Crypto Final Presentation

Jarmarri Green, Sean Stearns, Theodore Wu

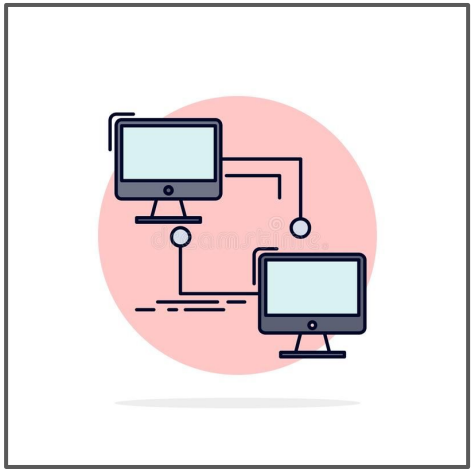
White Hat Component

Basic Idea

- Main goals of project:
  - Be able to communicate with bank and deposit/withdraw/check balance
  - Be able to do so securely
- Many ideas to consider
- Many vulnerabilities that need to be protected against
  - Man in the middle, replay attacks, passive listeners, fraudulent clients, etc.
- To be most protected, divided into 3 steps:
  - Setup
  - Authentication
  - Banking operations

Step 0: Basic Server/Client Setup

- Spun up basic TCP server hosted locally, and client process which connects
  - TCP is connection-oriented: more secure
  - Harder to insert extra data into TCP packets
- Localhost easier for black hat team to test at all times
- Downside: need to ensure processes work 100% of the time
  - Black hat team will run many times



Goal: Establish Secure Channel

- Need secure channel to communicate banking transactions
  - No outside listeners should be able to understand information
  - Only real bank/atm should be able to authenticate and receive symmetric key
- Thus, public key cryptosystem needed
- Process:
  - Handshake begins
  - Client and server authenticate to each other <- messages encrypted with PKC
  - Both parties receive symmetric key
  - Handshake over, transactions commence

Public Key Cryptosystem: RSA, and Setup

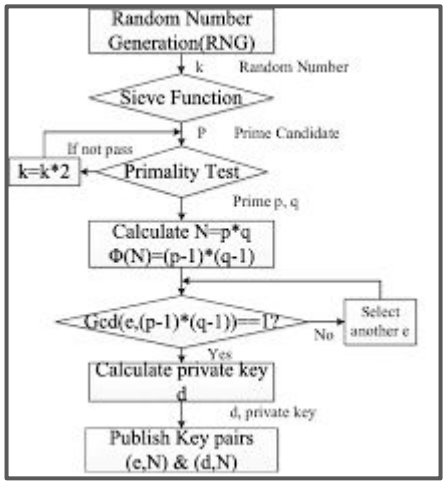
- RSA chosen as PKC, to generate and use public/private keys
- Decided to generate keys before server or client starts up
  - Generating during run could impose vulnerabilities
    - Bogus client
    - Server generating fake 3des key for client to use
  - Cripple security.
- Used config py file to generate keys and write them to appropriate txt files
- Then, client and server read keys during run

Why RSA?

- Alternative was diffie-hellman, which had drawbacks compared to RSA
- RSA assured
    - Confidentiality
    - Authentication
    - Integrity of messages
- Without RSA, would need additional measures in place to ensure authentication
    - Without this, could be vulnerable to man in the middle

RSA Key Generation

- Find 4 large primes, one for each key pair
- Ran loop to generate large (768) bit number
  - Run low-level prime test
  - Run miller-rabin algorithm
    - Determine primality with 75% likelihood, so run 15 times for a total likelihood of 9.17x10^-9 of the numbers not being prime
  - Running low-level primarily test significantly speeds up keygen
- Perform basic RSA arithmetic to get N, phi(N), e, and d
- Write public key {e, N} to file
- Write private key{d, p, q} to separate file
- Repeat for client



RSA encrypt and sign

- During the handshake the client and the server sign their messages with their private key.
- Using the RSA encrypt and decrypt functions, the sender first encrypts the message with its own private key and then the receiver’s public key.
- When the receiver gets the message the receiver decrypts the message twice, first with its private key and then with the public key of the sender
- This ensures that receiver can be certain who the sender is, as only they should be able to sign the message.

