Program (Basic Concept)

vehicle.py

The Vehicle Class represents a vehicle object with the parameters (passed) of **vehicle_id** and **secret**, both strings. ID is the vehicle's ID, and secret is the secret used by the vehicle internally for the OTP generation

The Vehicle Class uses *generate_otp()* on an instance of itself (a vehicle) to, as expected, generate the One time Password(OTP). The function for this is in otp.py

generate_otp(secret) takes a string as argument **secret** and then:

1. Stores the current Unix time as an integer in variable **timestamp**
2. Stores the **secret** and **timestamp** concatenated and encoded as bytes in variable **otp_input**
3. Stores the result of putting **otp_input** through the sha256 hashing algorithm as a human-readable hex digit only string with *hexdigest()* in variable **otp**
4. Returns the variables **otp** and **timestamp**

The Vehicle Class uses *create_zkp()*, using the **otp** and **timestamp** variables returned by *generate_otp()* as arguments for generating the Zero Knowledge Proof(ZKP). This happens by calling *generate_zkp_proof()*, a function located in zkp.py

***generate_zkp_proof()* is currently aliased as *generate_zkp_proof_simulated()* so that swapping to the real integration will be easier once it's set up in the future**

generate_zkp_proof_simulated(otp, timestamp) takes a string as argument **otp** and an integer as argument **timestamp**, and then:

1. Stores the **otp** and **timestamp** variables, concatenated as a string & encoded to bytes, in variable **combined**

2. Stores the result of putting **combined** through sha256 using *hexdigest()* to make human-readable in variable **proof** to simulate ZKP functionality

3. Returns variable **proof**

rsu.py

The RoadSide Unit (RSU) Class represents an object to be used as an RSU, and initializes with a passed dictionary variable named **vehicle_secrets**, which stores the mappings of a vehicle's ID to its secret.

The RSU Class uses *verify_zkp(),* taking the string parameters of **vehicle_id** and **zkp_proof**, as well as the integer parameter of **timestamp**, to verify the ZKP proof being given by a vehicle.

verify_zkp(vehicle_id, zkp_proof, timestamp) takes the parameters fed into it and then:

1. Stores the secret associated with the given **vehicle_id** (found from the RSU's stored secrets dictionary **vehicle_secrets**) in variable **secret**, storing a value of None if the vehicle is unknown

2. In cases where the secret isn't found, i.e. the vehicle isn't registered i.e. the value of **secret** is None, immediately returns False

3. When the secret is found, the *generate_otp()* function is called with the vehicle's secret to recreate the OTP, ignoring the returned timestamp by storing it in **_unused_timestamp** while storing the recreated OTP in **otp**

4. The function *generate_zkp_proof()* is used by passing the **otp** and **timestamp** variables to it to create the expected ZKP, stored appropriately in **expected_zkp**

5. The **zkp_proof** and **expected_zkp** variables are compared, returning the result of the comparison to represent the validity of the proof (returns True as result of comparison when the variable are the same, which signifies that the proof is valid)

The seeming redundancy in referring to a ZKP proof as well as naming the variable here **zkp_proof when the p in ZKP stands for proof is in place to designate the actual representation of the procured proof from the ZKP process, but it is inelegant at a first glance and I’m open to change on it**

blockchain.py

The Blockchain file uses *simulate_blockchain_verification()*, taking the parameters **vehicle_id** and **zkp_proof**, the integer parameter **timestamp**, and the bool parameter **verification_result**, to simulate invoking a smart contact in order to to verify the ZKP/OTP and log the event.

simulate_blockchain_verification (vehicle_id, zkp-proof, timestamp, verification_result) takes the parameters given to it and then:

1. Encodes the **vehicle_id** in bytes, uses the sha256 hash to anonymize it, then uses *hexdigest()* to get the hash as a hex string, taking only the first 10 characters (as this is only for display purposes) and storing that in variable **anonymized_id**

2. Prints a message using **anonymized_id** to simulate calling a smart contract with an anonymized vehicle id

3. A dictionary called **log_entry**, used to simulate the verification event being recorded and logged, is created with: **vehicle_hash**, the full anonymized hash found in the same way as earlier (except not shortened); **timestamp**, the time of the attempted authentication; and **authenticated**, which tells if the authentication was successful or not

4. The simulated blockchain event log given by **log_entry** is printed to screen

5. **verification_result** is returned