CSC4000W: Network and Internetwork Security

Implementation of Asymmetric and Symmetric Cryptography

with Provisions for Authenticity and Integrity Verification

Gianluca Truda
TRDGIA001
University of Cape Town
trdgia001@myuct.ac.za

Luke Neville
NVLLUK001
University of Cape Town
nvlluk001@myuct.ac.za

Zakariyah Toyer
TYRZAK001
University of Cape Town
tyrzak001@myuct.ac.za

Robbie Barnhoorn
BRNROB026
University of Cape Town
brnrob026@myuct.ac.za

1 INTRODUCTION

In the digital age, we cannot trust that personal and private information will remain secure when being transmitted across a network. As a result, cryptographic methods have emerged that can protect information across unsecured networks.

This document details the design and implementation of a program that uses *RSA-2048* asymmetric encryption to securely distribute *AES-256* session keys used in symmetric encryption, and then continues secure transmission with symmetric encryption using the newly acquired key. We also discuss some common attacks and how we have implemented measures to combat them.

2 FRAMEWORK

Solutions to both Task 1 and 2 were coded using *Python 3.6*. The project directory contains three modules relating to the implementation of cryptographic methods. *crypto_tools.py* acts as a utility class, providing useful functions to *crypto_asymmetric.py* and *crypto_symmetric.py*, which implement asymmetric and symmetric cryptographic methods respectively. These two modules do not actually implement the cryptographic algorithms themselves, but rather act as an API gateway to the *PyCrypto* python package [2] - which implements the chosen encryption and decryption algorithms. The project directory also contains a *network_interface.py* module. This implements all useful networking methods, making use of the Python *Sockets* library. Finally, the *network_client.py* and *network_server.py* modules make use of all the previously mentioned modules in order to demonstrate the use of cryptographic methods in networked communication. These two modules implement the requirements as per the assignment. Both Task 1 and Task 2 are demonstrated in one single client-server script pair. This is because our solution utilises the session keys distributed in Task 1 for the symmetric encryption in Task 2. Each task is discussed in more depth in sections 3 and 4.

Our solution made the assumption that all public keys are trustworthy - they have been signed and verified by a trusted Certification Authority (CA). This allowed us to verify *authenticity* throughout Tasks 1 and 2. That aside, no other assumptions were made. We favour the implementation of the communications over a TCP connection that is secured with a Secure Socket Layer (SSL), but our solution worked on the assumption that this was not guaranteed. As a result, the entire solution provided full security features at the *application* level.

2.1 Running the Program

This section is a summarised version of the *Installation* and *Usage* sections in the README file, which can be found in the project directory. After downloading and unzipping the project files, ensure that Python 3.6 is installed, and install the PyCrypto library via your package manager of choice (e.g. *pip*).

Start the *server* application in one terminal and the *client* application in another. When this runs, the client and server should connect over a local TCP connection and communication messages should be displayed.

For demonstration purposes, a plethora of debug statements are printed to the terminal which show and describe each stage in the process of communication. In practice, this would be hidden from the end-users.

3 TASK 1: ASYMMETRIC ENCRYPTION

This module is responsible for securely sharing a secret message (a symmetric encryption key) between a Client node and a Server node. The Client and Server establish a TCP connection and exchange two sets of public keys (a public key for RSA encryption and a verification key for RSA signature verification). The Client node generates a new 256-bit AES key (the payload) and pads it. The resulting text is hashed using the SHA-256 algorithm. That hash is then signed using the Clients signing key (RSA-2048) to produce a *signature*. That text is encrypted using the Server's public key (RSA-2048) to produce the *ciphertext*. The ciphertext and signature are concatenated into a single message that is sent over a (preferably-secure) TCP connection to the Server node. This process is illustrated in the top half of figure 1. The Server node does the inverse process. It splits the message into the *ciphertext* and *signature*, then decrypts the *ciphertext* using RSA-2048 and its private key. This gives the secret message (AES key) with padding. That entire text is hashed using SHA-256. The *signature* of the received message is decrypted using RSA-2048 and the Client's public verification key. This produces results in a piece of text that is compared