Why is this secure?

1) Want to prove to other party message is coming from intended sender
    a) Only intended sender can encrypt message with their private key
2) Possible to get public key from private key, but *hard* (discussed in class) to get private key from public key

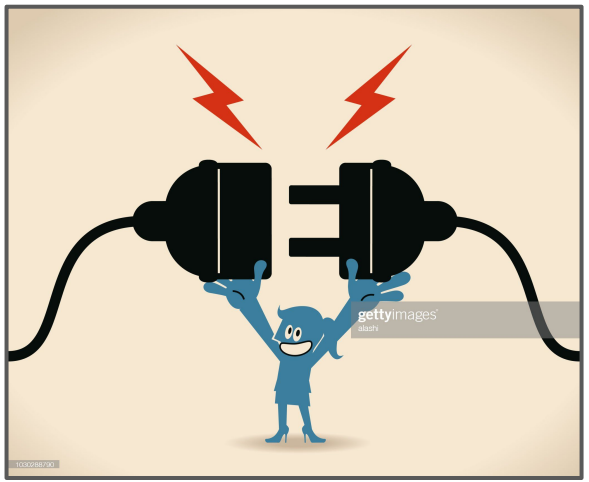
- Recipient can be reasonably sure the sender is who they say they are

1) Also want to prove recipient is who sender thinks they are
    a) Only intended receiver can decrypt with their private key

- RSA was crucial in providing authentication to our asymmetric messages
- Could induce vulnerabilities was it missing

Setup Complete: Time to Share Key

- Client and server can now authenticate securely
  - Not worrying about anyone posing as atm or bank
- Process:
  - Bank server starts up, reads in keys, opens port, listens for ATM client
  - Client starts up, reads in keys, connects to open port
  - Handshake begins



Handshake

1) Security capabilities are shared
    a) Only one set of securities here, but could expand into a larger system
2) Client sends nonce value (random large integer), server returns value
    a) Confirms sender isn't just mimicking messages from previous runs
    b) Confirms receiver can manipulate values (authentication) and send them back
3) Server generates large (192-bit) 3des key and sends to client
4) Client echoes key back
5) Client and server echo a goodbye message

Remember all messages are double RSA-encrypted here, so we get extra authentication

Generating 3DES Key For Banking

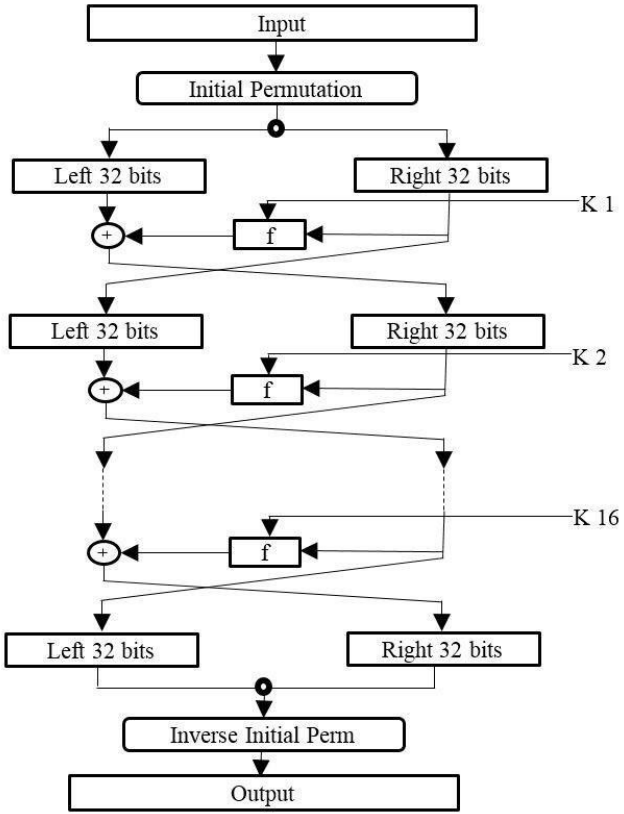
- Once handshake is almost complete, server generates 192 bit symmetric key
- Server and client are authenticated at this point, so no need to worry about faulty key
  - Generate key using secrets library
  - More secure than random.randbits
- Need for large key explained later in presentation

