Princess In Distress Solution

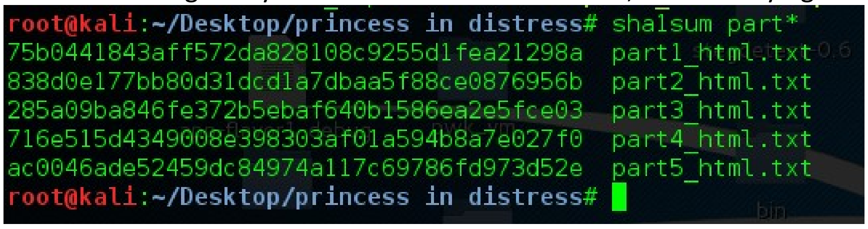
Tools Used

- Stegsolve http://www.caesum.com/handbook/Stegsolve.jar
- Various custom scripts (included throughout the writeup)
- Gronsfield cipher online cracker http://www.geocachingtoolbox.com/index.php?page=gronsfeldCipher
- Binwalk, xxd, strings, and other linux cmd line utilities.

Stage 1: The html file

The first stage of this challenge (and arguably the hardest to crack) was in the form of an html file. The first thing I noticed was that the file contained 5 copies of the same wikipedia entry about video games, making it a pretty big file.

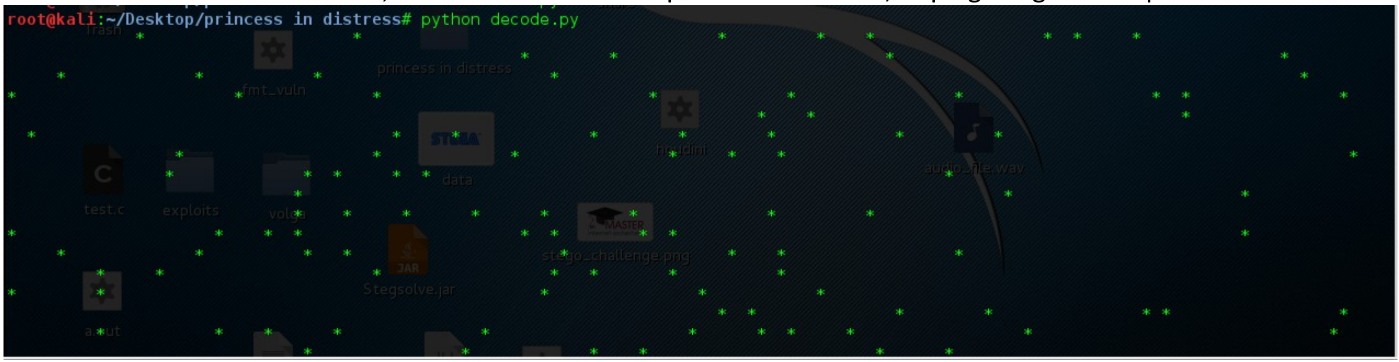
After looking for low hanging fruit such as hidden links or hints I found that the solution would take a bit more work. I decided to split the file into its five parts and see if they differed, I hashed each part, telling me that even though they looked the same in a browser, the underlying html was different in each part.



```
root@kali:~/Desktop/princess in distress# sha1sum part*
75b0441843aff572da828108c9255d1fea21298a  part1_html.txt
838d0e177bb80d31dcd1a7dbaa5f88ce0876956b  part2_html.txt
285a09ba846fe372b5ebaf640b1586ea2e5fce03  part3_html.txt
716e515d4349008e398303af01a594b8a7e027f0  part4_html.txt
ac0046ade52459dc84974a117c69786fd973d52e  part5_html.txt
root@kali:~/Desktop/princess in distress#
```

The obvious question to ask was, what data can be hidden in html that would appear normal in a browser that would cause the hashes to be different? The first answer would be the <html><body> tags in part1 and part5. However that should mean that parts 2-4 should be the same. Eventually I found that each part had differences in how each word was spaced (sort of). What I found was that in places within the html, double spaces had been added between words, this wouldn't be visible in a browser as it doesn't recognise multiple spaces.