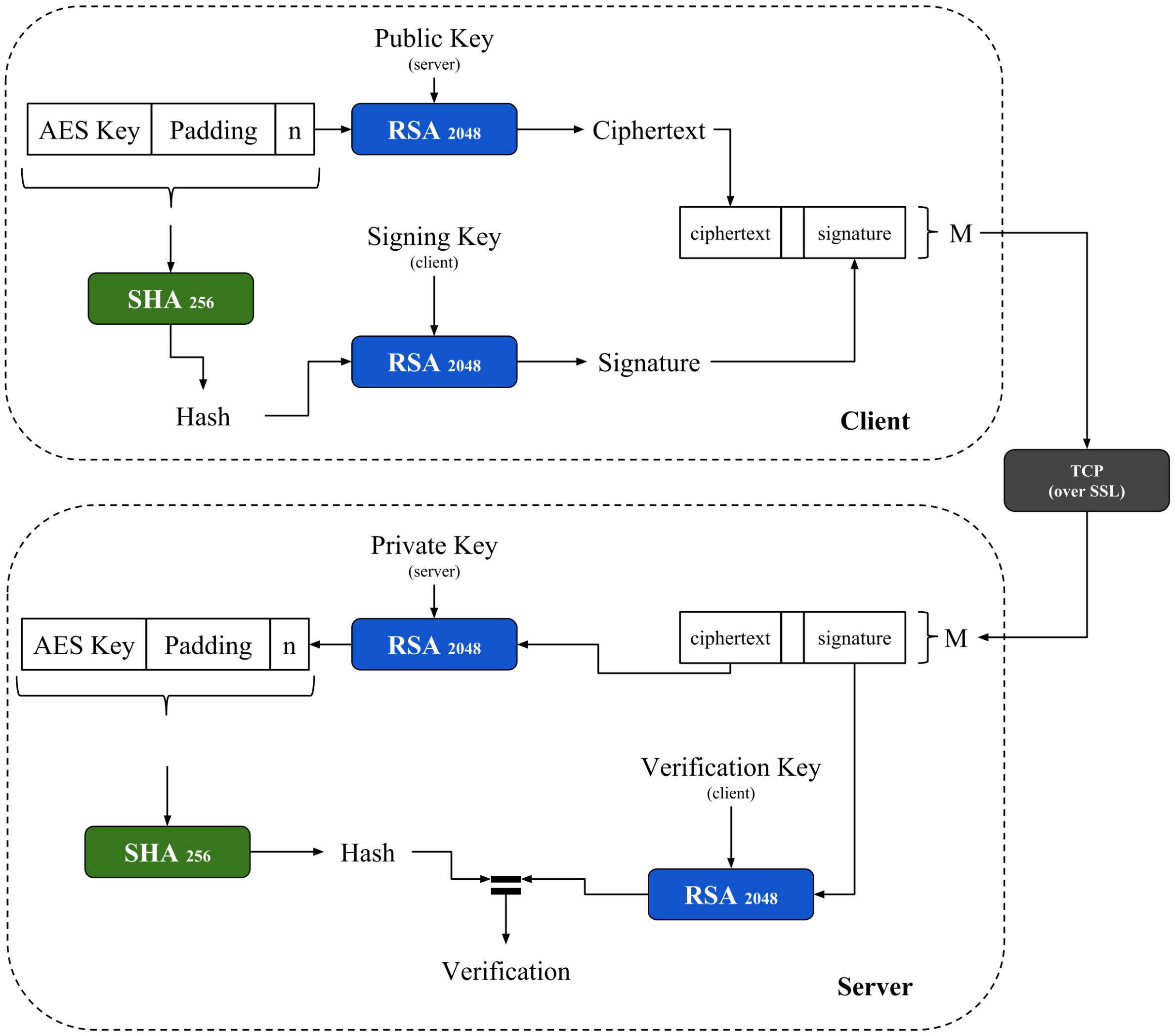


Figure 1: Signed public-key cryptography exchange between a Client and Server (Task 1). The payload of the message is a 256-bit AES key for later use in symmetric encryption (for Task 2).

with the hash to verify that the Client was indeed the source of the message and that it was unaltered. If the decrypted signature and hash do not match, either the Client was not the one to sign the message or the message was changed on the journey. This would indicate a malicious actor and the process would be aborted. This process is illustrated in the lower half of figure 1.

3.1 RSA-2048 Asymmetric Encryption

2048-bit RSA encryption is used for two functions: (1) to distribute a secret payload from Client to Server given no existing shared secret keys. (2) to digitally sign the sent message so that the Server can verify that it came from the intended Client.