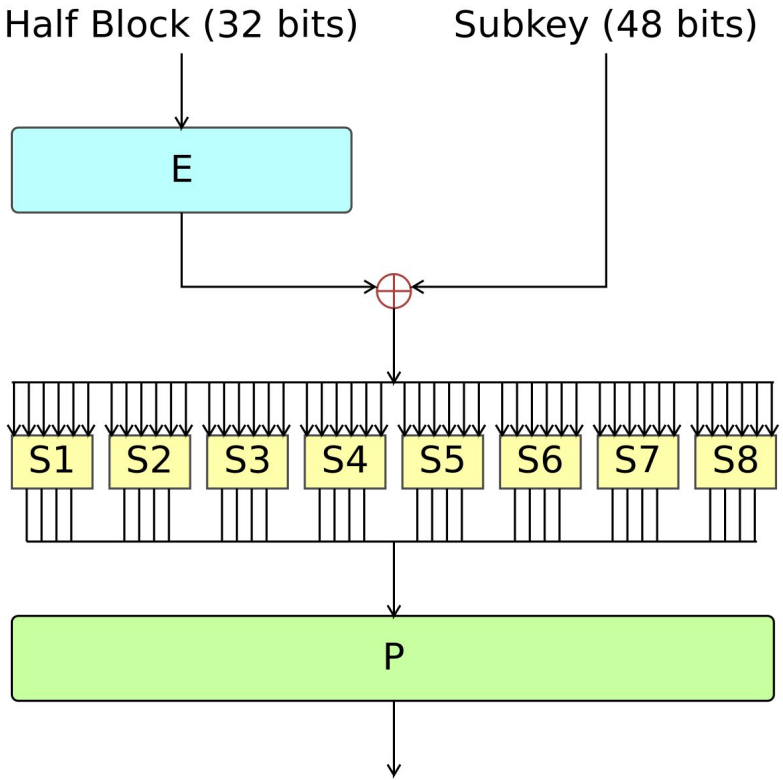
Banking Transactions Begin

Symmetric Encryption Scheme

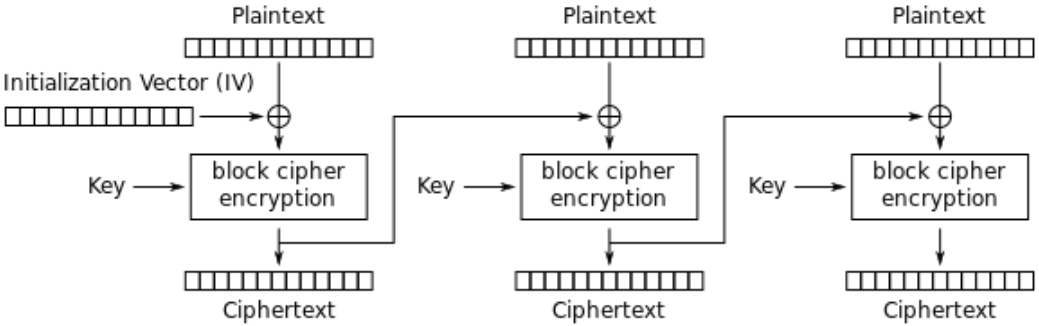
- Reasons for choosing Triple DES
  - Infeasible for Black Hat team to be able to crack 3DES in the time frame they are given
  - More experience with DES instead of a different Symmetric Cryptosystem like AES
  - The key-length for Triple-DES is also sufficiently long enough to deter any brute-force attacks.