The next question came naturally, how can someone embed data in a file based on the occurences of spaces and double spaces? I decided that the first thing I should do is strip all of the alphabetic characters and html code from the file, and make the double spaces more visible, hoping I might see a pattern.



```
root@kali:~/Desktop/princess in distress# python decode.py
```

Looking at this, I decided that the solution could be in the distance between each point, I then wrote a script that takes the distance from one point to the next and writes that integer as a byte to a new file, long story short this didn't work at all, but it did help brush up on some regex skills (which I'll need for the next part). The first attempt script was as such:

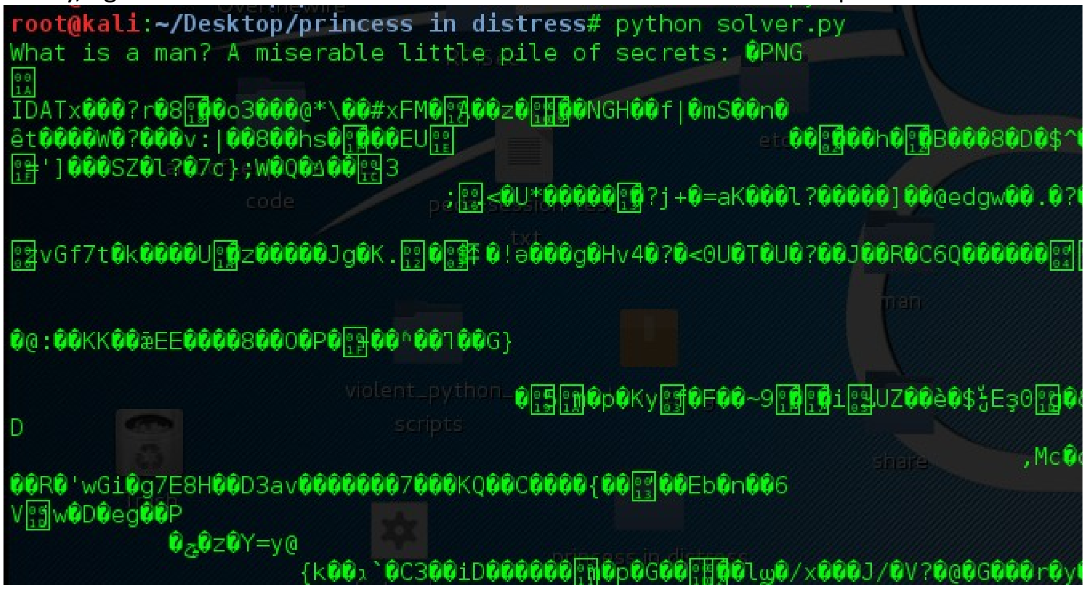
```
import struct
import re

newLine =  re.sub('\s\s', '*', open("video_games.html", "r").read()) #replace double space with *
newLine = re.sub('[^*]', '  ', newLine) #get rid of everything that isn't a *
```

```
print newLine
out = open("attempt1", "wb")


for i in range(0, len(line)):
	if(line[i] == '*'):
		i = i + 1
		for j in range(i, len(line)):
			if(line[j] == '*'):
				byte = struct.pack("<I", j-i)
				print byte
				out.write(byte)
				i = j
				break
```

After some further pondering, I decided to try something similar. This time I would take every occurence of a single space to indicate a 0, and every occurence of a double space to be a 1. Using similar regex replacement as before, I came up with the following script.

```
import re
import array

#start of by getting rid of any ~ characters in the file
newLine =  re.sub('~', '', open("video_games.html", "r").read())

#every double space becomes a ~
newLine = re.sub('\s\s', '~', newLine) #~=1

#every single space becomes a *
newLine = re.sub('\s', '*', newLine)

#strip everything from the file but * and ~
newLine = re.sub('[^*~]', '', newLine)


binary_string = ""

#Iterate over our string and build the bit string from each symbol
for i in range(len(newLine)):
    if(newLine[i] == '~'):
        binary_string += "1"
    elif(newLine[i] == '*'):
        binary_string += "0"

#Slice up our bit string into bytes and convert those to integers
data = [int(binary_string[x:x+8], 2) for x in range(0, len(binary_string), 8)]

#turn each integer to a character byte
data = ''.join(chr(i) for i in data)
print data
```