To perform the first function, a variant of the 2048-bit RSA algorithm with Optimal Asymmetric Encryption Padding (OAEP) is used. This expands upon traditional RSA by converting it from deterministic to probabilistic, which provides the following benefits:

(1) Security against the *Chosen Ciphertext Attack*.
(2) Security against partial decryption of ciphertexts.

It should be noted that this operates atop a message that is already padded with our own padding algorithm. This provides two layers of unpredictability to the resultant ciphertext.

For the second function, the *PKCS v1.5* standard of 2048-bit RSA encryption [1] is used. This is well suited to digital signatures as it performs no padding and is designed specifically for efficient signing.

3.2 Asymmetric Key Exchange

Upon establishing a TCP connection between Client and Server nodes, the public encryption and public verification keys must be exchanged between the two. For the purposes of demonstration, our code generates two new sets of keys for both the Client and

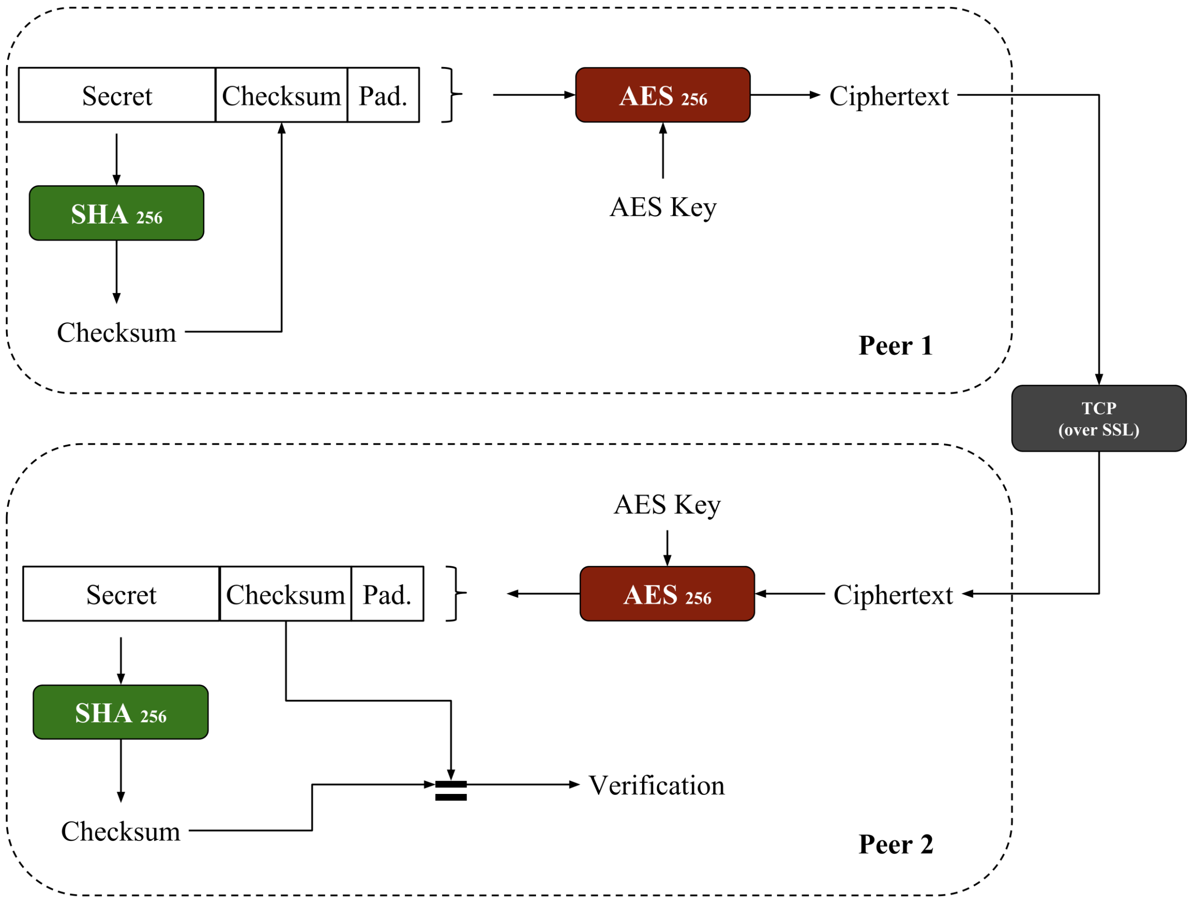


Figure 2: Checksummed symmetric encryption between two Peers (formerly the Client and Server) to exchange a secret message (Task 2).

the Server. In practice, both sets of keys would be pre-generated and stored (encrypted) on a secure keychain. The public encryption keys are certified by a Certification Authority (CA), as are the public verification keys (for signing). This provides third-party trust in the public keys, as it would be difficult for a malicious agent to obtain such certificates. Because both an *encryption* and a *signing* certificate are needed, the probability of a malicious agent successfully obtaining both is the product of the individual probabilities - a remarkably low chance. To fully utilise this double-certification approach, two different CAs should be used: one for public encryption keys, and one for public verification keys.

