Security Remediation

Group 1:
Tirth Kamlesh Kothari, Matthew Harper, Anirudh Sunil

Security remediation is the process of addressing system security weaknesses, this process aims to detect and contain threats to networks before they spread.

Remediations for Network Attacks (realistic):

Sniffing Attack Remediation: The best method of preventing sniffing attacks is to use **encryption**. For **Client** to **server** communications, we use the **SSL** wrapper object for **Python sockets** to implement **TLS** using a **certificate** for the **network server**. We were unable to complete the implementation of **SSL wrappers** for the **client-to-client** communications. However, we have planned out the implementation as follows:

1. Upon registration of **unique clients,** this is those whose IP, Name, and submitted Public keys are not the same as any already registered clients will have a Certificate Generated for them
    a. A CSR with the Client's information (Common name is it's IP) is generated
    b. The CSR is **signed** with the **server's** private key (This is because the server is already a trusted entity in the system, through its certificate, we can establish a chain of trust)
2. The certificate is stored in the Server's Memory.
3. The certificate is sent to the registering client
    a. This is so it can be used to establish a SSL socket wrapper
4. The certificate is included in requests for the list of clients

The SSL wrapper is secure because it is a well know and well tested implementation of a commonly used and well known standard.

Replay Attacks:
Timestamps! (we limit the time messages are valid)
Through the use of timestamps one can keep track of the time difference, if a message contains a timestamp that exceeds the threshold limit, one can directly ignore it and drop the packet. Nonces are another alternative, as one will store the unique nonces sent in each message then if someone attempts to use the same nonce value again. we can ignore the message that reuses the nonce this can prevent the system from replay attacks.

As this is a test case scenario, the grace period is unusually long - however in most cases we would attempt to make is as short as possible.

If the communications are **unencrypted** as they are in flow3 currently, an attacker can replace a nonce or timestamp with one that is valid. Our plan of action in remediating Sniffing attacks for client-to-peer communications would prevent this as the timestamp would be encrypted like all other messages in the secure tunnel SSL creates between hosts and its integrity guaranteed.

Reflection Attacks:
No challenge response is explicitly used in our system. Authentication cannot be achieved by reflecting messages on a host. If one is later necessary one can limit and restrict the use of nonces. For example allowing only odd nonces to authenticate the server and authenticating clients with even nonces.

Impersonation Attacks:
The aforementioned Certificates, the server and clients use which are implemented in the case of client-to-server communications and in progress for client-to-server communications, are what we use to prevent impersonations. The contents of the message are also compared to information gathered by the receiving socket. They are authenticated through the use of the TLS protocol, as session keys are encrypted with public keys which are guaranteed by certificates and additional values are signed - meaning only the expected recipient can receive the session key, and will be able to determine the expected sender was the one creating the messages.

The only time a user can bypass these protections is when initially registering to the server. The implementation requirements mention no Token (used by something like Docker swarm) to prevent registrations, and the required information is not present to validate users on their initial connection (like SSH) so we cannot validate users before they are registered.

Man in the Middle (MITM):
The aforementioned Certificates mentioned in the Sniffing and Impersonation attacks again, for the client-server communication, the use of SSL authentication prevents MITM attacks. The communications are encrypted, and their integrity is guaranteed as well through the cryptographic methods SSL implements.

These attacks can be mitigated by validating the authenticity of SSL/TLS certificates used for communications between the client and server. This involves checking the validity and integrity of the certificate, verifying the certificate chain, and ensuring the certificate is issued by a trusted Certificate Authority (CA). The TLS protocol (SSL socket) will handle the authentication of the party/parties - see Impersonation.

We currently only implement Client-Server SSL communications but as mentioned in the Sniffing section we have plans to implement SSL for Client-Peer connections which would mitigate MITM for them.

Denial of Service (DoS) and Distributed Denial of Service (DDoS):
The use of timeouts prevents malicious (or forgetful benign actors) from utilizing resources for an unlimited amount of time. These are present in the receiving/listening sockets and the sending sockets which will receive a response.

The number of connections that the client, and network (server) support **are limited**, this is to prevent the system from becoming overwhelmed by the number of connections, preventing timely response to active clients. As timeouts are set, inactive clients will be removed (preventing issues as described above).

For more destructive attacks that utilize botnets or techniques such as Syn-Flooding, to overwhelm the host system… Well those protections are left up to the system that hosts this program, and the network's configurations.

To prevent invalid responses, packets or even the errors thrown by timeouts we implement basic error handling wherever possible. We have plans to implement additional error-handling functionality wherever possible.

Key management attacks :
These attacks can be mitigated by following some rules for key:

-Full lifecycle management of keys
-Secure methods for key distribution
-Destroying keys at the end of their lifecycle
-Strong user authentication
-Ability to respond quickly to any detected compromise.

We generate a unique asymmetric key pair for each client (and manually do so for the server)
The SSL wrapper established a session key, using a form of Diffie Hellman for each unique session.

Race Conditions
Semaphore! (It's basically a variable or abstract data type used to control access to common resources by multiple threads and avoid collision) (Its a variable which is used to allow process synchronization) Without this data can be corrupted and functionality may be affected.

Perfect Forward Secrecy:
We do not directly implement or use functions to directly utilize Diffie Hellman. However, we use or plan to use SSL socket wrappers to provide security to communications. This implements TLS, which utilizes some form of Diffie Hellman to implement a scheme that provides perfect forward secrecy.

Injection/Overflow Attacks:
We do not directly defend against these attacks, we operate based on a predefined set of flags, so it is not possible to inject commands to be directly executed as no execs based on client/peer input are done. Additionally, the python language is not (really) susceptible to overflows as it checks the bounds of arrays at runtime.

Remediations of Outlandish Attacks:

Side Channel Attacks:
There is not much we can do, encase the host computer in cement. Ask the person with the electron microscope what they are doing in our home. Cameras?

Root Kit/Malware:
We would need to utilize hardware that creates a Trusted Execution Environment (ARM Trustzone or something based on RISC-V) and implement out code so that it runs as a package in that environment. Additionally we would need to have secure boot enabled to guarantee the integrity of our trusted system, relying on the trusted nature of our hardware. It is also possible to prevent and detect basic malware using antivirus tools/software.

Justin Marwad:
Distraction… quick mention Automation and Bing Chat!