Will Hawkins, Brian Hotopp, and Dan Stevens

November 2021

Cryptography and Network Security

Secure Banking Transactions: White Hat Final Project Write Up

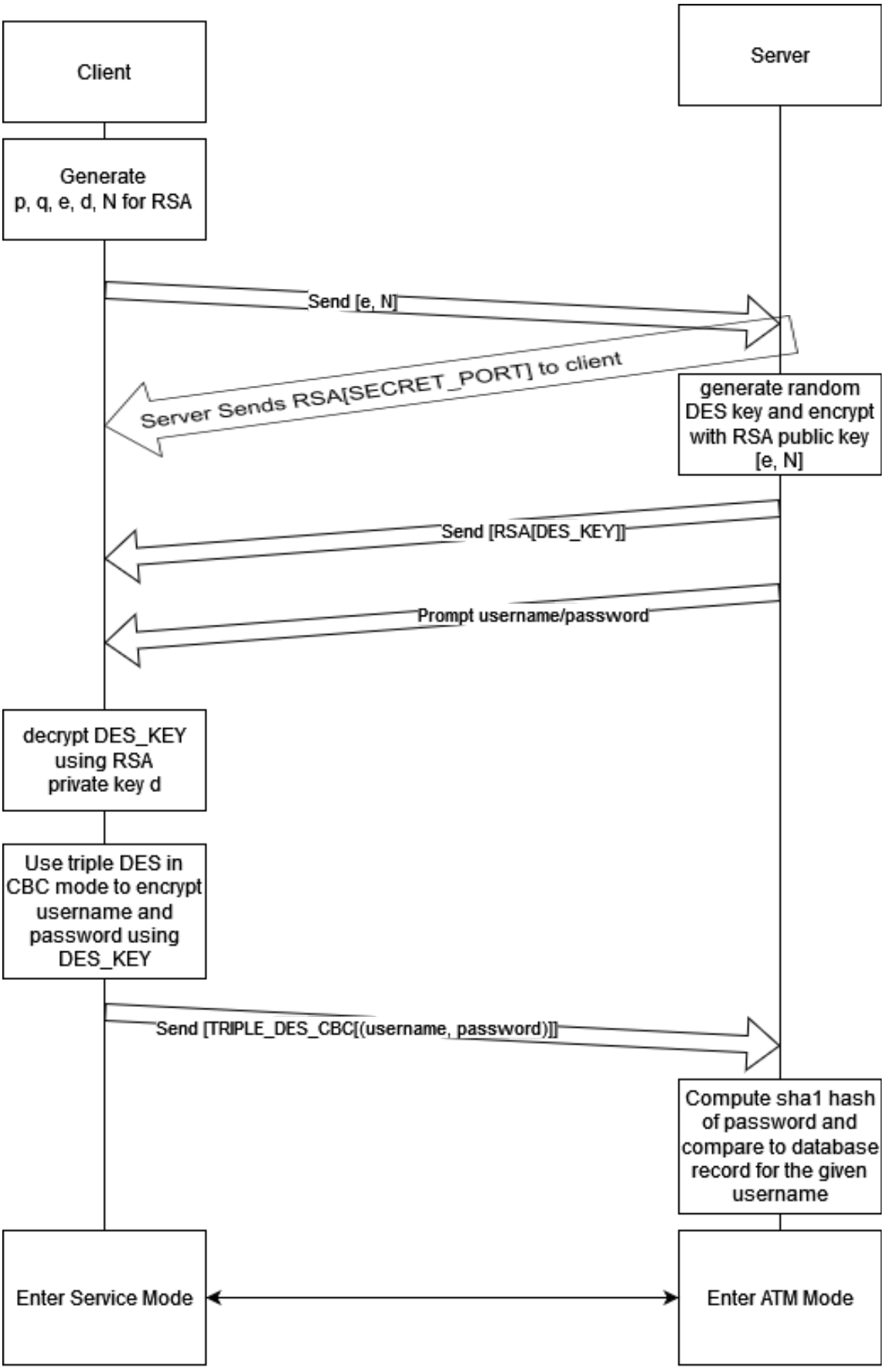
Client/Server

To implement our banking protocol, we used the python standard library socket interface. Our protocol has two main stages; first our client and server use RSA to exchange a 192-bit private DES key, and then the client and server enter “service” and “ATM” mode respectively to perform banking transactions. In this authenticated second stage, the client sends commands to the server, and polls for responses. Similarly, the server polls for commands from the user and sends responses to report successful or unsuccessful banking operations. Both the client and server operate asynchronously in this mode, each on their own thread. At this time our online banking system only supports a single user at a time, similar to the functionality of a conventional ATM.