4 TASK 2: SYMMETRIC ENCRYPTION

This module utilises the key-exchange performed in section 3 to perform 256-bit AES symmetric encryption and exchange a secret message between two peers. The holder of the secret is Peer 1, who hashes the secret message with SHA-256 to produce a *checksum*. This checksum is appended to the secret message. The resulting string is padded with nonsense characters to be an AES-compliant length and for additional security against a wide range of attacks. The resulting text is encrypted using AES-256 (with the shared AES key) to produce the *ciphertext*. This is then sent over a (preferabbly secured) TCP connection to Peer 2 - the intended recipient of the secret message. Because the shared AES key was exchanged using signed RSA encryption (see Task 1), the only other party who can be in possession of the secret key is Peer 2. This means that only Peer 2 can use the shared secret key to run AES-256 decryption on the *ciphertext*. This yields the padded secret message and checksum. The secret message is hashed with SHA-256 to produce a new checksum. This is compared to the checksum in the received message. If the checksums match, Peer 2 has verified the authenticity of the secret message. If they do not match, it is likely that a malicious agent has modified the ciphertext - rendering the entire message invalid and alerting both Peers. This entire process is illustrated in figure 2.

4.1 AES-256 Symmetric Encryption

To perform symmetric encryption, the Cipher Block Chaining (CBC) mode of the 256-bit AES algorithm is used. This offers a number of advantages over the more simplistic Electronic Codebook (ECB) implementation. Most notably, CBC has the property of *cryptographic* *diffusion*, as each block is XORed with the previous ciphertext block before encryption. This reduces the risk of malicious agents finding data patterns and reduces susceptibility to *replay attacks*. Because AES-256 is a block cipher, it requires the plaintext to be a multiple of 256-bits in length. Although the chechsum is always 256 bits, the secret text is of variable length. To ensure the total length is AES-compliant, a bespoke padding function is used to add nonsense characters (in this case exclamation points, to avoid confusion with the hexadecimal checksum that proceeds them) that add up to the nearest multiple of 256.

4.2 Obtaining AES Keys

When two Peers wish to exchange secret information (either one-way or two-ways) they need a shared secret AES key - the *session key*. This is generated by the Client (see Task 1) after public keys are exchanged. To generate the session key, a 32-byte random number is generated using operating system primitives. This is more secure than most pseudo-random number generating algorithms as it is less prone to repeating patterns that may be exploited by malicious agents. The 32-byte number is converted to a 256-bit hexadecimal, which is used as the AES *session key*. This is then shared with the Server node using public-key encryption (see section 3).

5 EXAMPLE OF SECRET MESSAGE EXCHANGE

The following example demonstrates both the solution to Task 1 (section 3) and Task 2 (section 4). This is because, in our implementation, the asymmetric encryption of Task 1 is used to distribute the AES session key needed for Task 2.

5.1 Scenario

In this example, two parties (Alice and Bob) wish to exchange secret information over a regional network. A malicious party on the network (Eve) wishes to attack/disrupt/decrypt this exchange. Alice and Bob each control a node on the network and have both RSA encryption and RSA signing keys certified by two separate Certification Authorities (CAs). Bob has a secret message (in this case an English sentence) that he needs to securely give to Alice. Because it is such sensitive information, both Alice and Bob need the integrity and authenticity of the message to be verified, but they share no secret keys at present.

5.1.1 Establishing Connection. First, Alice runs an instance of our software in *Client* mode. Bob has been running an instance in *Server* mode, awaiting Alice's connection. Alice establishes a two-way TCP connection with Bob. Ideally, the connection is protected with Secure Socket Layer (SSL), but our system makes no such assumption. Upon connection, Bob's system (Server) sends Alice's system (Client) its public encryption key and its public verification key. Both keys are 2048-bit RSA keys sent as hexadecimal strings. Alice's tools verify that these keys are valid and checks for current certificates with the two CAs. If everything checks out, Alice's system sends her public encryption and public verification keys to Bob. Bob's system performs the same checks on these keys. If all the keys check out for both parties, Bob's system sends a message to Alice to commence the asymmetric encryption (Task 1).

