Trust Issues Chat Detailed Documentation

System Basics

Client

The client-side application is a desktop application written in Python 3. The cryptographic work is performed with the help of the cryptogrpahy.io library. The REST API work is performed with the help of the Requests library.

Server

The server for the application is an NGINX server running Node.js (using the Express.js framework). The server is running on a t2.micro AWS EC2 instance. The server also uses a SQLite database that is stored locally on the EC2 instance. The URL for the server is https://donttrustthisgroup.me/. The server performs work with JSON web tokens with the help of the jwt library. The server does work regarding the salting and hashing of passwords with the help of the bcrypt library.

Client System Components

Modules

Authentication: This module is responsible for the handling and processing of user authentication. After the user enters his/her username (an email address) and password, the system makes a request to the server for a JSON web token, and once received maintains a Session object containing the user's credentials and the token in memory.

Message: The message module is responsible for representing messages in our system (either those to be sent or those that have been received). The two main classes are the PlaintextMessage class (contains a message in plaintext) and the EncryptedMessage class (contains an encrypted message and all the associated keys).

Encrypter: This module is responsible for handling the encryption of messages as well as producing HMAC tags. The primary cryptographic operations performed in this module make use of the cryptography.io library. It receives a Plaintext Message object as input and outputs an EncryptedMessage object. To do this, the AESEncrypt class first randomly generates a 256-bit AES key and a 128-bit initialization vector. In CBC (cipher block chain) mode using the generated IV, the message is encrypted using AES with the 256-bit key. Using the AES ciphertext, the HMAC class generates an HMAC tag by using a randomly-generated 256-bit key. The AES and HMAC keys are then concatenated and encrypted in the RSAEncrypt class using the 2048-bit RSA public key of the intended recipient. The EncryptedMessage object returns the AES ciphertext, the HMAC tag, the IV, and the concatenated key ciphertext. The figure below illustrates the work of the Encrypter module in more detail.

JSON_Data: This module is responsible for handling the creation of JSON objects from user-generated data as well as creating business class objects from JSON objects received from the server. The JSONConstructor class packages data generated by the user as prescribed by the JSONConstructorStrategy object that is passed into it. For each unique REST interaction that the client has with the server, a corresponding JSONConstructorStrategy class is defined that receives the relevant data from the user and packages it into a JSON object that the server will be able to handle. The JSONDeconstructor class works in much the same way. For each unique type of response that the client system may receive from the server, there exists a corresponding JSONDeconstructorStrategy class that receives the JSON object input and packages it into the appropriate business class objects.

Server: This module is responsible for handling all interactions with the server. The REST API operation performed by this class are handled by the Requests library. The ServerBoundary class executes the REST API calls as specified by the ServerCommand object that is passed into it. For each unique REST API call that the system needs to make, there exists a corresponding ServerCommand class. The ServerCommand classes all implement a method 'execute()' which performs the actual REST API call and provides the necessary data (that was received from the JSON_Data module). Because binary data is not JSON serializable (JSON only accepts Unicode data), the message ciphertext, key ciphertext, HMAC, and IV are all base64 encoded and then Unicode decoded prior to being transmitted. Analogously, upon receiving messages from the server, the message ciphertext, key ciphertext, HMAC, and IV were all base64 decoded and then Unicode encoded prior to being passed along for decryption.

Decrypter: This module is responsible for the decrypting of messages received from the server and again uses the cryptography.io library. Upon receiving the messages, the key ciphertext is decrypted in the RSADecrypt class using the user's RSA private key. Since the AES key and HMAC key are both 256 bits in length, the decrypted key ciphertext is split in half. The AuthenticateMessage class takes the obtained HMAC key and uses it in conjunction with the received HMAC tag to verify the integrity of the received message. Once verified, the AESDecrypt class uses the AES key to decrypt the message ciphertext, and the resulting plaintext is placed into a PlaintextMessage object. The figure below illustrates the work of the Decryption module in more detail.

UI: This module generates the UI for the app which involves creating the menu options for the user to select guiding the user through sending messages, displaying received messages, and sending friend requests.

Email_Action: This module is responsible for all logic relating to the sending of emails and uses the Python built-in smtplib and email modules. Email is used by the system to communicate RSA public keys between users. Emails are sent via SMTP via an SSL connection.

Friends: This module is responsible for all logic involving friend requests and handling new friends. Friend requests are sent to the client from the SendFriendRequestController class. When the recipient logs on, the GetPendingFriendRequestsController class shows a notification informing them of the friend request, if he/she accepts, the SendEmail class in the Email module prompts the user for their email credentials and then sends an email with their RSA public to the originator of the friend request. When the original user logs back into the system sometime later, he/she is informed by the AddRSADetailsOfFriendsController class that his/her friend request has been accepted and is then prompted to send his/her RSA public key via email. On each login, the AddRSADetailsOfFriendsController class checks for any newly accepted friends and prompts the user to supply paths to RSA public key files as needed. The paths to corresponding the RSA public keys for each friend are stored locally in configuration files for each unique user on a particular machine.

Configuration: This module is responsible for writing to and reading from various configuration files that the system relies upon to store information persistently. Local system file paths to RSA keys are the most crucial data that this module handles. Whenever the user encrypts or decrypts with RSA the paths for the corresponding file are read by this module and stored in a RSAConfig object. Whenever the user gets a new friend, the system adds a key-value pair for the friend email-RSA public key path.

The server supports the following 8 routes:

/signup: This route is responsible for creating new user entries in the User table and providing JSON web tokens to validate new users. When generating JSON web tokens, the server uses the username of the user as the payload and uses the plaintext password as the key to the HMAC password-based-key-derivation function that is used to sign. This route DOES not store received passwords in plaintext. Using the bcrypt library, the server generates a random 128-bit salt, concatenates the salt with the supplied password, and then hashes the concatenated string using the slow b-crypt hash function. The result of this hash along with the salt is what is stored in the password field in the SQLite database.

/login: This route is responsible for verifying credentials sent from users and providing JSON web tokens to those that supply valid credentials. To validate the password supplied, the server grabs the salt from the SQLite database and performs the salting and hashing functionality as described above and compares the results.

/send-message: This route is responsible for creating new message entries in the Message table that can eventually be polled by the recipient when he/she is online. New entries will only be created if the requester has supplied a valid JSON web token.

/check-messages: This route is responsible for polling for unread messages that are intended for the requester and returning them. This action is only performed if the requester supplies a valid JSON web token.

/send-friend-request: This route is responsible for populating the FriendRequests table SQLite database with friend requests. A JSON web token is required from the originator of the request.

/check-friend-requests: This route is responsible for polling the FriendRequests table and retrieving all the pending friend requests that a given user has. Requires a JSON web token from the user making the call.

/add-friend: This route is responsible for populating the Friends table with entries after friend requests have been accepted. Requires a JSON web token from the user who accepts the friend request.

/get-friends: This route is responsible for retrieving all the friends that a given user has. On the client-side, this list is compared to the records that the user has in the friends_rsa.txt configuration file to determine if the user has any new friends as a result of any recently accepted friend requests.