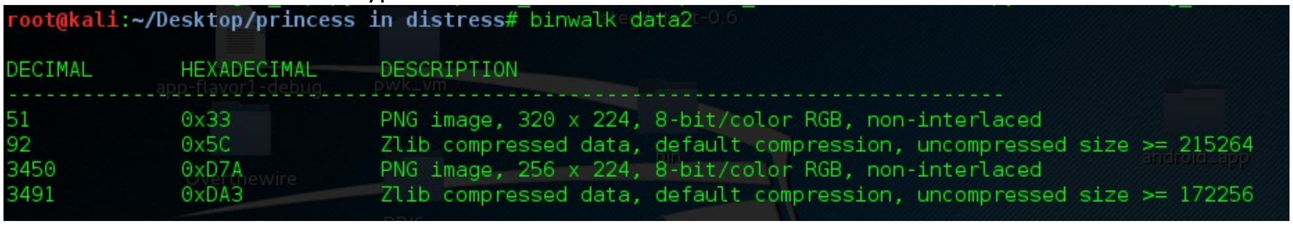
Finally, I got confirmation that this was the correct solution to the first part.



```
root@kali:~/Desktop/princess in distress# python solver.py
What is a man? A miserable little pile of secrets: �PNG

IDATx���?r�8��o3���@*\��#xFM��A��z����NGH��f|�mS��n�
ět����W�?���v:|��8��hs�����EU�                             et������h���B����8�D�$^
�'']���SZ�l?�7c};W�Q�A���3
                                   ;�<�U*������?j+�=aK���l?�����]��@edgw��.�?
�vGf7t�k����U��z�����Jg�K.�����!a���g�Hv4�?�<0U�T�U�?��J��R�C6Q������

�@:��KK��EE�����8���0�P����^��1��G}

                                        ����p�Ky��F��~9����i�UZ��e�$E30���
D
                                                                              ,Mc�
��R�'wGi�g7E8H��D3av�������7���KQ��C����{�����Eb�n��6
V��w�D�eg��P
          �z�Y=y@
                  {k��`�C3��iD�������p�G����l�/x���J/�V?�@�G���r�y
```

Stage 2: Getting the pictures and doing some Steganography

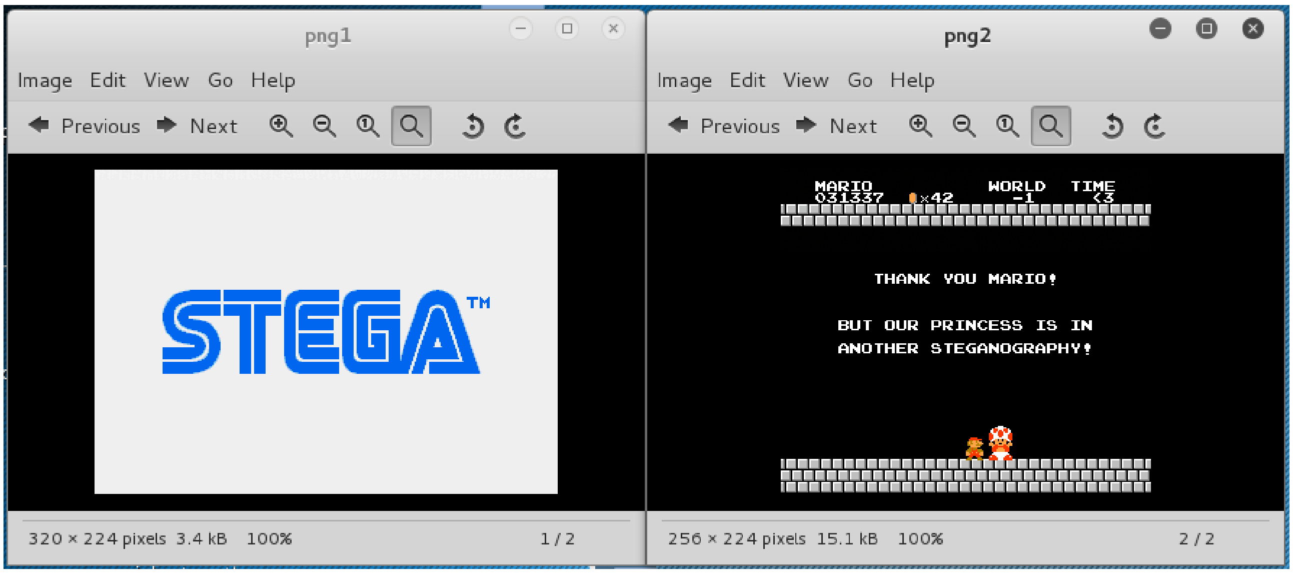
Instantly we see the magic bytes for a PNG file, I wondered if anything else was embedded so used binwalk to scan the file for known file types.



```
root@kali:~/Desktop/princess in distress# binwalk data2

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
51            0x33            PNG image, 320 x 224, 8-bit/color RGB, non-interlaced
92            0x5C            Zlib compressed data, default compression, uncompressed size >= 215264
3450          0xD7A           PNG image, 256 x 224, 8-bit/color RGB, non-interlaced
3491          0xDA3           Zlib compressed data, default compression, uncompressed size >= 172256
```