Authentication and Key Exchange:

Our server is launched by running python3 tcp.py <HOSTNAME> <PORT>. Once the server is running, customers can launch our client by running python3 client.py <HOSTNAME> <PORT> <SECRET PORT>. From this point, the client will generate p, q, e, d and N variables to perform RSA encryption and decryption. It sends the generated public key [e, N] to the server. Ther server then encrypts the secret port and sends it to the client. The server then closes the connection to the original port and opens a new connection on the secret port. The client decrypts

the secret port from the server using RSA and connects to the new port. The server also uses RSA to encrypt a randomly generated DES key, which is then sent back to the client. Once this handshake is completed, both the client and the server have possession of a 192-bit DES key, which is used for the rest of their interactions. Before the client can begin issuing banking commands to the server, it must send its banking username and password, encrypted using triple DES in CBC mode to the server. The server decrypts this information using the DES private key, and compares a hash of the given password to the database's recorded hash for the given user. If the hashes match, the server enters ATM Mode and notifies the client that it should enter Service Mode. If the username and password do not correspond to a valid login the connection is terminated. If the user wishes to attempt to login again they must restart the protocol. A diagram of this interaction can be seen below.

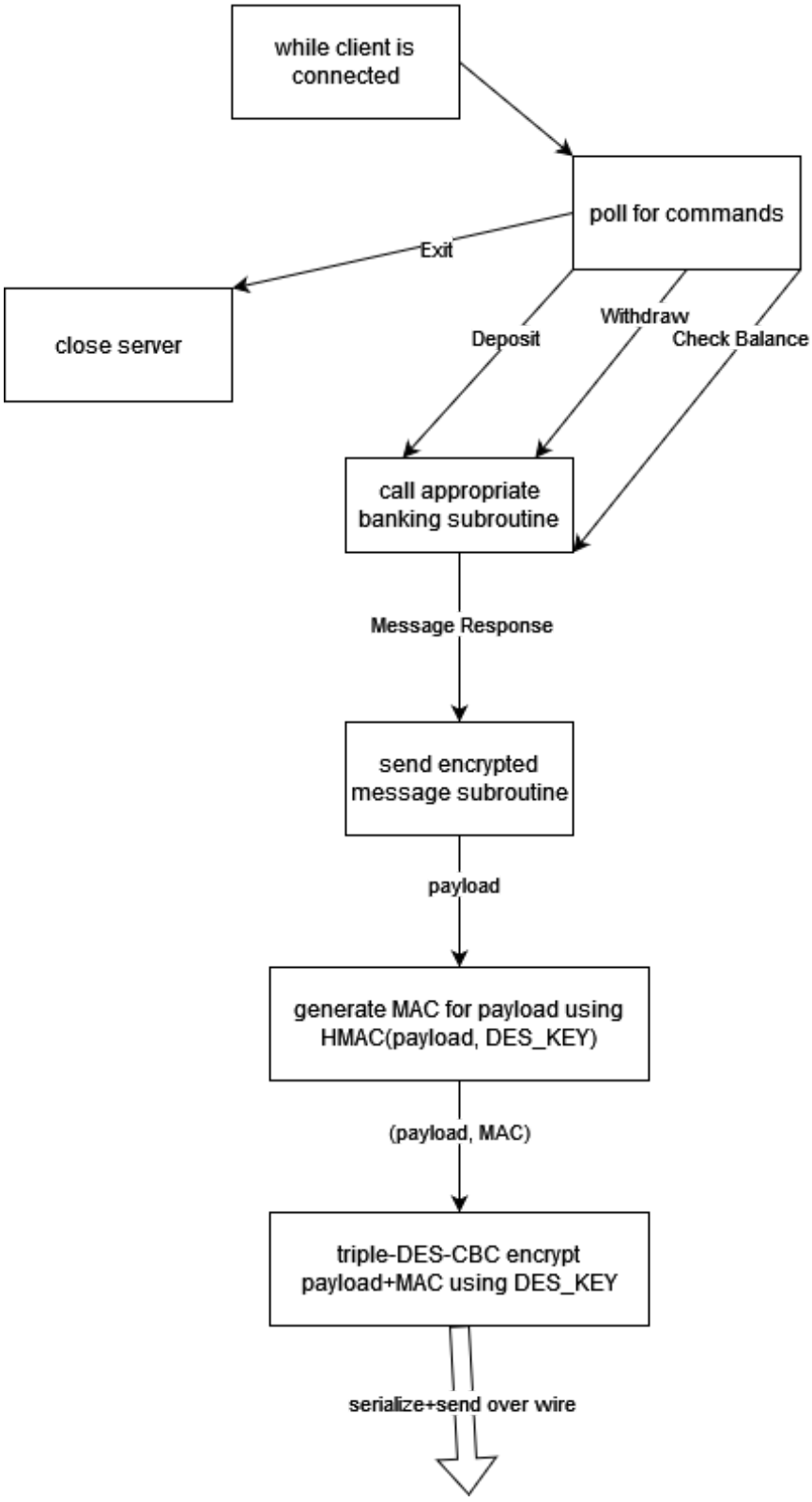

Client
Server
Generate
p, q, e, d, N for RSA
Send [e, N]
Server Sends RSA[SECRET_PORT] to client
generate random
DES key and encrypt
with RSA public key
[e, N]
Send [RSA[DES_KEY]]
Prompt username/password
decrypt DES_KEY
using RSA
private key d
Use triple DES in
CBC mode to encrypt
username and
password using
DES_KEY
Send [TRIPLE_DES_CBC[(username, password)]]
Compute sha1 hash
of password and
compare to database
record for the given
username
Enter Service Mode
Enter ATM Mode