5.1.2 Asymmetric (RSA) Encryption. Alice receives Bob's message and the system generates a new 256-bit AES session key. Alice's system passes this to the asymmetric encryption tools detailed in section 3 and illustrated in figure 1, which pad, sign, and encrypt the session key. The resulting message is sent over the TCP connection. Bob receives the message and his system decrypts it (as detailed in section 3 and illustrated in figure 1). The key is verified for *authenticity* and *integrity* by checking Alice's signature against Bob's own SHA-256 hashing of the plaintext. If anything is out of order, the session key is discarded and the session is restarted (with alerts to system administrators of a possible security risk). If everything checks out, Bob then proceeds to symmetric encryption (Task 2).

5.1.3 Symmetric (AES) Encryption. Bob gives his secret message plaintext to the symmetric encryption tools, which hash the secret message with SHA-256 to produce a checksum that is attached to the secret and padded (as detailed in section 4 and illustrated in figure 2). This plaintext is then encrypted (using the AES session key). The resulting ciphertext is sent via the TCP connection to Alice. Alice's system uses the symmetric decryption tools to decrypt the ciphertext with the AES session key. The message is verified for *integrity* against the checksum (as detailed in section 4). Because the symmetric exchange (Task 2) uses the secret AES session key shared with the asymmetric exchange (Task 1), the *authenticity* verified by the RSA signatures extends to this second task. As before, any irregularities abort the exchange and alert administrators. If all is in order, Alice's system hashes the received secret message (using SHA-256) and sends that to Bob over the same AES channel using the same steps as Bob used to send her the original secret message. Bob's system decrypts and verifies Alice's hash payload against the original secret message. If these do not match, the process is aborted and it is highly likely that a malicious actor is involved. If they do match, Bob sends a verification message to Alice. Both parties are now satisfied that the secret message has been successfully shared from Bob to Alice with no alterations or masquerading. The connection is then terminated.

5.2 Execution

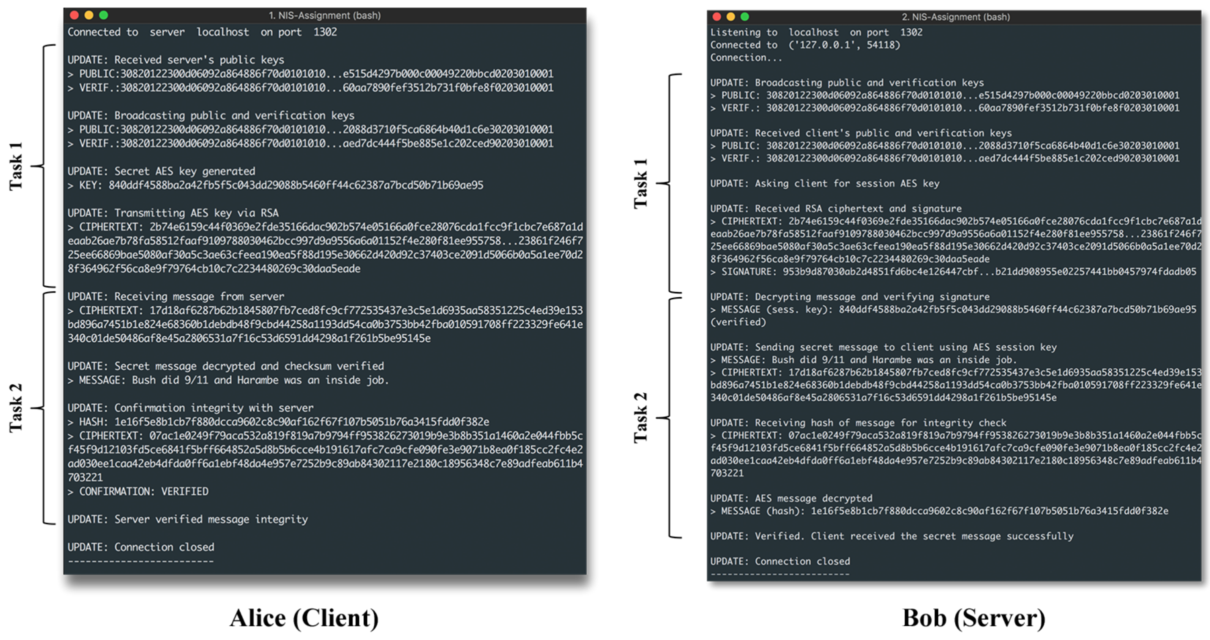
Running two terminals in the root directory allows for a good simulation of the above scenario. Bob's (server) node is demonstrated by running the *network_server.py* script, whilst Alice's (client) node is demonstrated by running the *network_client.py* script. The terminal outputs for an example of this demo code are presented in figure 3.