Use Cases

Use Case 1	SignUp
Actors	● Client application
Pre-Conditions	● n/a
Flow on Control	● The client application sends a request to the server along the /signup route ● The server adds the user's info to the database ● The server generates a JSON web token ● The server sends the token back to the user
Post-Condition:	● The user's credentials are stored in the database ● The user has a valid JSON web token
Error-Condition:	● The client application is unable to connect with the server

Use Case 2	Login
Actors	● Client application
Pre-Conditions	● The user has already created an account
Flow on Control	● The client makes a request to the server via the /login with inputted email and password ● The server validates the user’s credentials ● The server generates a JSON web token ● The server sends the token back to the user ● The client generates a new session ● The client opens the UI
Post-Condition:	● The user now has a new, valid JSON web token
Error-Condition:	● Invalid credentials are supplied ● The client application is unable to connect to the server

Use Case 3	Send Message
Actors	user
Pre-Conditions	● The user has a proceeded through signup or login ● The user has chosen the “send message” command
Flow on Control	● The user inputs message text ● The system asks which recipient they want to send the message too (chooses from list of friends for which an RSA public is assigned to) ● The user chooses a recipient ● The system asks the user if he/she wants to send the message ● The user confirms ● The system generates an 256 bit-AES key ● The system generates an initialization vector ● The system pads the data using PKCS #7 padding ● The system encrypts the data in CBC mode ● The system generates a 256-bit HMAC key ● The system computes the tag using SHA-256 ● The system concatenates the AES key and the HMAC key ● The system reads in the RSA public key from the configuration file ● The system encrypts the concatenation of the AES and HMAC keys using RSA encryption

		with OAEP padding ● The system packages the user email, recipient email, the message ciphertext, the key ciphertext, the tag, and the IV in a structure ● The system base64 encodes and then Unicode decodes all the binary data ● The system sends the data along with the JSON web token to the server via a REST API request
Post-Condition:	● A message is sent	
Error-Condition:	● The message is empty ● An invalid jwt is supplied ● The client application cannot connect to the server	

Use Case 4	Check Messages
Actors	● user
Pre-Conditions	● The user has proceeded through signup or login
Flow on Control	● The user selects the "check messages option" ● The client sends an API request to the server including the JSON web token to poll for messages ● The server verifies the token ● The server queries for new messages ● The server adds to a record of all message poll requests ● The server sends back the message data ● The binary data is base64 decoded and Unicode encoded ● The system read in the RSA private key from the configuration file ● The system decrypts the key ciphertext ● The system splits the decrypted key text to produce the AES key and the HMAC key ● The system regenerates the HMAC tag using the message ciphertext ● The system verifies that the computed tag

		matches the received tag ● The system decrypts the message ciphertext using the AES key and the IV ● The system unpads the decrypted message text ● The message text is Unicode decoded ● The message is displayed
Post-Condition:	● The user sees all their unread messages	
Error-Condition:	● An invalid jwt is supplied ● The client cannot connect to the server	

Security Goals

1. **Achieve message confidentiality**

   The system will achieve message confidentiality by encrypting all transmitted messages locally on the client's machine using AES-256. The AES key will be randomly generated using a cryptographically secure pseudo-random number generator. The AES keys will also be encrypted locally on the client's machine using 2048-bit RSA encryption. The message sender and message recipient will exchange keys in an out-of-band fashion via email.

2. **Achieve message integrity**

   The system will achieve message confidentiality by generating a hash-based message authentication code (HMAC) tag from the encrypted message. The HMAC tag will be generated using the SHA-256 cryptographic hash function and a 128-bit key generated from a cryptographically secure pseudo-random number generator. The HMAC key will also be encrypted with 2048-bit RSA encryption.

3. **User Authentication**

   The system will achieve user authentication through the use of passwords and JSON web tokens. All RESTful interactions with the server (with the exception of user signup and login) will require that the user provide a JSON web token in the request header. Users will only be able to obtain JSON web token by signing up and creating a valid account or logging into an existing account. The server DOES NOT store any passwords in plaintext but only those that have been salted (with a 128-bit salt) and hashed (using the slow hash function b-crypt) on the backend.

Assets

- Email address data
- Message data
- Key data
- Password data
- EC2 instance → detailed by examining the above assets
- Communication link between client and server→ detailed by examining the above assets
- Client code → will be in public repository
- Server code → will be in public repository

Stakeholders

- Users

Adversaries

1.
   a. Outsider
   b. Computationally limited
   c. Passive

   Example: teenager in the basement

2.
   a. Outsider
   b. Unlimited resources
   c. Passive

   Example: NSA

3.
   a. Outsider
   b. Computationally limited
   c. Active

   Example: average everyday hacker

4.
   a. Outsider

  b. Unlimited resources
  c. Active

  Example: Russian government

5.
  a. Insider
  b. Computationally limited
  c. Active

  Example: Server

Design Analysis

The use of SSL to encrypt communications between the client and the server will prevent all passive adversaries from acquiring any meaningful data. The only way an adversary would able to see the raw data being transmitted over the SSL connection (from the client to the server) would be to get ahold of the RSA private key. The only two ways to get ahold of the RSA private key would be to brute force it or somehow obtain the key file from the AWS instance. In theory, the adversary with unlimited resources could brute force RSA private key with a quantum computer in a likely lengthy but reasonable amount of time. For the computationally-limited adversary, however, it is fair to presume that this task would be impossible. If brute forcing is not employed, then the adversary would need to obtain the RSA private key file from the AWS instance. Whether the adversary wants to log into the instance or grab the file via SCP, he or she would need the SSH private key that is stored locally on my machine. Again the adversary with unlimited resources could likely brute force the SSH private key but the computationally-limited adversary would not be able to. Furthermore, limiting the inbound SSH traffic to a small range IPs that I control would make compromising the instance significantly harder for both types of adversaries. This would force them to have to compromise the LAN that controls the IPs allowed by the inbound rules. Therefore, we can reasonably assume that data transmitted over the SSL connection is confidential with respect to outsider adversaries.

Given the security of the SSL connection, in order to see the email addresses of the users, potential adversaries would have to either obtain them from the SQLite database or log the keystrokes of the users as they type into the client program. The seemingly very low probability that keystrokes would be logged and the associated complexity of obfuscating user keystrokes presented a large amount of effort relative to the severity of the vulnerability. Since the SQLite database exists locally in persistent storage on the AWS instance, the only way to access the data stored in it would be to compromise the instance as described above and obtain the database file. Given the need to SSH into the instance, we can reasonably assume that the database file is secure from being obtained by outside adversaries.

The passwords in the SQLite database would be just as difficult to obtain as the email addresses (however they are still vulnerable to keystroke logging on the client machine). However, if the SQLite database was in fact compromised, there is another layer of security in that the passwords are stored as the hash of the plaintext password concatenated with a random salt. Although the adversary would have access to the salt value, this salt prevents the adversary from using any type of pre-computed rainbow table which would force them to either carry out a dictionary attack or compute a brand new rainbow-table based upon the obtained salt. Given these circumstances, the computationally-limited adversary will likely not have enough resources to compute an extensive rainbow table, and the use of the slow hash function b-crypt makes the verification time too long for the adversary to carry out a successful dictionary attack. Therefore, it is reasonable to assume that the passwords in the SQLite database are confidential with respect to the computationally limited adversary. The adversary with unlimited resources, however, will most definitely be able to compute a new rainbow table and would likely be able to parallelize the dictionary attack enough to counteract the slow b-crypt hash function. Consequently, the system is assuredly vulnerable to the adversary with unlimited resources. However, it is reasonable to believe that the probability of this adversary acting against the system is very low.

The messages in the SQLite database are similar to the passwords in that they have their own additional layer of security via the AES encryption (again they are still vulnerable to keystroke logging on the client machine before they reach the SQL database). If the adversary manages to compromise the SQLite database, which we noted earlier as being difficult, the messages are still encrypted with AES-256 bit encryption. Given the assumption in the field of security that AES-256 bit encryption is virtually impossible to break by brute force, the only reasonable approach to obtaining the message data in plaintext would be to obtain the AES key. However, the AES keys, which are also in the database, are encrypted using 2048-bit RSA. Given this, the only way to decrypt the ciphertext for the AES key would be to either brute force the RSA private key or somehow acquire it from the local machine of the message recipient. While possible, compromising the local machine of the message recipient would be incredibly difficult given that the adversary would have no idea who the recipient is and no idea as to the security precautions that individual has taken. That leaves the brute forcing of the RSA key as the sole remaining approach which is possible for the limitless adversary (using a quantum computer as described earlier) and likely impossible for the computationally-limited adversary. Given the seemingly low probability of the limitless adversary, it is, therefore, reasonable to assume that the message data along with the AES/HMAC key data in the SQLite database is confidential.

