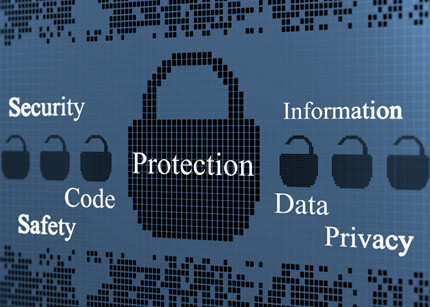
Computer Security Training Platform

## Introduction into Computer Security Competitions



Burns, Tanner | CS4395 – Independent Study | Dr. Qijun Gu

Abstract

Cybersecurity has become increasingly popular in the past few years. Recent hacks have brought cybersecurity into the spotlight, thus increasing the demand for cybersecurity professionals. Merging the gap between academic and real world experience is increasingly difficult because computer security is constantly evolving and becoming more complex.

To help push professionals and devotees, there are competition type events hosted by the government, companies, and schools. These events, known as “capture the flag”, test the competitors on their cybersecurity skillset. There are many different categories that these competition events test. However, for this study I focus on the coding and cryptography categories.

A way to help students learn cybersecurity is to allow them to have an open platform that mimics these competitions. Step by step solutions will also be included with the platform so that they can learn how to approach these problems and overcome unknowns that are found in cybersecurity issues. I will be presenting challenges and the step by step solutions for the challenge to help teach students learn how to solve these competition type problems.

Approach

The information presented in this paper was conducted over this semester. The training platform has a large portion of topics that are common in cyber security. The topics include coding, cryptography, exploitation, reverse engineering, web, and forensics. Security of the training platform itself had to be secure so that they could not cause any denial of service if the platform were to be hosted.

Step by step solutions were made with all the coding and cryptography questions that were created for the platform. This is for the student to be able to learn while still understanding how the real-world competitions work, but also understanding how difficult cybersecurity is. I constructed easy and medium difficulty problems that range in many different factors of coding and cryptography. Abstract and abnormal coding knowledge is essential for the cybersecurity competitions. Therefore, I try to teach the basics of what is involved coding wise so that it is easier when solving a cryptography problem that requires a lot of different types of coding. For cryptography, the problems range from very simple encoding and decoding to the more complex RSA encryption to give a good understanding of the different type of cryptography problems that could rise in cybersecurity.

Outcome

Over the semester, I created a total of 14 problems for the training platform. Four of the problems were coding, eight of the problems were cryptography, and two of the problems were exploitation. Each of the categories had sub categories that were key factors of that main category. I will continue to explain the categories and the security set up further to make the platform safe.

For the coding category, there were several important factors since coding is the basis of all the other categories. Coding for cybersecurity is different than normal coding. Coding, in this sense, is more how to use preexisting tools in a script or how to connect to servers using a script. The exact factors were code conversion (i.e. base64 and hex), string manipulation (i.e. strip and split), file manipulation (i.e. open and close), and remote connection (i.e. pwntools and socket). These sub categories of coding are used in many of the other main categories as well which makes them very essential.

The cryptography category is a challenging one because many factors go into cryptography challenges. Usually these types of problems take more steps and involve math a good portion of the time. The sub categories that are involved in the cryptography problems are substitution xor cipher (i.e. Vigenère cipher), Caesar cipher, AES plaintext and AES EBC, RSA weak public key, hash collision, and hash length extension. These are all common cryptography problems in the competitions.

Exploit is another important category. However, I did not fully implement all the exploit problems that will eventually be on the training platform. The two categories I was able to implement for the training platform were format string exploitation and shell script exploitation. These are just two of the many that are in the competitions.

Servers were required for a good portion of the problems that are in the training platform. With this being the case there must be security put in place so that they cannot be used to exploit the entire platform. To accomplish this, the Linux file system must be secure. I created a script for the platform that sets up the proper security for these servers and starts the servers when the platform is started. This was done by creating users and groups, setting the proper permissions, granting the proper ownership, and running the server as the proper user. When this is correctly set up, even if a user can exploit into their platform, they cannot harm any of the essential tools, files, or folders that keep the platform intact.

Conclusion and Future Works

Cyber security competitions are a great way to learn. Therefor a training platform that gives students the tools they need to learn the basics is the best way to get them interested in being involved in the more challenging competitions. This is truly the closest experience you can get to cybersecurity without being in the workplace. From using statistics gathered during summer research the most useful categories were chosen. These categories are then presented in different problems in the platform to help students learn.

I would like to continue advancing the research and I would like to have the training platform turn into a platform for Texas State to use and train other security students. I believe it would be very beneficial to the Computer Science students and hopefully the entire university. Students will get to have a true taste of computer security through these competitions and platforms.

References

All the security competitions that are hosted every year.

Coding – easy – Gatta Catch Em All

Problem: Find flag in the given text file

Hint: That’s a lot of lines, a script might be useful.

Given: given-easy.txt

Steps:

1) Notice that most the lines end in ==. This is a sign of base64.

Note: Base64 is an ASCII encoding scheme. The = at an end of a string is a good clue. To read more on base64 visit <https://en.wikipedia.org/wiki/Base64>.

2) Write a short python script to open the file and decode each line. Something similar to:

import base64

with open ("given-easy.txt", "r") as file1:

lines=file1.readlines()

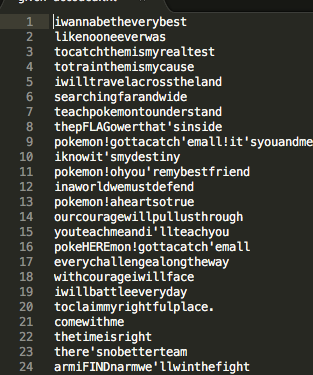
with open (“output.txt”, “w”) as file2:

for i in lines:

file2.write(base64.b64decode(i)+”\n”)

Note: this short script will read in the given file into a list, then open a file called output and write the decoded text to the file. This code uses the base64 library which allows the base64.b64decode(string) and base64.b64encode(string). To learn more about this library visit <https://docs.python.org/2/library/base64.html>.

This is a screenshot of the output file after the text has been decoded. Notice the capital letters at each 8th line.



3) Solution:

8: thepFLAGowerthat'sinside

16: pokeHEREmon!gottacatch'emall

24: armiFINDnarmwe'llwinthefight

32: teacPATThpokemontounderstand

40: youtERNEeachmeandi'llteachyou

From these lines we can see a couple things going on, every 8 lines we are giving text we want and there are only 4 letters given at a time.

