub_401271
add   esp, 6Ch
```

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:



Conclusion

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

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