6 ATTACKS

In this section we briefly explain a number of typical attacks made on network communications.

6.1 Modification

In a message modification attack, the attacker intercepts the message and alters its contents in some way. This results in an incorrect packet on the receiving end, or no received packet at all. If the attacker alters, for instance, the destination address of the header, the message will not reach the intended recipient. The attack may also alter the actual contents of the message.
In order to do this, the attacker needs to be able to intercept and



```
Connected to  server  localhost  on port  1302

UPDATE: Received server's public keys
> PUBLIC:30820122300d06092a864886f70d0101010...e515d4297b000c00049220bbcd0203010001
> VERIF.:30820122300d06092a864886f70d0101010...60aa7890fef3512b731f0bfe8f0203010001

UPDATE: Broadcasting public and verification keys
> PUBLIC:30820122300d06092a864886f70d0101010...2088d3710f5ca6864b40d1c6e30203010001
> VERIF.:30820122300d06092a864886f70d0101010...aed7dc444f5be885e1c202ced90203010001

UPDATE: Secret AES key generated
> KEY: 840ddf4588ba2a42fb5f5c043dd29088b5460ff44c62387a7bcd50b71b69ae95

UPDATE: Transmitting AES key via RSA
> CIPHERTEXT: 2b74e6159c44f0369e2fde35166dac902b574e05166a0fce28076cda1fcc9f1cbc7e687a1d
eaab26ae7b78fa58512faaf9109788030462bcc997d9a9556a6a01152f4e280f81ee955758...23861f246f7
25ee66869bae5080af30a5c3ae63cfeea190ea5f88d195e30662d420d92c37403ce2091d5066b0a5a1ee70d2
8f364962f56ca8e9f79764cb10c7c2234480269c30daa5eade

UPDATE: Receiving message from server
> CIPHERTEXT: 17d18af6287b62b1845807fb7ced8fc9cf772535437e3c5e1d6935aa58351225c4ed39e153
bd896a7451b1e824e68360b1debdb48f9cbd44258a1193dd54ca0b3753bb42fba010591708ff223329fe641e
340c01de50486af8e45a2806531a7f16c53d6591dd4298a1f261b5be95145e

UPDATE: Secret message decrypted and checksum verified
> MESSAGE: Bush did 9/11 and Harambe was an inside job.

UPDATE: Confirmation integrity with server
> HASH: 1e16f5e8b1cb7f880dcca9602c8c90af162f67f107b5051b76a3415fdd0f382e
> CIPHERTEXT: 07ac1e0249f79aca532a819f819a7b9794ff953826273019b9e3b8b351a1460a2e044fbb5c
f45f9d12103fd5ce6841f5bff664852a5d8b5b6cce4b191617afc7ca9cfe090fe3e9071b8ea0f185cc2fc4e2
ad030ee1caa42eb4dfda0ff6a1ebf48da4e957e7252b9c89ab84302117e2180c18956348c7e89adfeab611b4
703221
> CONFIRMATION: VERIFIED

UPDATE: Server verified message integrity

UPDATE: Connection closed
------------------------
```

```
Listening to  localhost  on port  1302
Connected to  ('127.0.0.1', 54118)
Connection...

UPDATE: Broadcasting public and verification keys
> PUBLIC: 30820122300d06092a864886f70d0101010...e515d4297b000c00049220bbcd0203010001
> VERIF.: 30820122300d06092a864886f70d0101010...60aa7890fef3512b731f0bfe8f0203010001

UPDATE: Received client's public and verification keys
> PUBLIC: 30820122300d06092a864886f70d0101010...2088d3710f5ca6864b40d1c6e30203010001
> VERIF.: 30820122300d06092a864886f70d0101010...aed7dc444f5be885e1c202ced90203010001

UPDATE: Asking client for session AES key

UPDATE: Received RSA ciphertext and signature
> CIPHERTEXT: 2b74e6159c44f0369e2fde35166dac902b574e05166a0fce28076cda1fcc9f1cbc7e687a1d
eaab26ae7b78fa58512faaf9109788030462bcc997d9a9556a6a01152f4e280f81ee955758...23861f246f7
25ee66869bae5080af30a5c3ae63cfeea190ea5f88d195e30662d420d92c37403ce2091d5066b0a5a1ee70d2
8f364962f56ca8e9f79764cb10c7c2234480269c30daa5eade
> SIGNATURE: 953b9d87030ab2d4851fd6bc4e126447cbf...b21dd908955e02257441bb0457974fdadb05

UPDATE: Decrypting message and verifying signature
> MESSAGE (sess. key): 840ddf4588ba2a42fb5f5c043dd29088b5460ff44c62387a7bcd50b71b69ae95
(verified)

UPDATE: Sending secret message to client using AES session key
> MESSAGE: Bush did 9/11 and Harambe was an inside job.
> CIPHERTEXT: 17d18af6287b62b1845807fb7ced8fc9cf772535437e3c5e1d6935aa58351225c4ed39e153
bd896a7451b1e824e68360b1debdb48f9cbd44258a1193dd54ca0b3753bb42fba010591708ff223329fe641e
340c01de50486af8e45a2806531a7f16c53d6591dd4298a1f261b5be95145e

UPDATE: Receiving hash of message for integrity check
> CIPHERTEXT: 07ac1e0249f79aca532a819f819a7b9794ff953826273019b9e3b8b351a1460a2e044fbb5c
f45f9d12103fd5ce6841f5bff664852a5d8b5b6cce4b191617afc7ca9cfe090fe3e9071b8ea0f185cc2fc4e2
ad030ee1caa42eb4dfda0ff6a1ebf48da4e957e7252b9c89ab84302117e2180c18956348c7e89adfeab611b4
703221

UPDATE: AES message decrypted
> MESSAGE (hash): 1e16f5e8b1cb7f880dcca9602c8c90af162f67f107b5051b76a3415fdd0f382e

UPDATE: Verified. Client received the secret message successfully

UPDATE: Connection closed
------------------------
```