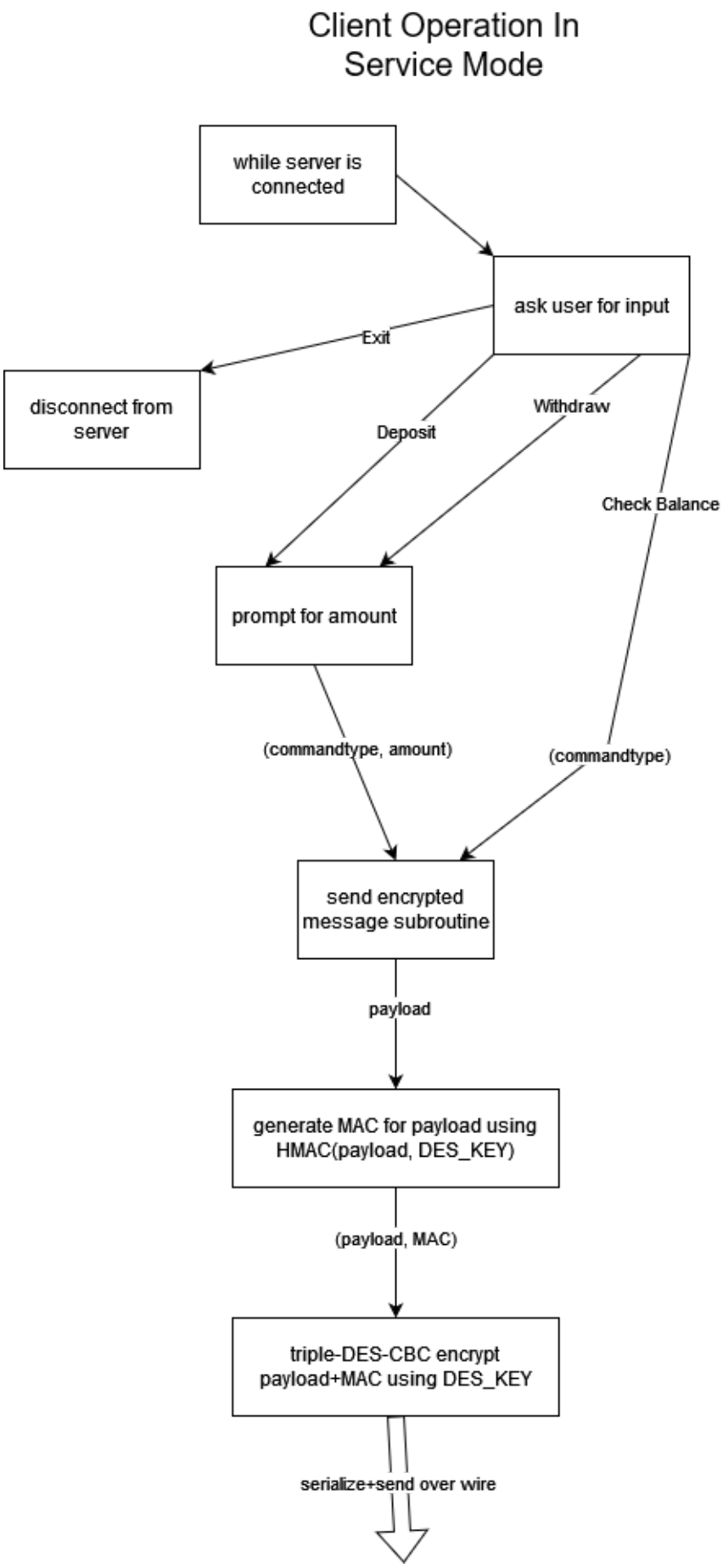
Storing the hashes of the password increases the security of the system. Even if the accounts files were to be exposed an adversary would not be able to login with a user's credentials. If the passwords were to be stored, however, an adversary gaining access to the accounts file would be a much more serious breach.

ATM/Service Mode:

The banking service supports three operations. A user may withdraw funds, deposit funds, check their balance, or exit operation. The user must first choose which operation they wish to perform. If they choose to withdraw or deposit they are then prompted for an amount. For withdrawals the bank checks with the database to verify that the user has sufficient funds in their account. After