Let's make a python script to read this pattern through the entire file and see what we find. Something similar to this code:

with open ("output.txt","r") as file1:

listoftext=file1.readlines()

count=0

flag=""

for i in listoftext:

if ((count+1)%8)==0:

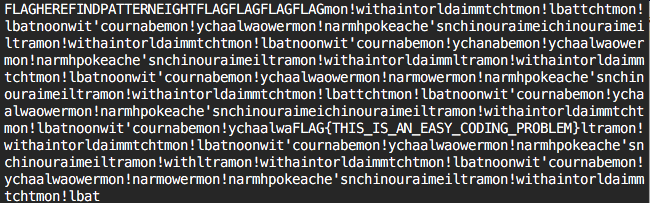
flag+=i[4:8]

count+=1

print flag

Note: this script goes through the list, keeps a counter of the line it is at which is actually (counter +1), since a list starts at 0 index and a file starts at like 1 there is an offset. Then you can mod this number by 8 to check if you are on one of the eighth lines. Then if the remainder is 0 take just the 4th to 8th character in the line. Again index starts at 0 but you want the 5 character until the 10th character but those are actually in locations 4 and 9 index.

This will give the following output:



There is the flag:

FLAG{THIS\_IS\_AN\_EASY\_CODING\_PROBLEM}

Example script:

import base64 #

with open ("given-easy.txt", "r") as file1:

lines=file1.readlines()

#with open("output.txt,"w") as file2: # STEP 1

# for i in lines: # for generating output file with

# file2.write(base64.b64decode(i) # readable text

listoftext=[] #

for i in lines: #

listoftext.append(base64.b64decode(i)) # list contains decoded text

count=0 #STEP 2

flag="" #this is the logic used to extract the

for i in listoftext: #flag from the file

if ((count+1)%8)==0: #

flag+=i[4:8] #

count+=1 #

print flag

Coding-Remote – The Race

Problem: If you can beat this server in a race, it will give you the flag

Hint: The proper tools must be used to pwn this race

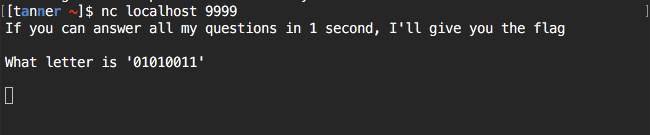
Given: nc 127.0.0.1 9999

Starter notes: Pwntools library will be used in this solution. To learn more about how to use and get pwntools visit this [link](https://github.com/Gallopsled/pwntools).

Steps:

1) Connect to server and see what the output is.

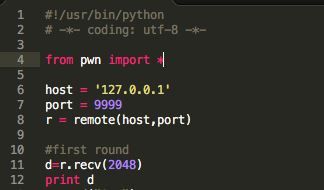
Note: Connect using nc the first time to see the first output of the server



2) Start creating a script that you will build on to beat the speed of the server timeout.

Note: Pwntools will need to be used in a python script to make it easy to connect to the server

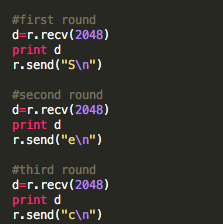
Now that we have seen the first question, let’s start putting the script together.



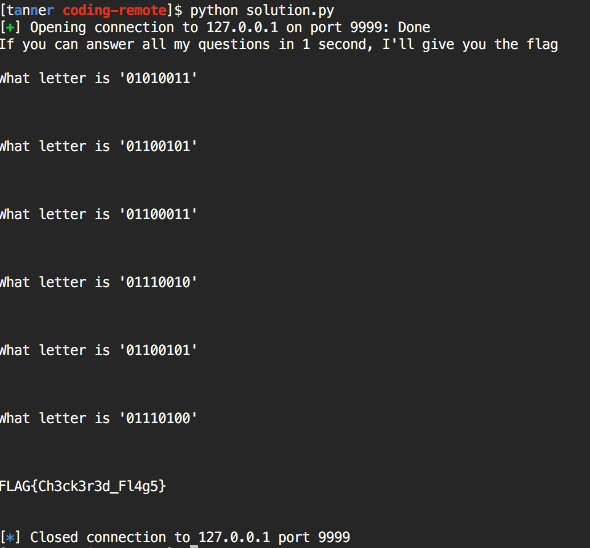
This script will connect to the server and receive that data that the server gives out, which can be seen in the first screenshot. When running this script you would see the same output as in the screenshot.

3) Build on this script until you eventually get a flag as output from the server. The reason a script must be used in this problem is because the server will timeout in 1 second. Meaning there is no possible way you can answer all the questions with human input. The input must be sent as a stream to the server and interpreted by the server in faster than 1 second.

This next screenshot is part of the solution to show how to send data back to the server because you will need to send the answers to the server.



This shows how to send data back to the server. And eventually if you run your solution script the following will show:



Example Solution Script:

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

from pwn import \*

host = '127.0.0.1'

port = 9999

r = remote(host,port)

#first round

d=r.recv(2048)

print d

r.send("S\n")

#second round

d=r.recv(2048)

print d

r.send("e\n")

#third round

d=r.recv(2048)

print d

r.send("c\n")

#fourth round

d=r.recv(2048)

print d

r.send("r\n")

#fifth round

d=r.recv(2048)

print d

r.send("e\n")

#sixth round

d=r.recv(2048)

print d

r.send("t\n")

#flag

d=r.recv(2048)

d=r.recv(2048)

print d

r.close()

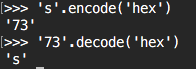
Coding-SuperEasy-Decoding

Problem: I have encoded a string; can you turn it into a flag?

Hint: Hex -> Ascii

Given: 464c41477b336e436f64316e475f346e445f6433633044316e477d

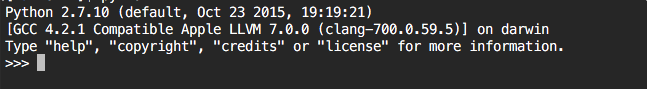
Notes: Encoding and decoding are an easy way to convert strings to a certain encoded type of text or to turn an encoded type of text back to another form of text by decoding it. Simple examples would be if I encode “s” with hex, I am given ‘73’. Now if I decode ‘73’ with hex I am given “s”. This can be seen in the following screenshot.



Steps:

1) We are given a string that we know is encoded. When looking at this string you can see a pattern of characters such as c, e, f, and d. Hopefully these stick out as the letters that are used when counting in hex. Therefore, we know we have a string that was encoded with hex.

2) Next we can easily use python to decode this string for us. First just open the terminal. Then, go to the python interpreter by typing ‘python’ in your command line. It should look like the following:



3) Now enter the command to decode the given string with the argument as hex. The command should look as: (do not copy and paste from Microsoft word, command will not work)

‘464c41477b336e436f64316e475f346e445f6433633044316e477d’.decode(‘hex’)

Then the string that was encoded with hex will be returned to you. This can be seen in the following screenshot.