Figure 3: Terminal output for Alice (Client) and Bob (Server) in the example scenario. Task 1 leads directly into Task 2, but the separation between tasks is shown for evaluation purposes.

block communication between the two parties, and masquerade as the sender.

6.2 Replay

A replay attack, also known as a playback attack, is when the attacker intercepts a message and fraudulently delays or re-sends the message in order to manipulate the receiver. For example, the attacker might resend a log-on message to a server, which would think that the message is legitimate and return some access keys to the attacker. It is important to note that the attacker need not know the contents of the message, but can still manipulate the receiver by making it process or respond to a message multiple times.

6.3 Traffic Analysis

A traffic analysis attack is when the attacker intercepts messages sent between two parties and, even without knowing the content of the message, can make useful deductions on the communication by looking at the metadata, frequency, and timing of messages. For example, the attacker might observe that a message is sent from A to B every day at exactly 8am. The attacker can also use this to determine what type of information is being sent, such as chat messages, emails or web page requests.

6.4 Meet in the Middle and Short Message

These attacks can be applied on the RSA crypto-system, and are a result of the sent message (M) being being too small. This allows the attacker to exploit the underlying mathematics of RSA to deduce the contents of the message without knowing the encryption keys.

If the message M is shorter than $N^{1/e}$ ($M < N^{1/e}$), where N is the modulus used and e is the encryption exponent, then the attacker can easily compute the message M by taking the e^{th} root of M. The attacker can also work out M if it is the product of two small integers A and B.

6.5 Spoofing

A spoofing attack in a general name of any attack where the attacker successfully masquerades as someone they are not.

6.6 Side Channel

A side channel attack is a class of attack where the attacker exploits the physical implementation of the computer system, rather than exploiting a weakness in the crypto-system. The attacker can find exploits by monitoring power consumption, timing functions, or even monitoring sound.

7 SECURITY FEATURES

In this section we detail the security features that are implemented in our solution, pointing out which potential attacks they avoid.

7.1 Digital Signatures

A digital signature is a unique signature intrinsically attached to a digital document, such as a message, that ties an identity to the document. They are designed such that only the person who created the document could have signed the document. This means it can be used to irrevocably prove that a message originated from the person who claims to have created it. In our solution, the exchange of secret AES keys via RSA encryption makes use of digital signatures.