DES encryption

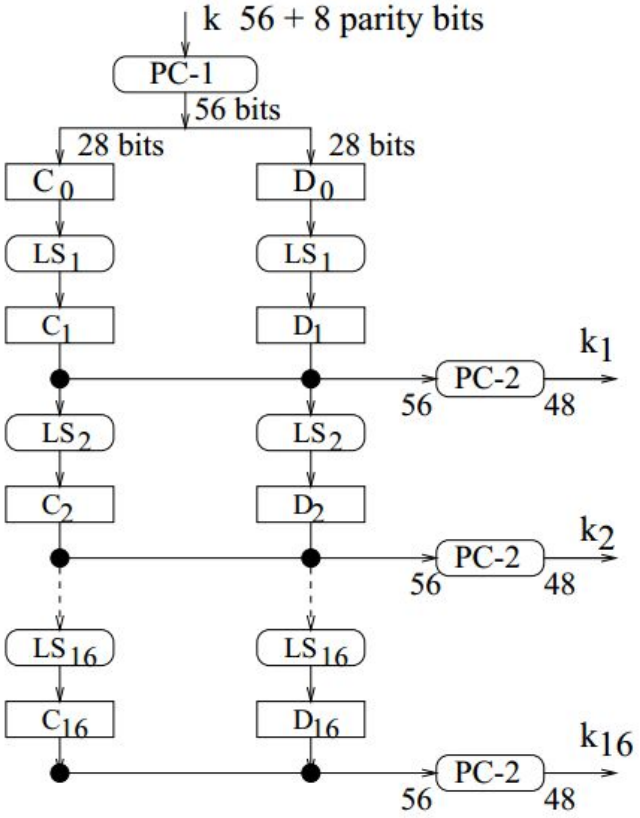




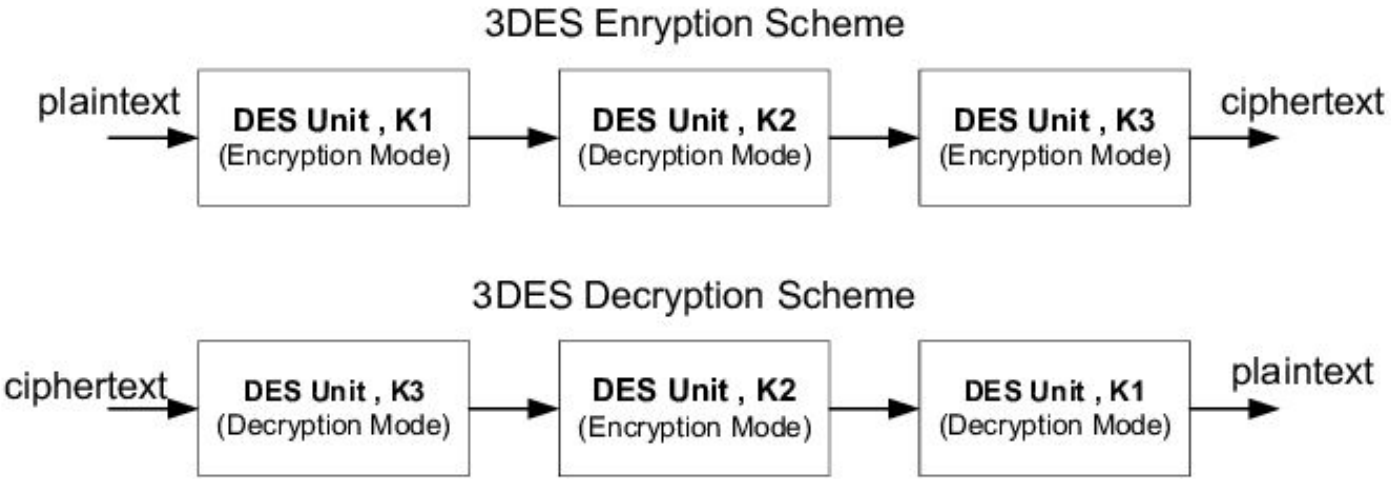
CBC Mode and Subkey generation



Cipher Block Chaining (CBC) mode encryption



Triple DES



Total key length = 64 + 64 + 64 = 192 bits

Message Sending Process

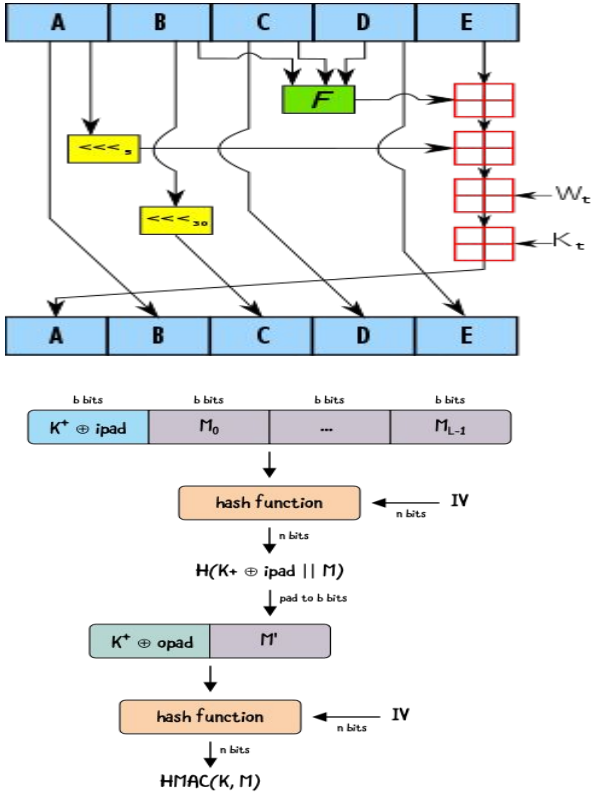
- The message is first constructed by the client as a command + “ “ + amount.
- There are three commands:
  1. “deposit”
  2. “withdraw”
  3. “check”.
- The “deposit” and the “withdraw” command also have an amount tacked on the end separated by a space. (“check” does not)

Message Sending Process

- The timestamp is then tacked on to the end of the message separated by a “.”
- Both the client and the server keep an internal timestamp that is incremented each time it receives a message.
- The message currently looks like:
  - Message = command + “ “ + amount + “.” + timestamp + “.”
- The hash of the message is then computed and tacked on to the end of the message.
  - Final_message = message + Hash(message)

Message Hashing Process (Overview)

- How the client and server to create/verify signatures through hashing:
  - Implemented the hashing function SHA-1
  - Coupled with HMAC (i.e. HMAC-SHA-1) to generate MACs for incoming and outgoing messages



Message Hashing Process (Details)

- Creation of any of these signatures begins with the choice of constants for the ipad and opad values that all HMACs need to utilize in their generation.
- The importance of these two values lies in the fact that their specific differences determine the computational independence of two subkeys that must be used in the generation of any HMAC.
- As such, these two values must be chosen in a way that their related Hamming distance is maximized.

Message Hashing Process (Details; continued)

- That is to say, their Hamming distance, which is a measure of the number of bits that differ between the two n-bit binary strings of opad and ipad, must be maximized.
- There is a “default” set of two values, repeated bytewise as needed for the opad and ipad.
- These values, 0x36 and 0x5c, were chosen as such for any HMAC’s values of opad and ipad because of their “optimal” Hamming distance when repeated.

Message Hashing Process (Details; concl.)

- Usage:
  - Evaluate the bitwise XORs of the respective values with some private key to determine subkeys.
  - hash the ipad's resulting subkey after the message of interest has been appended onto its end.
  - Append that result upon the end of the opad’s previously evaluated subkey.
  - A final hash of that newly created value will serve as the HMAC of the message.