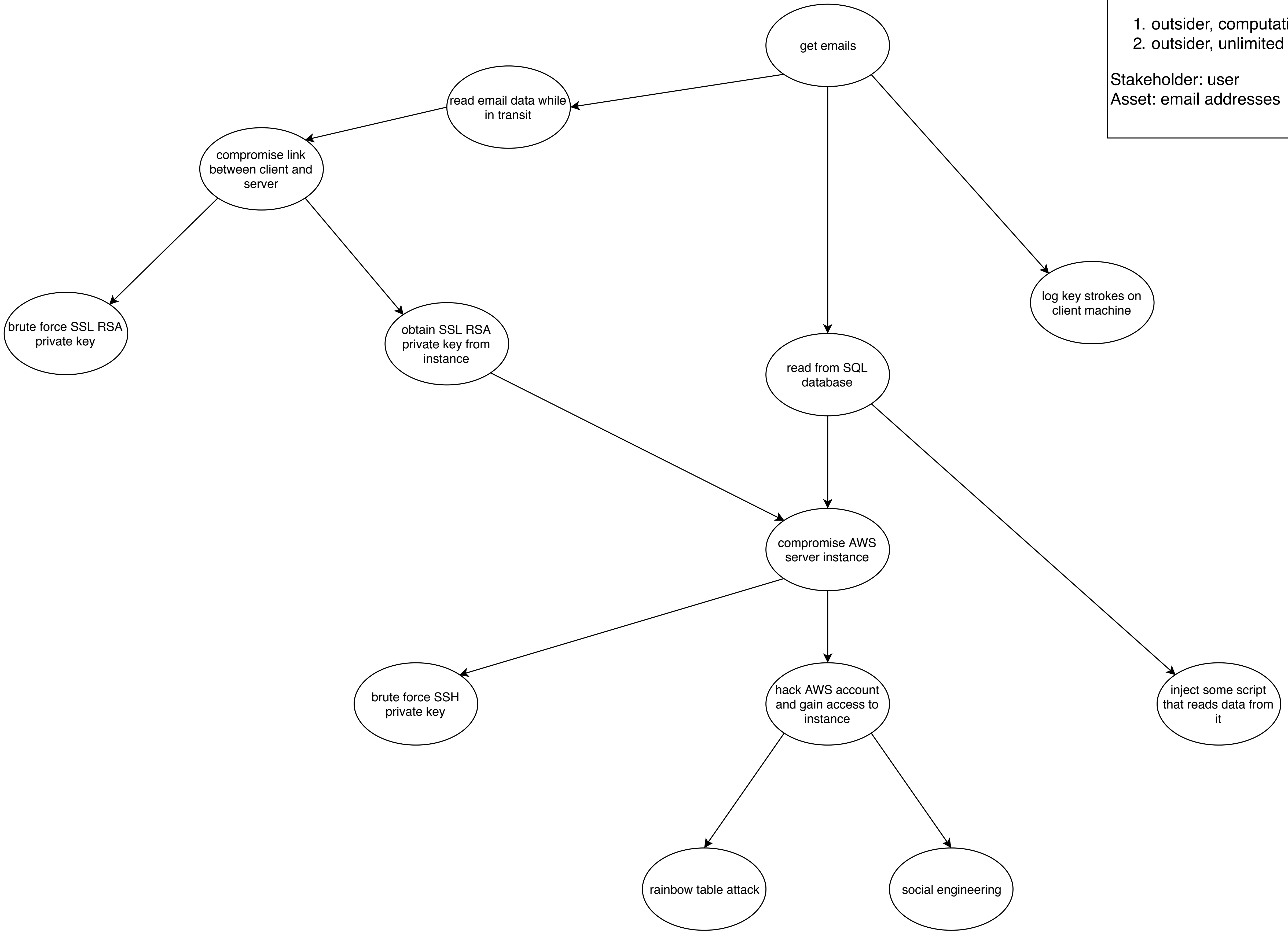
The difficulties of obtaining any of the noted assets as described above demonstrates that the system adequately ensures confidentiality with respect to the outsider adversary.

When the server itself is viewed as an insider adversary the analysis changes slightly. This situation highlights the true downsides of the locally stored SQLite database. With respect to the outsider adversary, the SQLite database was very effective in that its security and ability to be accessed was tied to that of the server which itself is fairly secure. In addition, the existence of the database on persistent storage eliminated an entire communication link (one that would typically exist between the server and the database) and prevented the need for having to handle the security of the database separately. With the server acting as an insider adversary, however, it is impossible to ensure the confidentiality of non-encrypted data. This is because there is no way to stop the server from reading data such as email addresses and passwords as they come in in the body of REST API requests. The confidentiality of the message ciphertext along with the AES/HMAC ciphertext is still assured, however, because the server does not have access to the corresponding RSA private key just like the outsider adversary.

With regards to message integrity, the system generates an HMAC tag from the AES ciphertext using a 256-bit randomly generated key and the SHA-256 cryptographic hash function. Given the assumption in security that HMAC guarantees that an adversary cannot generate a valid message, tag pair for a given key without knowing that key, it is reasonable to assume that message integrity could only be violated if an adversary were to get ahold of the HMAC key. Given that it is 256-bits in length, it would be virtually impossible to compute via a brute force approach. Therefore, an adversary attempting to undermine the message integrity would need to acquire the plaintext HMAC key. Just as with AES key described earlier, we demonstrated that the system could reasonably ensure the confidentiality of the HMAC key, which in turn allows the system to ensure message integrity.

With regards to user authentication, the system uses password-based authentication and then provides JSON web tokens to those users that provide valid credentials. The JSON web tokens are generated using an HMAC password-key-derivation function which uses the user’s password as the key. Earlier we demonstrated that the system is able to ensure the confidentiality of passwords with respect to the outsider adversary. Given the confidentiality of passwords with respect to outsiders, we can, therefore, ensure that outsiders will be unable to supply the necessary data needed to generate valid JSON web tokens (granted the user’s supply decent passwords). Thus, the system ensures proper user authentication with respect to outsiders.

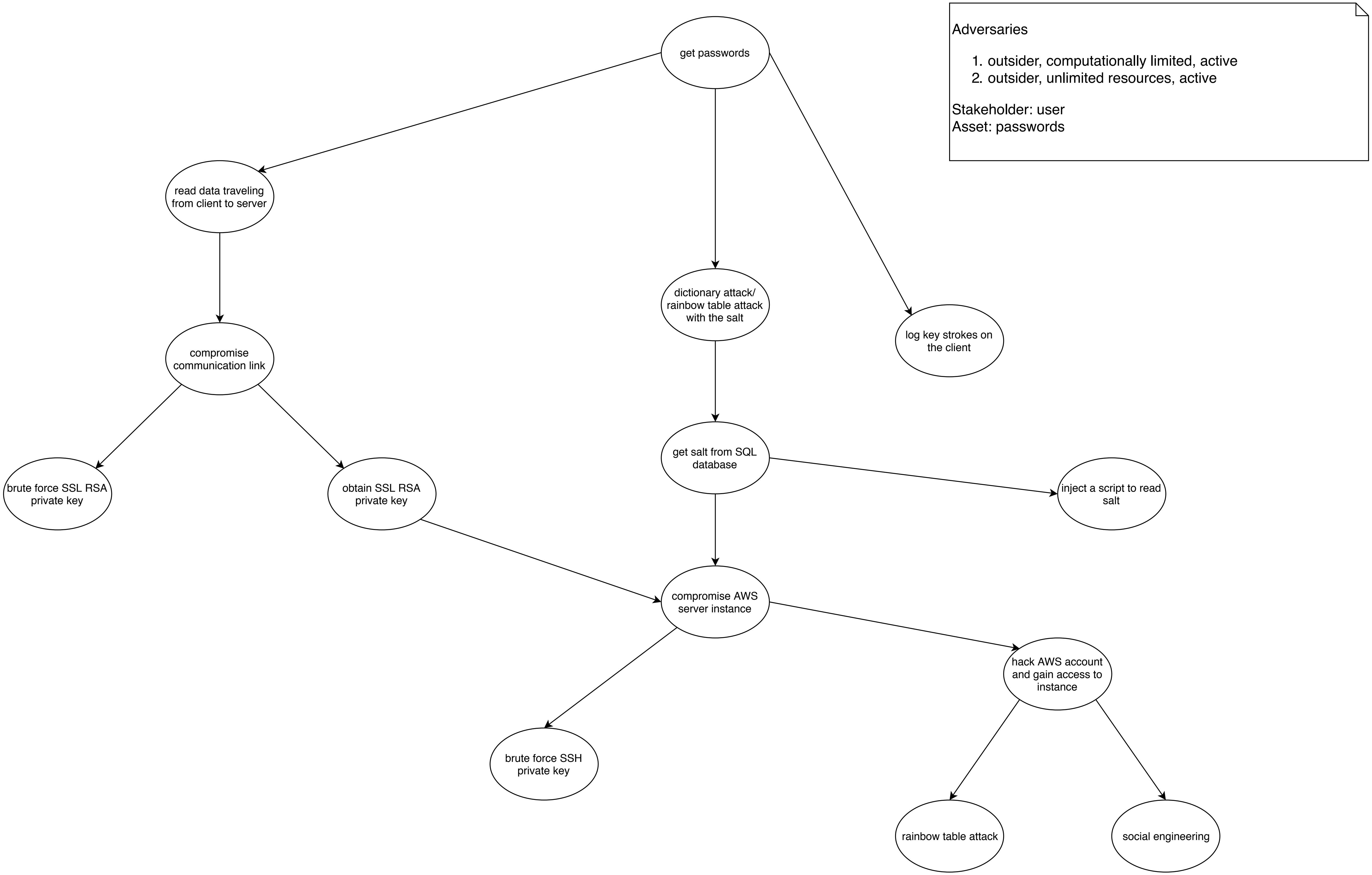
Unfortunately, the system is unable to ensure the confidentiality of passwords with respect to the insider. As a result, there is no way to stop the server from generating valid JSON web tokens and acting as if it were an arbitrary user. In spite of this vulnerability, the server still does not have the corresponding RSA public key necessary to send encrypted messages to other users that they would be able to decrypt. As noted earlier, the server would also not have any RSA private key values needs to decrypt messages sent by other users. Given that the message sender and message recipient communicate their RSA public key in an out-of-band fashion (via email), the only way for the server to obtain the private key would be to compromise the email of a user or spoof this out-of-band communication with an email address that the server controls. Consequently, this aspect of the system presents the biggest vulnerability which is as of now unaccounted for.


