Report for the Course on Cyber-Physical Systems and IoT Security

Title of the reference paper: Authentication of IoT Device and IoT Server Using Secure Vaults

Name of who is involved in the project: Michael Amista' (2122865), Marco Brigo (2121727)

Source code: https://github.com/MikeIsBack/CPS-Final-Project

Objectives

The primary objective of this project was to implement a secure communication protocol for mutual authentication between an IoT client device and an IoT server. Unlike the reference work, this project focused on delivering a fully software-based simulation of the protocol. The aim was to emulate the cryptographic operations, state management, and communication flows entirely in Python, shaping a flexible simulation environment for experimentation and testing.

Specifically, the protocol required establishing a multi-session communication system where both the client and the server exchange challenges and responses securely, update their shared cryptographic vault, and maintain synchronized states across different sessions. The shared vault serves as a critical security feature, with periodic updates on vault keys to ensure evolving security properties. The protocol incorporates robust cryptographic primitives such as AES encryption (that can be freely chosen among AES-128 and AES-256), random number generation, ensuring the confidentiality and integrity of transmitted data.

System Setup: programming language, libraries and tools

The project was developed and tested using Python, leveraging several libraries and tools to simulate cryptographic operations, inter-process communication, and vault management. A key component of the implementation was the use of the cryptography module for AES encryption. Sensitive data exchanged during communication sessions was encrypted and decrypted securely, with padding utilities ensuring that data blocks met the AES-128 or 256 encryption requirements.

Inter-process communication between the client (IoT device) and the server was established using Python's built-in socket library. A TCP connection was implemented where the server listened on a specified port, and the device initiated the connection. This setup enabled real-time emulation of the message exchange process in a controlled network environment.

For vault management, a shared repository of cryptographic keys was stored in a binary file (vault.pkl). The vault is updated after each session using an HMAC-based mechanism. An HMAC is computed with the exchanged session data as the key and the current vault as the message. The HMAC output is used to update the vault partitions via XOR operations. This process ensures the updates are tied to session-specific data, maintaining synchronization and security between the client and server for future sessions.

The randomness necessary for cryptographic operations was generated using the random module, which provided cryptographically secure random indices and keys. These random values were integral to creating challenges (C1 and C2) and session-specific random numbers (r1 and r2), ensuring the unpredictability and robustness of the protocol.

To simulate a networked communication environment, the client and server scripts were run as separate processes on the same machine.

Multi-session simulation was a crucial aspect of testing. The protocol's behavior was evaluated across different sessions (by default five) in each execution of the client and server scripts. These sessions included essential stages such as initial message exchanges (e.g., session and device identification), challenge-response generation and verification, and cryptographic vault updates.

System Setup: file structure

The different files that shape the simulation can be found under the directory “Simulation/” in the linked repository.

The **constants.py** file serves as the centralized file for all configuration settings and global parameters used throughout the simulation. By defining essential attributes such as the size of the vault, the length of cryptographic keys, session duration, and network settings, it ensures consistency across the implementation and provides a single location to modify these values.

The **vault.py** file is responsible for managing the secure cryptographic vault, which is central to the authentication protocol. It includes functions to initialize the vault with a set of random keys, load the vault content from a file, and save updated versions of the vault back to persistent storage. The file also handles the vault update process, ensuring that its state evolves securely after each session using cryptographic techniques.

The **utils.py** file provides essential support for the cryptographic operations required in the protocol. It includes functions for encrypting and decrypting data, applying padding to align data with block sizes, and generating random indices for cryptographic challenges. Additionally, it contains functionality for combining keys from the vault through XOR operations.

The **server.py** file implements the server's logic for handling authentication and vault management. It listens for incoming connections, initiates sessions, and manages the exchange of cryptographic messages with the client. During each session, the server generates and sends challenges, verifies the client’s responses, and computes session-specific keys for encryption and decryption. At the conclusion of each session, the server securely updates the vault to prepare for future interactions.

The **client.py** file represents the client side of the protocol, simulating the behavior of an IoT device. It establishes a connection with the server, participates in the exchange of authentication messages, and performs cryptographic computations similar to the server. The client processes received challenges, generates appropriate responses, and verifies the server’s authenticity to ensure mutual authentication. Like the server, the client updates its vault at the end of each session, maintaining synchronization and readiness for subsequent communications.

The **experiments.py** file is dedicated to testing the strength of the vault implementation against brute force attacks. It includes simulations where an attacker attempts to guess the vault content either by generating random vaults or by sequentially guessing individual keys. This file demonstrates the infeasibility of such attacks due to the sheer size of the cryptographic key space and highlights the robustness of the vault’s design. The experiments reinforce the security of the implemented protocol by illustrating the impracticality of brute force predictions in realistic scenarios.