a user has completed an action they may request another or exit. We use multithreading in our implementation to enable simultaneous and continuous communication between the client and server. The program will continue looping until the user exists or the connection is terminated.

Server Operation in ATM Mode




Client Operation In Service Mode
while server is connected
ask user for input
Exit
disconnect from server
Deposit
Withdraw
Check Balance
prompt for amount
(commandtype, amount)
(commandtype)
send encrypted message subroutine
payload
generate MAC for payload using HMAC(payload, DES_KEY)
(payload, MAC)
triple-DES-CBC encrypt payload+MAC using DES_KEY
serialize+send over wire

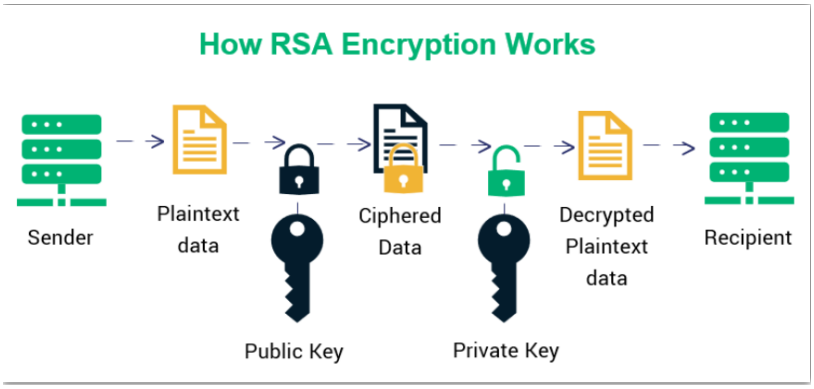
Protocol Implementations:

RSA:

For the public key cryptography we used RSA to distribute the secret keys for HMAC and Triple-DES. The set up for RSA is you have 2 very large prime numbers p and q where $N = p \cdot q$. You select an encryption key, e such that $\gcd(e, \Phi(N)) = 1$ and $1 < e < \Phi(N)$, where $\Phi(N) = (p-1)(q-1)$. Then you can solve for the decryption key d using the extended euclidean algorithm to find $d = e^{-1} \bmod \Phi(N)$ thus, $d \cdot e = 1 \bmod \Phi(N)$. You can encrypt messages where the cipher text is calculated as follows: $C = P^e \bmod N$. Decryption follows similarly where the original plain text can be calculated as $D = C^d \bmod N$.

RSA is public key cryptography because e and N are public knowledge and p, q, and d are kept private. Therefore anyone can encrypt a message but only someone with the private key d can decrypt one.

Diagram of how RSA Works below:



RSA works based on the fact that $P^{e \cdot d} \bmod N = P$ due to the fact that e and d are inverses mod $\Phi(N)$. Then from Euler's Theorem we have $P^{\Phi(N)} = 1 \bmod N$ and in RSA we have $P^{e \cdot d} =$

$P^{1+k \cdot \Phi(N)} = P \cdot P^{k \cdot \Phi(N)} = P \cdot 1^k = P$. Therefore when both keys are applied to some plain text you get the same plain text back.

RSA security relies on the fact that factoring large integers is hard. If you have a 256-bit N then it can be factored in a couple of minutes. A 512-bit N takes several weeks on modern consumer hardware. Factoring 1024-bit N is not possible in any reasonable amount of time given our current computing abilities. If you go up to a 2048-bit N then you should be fairly secure from any brute force attackers.

Our implementation of RSA is in the RSA.py file and the function RSA.RSA_params(size) will return (p,q,e,d) where size is the length of p and q that are generated. We use 512 bits for p and q thus N is about 1024 bits. Note: when encrypting and decrypting use the function pow(Ptxt, e, N) and pow(Ctxt, d, N). The function pow has optimizations when given the modulus that makes calculation of the exponents much quicker.

Example of RSA:

Select $p = 17$, $q = 11$, then $N = p \cdot q = 187$

$\Phi(N) = (p-1)(q-1) = 160$, select $e = 7$ sense $\gcd(7, 160) = 1$

Find $d = e^{-1} \bmod \Phi(N) = 23$ since

Public: {e,N}, Private {d,p,q}

Encryption:

Let $m = 88$, thus $C = m^e \bmod N = 88^7 \bmod 187 = 11$

Decryption:

To get back m we solve $m = C^d \bmod N = 11^{23} \bmod 187 = 88$

Paillier

Paillier is another public key crypto system that we implemented. Similar to RSA. In Paillier you generate p and q the same as in RSA and then $N = p \cdot q$ but then you define a function $L(x) = (x\text{-}1 \bmod n^2) / n$ and $\lambda = \text{lcm}(p-1, q-1)$. Randomly select g between 1 and N^2 then encrypt the plain text where for plain text m, $c = g^m \cdot r^n \bmod n^2$.

To decrypt c, calculate $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$, then you can solve for the plain text $m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$. In this system the public keys are n and g. The private key is λ which is calculated from p and q. Again this relies on the fact that factoring large integers is hard. Thus to make Paillier secure we increase the size of p and q. Paillier cryptography is also homomorphic where addition and multiplication are preserved during encryption and decryption.

Homomorphic encryption is beneficial because it allows computation on encrypted data. The two basic types of homomorphic encryption are additive and multiplicative homomorphism. Additive homomorphism is where $E(m_1) \oplus E(m_2) = E(m_1 + m_2)$ where $\oplus$ represents additive homomorphism. Multiplicative homomorphism follows similarly where $E(m_1) \otimes E(m_2) = E(m_1 * m_2)$ where $\otimes$ is multiplicative homomorphism. RSA from the last section is only partially homomorphic where it has multiplicative homomorphism but not additive homomorphism. Paillier is fully homomorphic.