Coding-Remote-Dynamic

Problem: If you answer 10000 problems, the flag is yours. I want the decimal version of these numbers.

Hint: Your pwn level is over 9000!

Given: nc 127.0.0.1 11111

Notes: Pwntools library will be used. To learn more about how to use and get pwntools visit this [link](https://github.com/Gallopsled/pwntools).

Steps:

1) Connect to the server and see what the output is.

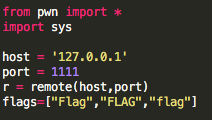
Note: before you start making your script is easier to connect using netcat and see the output.



2) From connecting to the server, you can notice that the information that is sent back is in base64. Let’s decode this base64 string that the server gave us and see what it is.

NTY1== 🡪 565

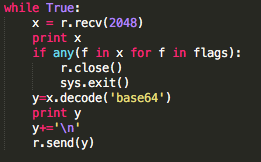
3) Now we know that we are given a base64 encoded version of a decimal number. From the problem statement we know we have to give the server the correct decimal value for 10000 different problems. Time to build a script. This script should, 1) connect to the server, 2) get info from the server, and 3) send the data back to the server. This needs to be in a loop to go until the server gives you the flag.



Note: This short snippet of the script connects to the remote server at 127.0.0.1 at port 1111.

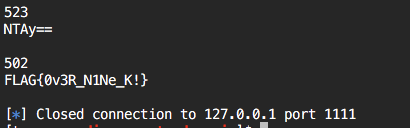
Also, notice that there is a list of flags keywords. Since we know the format of the flags start with the word flag, then keep this list to check if the information we get back from the server is the flag.

4) Now we will create the loop to get info from the server and send information back to the server.



Note: We make a loop of while true to keep going while the server is sending back information. The variable ‘x’ receives the data for the server. We then use the list of flag keywords we created to check if the data we are getting from the server is the flag. If we receive the flag, we want to close the connection. Then if ‘x’ is not the flag we want to decode the base64 string we received to turn it back to the decimal value. We then can print out the number we are sending back (more of a debugging feature). IMPORTANT: we are mimicking a raw\_input() type of input in python. This means our input we send to the server needs to have a newline included which is copying when you hit enter after giving the string. We then send this data to the server.

If the data we send back passes 10000 answers, we should receive the flag.



Sample script:

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

from pwn import \*

import sys

host = '127.0.0.1'

port = 1111

r = remote(host,port)

flags=["Flag","FLAG","flag"]

while True:

x = r.recv(2048)

print x

if any(f in x for f in flags):

r.close()

sys.exit()

y=x.decode('base64')

print y

y+='\n'

r.send(y)

r.close()

Crypto – Caesar Cipher

Problem: If you can decrypt this file, you might find a flag

Hint: I like my salad with two ingredients

Given: file.enc

Steps:

1) When the file is opened, it will be obvious that the first step is to decode the base64.



A command to decode this is as follows (note: this is a python command but there are many other commands that can be used to decode base 64)

Command: python –m base64 –d <<< REpZRXtBeUNxeVBfR3FfQ3lRd30K==

Produces: DJYE{AyCqyP\_Gq\_CyQw}



2) Now there is a few different things you could do. These easiest would be to think about what this text looks like, four letters followed by {. That is a good indicator that it said flag{. One that has seen Caesar cipher would know that it looks like these characters have been shifted in some way.

Easiest solution: take the text ‘DJYE{AyCqyP\_Gq\_CyQw}’ and put this into an online Caesar cipher decoder. Such as this [link.](https://www.nayuki.io/page/automatic-caesar-cipher-breaker-javascript)

After you insert the text you will be given the flag and the key.

Key: 24

Flag: FLAG{CaEsaR\_Is\_EaSy}

3) Another solution could be to write a python script to crack caesar ciphers in the future.

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

import sys

def main():

key = sys.argv[1]

text = "DJYE{AyCqyP\_Gq\_CyQw}"

print decryptMessage(key,text)

def decryptMessage(key,message):

translated = ""

for i in range(0,len(message)):

if message[i].isalpha():

if message[i].isupper():

char = chr((ord(message[i])-ord('A')+int(key))%26 + ord('A'))

translated+=char

else:

char = chr((ord(message[i])-ord('a')+int(key))%26 + ord('a'))

translated+=char

else:

translated+=message[i]

return translated

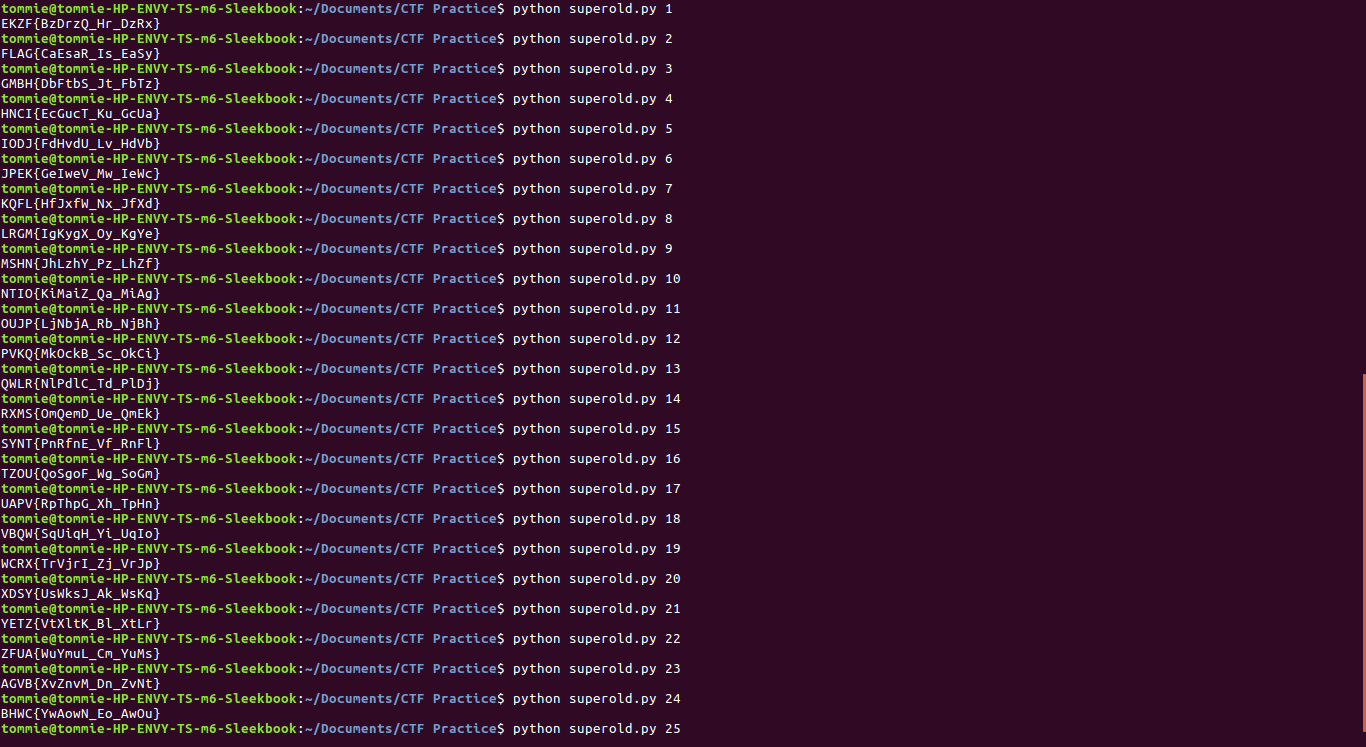
if \_\_name\_\_ == '\_\_main\_\_':

main()