get emails
read email data while in transit
compromise link between client and server
brute force SSL RSA private key
obtain SSL RSA private key from instance
read from SQL database
log key strokes on client machine
compromise AWS server instance
brute force SSH private key
hack AWS account and gain access to instance
inject some script that reads data from it
rainbow table attack
social engineering

Adversaries

1. outsider, computationally limited, active
2. outsider, unlimited resources, active

Stakeholder: user
Asset: email addresses

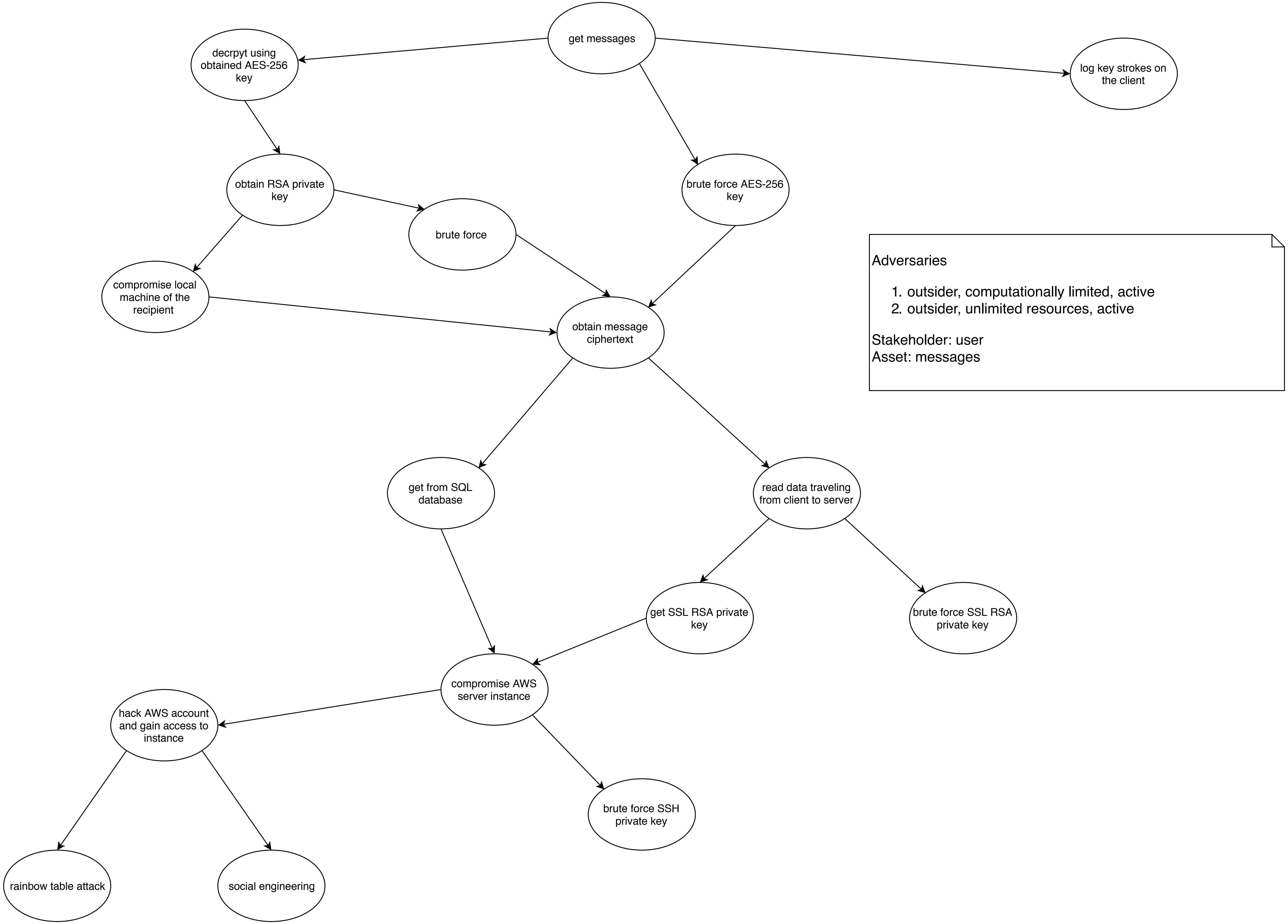


get passwords
read data traveling from client to server
dictionary attack/ rainbow table attack with the salt
log key strokes on the client
compromise communication link
get salt from SQL database
brute force SSL RSA private key
obtain SSL RSA private key
inject a script to read salt
compromise AWS server instance
hack AWS account and gain access to instance
brute force SSH private key
rainbow table attack
social engineering

Adversaries

1. outsider, computationally limited, active
2. outsider, unlimited resources, active