By signing the message with their private signing key, the sender allows the recipient to verify the authenticity of the message using the sender's public verification key. Because all subsequent communications make use of the shared AES session, and because the authenticity of that key is verified, all further communications in that session are guarded against *modification* and *spoofing* attacks.

7.2 Message Padding

In order to prevent *meet-in-the-middle* and *short message* attacks, as well as some forms of *traffic analysis* attacks, each message that is send via asymmetric (RSA) encryption is padded with a random number of zeros between 10 and 99. This extends the length of the message such that it is not less than $N^{1/e}$, and also means that the the resulting ciphertext is different every time, even if the plaintext is the same. This padded message is used both for the main RSA encryption and for the signing, so that both are non-deterministic and robust to *known-ciphertext* attacks. For the asymmetric encryption of the message, the Optimal Asymmetric Encryption Padding (OAEP) mode of the RSA-2048 algorithm is used. This adds an additional layer of non-determinism to the padding security.

7.3 Session Key Verification

In Task 1, a new AES session key is generated and shared from the Client to the Server through RSA encryption. As detailed above, two-layer padding and digital signatures are used in this RSA encryption. Upon receiving the session key, the Server node hashes it (with SHA-256) and sends this hash back to the Client node using the same RSA encryption (with two-layer padding and digital signatures). The client decrypts the message and compares the hash with its own hash of the session key. This means that, in addition to the authentication and other security features, the session key has verified integrity. This means that the sender of the shared secret key can be certain that the receiver got the correct secret key and nothing was tampered with in transit.

7.4 Message Checksums

In Task 2, symmetric encryption (AES-256) is used to share secret messages between an establish pair of Peers using a shared session key. The messages to be sent are hashed (with SHA-256) to produce a checksum that is concatenated with the secret message to produce the plaintext. That plaintext is then encrypted and the resulting ciphertext is transmitted over the TCP connection. This checksum allows the receiver to verify the integrity of the received messages, as any changes to the ciphertext would result in a mismatch between the checksum and the hash of the secret message. This, on top of the Cipher Block Chaining (CBC) mode of AES used, provides robust security against *modification attacks*.

7.5 Secret Message Verification

The secret messages in symmetric (AES) encryption are verified for integrity by both Peers. For instance, when Alice sends Bob a secret message, Bob verifies it with the checksum, but then hashes the secret message (with SHA-256) and sends that back to Alice via the same symmetric encryption channel (with checksum). Comparing the received hash with the hash of the sent secret message allows Alice to be sure that Bob received the exact message she intended. This is yet another layer of protection against a host of attacks, including *modification* and *replay* attacks.

7.6 Message IDs

As described in section 6.2, *replay attacks* potentially result in illegitimate messages being sent to a receiver. This can be prevented in two ways. The first option is for the sender to *timestamp* each message. The receiver will then look at this timestamp and only accept the message if it was sent within a the requisite time window. This prevents the attacker holding onto a message or delaying messages by reducing the time in which they can resend messages. But this is imperfect, as the attacker can still perform a replay attack as long as it happens within the window period.

A better solution is for each party to attach a unique ID to each message. The receiver can then store this ID, and will not accept another message if it contains the same ID. This prevents the attacker from replaying any message at all as it will instantly be rejected by the receiver. In practice, each user is not required to store the ID of every message received, as the message IDs are sequential and the receiver only needs to store the previous message ID. When a message is received, the ID is checked against the previous ID to ensure that it follows sequentially. If it does not, it is rejected. If it does, the message is accepted and the new message ID is saved, overwriting the previous one. This superior method was chosen for our system.

7.7 Dummy Messages

Traffic analysis attacks are difficult to prevent, as they involve studying only the metadata (which is impractical to hide). Our solution attempts to reduce the effectiveness of traffic analysis attacks by sending dummy messages. These messages look like they may be important to an attacker, though in reality the receiver will know they are useless. These messages are sent at random intervals, interspersed in the legitimate communication. In the eyes of the attacker this scrambles the communication and adds noise which makes interpreting the message metadata difficult.

REFERENCES

[1] B Kaliski. 1998. PKCSv1.5 Documentation. https://tools.ietf.org/html/rfc2313
[2] Darsey Litzenberger. [n. d.]. PyCrypto. https://www.dlitz.net/software/pycrypto/