This script takes each character and cycles it by the key. With this script the key is input in the system argument so you can test different keys in the future for a different Caesar cipher.

The equation in python has to use capital A and mod 26 to rap around with capital letters and a with lower case letters.

Below is the result of using brute force to test keys until you get an obvious result. Notice that when you enter a key argument of 2, you get a legible result, “FLAG{CaEsaR\_Is\_EaSy}”.



Easy – Hash

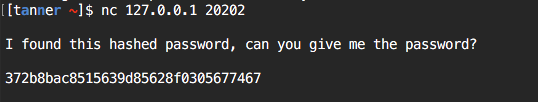
Problem: I found a hashed password. If you can tell me the password, I can log in and get you the flag.

Hint: It looks like an md5 but who knows.

Given: nc 127.0.0.1 20202

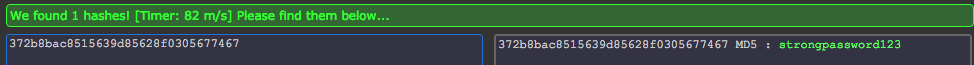
Steps:

1) Connect to the server and get the hash.



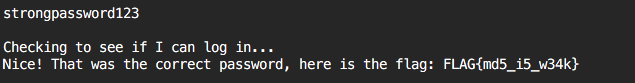
Note: This is an unsalted md5 hash. To learn more about md5 hashing visit this [link](https://en.wikipedia.org/wiki/MD5).

2) Now that we have the hash, we need to figure out the password and send it back to the server. There are many sites to crack a simple md5 hash. Such as this [link](https://hashkiller.co.uk/md5-decrypter.aspx). Copy and paste the hash into this website and it will give you the password.



Note: An md5 hash can easily be cracked nowadays. There are many sites that have databases stored with md5 hashes to easily determine what the hash was before the encryption.

3) Now that we have the password, we can give it to the server and receive the flag.



Flag = FLAG{md5\_i5\_w34k}

Easy RSA

Problem: We encrypted the flag with RSA, can you crack it?

Hint: Short public key

Given: easy-rsa.tar.gz (custom1.enc, custom2.enc, custom3.enc, and public\_key.pem)

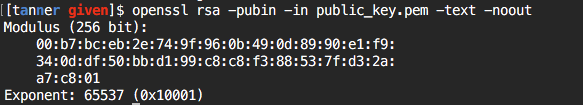
Steps:

1) Understanding the public key that is in the given folder. The public key file is the public\_key.pem file.

If you try to open this file regularly it will not open correctly. Lucky there is a tool called openssl that can make this file very readable. The following command will give you your public\_key file in a readable text

$openssl rsa –pubin –in public\_key.pem –text –noout

The following will be given:



We now have our public key(modulus), and our exponent that will be needed for decryption.

The current modulus is not usable as is. It needs to be converted to the decimal version. This can be done easily by using the python interpreter. We get to this by simply typing python in our terminal. After we are inside the python interpreter we need to create a few variables that will be used. Something like the following:

>>> line1=’00:b7:bc:eb:2e:74:9f:96:0b:49:0d:89:90:e1:f9:’

>>> line2=’32:0d:df:50:bb:d1:99:c8:c8:f3:88:53:7f:d3:2a:’

>>> line3=’a7:c8:01’

>>> total=””

>>> line1=line1.split(‘:’) #going to split up the string at each colon and put it in a list

>>> line2=line2.split(‘:’)

>>> line3=line3.split(‘:’)

>>> for i in line1:

… total+=i #this will append each of the list elements together 00b7bceb2e749f…

>>>for i in line2:

… total+=i

>>>for i in line3:

total+=i

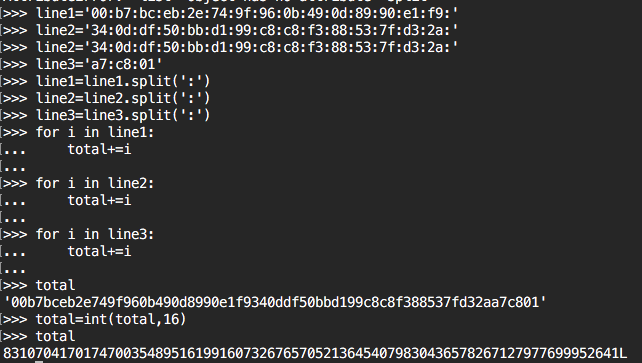
Now total will have all the individual list elements in a single string and should look like the following:

00b7bceb2e749f960b490d8990e1f9320ddf50bbd199c8c8f3885375d32aa7c801

Now we want to convert this to a decimal value.

>>>total=int(total,16) #this convers hex to decimal

The following is a screenshot of the above commands (note total=”” was already initialized before screenshot)

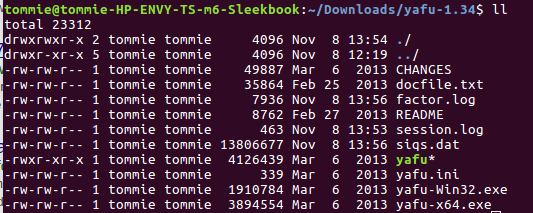


We now have to long integer which is our new modulus in decimal:

831070417017470035489516199160732676570521364540798304366578267127977699952641L

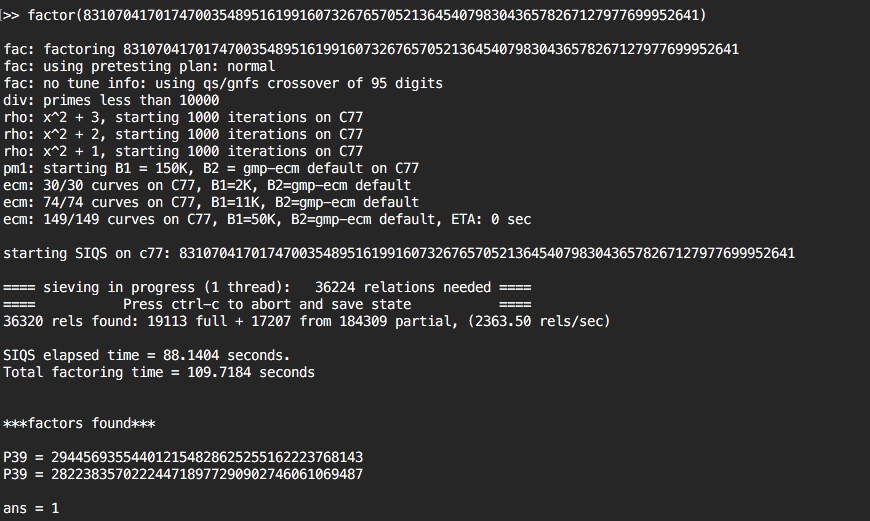
2) Now that we have the decimal version the modulus we need to get the two largest prime factors of this modulus. There is a great tool called ‘yafu’ that can be used to do this for us.

Yafu can be read about at this [link](https://github.com/DarkenCode/yafu). Yafu can be installed [here](https://sourceforge.net/projects/yafu/). After installation, make sure that you have the correct permissions to execute the tool. Enter ‘ll’ in the directory containing yafu to list permissions.



If yafu is not executable (permissions don’t contain the character ‘x’), try running ‘chmod 755 yafu’. Run yafu by entering ‘./yafu’.

Now we will factorize this integer to get our P1 and P2 which are needed for the decryption of rsa. When running this tool it looks like the following and will take a little time.



We now have out P1 and P2. We can now start the decryption process.

P1 = 294456935544012154828625255162223768143

P2 = 282238357022244718977290902746061069487

3) Now we can begin understanding our script to decrypt the rsa files we were given.

I will go over some variables that you will need to know to go over decryption.

z = our file

c = cipher text as number

n = our long decimal value of the modulus from the public key

e = the exponent given in the public key

P1=prime factor 1 of the modulus key

P2=prime factor 2 of the modulus key

r=(prime factor 1 – 1) multiplied by (prime fact2 -1) provides our totient number

d=the decryption key

p=the plaintext as number

pt=plaintext in a string representation

There are also a couple libraries in python that we will need to have for our decryption script.

-gmpy

-Crypto.Util.number

-base64

4) Now we can begin to build our script that will decrypt the given rsa files.

First lets open the first file custom1.enc and read it as encoded hex values

z = open(‘custom1.enc’,’r’).read().encode(‘hex’)

Now we turn the cipher text to a number

c = int(z,16)

Set n to be our long modulus number we got in step 2

n=83107041701747003548951619916073267657052136454079830436578267127977699952641

Set e to the exponent we were given in our public key

e = 65537

Set P1 and P2 that we obtained from the yafu tool

P1 = 282238357022244718977290902746061069487

P2 = 294456935544012154828625255162223768143

Set r to the equation to generate totient number

r = (P1-1)\*(P2-1)

For the d variable we user gmpy to get our invert of the e over r. We need to use the gmpy library that we imported earlier

d = int(gmpy.invert(e, r).digits())

Now set p to generate our plaintext using the rsa equation c^d%n

p = pow(c,d,n)

Let pt be your new plaintext that is decoded from hex into a readable string then print it to receive the plaintext. To use the decode, pt must be an even number.

pt = hex(p).strip('0x').strip('L').decode('hex')

print pt

We now have our script set up to decrypt rsa with a short public key. Go through and replace each file name with the custom1.enc, custom2.enc, and custom3.enc and concatenate the new string into your flag

FLAG{R5A\_i5\_n00b}

Sample Script:

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

import gmpy

from Crypto.Util.number import \*

import base64

z = open('custom3.enc','r').read().encode('hex')

c = int(z,16)

n = 83107041701747003548951619916073267657052136454079830436578267127977699952641

e = 65537

P1 = 282238357022244718977290902746061069487

P2 = 294456935544012154828625255162223768143

r = (P1-1)\*(P2-1)

d = int(gmpy.invert(e, r).digits())

p = pow(c,d,n)

#print hex(p)

#print p

pt = hex(p).strip('0x').strip('L').decode('hex')

print pt

Messy Aes

Problem: We were playing around with AES encryption. We seem to have lost our flag. Can you find it?

Given: po6vko2hb574zjwxurkayebnio33mnmuhwhhvtwk2tlqsghnasyb2kfmzu7ckz7lrvcabdsmisereryubq

Hint:

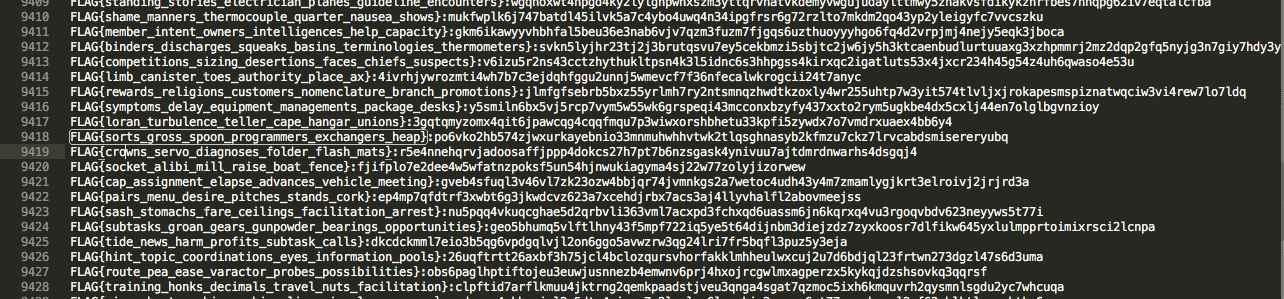
Solution)

This problem has a very easy solution. It is simply a lot of encrypted flags and you just have to find the correct one.

Simple way:

Ctr-f and search for the first 8 bytes: po6vko2hb574zjwx

This leads to the flag : FLAG{sorts\_gross\_spoon\_programmers\_exchangers\_heap}



Longer but simple way: make a short script to search the file.

##needed string##

#po6vko2hb574zjwxurkayebnio33mnmuhwhhvtwk2tlqsghnasyb2kfmzu7ckz7lrvcabdsmisereryubq

##first 8bytes

fsb='po6vko2hb574zjwx'

lines=[]

with open('given.txt') as f:

for line in f:

if fsb in line:

print "Found flag:\n"

print line

This also gets the flag: FLAG{sorts\_gross\_spoon\_programmers\_exchangers\_heap}

Hash Collision

Problem: Can you log into our application and retrieve the flag?