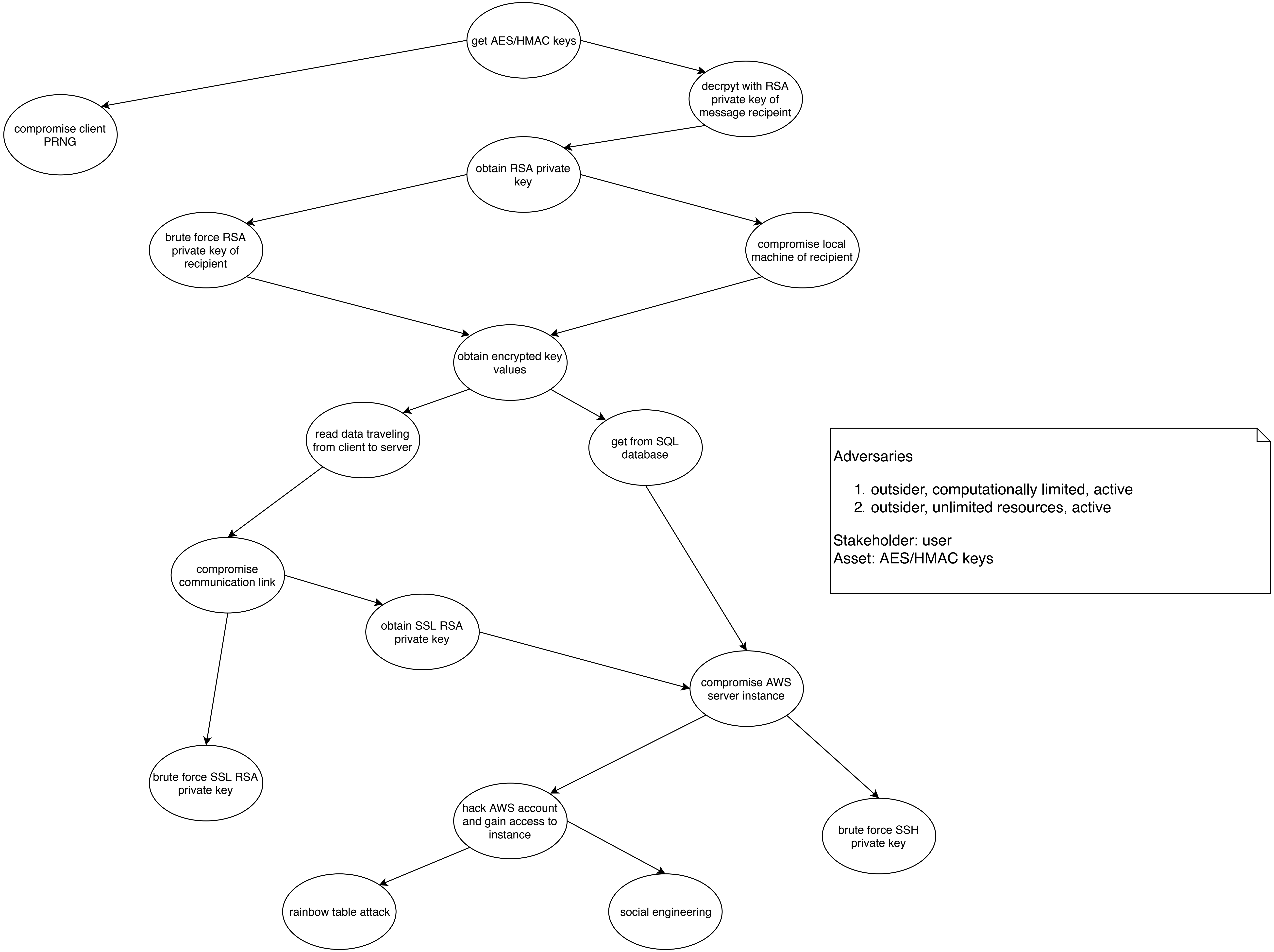
Stakeholder: user
Asset: passwords


get messages
decrpyt using obtained AES-256 key
log key strokes on the client
obtain RSA private key
brute force AES-256 key
brute force
compromise local machine of the recipient
obtain message ciphertext
get from SQL database
read data traveling from client to server
get SSL RSA private key
brute force SSL RSA private key
compromise AWS server instance
hack AWS account and gain access to instance
brute force SSH private key
rainbow table attack
social engineering

Adversaries

1. outsider, computationally limited, active
2. outsider, unlimited resources, active

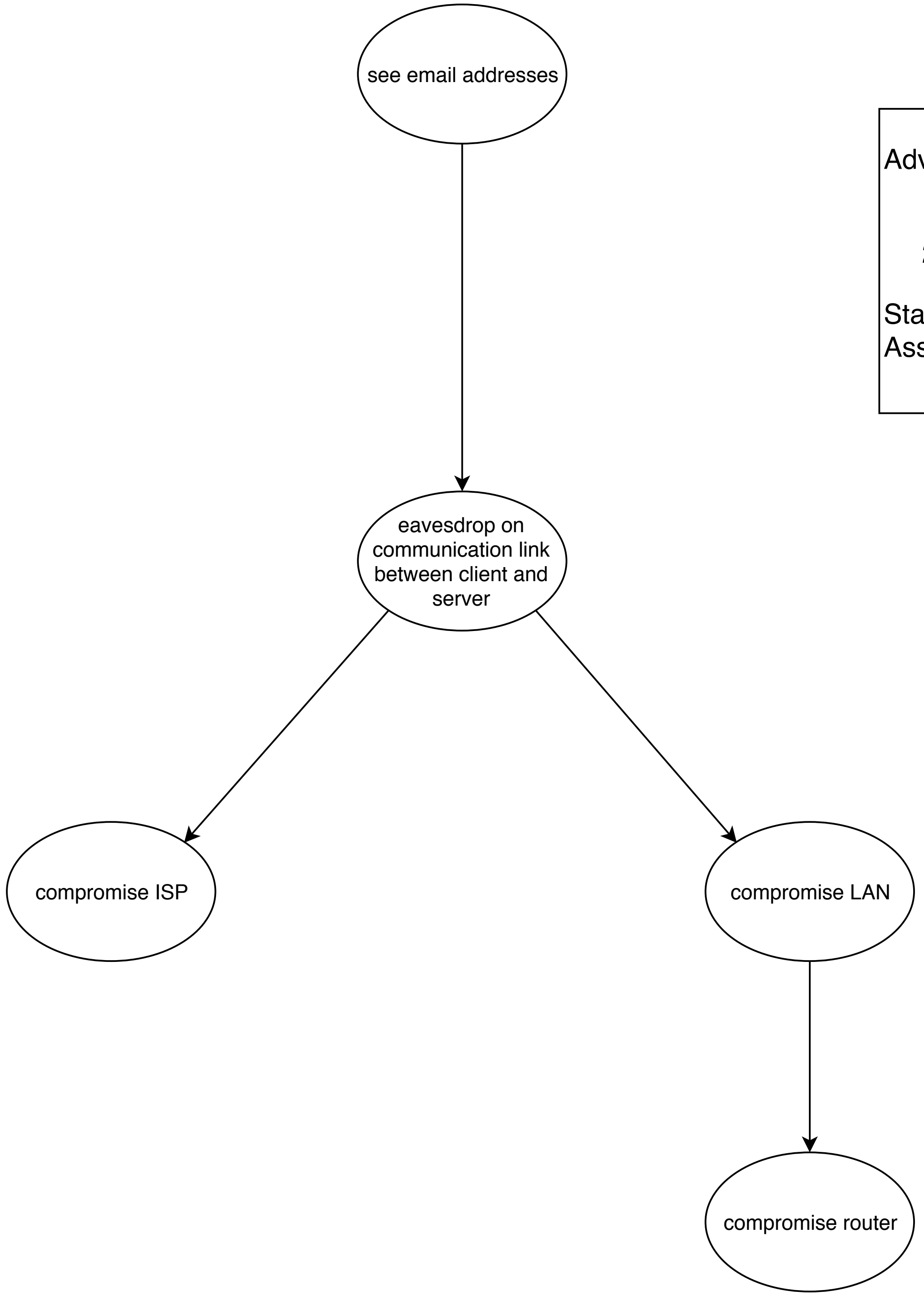
Stakeholder: user
Asset: messages


get AES/HMAC keys
compromise client PRNG
decrpyt with RSA private key of message recipeint
obtain RSA private key
brute force RSA private key of recipient
compromise local machine of recipient
obtain encrypted key values
read data traveling from client to server
get from SQL database
compromise communication link
obtain SSL RSA private key
compromise AWS server instance
brute force SSL RSA private key
hack AWS account and gain access to instance
brute force SSH private key
rainbow table attack
social engineering

Adversaries

1. outsider, computationally limited, active
2. outsider, unlimited resources, active