To run the simulation, start by initializing the shared vault by executing “python initialize_vault.py --key-size 16” for AES-128 or “python initialize_vault.py --key-size 32” for AES-256. Then, open two terminal windows: one for the server and one for the client. In the first terminal, run “python server.py” to start the server, which will listen for incoming connections. In the second terminal, run “python client.py” to simulate the IoT device initiating communication. As the simulation progresses, the print statements in each terminal will display the step-by-step communication flow, including challenges and responses.

System Setup: simulation flow

The simulation is structured as a client-server model, where the client represents the IoT device and the server acts as the IoT server, as discussed in the reference work. This setup mimics the authentication protocol's real-world functionality by simulating multiple sessions of secure communication.

The simulation is divided into the following sequential steps, representing the logical flow:

1. The first vault initialization is achieved by running “initialize_vault.py” that will generate a file named “vault.pkl”. At the start of each session, both the client and server load the shared cryptographic vault. The vault contains a set of pre-initialized keys, stored persistently in the binary file “vault.pkl”. This vault serves as the foundation for cryptographic operations throughout the protocol.
2. The protocol begins with the client initiating communication with the server over a TCP connection. This is achieved by running server.py and client.py on two different terminals. The client sends a message containing its device ID and the current session ID. This marks the first message (M1) of the protocol.
3. Upon receiving the device ID and session ID, the server generates a random challenge composed of two elements: a set of random indices (C1) and a random value (r1). These indices point to keys within the shared vault, and the random value r1 serves as the challenge's non-repeating component. The server sends this challenge back to the client as message M2.
4. Upon receiving the challenge, the client retrieves the relevant keys from the vault using the provided indices (C1) and XORs them to derive a session-specific key (k1). The client then generates its response, which includes the received random value r1, a new random value (t1), a second set of random indices (C2), and a new random challenge value (r2). This response is encrypted using the session key (k1) and sent back to the server as message M3.
5. When the server receives M3, it decrypts the message using its own derived session key (k1) and verifies the integrity of the received data by comparing the random value r1 with the one it originally sent. If the values match, the server generates its response. The server computes a second session key (k2) using the indices provided in C2, then generates a new random value (t2) and encrypts the response (r2 and t2) with a combined key derived from k2 and t1. This encrypted response is sent back to the client as message M4.
6. The client decrypts M4 using its derived combined key and verifies the random value r2 to confirm the server's authenticity. If the values match, the authentication process is deemed successful, and both the client and server proceed to update their shared cryptographic vault. This is achieved by XORing all vault keys with a combined value

derived from the session keys (k1 and k2). The updated vault is then saved for future sessions.

7. The simulation runs for multiple sessions, repeating the same sequence of operations to ensure that the protocol handles multiple rounds of authentication seamlessly. Each session involves the loading, usage, and updating of the shared vault, as well as the secure exchange and verification of cryptographic challenges and responses.

This general logic flow ensures that both the client and server maintain a synchronized vault state while verifying each other's authenticity. The use of cryptographically secure random values, session-specific keys, and consistent vault updates emulates the behavior of a robust mutual authentication protocol as discussed by the authors of the reference paper.

Experiments

Considering the few experiments conducted by the authors of the reference paper and the nature of our simulation we were not able to replicate the main experiment, the analysis on the device power consumption. So, we decided to conduct some experiments based on the results highlighted in Table 2. The experiments are reported in the file "experiments.py", where the aim was to prove the strength of the vault security considering two different brute force attacks that try to guess the content of the vault. We assume the attacker has some knowledge about the vault implementation such as number of keys and length of keys and he tries to perform a brute force attack to predict the secret keys.

In the first brute force implementation, the attacker generates a random vault for every attempt and compares it against the actual vault. This approach simulates the theoretical process of an attacker exhaustively searching the entire vault space ($256^{\text{key\_size}\times\text{vault\_size}}$). However, given the large number of combinations, this approach becomes computationally infeasible even for relatively small vault configurations (e.g., 4 keys of 16 bytes each). During the experiment, no matches were found even after an extended period of computation, and the method does not terminate due to the massive prediction complexity.

In the second brute force implementation, the attacker guesses the vault content key by key. Starting with the first key, the attacker tests all $256^{\text{key\_size}}$ possible values before moving on to the next key. This strategy reduces the search space by addressing each key individually but still faces exponential growth in prediction complexity as the key size increases. For example, with a key size of 16 bytes, the attacker would need to test 256^{16} combinations for a single key before moving to the next. During our experiments, even after several hours of computation, the attacker was unable to guess the first key of the vault. This highlights the impracticality of brute force attacks under realistic conditions and also considering the content of the vault changes after each session.

Results and Discussion

Both brute force implementations demonstrate the robustness of the vault against prediction attacks. Even with knowledge of the number and length of keys, an attacker is effectively blocked by the size of the search space. The continuous updating of the vault after each session further reduces the feasibility of such predictions, as the vault's state evolves over time, invalidating any previously attempted guesses. These findings prove the strength of the implemented protocol, as highlighted by the authors when discussing security issues.