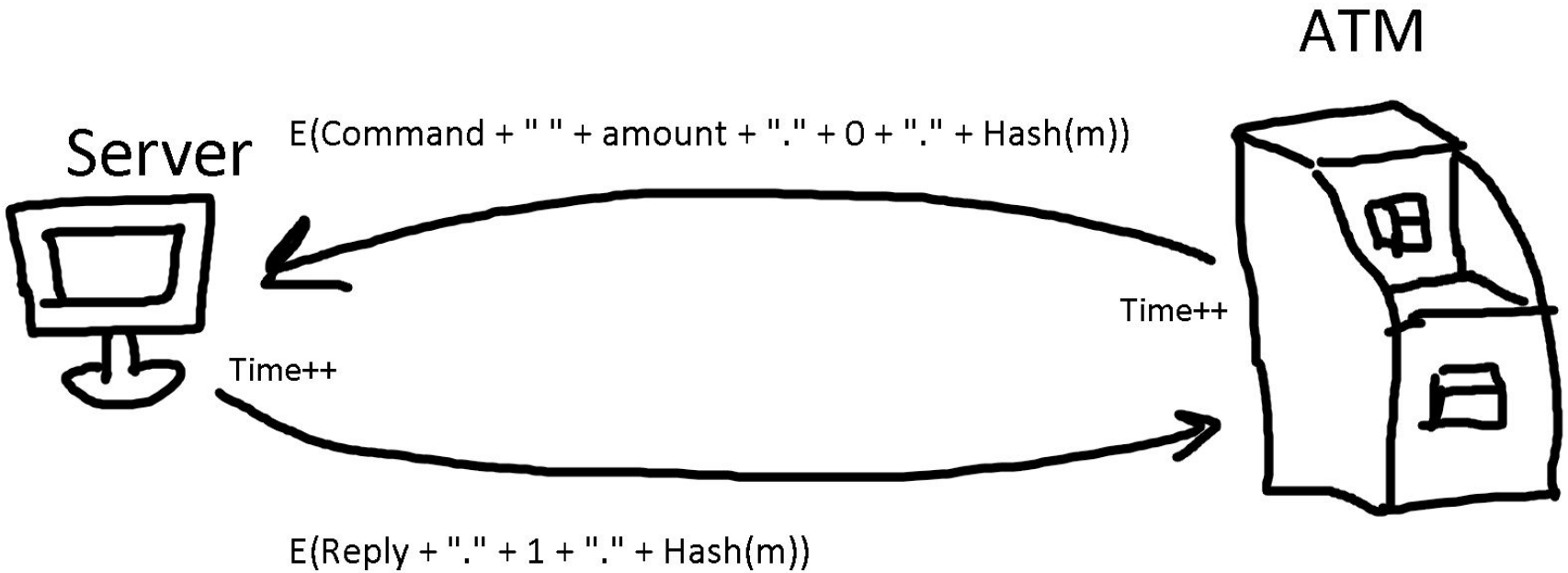
Message Hashing Process (Verification)

- The resulting HMAC, or signature, of any sender's message can then be verified by the hash of that message and the decryption of its signature with the public key of the receiver.
- Note: Technically, hash functions are considered "secure" because there isn't a powerful enough way to find two identical hashes of different strings.
- That works in a hasher's favor however, because nor would an attacker.
- Lastly, of course after an HMAC is appended onto its related message, it can't just be sent as it is…..

Final message packaging

- The final message is then encrypted with 3DES with the shared symmetric key and sent over to the receiver.
- When the receiver gets the message they first decrypt the message and check the hash of the message.
- The receiver ensures that:
  - Hash(first half of message) == second half of message
- Then the timestamp is extracted from the message and compared to the internal timestamp.
- If the hash or timestamp is different the message is flagged as suspicious and further communication is stopped.

Illustration



Black Hat Component

Some Notes:

- Code we were given has a bug in RSA key generation
- Mentioned in readme: On starting the application, the client and the server will initiate a handshake to exchange the session key. On certain runs you may encounter an error with an improperly decrypted key due to the use of the BitArray library. If this scenario is encountered simply start the server and client once more.
- Causes frequent crashes (possible during the demo), especially when generating multiple pairs of RSA keys

Frequent Crashes (expanded)

- The reasons for these (runtime) errors vary, and they can happen on either the client or server side of the system.
- The errors can be classified under four different categories:
    - An error in the event of a mistyped bank command (client)
    - An error during client-server startup in which a key is improperly generated. (RSA)
    - An error after a user has validly entered their command type (client)
    - An error in which working with decimal values causes the code to crash. (client/server)

Frequent Crashes (cont.)