So we have 2 PNG's. The problem was binwalk didn't seem to want to extract them, just the zlib files. I decided to update my script from above so that it would solve the first part of the challenge, and then carve the png files from the produced data. The final code for this part was as follows.

```
import re
import array

#start of by getting rid of any ~ characters in the file
newLine =  re.sub('~', '', open("video_games.html", "r").read())

#every double space becomes a ~
newLine = re.sub('\s\s', '~', newLine) #~=1

#every single space becomes a *
newLine = re.sub('\s', '*', newLine)

#strip everything from the file but * and ~
newLine = re.sub('[^*~]', '', newLine)



binary_string = ""

#Iterate over our string and build the bit string from each symbol
for i in range(len(newLine)):
        if(newLine[i] == '~'):
                binary_string += "1"
        elif(newLine[i] == '*'):
                binary_string += "0"

#Slice up our bit string into bytes and convert those to integers
data = [int(binary_string[x:x+8], 2) for x in range(0, len(binary_string), 8)]

#turn each integer to a character byte
data = ''.join(chr(i) for i in data)

print data
outfile = open("png1", "wb")

index = 0
#We need to iterate past the ascii text
for i in range(len(data)):
        if(data[i] == "P"):
                index = i-1
                break

#82 marks the end of the first png file and the beginning of the next
#82 89 50 4e 47 0d 0a
for i in range(index, len(data)):
        #this if statement is a little messy, but I wanted to make sure it would be stable
        #we look for \x82 and if the following bytes are the png magic bytes
        #and the current index is larger than 100 (basically we arent at the beginning of the file)
        #then we are at the end of the first png, so break
        if(data[i] == "\x82" and data[i+1] == "\x89" and data[i+2] == "\x50" and i > 100): #looking for
end of PNG
                print "BREAKING"
                index = i+1
                break
        else:
                outfile.write(data[i]) #otherwise write the data to the first png file

png2 = open("png2", "wb")
#now we just iterate over the rest of the file and write the data to the next png file
for i in range(index, len(data)):
        png2.write(data[i])

png2.close()
outfile.close()
```