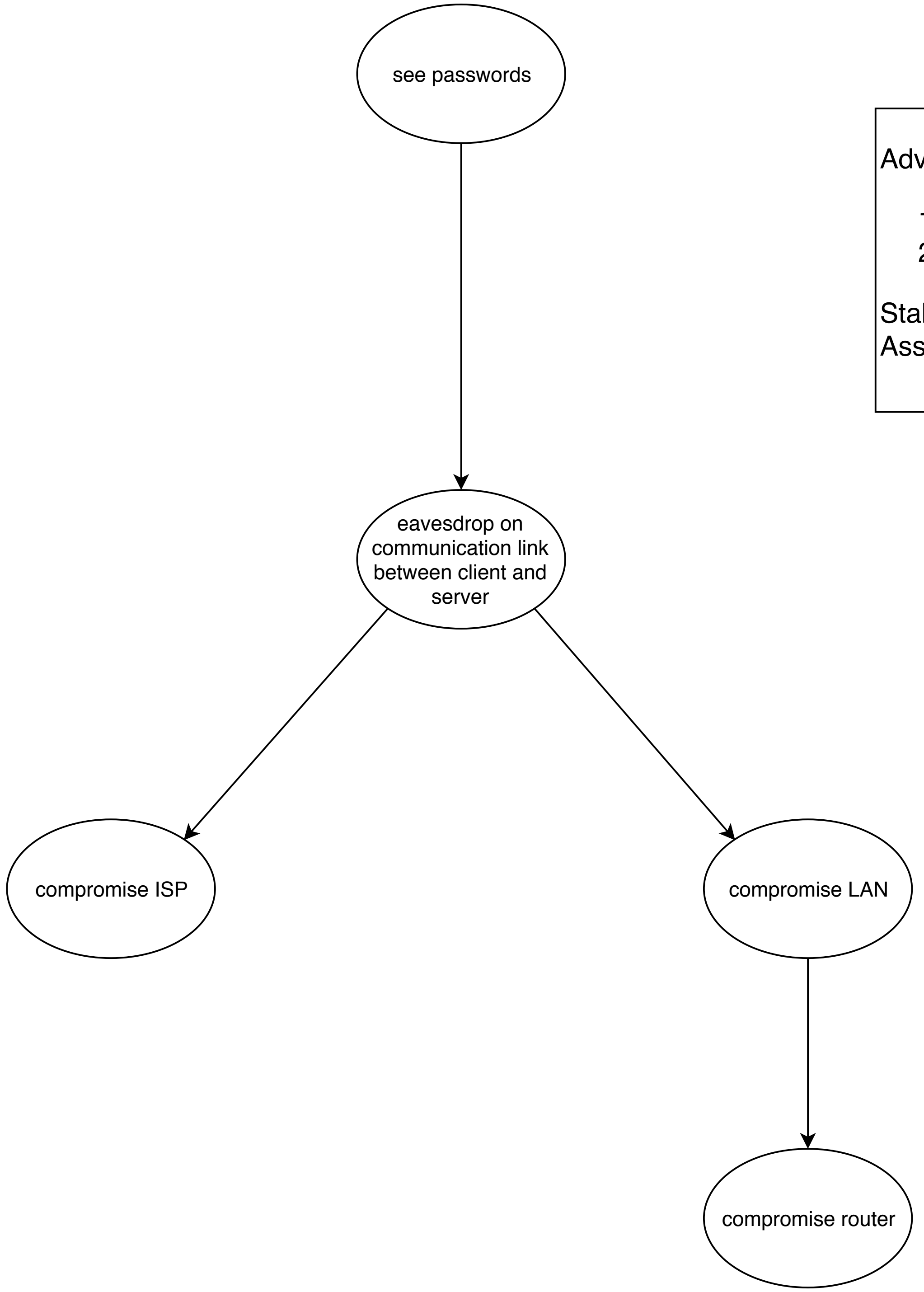
Stakeholder: user
Asset: AES/HMAC keys



Adversaries

1. outsider, computationally limited, passive
2. outsider, unlimited resources, passive

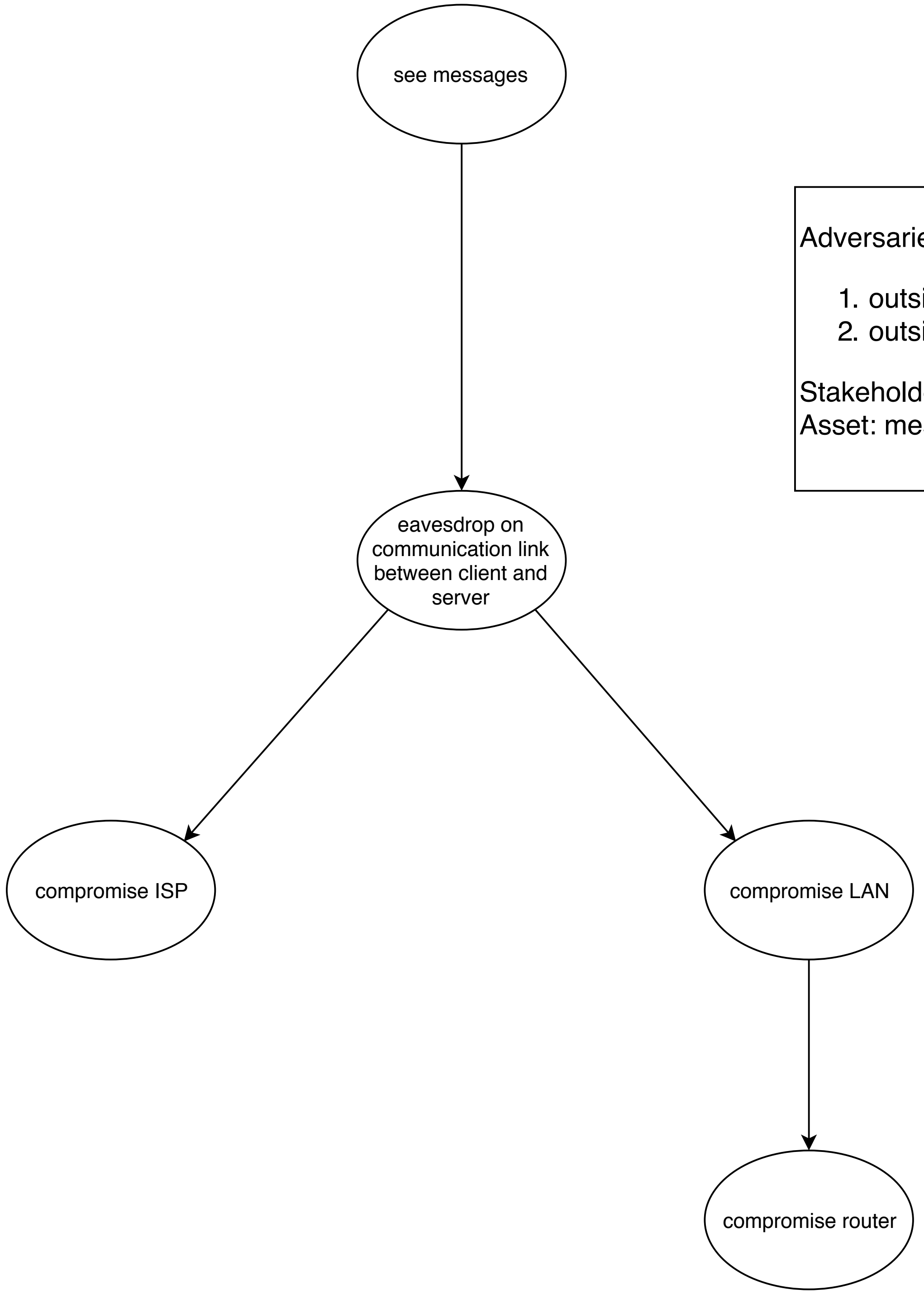
Stakeholder: user
Asset: email addresses



Adversaries

1. outsider, computationally limited, passive
2. outsider, unlimited resources, passive