Given: nc 127.0.0.1 30003; logic file

Hint: There might be a collision in our logic.

Steps)

1) Understanding the given logic file:

-We see that there is a username and password required.

-username = boko

-password = complexPasswordWhichContainsManyCharactersWithRandomSuffixeghjrjg

-More importantly we see that the password is encrypted with a hash function called pbkdf2sync. This is a function for computing a shasum of the password given. If you research this function you will find out that it is vulnerable because it can also take a already hashed version of the same password that is computed by another shasum function and pass the password checking logic. #Add link about the vulnerable function

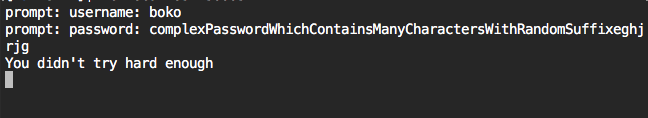
What this means if when this functions encrypted the password:

complexPasswordWhichContainsManyCharactersWithRandomSuffixeghjrjg

It computes a shasum of this password.

2) Connect to the server and try and log in.

- Let’s try the username and password that we can see in the logic first.



-This looks like we hit the if password if statement and the logic in there told us we did not try hard enough.

3) Generate hash and try to create hash collision

-From step 1 let’s recall that the function for the encryption is using a shasum. Let’s explore this in more detail.

There are also other ways to compute a shasum of a password. Such as the following:



The command:

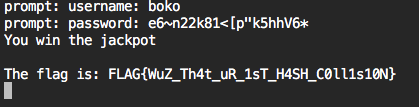
echo -n 'complexPasswordWhichContainsManyCharactersWithRandomSuffixeghjrjg' | shasum | xxd -r –p

Returns:

e6~n22k81<[p"k5hhV6\*

-We now have a hashed version of the password we want to use to log in. If we feed this hash string, we will create a hash collision inside the pbkdf2Sync function. To learn more about hash collision visit this [link](https://en.wikipedia.org/wiki/Collision_(computer_science)).

-Lets now try and log in and create the hash collision



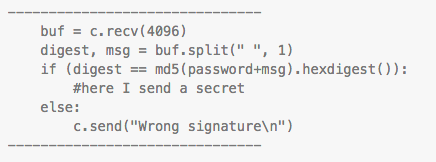
-We now created a hash collision and were able to exploit the pbkdf2Sync function and log into the application and get the flag.

Hash-Length-Extension(Hle)

Problem: We have found one authorized message “c974b779d095f5772a36e2139276ffdc testing connection”.

Can you find another authorized message and send it to nc 192.168.3.5 41300.

Given:



Hint: Can you extend your luck?

Intro:

This is a clear hash length extension problem. Luckily there are tools to help solve this so it does not have to be brute forced. To learn more about hash length extension visit this [link](https://en.wikipedia.org/wiki/Length_extension_attack). There are a few tools out there to help with this type of attack. The one that will be used for this write up is called hashpump and can be found [here](https://github.com/bwall/HashPump).

Steps:

1) Lets discuss what we know and don’t know from the given code and the authorized message we have.

First, we have a few important variables. digest and message which are what you want to manipulate and the password that is hidden. Since we don’t know the password we need to generate a digest with the part of the msg that we know.

We know that part of the message is ‘testing connection’. We also have the hash given from their password and the message we know. We now need to generate a new hash with the known message that will then match the digest.

2) Start making a script for the hash length extension. The following photo shows our known information and how to connect to the server to send and receive data.



Now we want to create a loop to use our test function to send our extended hashes that are generated by hashpump to the server and check if it is the correct answer. We want to create a for loop that calls the hashpump executable with certain options. This command will generate the hash and send it to the server. The following screenshot is how to set up the command line for hashpump.



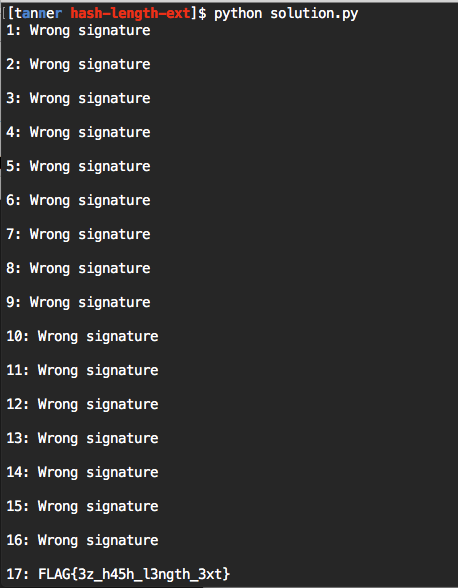
In this screenshot you can see how the full command will look like when calling hashpump:

./hashpump –s known\_sig –d \”+known\_text+\” –k str(i) –a “AAAA”

Then the next line uses the command call to call the hashpump file and get its output into a variable. Then split the hex string at ‘\\x80’(space) of the returned string from the hashpump command. Then strip out the ‘\\x’ hex beginning of each string and decode in hex. Now lastly, put the complete hash length extended string together and send it to the server.

Then we continue until we are not given wrong as the response from the server.

The solution then gives the following ouput:



Sample Script:

import socket

import commands

host = "192.168.3.5"

port = 41300

known\_sig = "c974b779d095f5772a36e2139276ffdc"

known\_text = "testing connection"

def test(sign, text):

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM);

sock.connect((host, port))

data = sign + " " + text

sock.send(data)

return sock.recv(4096)

for i in range(1,32):

cmd = "/usr/local/Cellar/hashpump/1.2.0/bin/hashpump"

cmd += " -s " + known\_sig

cmd += " -d \"" + known\_text + "\""

cmd += " -k " + str(i)

cmd += " -a " + "AAAA"

sign, text = commands.getoutput(cmd).split('\n')

a, b = text.split('\\x80')

b = b[:-4].replace('\\x','').decode('hex')

data = a + '\x80' + b + 'AAAA'

res = test(sign, data)

print str(i) + ': ' + res

if not 'Wrong' in res:

break

Aes-cbc-IV(Poison IV)

Problem: Can you give me the IV to make the current plain text that is encrypted with aes to decrypt to the second plain text. Send the IV needed to chain the plain text to nc 192.168.3.5 64444

Given:

**Original**:

Pass: sup3r31337. Don't loose it!

**Encrypted**:

4f3a0e1791e8c8e5fefe93f50df4d8061fee884bcc5ea90503b6ac1422bda2b2b7e6a975bfc555f44f7dbcc30aa1fd5e

**IV**:

19a9d10c3b155b55982a54439cb05dce

**Decrypted**:

Pass: notAs3cre7. Don't loose it!