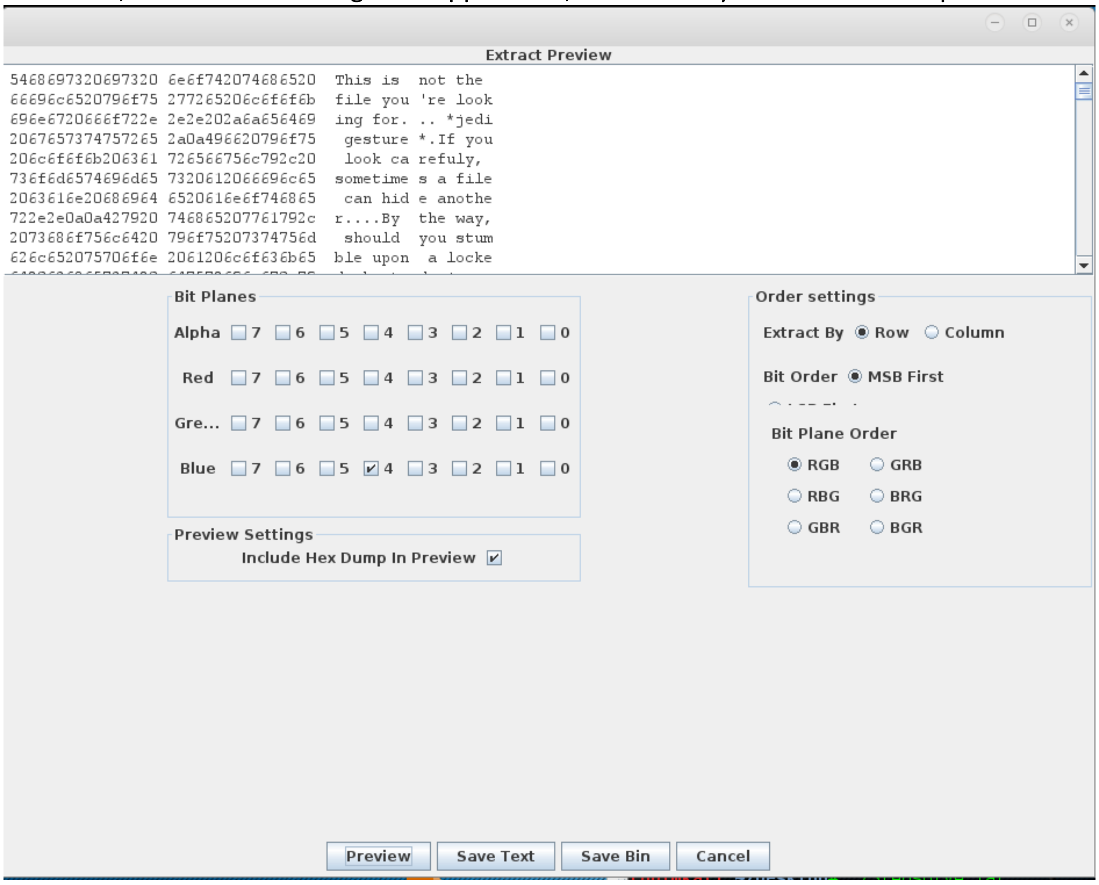
THANK YOU MARIO!

BUT OUR PRINCESS IS IN
ANOTHER STEGANOGRAPHY!

320 × 224 pixels 3.4 kB 100% 1 / 2

256 × 224 pixels 15.1 kB 100% 2 / 2

We now have two images to work with, and a pretty obvious hint as to how to crack them. Admittedly I don't have much experience in writing programs to solve steganography challenges, and so with not much finesse, I turned to the StegSolve application, and basically tried random bit planes until I got a hit.



Extract Preview

```
5468697320697320 6e6f742074686520  This is  not the
66696c6520796f75 277265206c6f6f6b  file you 're look
696e6720666f722e 2e2e202a6a656469  ing for. .. *jedi
2067657374757265 2a0a496620796f75   gesture *.If you
206c6f6f6b206361 726566756c792c20   look ca refuly,
736f6d6574696d65 7320612066696c65  sometime s a file
2063616e20686964 6520616e6f746865   can hid e anothe
722e2e0a0a427920 746865207761792c  r....By  the way,
2073686f756c6420 796f752073747564d   should  you stum
626c652075706f6e 2061206c6f636b65  ble upon  a locke
```

