表格

描述已自动生成

Coding Issues

Inadequate exception handling:

In several places, exception handling is very general and does not provide enough context.

In particular, try-except blocks only catch Exception without distinguishing between different types of exceptions.

Example:

except Exception as e:

print("Exception: ", {e})

traceback.print\_exc()

stop\_event.set()

break

Improvement suggestion: Specific exception types should be caught and handled appropriately according to different exception types.

Missing validation of user input:

When processing user input, such as sendPrivateMessage function and setFingerprint function, no input validation is performed.

Example:

participantsString = input("Who would you like to send the message to: ")

myFingerprint = input()

Improvement suggestion: Add input validation to ensure that the value entered by the user is valid and in the expected format. Especially when it comes to security-sensitive data (such as fingerprints, message targets, etc.), the security of the input should be ensured.

Sensitive data exposure:

The getPrivateKey function directly exports and sends the private key.

Example:

message = myPrivateKey.export\_key().decode()

Improvement suggestion: The private key should not be exposed or sent to any external entity. Ensure that the private key is always stored in a secure environment and that encryption operations are only performed when used locally.

Lack of verification of encryption details:

In the implementation of RSA and AES encryption and decryption in many places, the decryption results are not sufficiently verified, which may result in tampered messages passing.

Example:

receivedAESKey = myRSACipher.decrypt(receivedEncryptedKey)

Improvement suggestion: The decryption result should be verified to be in the expected format (such as signature verification, message structure integrity, etc.).

Backdoor:

Backdoor #1:

Location:

In the receiveMessages function, there is a piece of code

if data["message"] == "admin":

await asyncio.gather(getPrivateKey(websocket))

Problem: If the received message is "admin", the code will call the getPrivateKey function, which will transfer the private key to the server.

Risk: This is a very obvious backdoor implementation. An attacker can easily obtain the private key of the server or client by sending an "admin" message, resulting in a serious security vulnerability. Anyone who sends an "admin" message can trigger this mechanism.

Backdoor #2:

Location: setFingerprint function

async def setFingerprint():

print("Input fingerprint to change to: ")

myFingerprint = input()

Problem: This function allows users to arbitrarily change their fingerprints by entering a new fingerprint. This means that an attacker can pretend to be another client or user and bypass authentication.

Backdoor #3:

Location: In the message processing logic of the clientHandler function

elif data["type"] == "client\_update":

serverSockets = []

currentServerIndex = 0

for server in server\_list:

serverSockets.append(server["socket"])

if server["socket"] == activeSocket:

break

currentServerIndex += 1

if currentServerIndex == len(server\_list):

print("An unverified server is trying to send a message!!!")

continue

server\_list[currentServerIndex]["clients"] = data["clients"]

elif data["type"] == "client\_update\_request":

await asyncio.gather(sendClientUpdate(activeSocket))

else:

print("SERVER SHUTTING DOWN - BACKDOOR #3 FOUND")

break

Problem: If the message type does not match the expected processing path, the program will print "SERVER SHUTTING DOWN - BACKDOOR #3 FOUND" and shut down the server. This means that the server can be shut down remotely by sending certain types of messages.

I. Potential coding issues

1. Security issues of file upload paths:

file\_path = os.path.join(os.getcwd(), f'./upload/{filename}')

When downloading files, the code does not strictly check the content of filename, which may allow malicious users to construct filenames for path traversal attacks and access any file on the server. This attack can cause attackers to read sensitive files on the server, such as configuration files, password files, etc.

Suggestion: Add a validity check for filename, avoid any path traversal behavior, or use a secure file path management tool to manage file storage and reading.

2. Unverified file upload processing:

uploaded\_file = data.get("file")

with open(f"./upload/{uploaded\_file.filename}", "wb") as fout:

fout.write(uploaded\_file.file.read())

When uploading files, no check or verification is performed on the uploaded files, and malicious code or oversized files may be uploaded. Uploaded malicious code may be directly executed, resulting in code injection attacks.