The Paillier cryptosystem is in the file paillier.py and to get the parameters call paillier.paillier(size) and this returns (g,p,q,lambd,u). Again the parameter size is the length of p and q. Note: the limiting factor for the paillier encryption is the way I implemented lcm. It takes

a while when using large values for p and q thus we decided on RSA so we could use larger primes in an efficient amount of time.

Example of Paillier:

$P = 13$, $q = 17$, $N = p \cdot q = 221$, $N^2 = 48841$

$\lambda = \text{lcm}(p-1, q-1) = 48$, pick $g = 4886$

$\mu = (L(g^{\lambda} \bmod n^2))^{-1} \bmod n = 159$

Encryption:

Plaintext: $m = 123$, select $r = 666$

Then ciphertext: $c = g^m \cdot r^n \bmod n^2 = 25889 \bmod 221^2$

Decryption:

To get back the original plain text: $m = L(c^{\lambda} \bmod n^2) \cdot \mu \bmod n = 123 \bmod 221$

<u>HMAC (Message Authentication):</u>

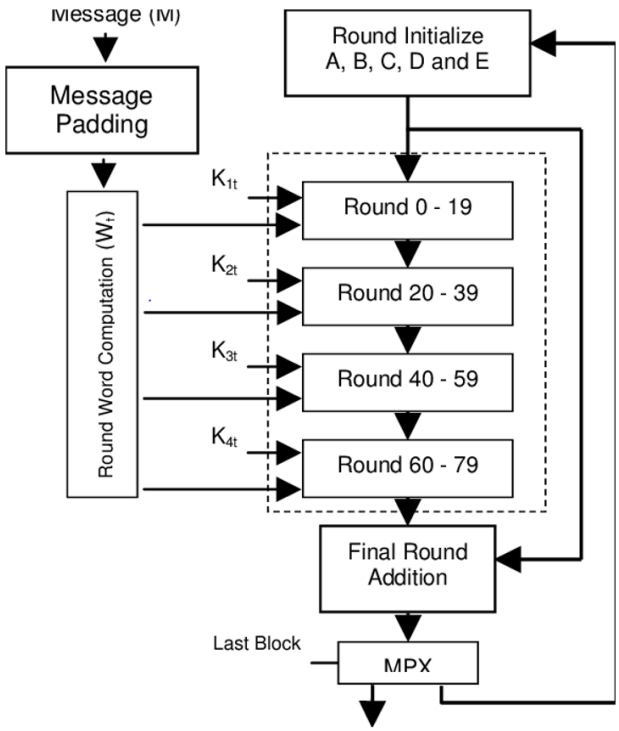
For message Authentication we used HMAC with the SHA1 hash function to provide message authentication to prevent messages being tampered with.

SHA1 is a hash function which stands for Secure Hash Algorithm 1. This is a hash function that returns a 40 digit hexadecimal number regardless of the length of the input to the hash. Our SHA1 Hash function is in the file sha1.py and to call the function write sha1.sha1(message) where message is a string of characters.

Example of SHA1: sha1('hello world.') → ‘0190e761bba7bf93fac099718ddb33fd9b3bea1f’

We used the SHA1 hash function inside the HMAC to provide message authentication. We calculated HMAC as with a private key k as follows: $HMAC_k = SHA1[(K^+ \text{ XOR opad}) \| SHA1[(K^+ \text{ XOR ipad}) \| M)]]$. Here opad and ipad are 64 bit randomly generated padding constants. We essentially XOR the key with the inner pad then add the message and send the whole thing though the SHA1 hash algorithm. We then take the key and XOR it with the outer pad and then add the output of the previous step. We put this entire result through the SHA1 hash function to get the final HMAC result.