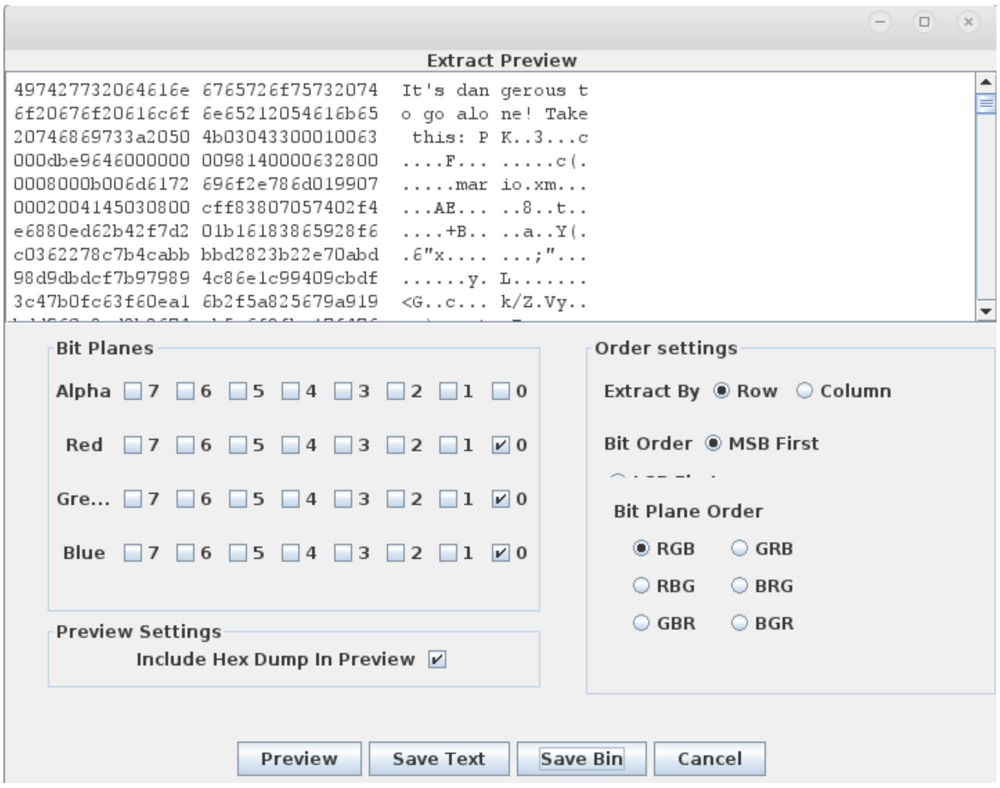
Bit Planes: Alpha 7 6 5 4 3 2 1 0; Red 7 6 5 4 3 2 1 0; Green 7 6 5 4 3 2 1 0; Blue 7 6 5 [✓ 4] 3 2 1 0

Order settings: Extract By [✓ Row] Column; Bit Order [✓ MSB First]; Bit Plane Order [✓ RGB] GRB RBG BRG GBR BGR

Preview Settings: Include Hex Dump In Preview ✓

This was the first hint to be uncovered. It tells me that I should find a "locked chest", to which the ciphered key is BQRAIHUJBVWSF, and that key can be deciphered using the magic number 1337.

I then moved on to the second png file. I don't know if this is correct or not, but for some reason the time in the picture "< 3" made me think that I should try bit planes smaller than 3, either way it worked and I found a hidden zip file.



Extract Preview

```
497427732064616e 6765726f75732074  It's dan gerous t
6f20676f20616c6f 6e65212054616b65  o go alo ne! Take
2074686973a2050 4b03043300010063   this: P K..3...c
000dbe9646000000 0098140000632800  ....F... .....c(.
0008000b006d6172 696f2e786d019907  .....mar io.xm...
0002004145030800 cff83807057402f4  ...AE... ..8..t..
e6880ed62b42f7d2 01b161838659 28f6  ....+B.. ..a..Y(.
c0362278c7b4cabb bbd2823b22e70abd  .6"x.... ...;"...
98d9dbdcf7b97989 4c86e1c99409cbdf  ......y. L.......
3c47b0fc63f60ea1 6b2f5a825679a919  <G..c... k/Z.Vy..
```

Bit Planes: Alpha 7 6 5 4 3 2 1 0; Red 7 6 5 4 3 2 1 [✓ 0]; Green 7 6 5 4 3 2 1 [✓ 0]; Blue 7 6 5 4 3 2 1 [✓ 0]

Order settings: Extract By [✓ Row] Column; Bit Order [✓ MSB First]; Bit Plane Order [✓ RGB] GRB RBG BRG GBR BGR

Preview Settings: Include Hex Dump In Preview ✓

A similar problem came about with the png files, so I wrote another script that carved it out of the binary.