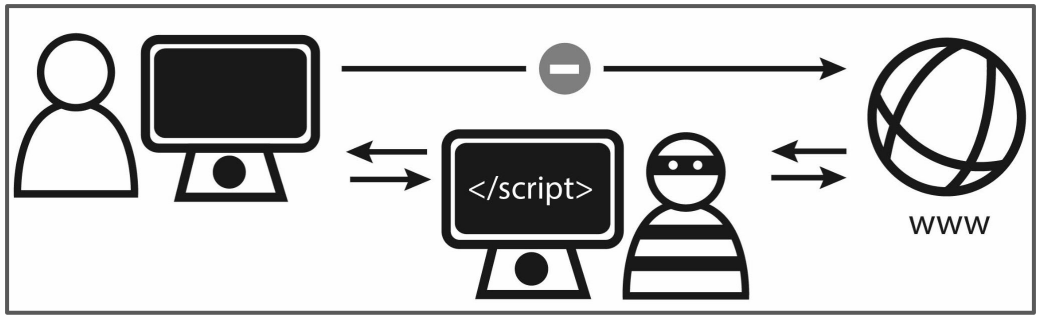
- These errors of course do not always occur.
- However they happen relatively often, and to briefly quantify what “often” means:
    - a temporary modification of the code was made so that when ran, it automatically tried some command assuming it could get to that point:
        - If successful it appends to a text file an indication of such an event, otherwise it appends an indication of failure.
        - In 100 consecutive runs of this modified code, it only “made it” 76 times.
    - At first glance, 76:24 may seem reasonable.

Frequent Crashes (concl.)

- But, MITM, for example, needs four “good” runs. The probability of us getting those four good runs would be:
  - $19^4/25^4$, or somewhere around 33%
- Other than testing successful command reception averages, doing only successful “did-it-even-run” averages was considered.
- However, while potentially informative, it was decided that it was not worth the time remaining to test/try other more pertinent things.
- So the idea was ultimately scrapped.
- Moving on, despite this problem, we were still able to try different attacks….

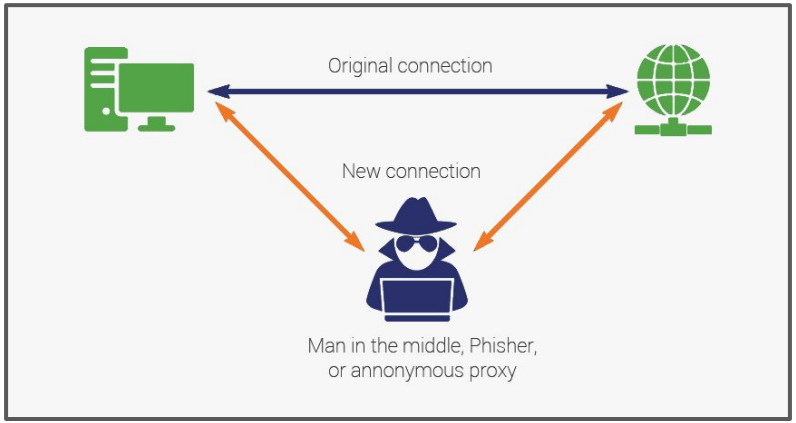
Man in the Middle Attacks

- General idea: adversary inserts themself between client and server
- Passive adversary: listen in on messages
  - Steal information
  - Learn about cryptosystems or even key data
- More active adversary: edit traffic
  - Change data being sent
  - Use client credentials to steal data/money/etc
- Could even pose as other party entirely (most effective attack)



How Can MITM Happen?

- Usually a result of a flaw in authentication
- If others can pose as one or both parties, major systems fall apart
- In enemy code:
  - public/private keys for RSA were generated during the run, instead of in a setup beforehand
- This means an adversary can pose as one or both parties, tell the interested party “this is my public key” and begin a handshake