Hint: Study up on aes-cbc first block byte swapping

Steps)

1. Understanding aes-cbc. Aes is a type of encryption that encrypts blocks of data. To learn more about aes and the different types visit [link](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard).



AES is a block cipher, which means that plaintext is split into blocks: every block is encoded with an encryption key of an equal length (128, 192 or 256 bits in case of AES). By itself, a block cipher is only suitable for secure transmission of one block; in order to encode larger amounts of data, various modes of operation were introduced. CBC (Cipher Block Chaining) is one of such modes. To encrypt a block in CBC mode each block's plaintext is XORed with the preceding block's ciphertext (or IV for the first block), then encoded with a chosen algorithm (AES in our case). CBC is widely-used, but because of its properties it's vulnerable to byte-flipping attacks: when you change a byte in a block's ciphertext, the byte in the same position of the next block's plaintext gets changed because of the XOR operation.

1. Understand the problem given. Firstly, we are given the original plain text. Second, we don’t know any key size. Third, we are given the encrypted version of the original plain text. Fourth, which is important, the initial vector used for the encryption. Lastly, the decryption version of text that we want to obtain.

What we want: We want to rearrange the first block of the plain text to be the desired plain text after decryption. The initial vector that we are given would produce the original plain text when used to decrypt so we want to make a new initial vector that would change the first block of the aes decryption to the decrypted plain text.

1. The way we start to get our answer is by using the mathematical equation used by aes.

C = Enc(IV xor M)

C = Enc(IV' xor M')

So, IV xor M == IV' xor M'

C=cipher text

IV=initial vector

IV’=new initial vector to produce our wanted decrypted text (need to find)

M=first plain text

M’ = second plan text

1. Start a script to rearrange the IV to produce the desired plain text.

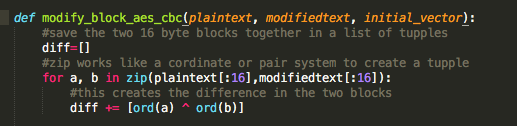
First, create a list variable. Ie such as diff=[]

Second, use a for loop and zip to make a list of tupples. This is basically a coordinate type system that would look like a list like the following:

[x,y],[x1,y2],[x2,y2],…

Third, in the for loop make your list contain (x[plaintext] xor y[modifiedtext])

Such as :

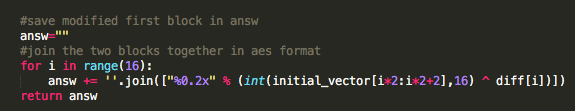


Now, we have a list generated of the given plaintext and the desired plaintext xored together.

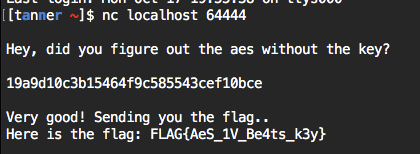
We will begin to generate the new IV now. First a variable will be needed to store the new IV. Such as answ=””

Now we want to loop through 0 to 16 which is the first 16 bytes. In this loop we want to join the bytes using a certain equation to generate the new IV that swaps the bytes to the desired letters. We use the initial vector and swap it around to generate the new IV that gives the desired plain text.

We want to take the IV turn it to a decimal value and swap it using the following equation for the index: i\*2:i\*2\*2. Then xor that part of the IV with the list we created earlier. This looks like the following:



1. Then you want to connect to the server and send it the IV you have created and try to get back the flag.



Sample Script:

def modify\_block\_aes\_cbc(plaintext, modifiedtext, initial\_vector):

#save the two 16 byte blocks together in a list of tupples

diff=[]

#zip works like a cordinate or pair system to create a tupple

for a, b in zip(plaintext[:16],modifiedtext[:16]):

#this creates the difference in the two blocks

diff += [ord(a) ^ ord(b)]

#save modified first block in answ

answ=""

#join the two blocks together in aes format

for i in range(16):

answ += ''.join(["%0.2x" % (int(initial\_vector[i\*2:i\*2+2],16) ^ diff[i])])

return answ

#modies first block aes-cbc $1=plaintext $2=modifiedtext $3=given Initial Vector

print modify\_block\_aes\_cbc('Pass: sup3r31337','Pass: notAs3cre7','19a9d10c3b155b55982a54439cb05dce')

V-Cipher(Vigenere ^ Cipher)

Problem: We encrypted a file using a vigenere xor cipher. Can you get the flag?

Given: vfile.enc

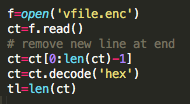
Hint: Frequency

Info: A vegenere cipher is a repeating key cipher, in this case it was xored with the plain text.

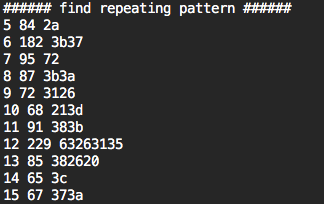
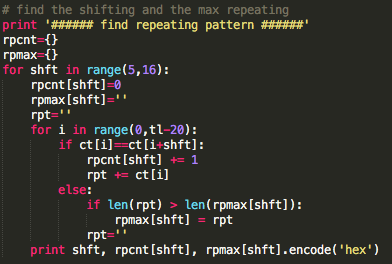
To learn more about how a vegenere cipher works visit this [link](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher).

Steps:

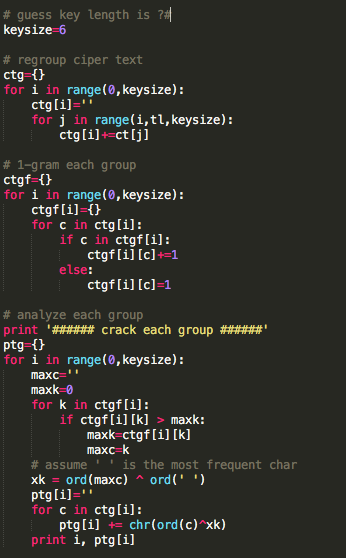
1) First thing to start with is opening the given file and reading it into a variable. Then you want to make sure it is an even length so that you can decode the hex version. Most likely there will be a new line which is the reason you want to strip the last character from the file. The following is a screenshot of how it would look. You also want to save the length of the text.



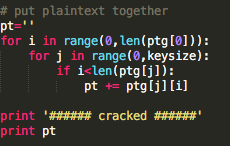
2) The second part and most important is finding the max repeating patterns in the cipher text and finding the largest shifting point. This can be done in many ways, but the rough idea is to take a pattern of a string such as ‘abc’ and see if ‘abc’ is repeated again later in the cipher text. You want to repeat this for strings from length 2 to roughly 16. This is also known as calculating the frequency of the text which is based on the frequency of the English alphabet which can be found easily online. After you have done this part of your script you should know a rough keysize. The following screenshots show the code for analyzing the patterns and the output of the patterns.