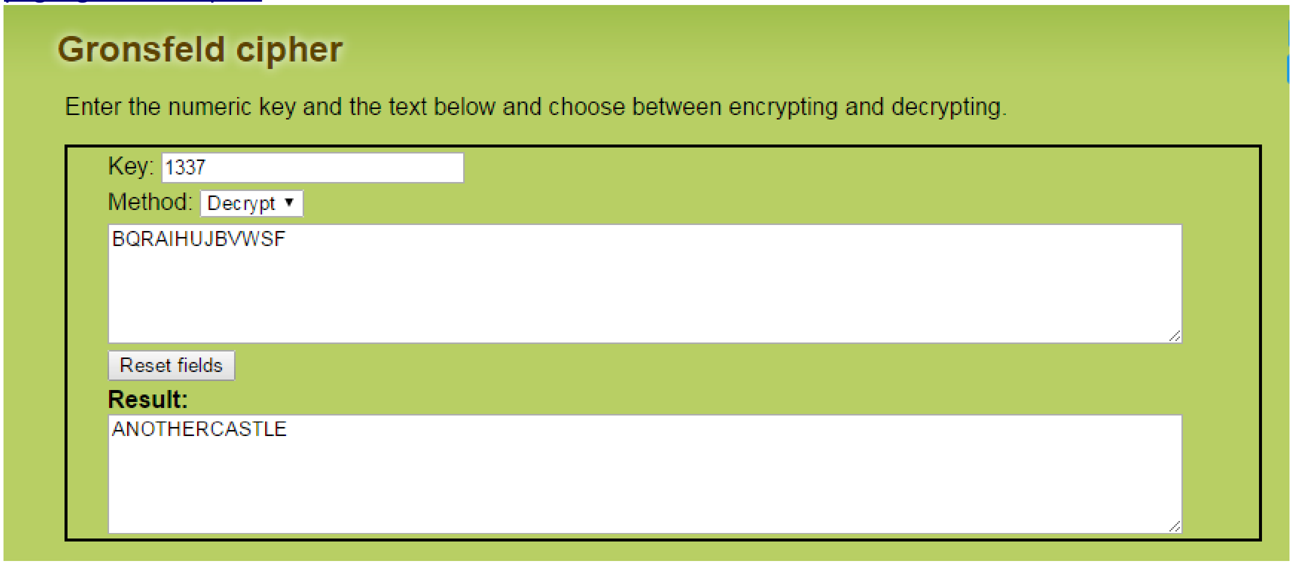
```
data = open("extracted_zip", "r").read()
out = open("cleaned_zip", "wb")
index = 0
for i in range(len(data)):
        #\x50 marks the start of a zip file
        if(data[i] == "\x50"):
                index = i
                break

for i in range(index, len(data)):
        out.write(data[i])

out.close()
```

And finally, I had my zip file.

Stage 3: Unlocking the zip and saving the princess!

This stage took me a little longer than I would have liked. The key to the hint is the use of the word "uncipher", which made me look at classical ciphers such as caeser and viginere. Eventually I decided to google "cipher uses numeric key", and found the gronsfeld cipher uses a numeric key.

I then used the following website to decipher the key: http://www.geocachingtoolbox.com/index.php?page=gronsfeldCipher.



Gronsfeld cipher

Enter the numeric key and the text below and choose between encrypting and decrypting.

Key: 1337

Method: Decrypt

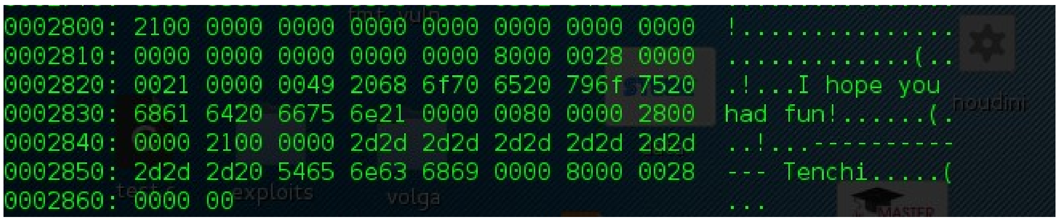
BQRAIHUJBVWSF

Result:

ANOTHERCASTLE

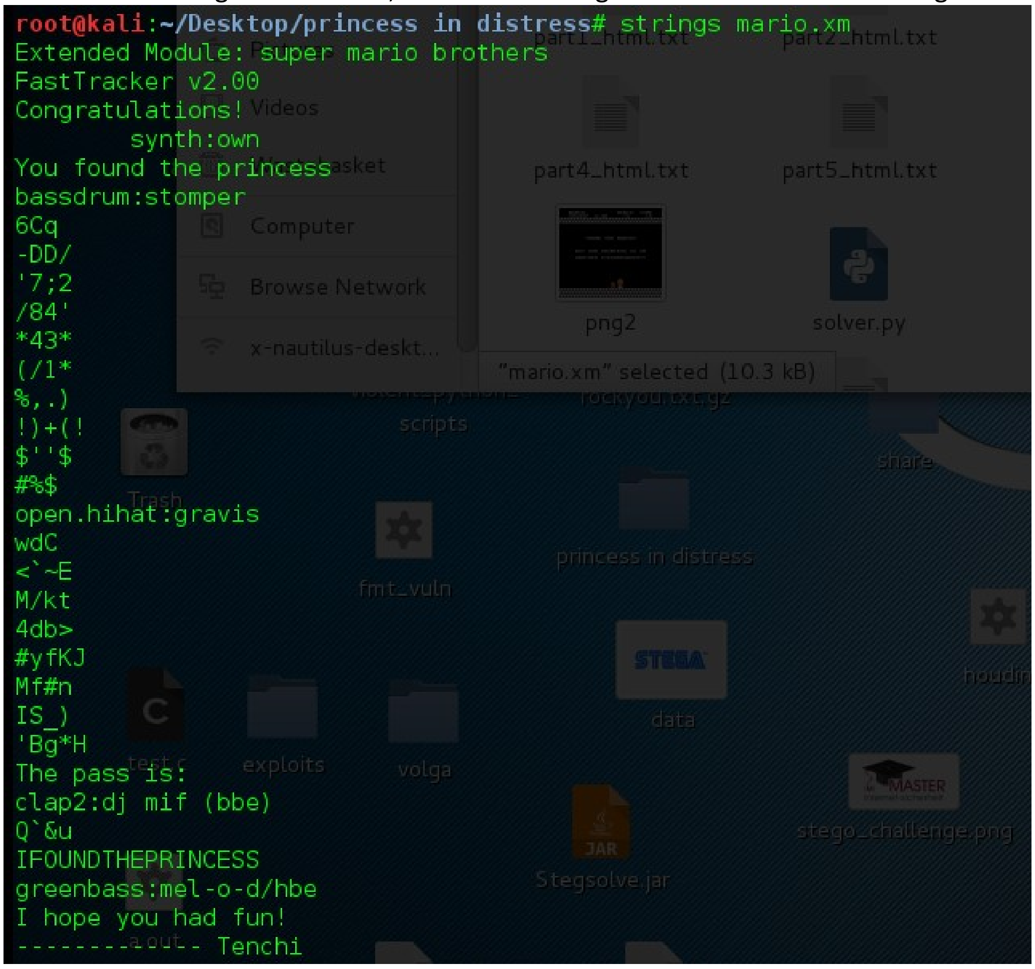
That looked pretty promising, and indeed was the password required to extract an audio file from the zip.

I decided to play the audio file and see if I could spot anything, it sounded a bit sped up, but nothing out of the ordinary. Without expecting too much, I figured it couldn't hurt to inspect the hex dump of the audio file, which showed me it contained ascii strings. Using xxd:



```
0002800: 2100 0000 0000 0000 0000 0000 0000 0000  !...............
0002810: 0000 0000 0000 0000 0000 8000 0028 0000  .............(..
0002820: 0021 0000 0049 2068 6f70 6520 796f 7520  .!...I hope you
0002830: 6861 6420 6675 6e21 0000 0080 0000 2800  had fun!......(.
0002840: 0000 2100 0000 2d2d 2d2d 2d2d 2d2d 2d2d  ..!...----------
0002850: 2d2d 2d20 5465 6e63 6869 0000 8000 0028  --- Tenchi.....(
0002860: 0000 00                                  ...
```

Instead of searching all of the hex, I then used strings to find all of the ascii strings hidden in the audio:



```
root@kali:~/Desktop/princess in distress# strings mario.xm
Extended Module: super mario brothers
FastTracker v2.00
Congratulations!
        synth:own
You found the princess
bassdrum:stomper
6Cq
-DD/
'7;2
/84'
*43*
(/1*
%,.)
!)+(!
$''$
#%$
open.hihat:gravis
wdC
<`~E
M/kt
4db>
#yfKJ
Mf#n
IS_)
'Bg*H
The pass is:
clap2:dj mif (bbe)
Q`&u
IFOUNDTHEPRINCESS
greenbass:mel-o-d/hbe
I hope you had fun!
---------- -- Tenchi
```

And finally, it looks like we have saved the princess, the password (flag) being IFOUNDTHEPRINCESS.