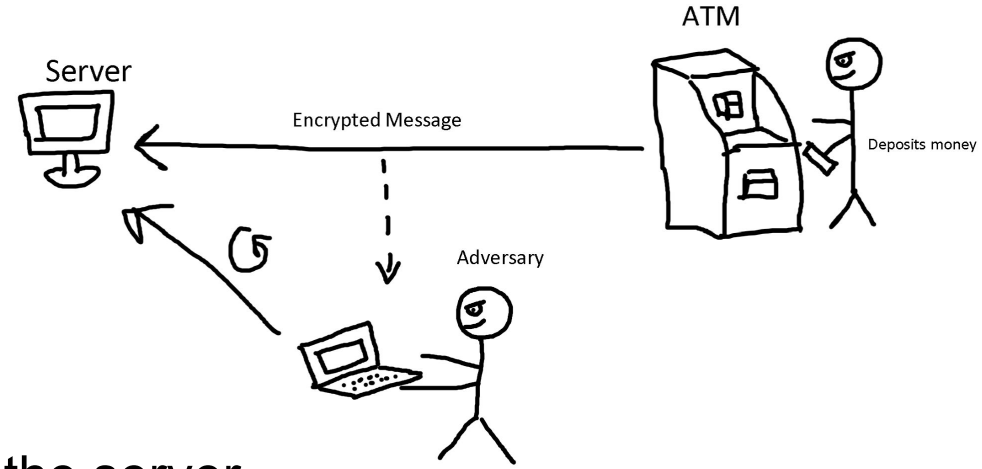


MITM Code Demo

- Will now demo MITM attack on other team’s implementation
- Using file which acts as server and client
  - Poses to bank as ATM
  - Poses to ATM as bank
- Authenticates with both parties, then passes messages back and forth
  - Can listen in on data
  - Or be more active by editing data
  - Keeps track of money stolen
- Client and ATM have no indication that traffic is being listened to or changed
- Ports switched (crude, but could be the result of phishing or something in the real world)

Replay Attacks

- As the adversary we were able to gain infinite money in our account through a replay attack on the server.



- Since the messages between the server and the ATM do not have any method of preventing a replay attack, the same encrypted message sent twice will give the account twice as much money.

Replay attack in action

```
9041963532029114894707185069380949890635279
7068294125253533153320342067287346684429237
6270076785018437132380282878487235359747266
2816759896160691
got: Client: deposit 100
Server: Now you have 100 in your account
got: Client: deposit 1000
Server: Now you have 1100 in your account
got: Client: deposit 1000
Server: Now you have 2100 in your account
got: Client: deposit 1000
Server: Now you have 3100 in your account
```

```
8734668442923762700767850184371323802828784872353
Welcome To The Bank
 1. Deposit, 2. Withdraw, 3. Check Balance
  | Deposit format: deposit 100    |
  | Withdraw format: withdraw 100  |
  | Check Balance format: check    |
  | Quit: quit                     |
Client: deposit 100
Server: Now you have 100 in your account
Client: deposit 1000
Server: Now you have 1100 in your account
Client: deposit 1000
```

```
CLIENT: b'd284780d04a7fe6c3a95bdd4d99665027ed242545bbfd16619fe9453beb827
9fb5e4e623eadaaafc83851de17f0f84a63be5b3ea4259dc9befc1e809fd4ca1a1'
Forward?twice
```

3DES Vulnerabilities

- 3DES is no longer considered highly secure anymore.
- NIST has proposed retiring 3DES as early as 2017 and as more vulnerabilities have been found the transition from using it has accelerated.
- Furthermore the ATM uses three different keys for each DES encryption/decryption in 3DES. This has been shown to be vulnerable to an attack that can be run on commercial-grade computers, as long as enough data can be collected.

Random.Randint vulnerabilities

- Other team's client uses random.randint() to generate their 3des key
- Any random.random in python is a pseudo-random number generator, meaning an attacker could reproduce the sequence
- Uses Mersenne twister to generate "random" string of numbers
  - Not implemented here, but definitely a possible vulnerability
- If attacker were able to reproduce the sequence and get the key, they could decrypt passing traffic and learn information

Denials Of Service (Slight)

- A few small cases of DOS
  - Not super important as there is only one client, but could be considered

1) Sometimes server crashes on startup, as we spoke about
2) Many messages to server from client will result in multiple seconds or minutes of wait time due to RSA exponentiation times
    a) Because of sleep calls during arithmetic, makes enemy code not vulnerable to timing attack (cool) but is very slow

- Not very useful DOS for an attacker, but definitely could deny service to other clients if other clients could connect at the same time

Sources

- https://leeneubecker.com/3des-insecurities-pose-risk-to-many-financial-institutions-and-us-military/

Any Questions?