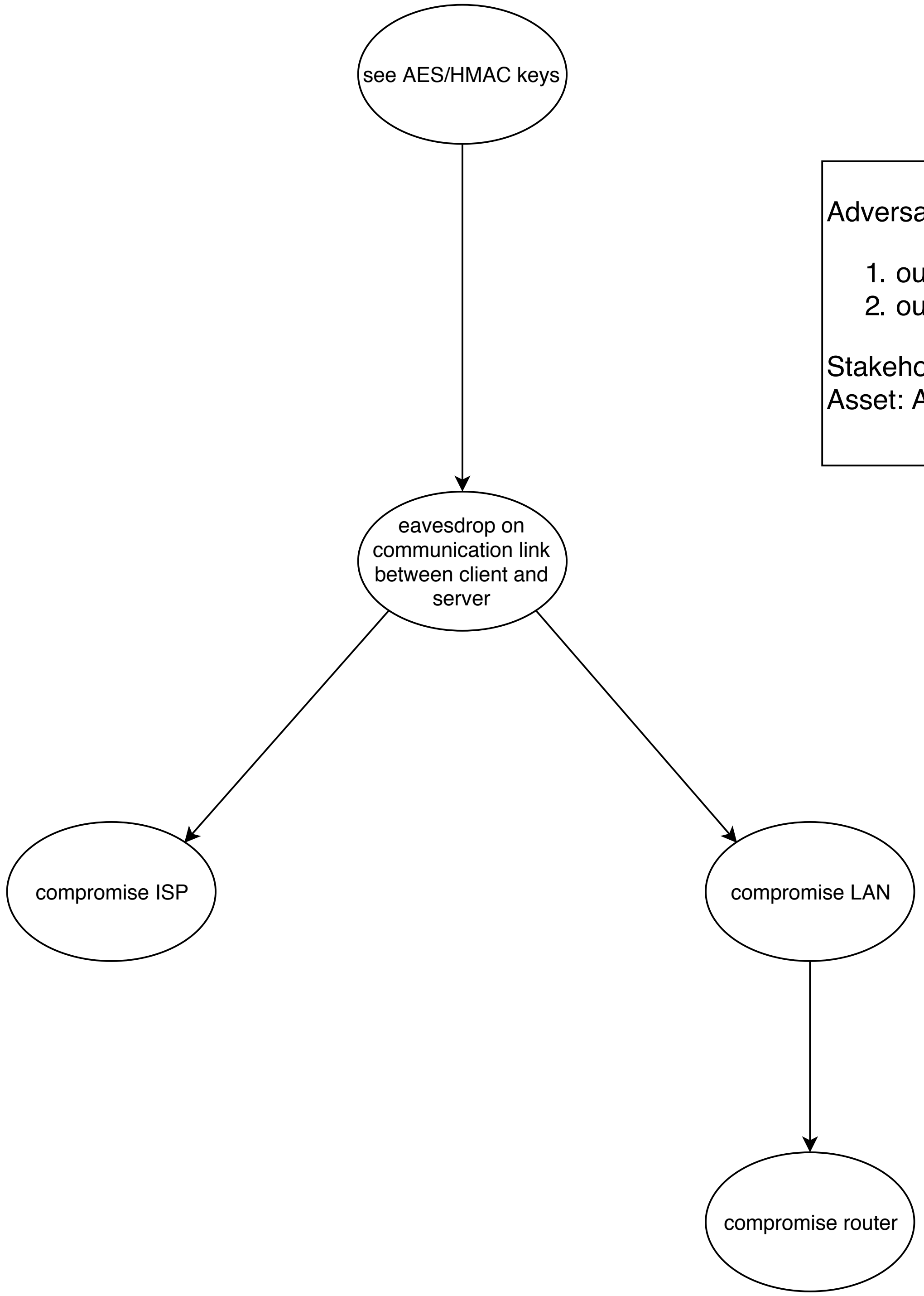
Stakeholder: user
Asset: passwords



Adversaries

1. outsider, computationally limited, passive
2. outsider, unlimited resources, passive

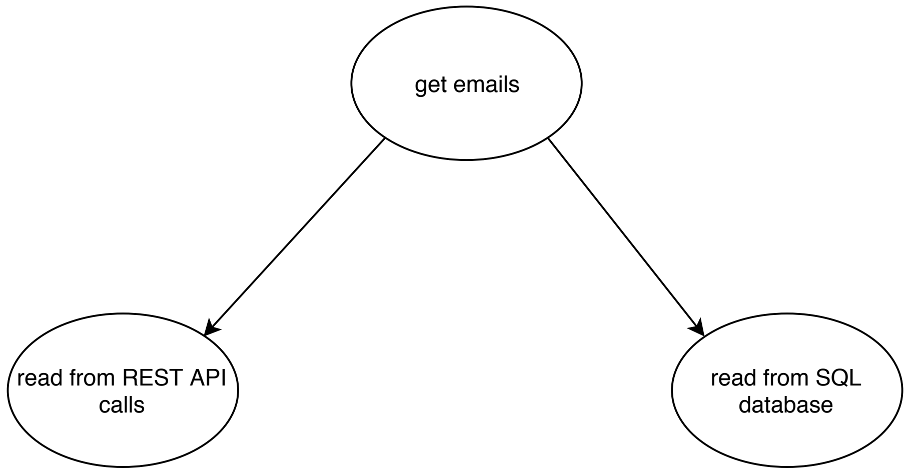
Stakeholder: user
Asset: messages



Adversaries

1. outsider, computationally limited, passive
2. outsider, unlimited resources, passive