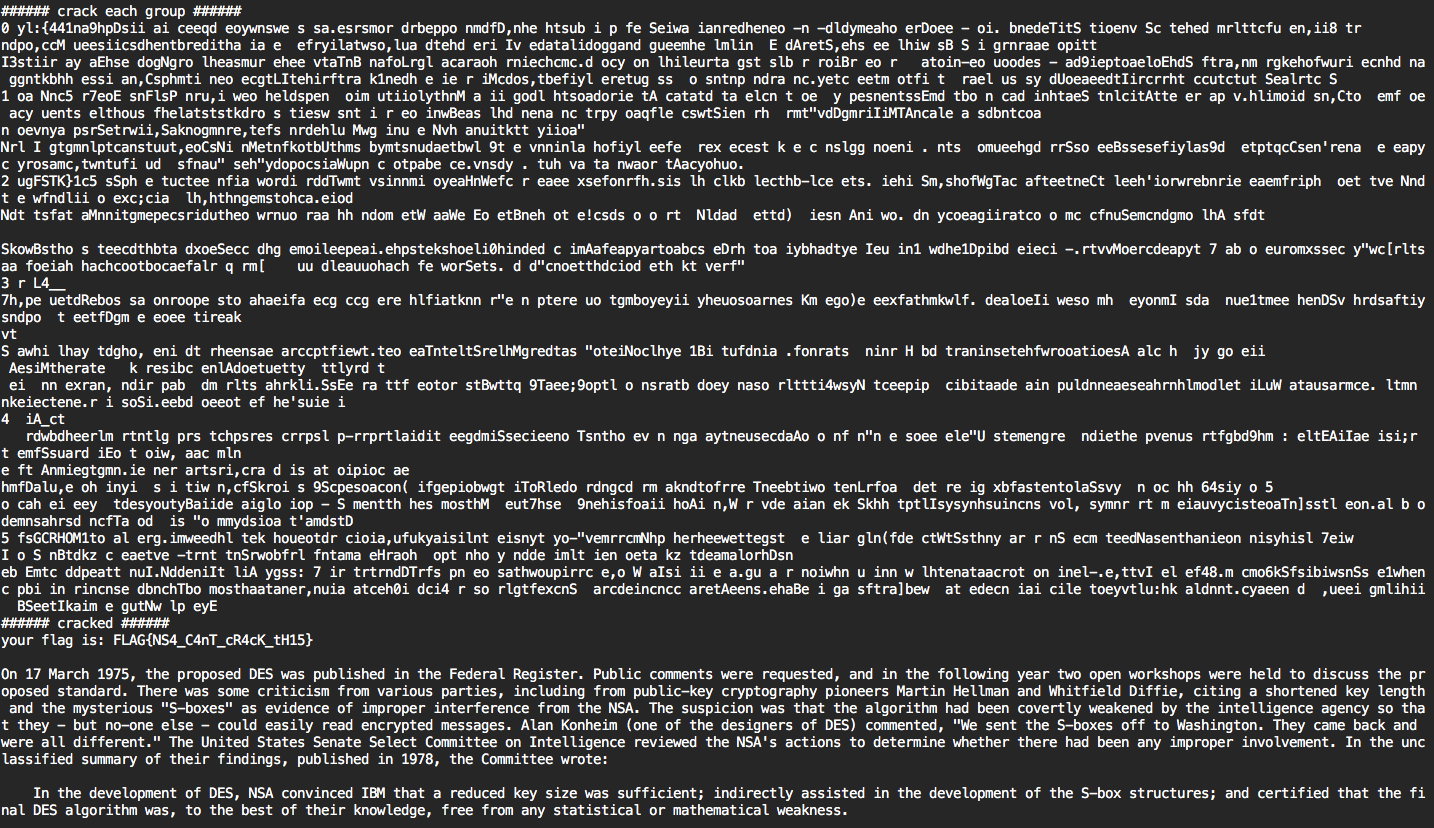
3) It can easily be seen that 12 and 6 have the most repeating patterns, meaning one of these is most likely the key length. Usually it is better to start at the lowest factor number, meaning try 6 first. Then you will want to guess the key size is equal to 6. Then you want to regroup the cipher text based on the key size. Which puts the cipher text into smaller sections(columns) of the cipher text with the length of 6. Then you want to use the 1 gram frequency for the alphabet and swap the characters accordingly to the English alphabet frequency. Normally in a frequency chart. The most used character is not a space because the normal frequency is only 0-25 which is A-Z. In our cipher text we want to assume that the most used character is a space(‘ ‘). We then analyze the cipher text blocks that we created to swap the character based on their high frequencies. This can be seen in the following screenshot.



4) Now after we have analyzed each group with our guessed key length as 6. We can now begin to put the cipher text back together which will now be converted into our plain text. Since we had our blocks of text we need to merge them back together which will become our plain text. This can be done like the following:



5) When you run the full script you should get some output like the following:



Sample Script:

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

f=open('encrypted.txt')

ct=f.read()

# remove new line at end

ct=ct[0:len(ct)-1]

ct=ct.decode('hex')

tl=len(ct)

# find the shifting and the max repeating

print '###### find repeating pattern ######'

rpcnt={}

rpmax={}

for shft in range(5,16):

rpcnt[shft]=0

rpmax[shft]=''

rpt=''

for i in range(0,tl-20):

if ct[i]==ct[i+shft]:

rpcnt[shft] += 1

rpt += ct[i]

else:

if len(rpt) > len(rpmax[shft]):

rpmax[shft] = rpt

rpt=''

print shft, rpcnt[shft], rpmax[shft].encode('hex')

# guess key length is ?#

keysize=6

# regroup ciper text

ctg={}

for i in range(0,keysize):

ctg[i]=''

for j in range(i,tl,keysize):

ctg[i]+=ct[j]

# 1-gram each group

ctgf={}

for i in range(0,keysize):

ctgf[i]={}

for c in ctg[i]:

if c in ctgf[i]:

ctgf[i][c]+=1

else:

ctgf[i][c]=1

# analyze each group

print '###### crack each group ######'

ptg={}

for i in range(0,keysize):

maxc=''

maxk=0

for k in ctgf[i]:

if ctgf[i][k] > maxk:

maxk=ctgf[i][k]

maxc=k

# assume ' ' is the most frequent char

xk = ord(maxc) ^ ord(' ')

ptg[i]=''

for c in ctg[i]:

ptg[i] += chr(ord(c)^xk)

print i, ptg[i]

# put plaintext together

pt=''

for i in range(0,len(ptg[0])):

for j in range(0,keysize):

if i<len(ptg[j]):

pt += ptg[j][i]

print '###### cracked ######'

print pt