Diagram of the SHA1 hash function shown below:



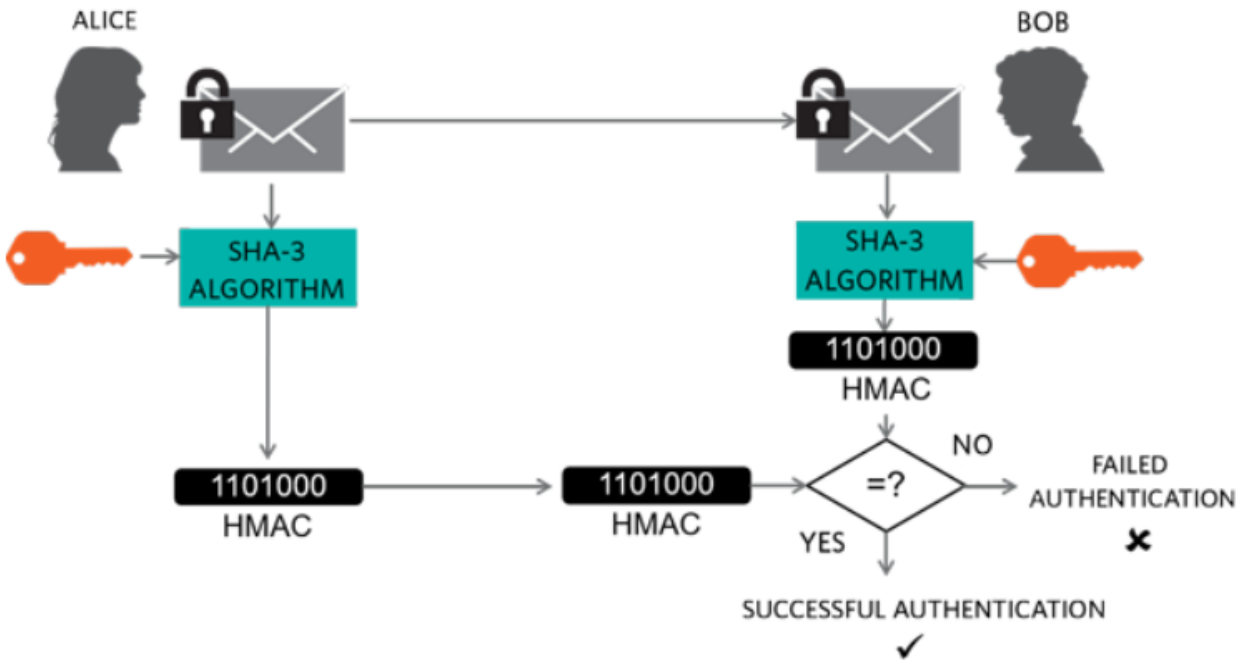
The idea of using this HMAC function is if an attacker was to change some part of the message sent, (flip a bit or replace the message with something of his own), the receiver would know. When the message is received it is first decoded then the sender separates the message from the MAC and runs the message through the same MAC that the sender used. Thus, the only

way the MACs match is if the message was not tampered with. A MAC is secure if it is hard to get a hash collision where it is difficult to get 2 inputs to map to the same hash value. HMAC with SHA1 is a secure MAC and even with the birthday attack it is very hard to get 2 messages that map to the same MAC.

Our HMAC function is stored in MAC.py and can be called using MAC.hmac_fn(message,key) where the message is a string and the key is a 64 bit integer.

Example of HMAC: hmac_fn('hello world', 20538416520752178210) → '9c569fdfafd9fafa07470d5cf8c7ec2cf40e60ee'

Diagram of how HMACs are used bellow: (Note: we used SHA1 as opposed to SHA3 in the diagram)



DES:

DES is a symmetric-key block cipher that encrypts 64 bit blocks with a key of length 64 bits. Our implementation of DES is to-specification. It uses 16 rounds and eight S-boxes to provide so-called “confusion and diffusion.” In each round, the “Feistel” function is called, which expands the 32-bit input to 48 bits, mixes in the 48 key bits by XORing them with the input, and generates an output for the round by passing the combination of the two through the eight S-boxes and combining this output with the other 32 bits of the round input. Our implementation of DES lives in DES.py. It takes both its inputs as bitarrays (data structure from a third-party python library) because it is an internal function that is never called by our client/server code.