Suggestion: Strictly check the size and type of files before uploading them to avoid uploading executable files or oversized files.

3. Lack of exception handling mechanism:

Some codes do not have comprehensive exception handling when handling WebSocket connections or file operations, for example:

async with websockets.connect(distant\_address, ping\_interval=10) as server\_websocket:

try:

...

except Exception as e:

print(f"An error occurred: {e}")

Some exceptions only print error information without further recovery mechanism or error status processing, which may cause system instability or even downtime.

Suggestion: Add more detailed exception handling, not only capture and print errors, but also consider system recovery or retry mechanism, especially in network operations or file system operations.

4. Hard-coded keys in RSA encryption:

PASSPHRASE = "G40"

The password of the RSA private key is hard-coded, and anyone who can read the code can easily obtain this password. If the key file or source code is leaked, the attacker can easily decrypt all communication content.

Suggestion: Do not hard-code sensitive information into the code, and manage passwords and keys through secure configuration files or environment variables.

5. Insufficient user authentication in message processing:

When sending and receiving WebSocket messages, the code confirms the user identity by comparing the recorded WebSocket object:

if internal\_online\_users[id]['socket'] == websocket:

from\_user = id

If an attacker hijacks the WebSocket object or forges a request through a malicious WebSocket connection, the system may not be able to detect it.

Recommendation: Re-verify the user identity through a secure authentication mechanism (such as JWT, OAuth) at each message delivery, rather than relying solely on the WebSocket object.

6. Lack of cleanup mechanism in user exit logic:

When the user exits, there is only a simple connection cleanup:

if (internal\_online\_users[online\_user\_id]['socket'] == websocket):

del internal\_online\_users[online\_user\_id]

But other resources related to the user (such as message queues, caches, etc.) are not cleaned up during cleanup, which may cause memory leaks.

Recommendation: When the user exits, make sure to clean up all resources related to the user to avoid memory leaks when the system is running for a long time.

2.bakdoor

1. RSA key pair generation and storage mechanism:

When the code generates an RSA key pair, the key pair is stored in the file system:

private\_key = RSA.import\_key(private\_file.read(), passphrase="G40")

If one party (such as a developer or system administrator) pre-generates a set of keys and presets these keys in the code, then the party will always be able to decrypt the communication content or forge the user's signature. This may be used as a backdoor.

Recommendation: Avoid using shared RSA key pairs and ensure that each user has their own independently generated key pair. Key management should be done through a secure mechanism, such as an encrypted keystore or hardware security module (HSM).

2. Unauthorized server neighbor registration:

In the code, NEIGHBOURS defines the neighbor relationship between servers, but there is no sufficient verification mechanism to ensure the legitimacy of the neighbor server. If a malicious server is registered as a neighbor, it may be able to monitor all communications or inject forged messages into other legitimate servers.

for neighbour in ONLINE\_NEIGHBOURS:

await ONLINE\_NEIGHBOURS[neighbour]['socket'].send(message)

Recommendation: When adding a neighbor server, use a strict authentication mechanism (such as certificate-based SSL/TLS authentication) to ensure that only legitimate servers can participate in the communication.

3. Potential message replay attack:

Although the code uses recv\_counter to prevent message replay attacks, the management of this counter is not perfect. If an attacker can predict or hijack the counter, he can forge messages or replay previous messages.

ValidateMessage(parsed\_message['counter'], server\_state['clients'][sent\_from]['counter'])

Recommendation: Use more secure methods (such as timestamps, nonce) in each communication to prevent replay attacks and ensure the uniqueness of each message.

4. Unencrypted client state management:

with open("client\_state.json", "r") as client\_state\_json:

client\_state = json.load(client\_state\_json)

The client state is stored in plain text in a local file, which can be modified or read by any user with file system access rights. This may allow an attacker to forge identities, tamper with public keys, or change other important client information.

Recommendation: Encrypt the client state file or use a more secure storage mechanism (such as the key storage service provided by the operating system).