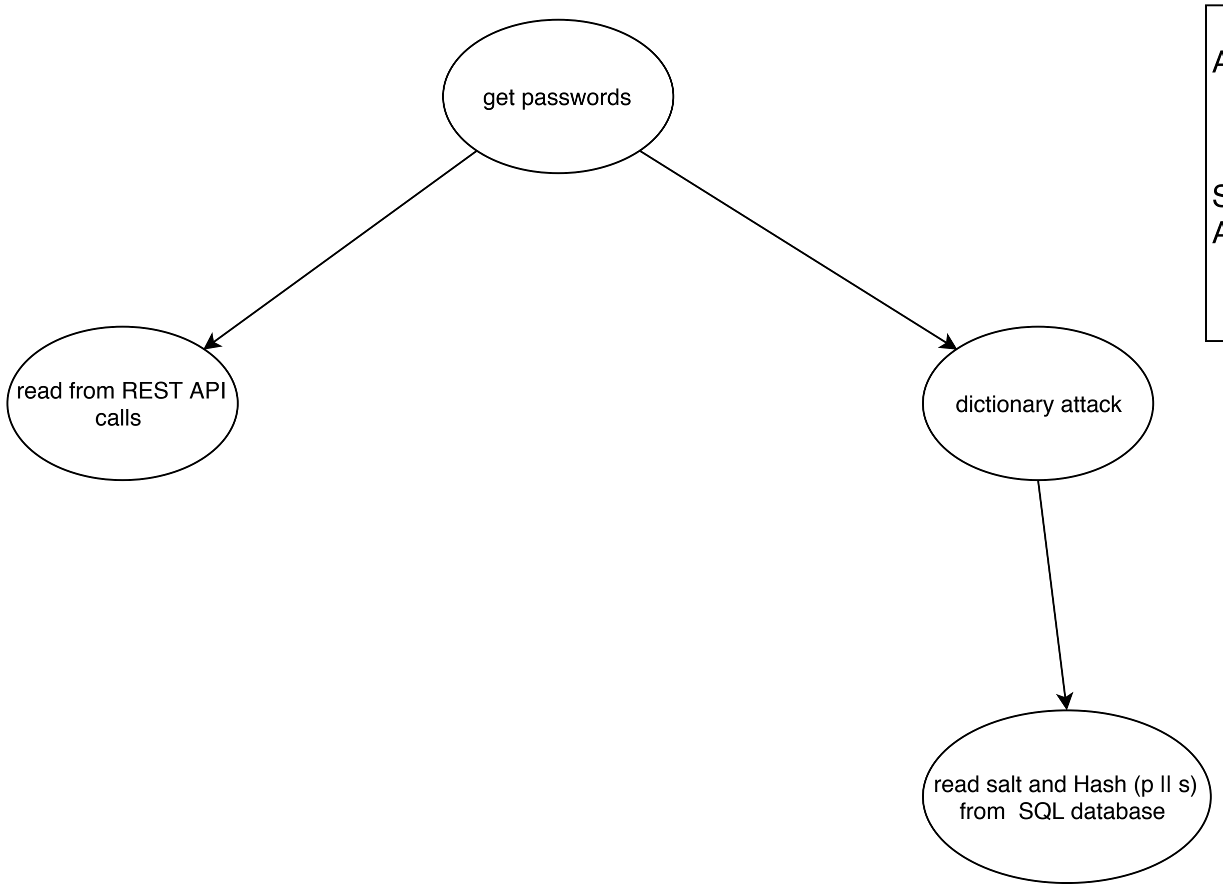
Stakeholder: user
Asset: AES/HMAC keys



Adversaries

1. insider, computationally limited, active

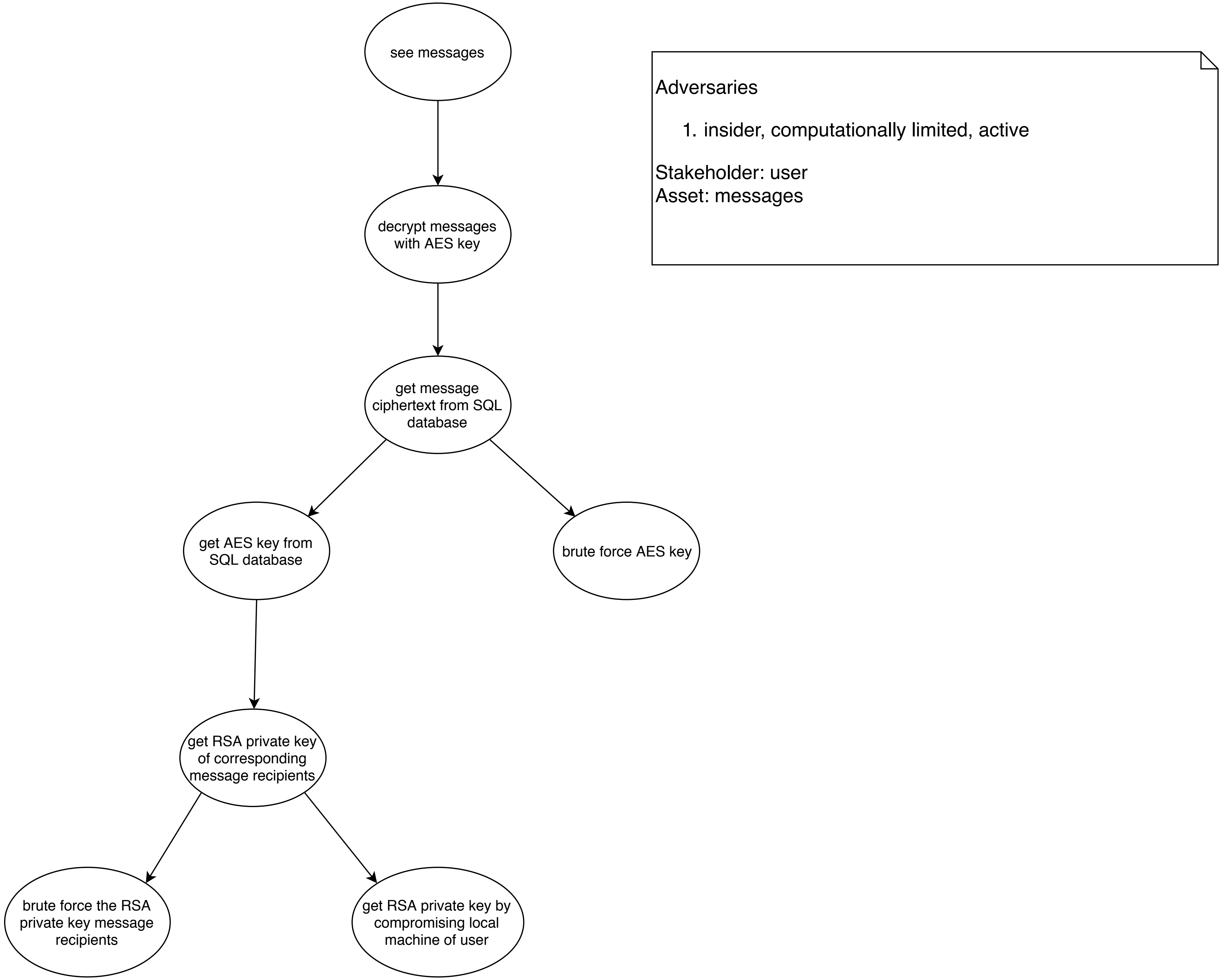
Stakeholder: user
Asset: email addresses



Adversaries

1. insider, computationally limited, active

Stakeholder: user
Asset: passwords

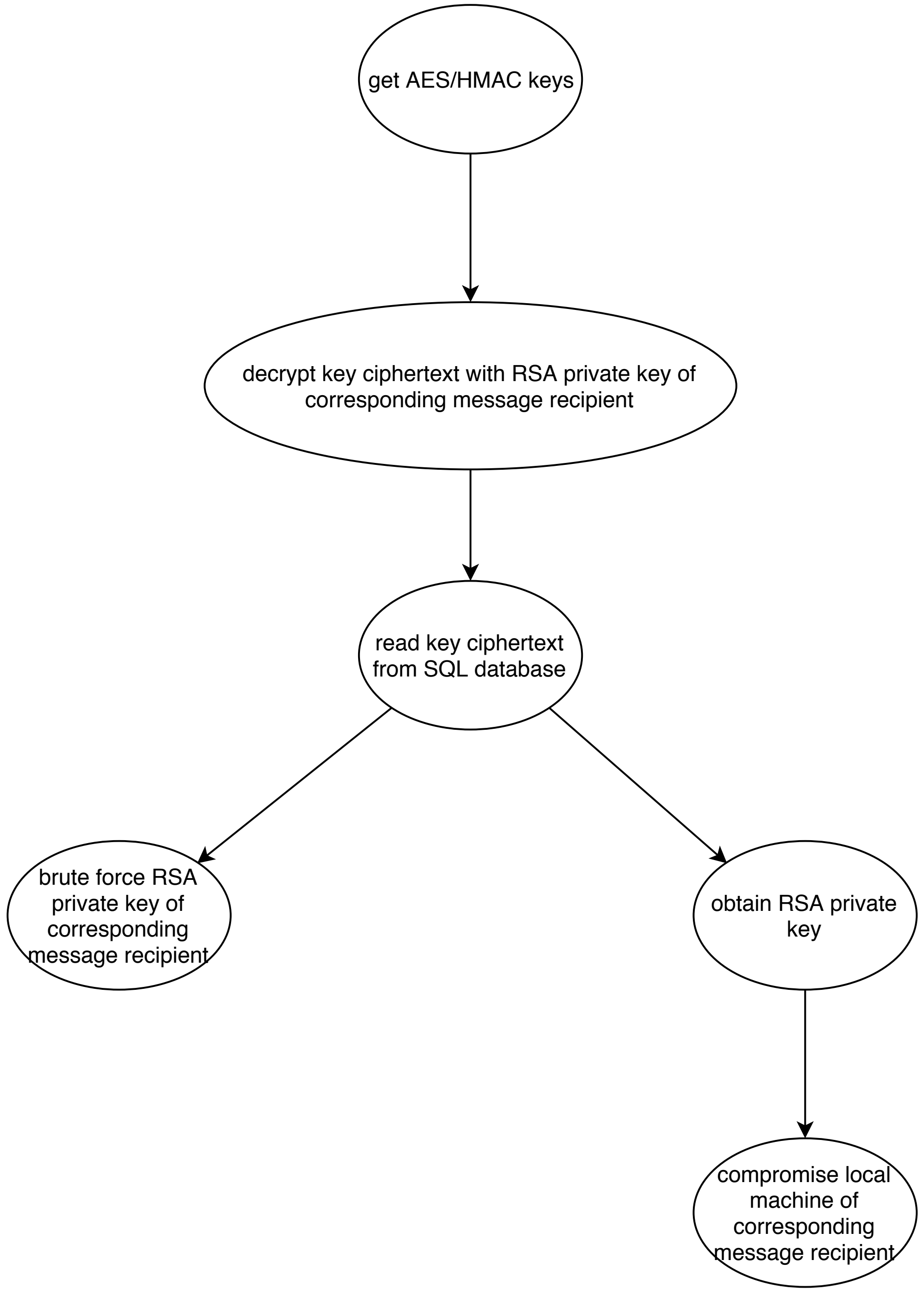


see messages
decrypt messages with AES key
get message ciphertext from SQL database
get AES key from SQL database
brute force AES key
get RSA private key of corresponding message recipients
brute force the RSA private key message recipients
get RSA private key by compromising local machine of user

Adversaries

1. insider, computationally limited, active

Stakeholder: user
Asset: messages


get AES/HMAC keys
decrypt key ciphertext with RSA private key of corresponding message recipient
read key ciphertext from SQL database
brute force RSA private key of corresponding message recipient
obtain RSA private key
compromise local machine of corresponding message recipient

Adversaries

1. insider, computationally limited, active

Stakeholder: user
Asset: AES/HMAC keys