Triple-DES:

Our banking protocol uses triple DES to secure all transactions after the user is authenticated to the server. Triple DES is an improved version of DES that applies the original cipher three times to each block. Triple DES requires three times the key bits as DES (hence our usage of a 192-bit key). The three steps of triple DES are as follows (each encryption/decryption is using single-DES as described above). First it encrypts the plaintext using the first 64 bits of the key, then it decrypts this result using the “middle” 64 bits of the key, and finally it encrypts this result using the final 64 bits of the key. In order to decrypt a ciphertext created using triple-DES, the inverse of these operations are performed, namely decryption using the last 64 bits of the key, encryption using the middle 64 bits, and decryption using the first 64 bits. Because our implementation uses three independently generated random 64-bit keys, it is not backwards compatible with DES. This however is the strongest possible keying option for triple DES (compared to only one or two independent keys forming the complete 192-bit key). Triple DES

is known to be more secure than single DES because it has a larger effective key length, although its effective key length is not 192 bits (only 168) due to its vulnerability to known-plaintext attacks [1].

TDES-CBC:

CBC is a technique we used to enable our implementation of triple DES to encrypt block sizes of arbitrary length. It is superior to independently encrypting each block because it prevents each block from being independently brute forced. An arbitrary length input can be encrypted using triple DES in CBC mode by the following algorithm: first the message is segmented into 64 bit blocks. If the last block is less than 64 bits, it is padded with zeros. For each block, we XOR the result of applying triple DES with a 64 bit “initialization vector” (IV). Every new block to be encrypted has a unique initialization vector which is the result of the previous encryption. Since there is no “result of the previous encryption” for the first block, our implementation uses a first initialization vector composed of all zeros. Our triple DES CBC mode encryption function lives in tripleDES.py. It is a wrapper function that takes an arbitrary python object which is encoded to bits using the pickle module. This is then our input to the inner tripleDESCBCEncrypt function that performs encryption and returns a string of 1s and 0s we can send over the wire.

Server Execution Example

```
oject$ python3 tcp.py localhost 8000 8008
Opening Bank on localhost at port 8000.
Connected by ('127.0.0.1', 64670)
Recieved Public RSA Keys: 69715157663507846872757491134050778536908290442322
43491104407491010092497476232817902507618537542131931233281663618893567818
77674954398714759027557510967 3, 79892136135955871435087688308663955954020
85991496381272354950048377935493733840000024127786346920171126356413209211
79136921055747817838911855490761760491627543522738158067169832246280418701
67710772303863555353199921770668846916296736416737085174462452246214659651
51512062313058452354166544540023325510456693 07
Sending Secret Port 8008 to client.
Opening Secret Bank on localhost at port 8008.
Connected by ('127.0.0.1', 64671)
Sending encrypted DES key
beta has logged in.
received check command from client
received withdraw command from client
WITHDREW
received withdraw command from client
received deposit command from client
DEPOSITED
```

Client Execution Example

```
oject$ python3 client.py localhost 8000
Welcome To HHS Bank.
 Please send Public RSA Keys >>>
Sending RSA Public Keys [e,N].
Welcome to the secret port.
DES key is
0000010101111001110111001111011100011001110010011101100101101001000100011
1100001101100001111101010100000000010111111010011001101101111011100011010
1011101010111100110111000110000010110000110 1
Please enter username and password >>>
Enter your username: beta
beta
Enter your password: jasongibbons
jasongibbons
Sending username and password
Hello beta, you are logged in!
Type W for withdraw, D for deposit, C for check balance
C
Current balance: $899.99
Type W for withdraw, D for deposit, C for check balance
W
Enter an amount to withdraw >>> $100
Successful withdrawal
Current balance: $799.99
WType W for withdraw, D for deposit, C for check balance
W
Enter an amount to withdraw >>> $1000
Failed withdrawal, insufficient funds!
Current balance: $799.99
Type W for withdraw, D for deposit, C for check balance
D
Enter an amount to deposit >>> $10
Successful withdrawal
Current balance: $809.99
Type W for withdraw, D for deposit, C for check balance
```

Code in GitHub: https://github.com/dan11109/Cryptography-and-Network-Security-Project

References:

[1] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

https://codereview.stackexchange.com/questions/37648/python-implementation-of-sha1

https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/

https://blog.openmined.org/the-paillier-cryptosystem/v