import json

from flask import Flask, request, jsonify, Response

from flask\_restful import abort

from configuration import Configuration

from pollen.attack\_reporting import AttackReporting

from logger import Logger

logger = Logger("BloSS")

app = Flask(\_\_name\_\_)

pollen\_blockchain = None

config = Configuration()

attack\_reporting = AttackReporting(config)

@app.route('/api/v1.0/report', methods=['POST'])

def report():

if not request.json:

abort(400, message="No attack reports provided")

json\_data = None

attack\_reports = []

try:

json\_data = json.loads(request.get\_json(force=True))

for message in json\_data:

attack\_reports.append(attack\_reporting.parse\_attack\_report\_message(message))

except Exception as e:

logger.info(e)

if json\_data is None:

try:

json\_data = request.json

logger.info("[BLOSS/report (second try block) type(json\_date):{}".format(json\_data))

attack\_report = attack\_reporting.parse\_attack\_report\_from\_node(json\_data)

attack\_reports.append(attack\_report)

except Exception as e:

logger.info(e)

logger.info(attack\_reports)

try:

pollen\_blockchain.report\_attackers(attack\_reports)

except:

return "Failed to report attackers to blockchain", 500

return "Successfully reported attackers to blockchain", 201

@app.route('/api/v1.0/set\_blocked', methods=['POST'])

def set\_blocked():

if not request.json:

abort(400, message="No hash provided.")

json\_data = json.loads(request.get\_json(force=True))

try:

pollen\_blockchain.set\_blocked(json\_data['hash'])

except:

return "Failed to mark attack report hash as blocked", 500

return "Successfully marked attack report hash as blocked", 201

import json

import time

from threading import Thread

import requests

import api

from configuration import Configuration

from logger import Logger

from pollen.blockchain import PollenBlockchain

class BloSS:

def \_\_init\_\_(self):

self.\_config = Configuration()

self.\_logger = Logger("BloSS")

self.\_pollen\_blockchain = PollenBlockchain(

enable\_encryption=self.\_config['DATASTORE']['ENCRYPTION']

)

retrieval\_thread = Thread(target=self.\_retrieve\_attackers\_periodically)

retrieval\_thread.daemon = True

retrieval\_thread.start()

self.\_api\_thread = Thread(target=self.\_start\_api)

self.\_api\_thread.daemon = True

self.\_api\_thread.start()

def \_start\_api(self):

api.pollen\_blockchain = self.\_pollen\_blockchain

api.app.run(debug=False, host='0.0.0.0', port=6000)

def \_retrieve\_attackers\_periodically(self):

while True:

try:

attack\_report = self.\_pollen\_blockchain.retrieve\_attackers()

if attack\_report:

requests.post(self.\_config['ENDPOINT']['NODE']

+ "/api/v1.0/report",

json=json.dumps(

json.loads(

str(attack\_report)

)

)

)

self.\_logger.info("[BLOSS]:attack\_report {}".format(attack\_report))

self.\_logger.info("Successfully retrieved {} attackers "

"targeting {}"

.format(len(attack\_report.addresses),

attack\_report.target))

time.sleep(self.\_config['INTERVAL']['RETRIEVE\_SECONDS'])

except Exception as e:

time.sleep(self.\_config['INTERVAL']['RETRIEVE\_SECONDS'])

if \_\_name\_\_ == '\_\_main\_\_':

BloSS()

import json

from datetime import datetime

from logger import Logger

class AttackReportingException(Exception):

def \_\_init\_\_(self, message):

self.message = message

def \_\_str\_\_(self):

return repr(self.message)

class AttackReport(object):

def \_\_init\_\_(self, target, action, timestamp,

subnetwork, addresses, hash=None):

self.\_target = self.target = target

self.\_action = self.action = action

self.\_timestamp = self.timestamp = timestamp

self.\_subnetwork = self.subnetwork = subnetwork

self.\_addresses = self.addresses = addresses

if (hash is None and target is not None

and action is not None

and timestamp is not None):

self.\_calculate\_hash()

else:

self.\_hash = hash

def \_\_str\_\_(self):

dict\_representation = {"target": self.target,

"action": self.action,

"timestamp": self.timestamp,

"subnetwork": self.subnetwork,

"addresses": list(self.addresses),

"hash": self.\_hash}

return json.dumps(dict\_representation)

def \_\_hash\_\_(self):

return self.\_hash

def \_\_eq\_\_(self, other):

try:

return (self.target == other.target and

self.action == other.action and

self.timestamp == other.timestamp and

self.subnetwork == other.subnetwork and

self.addresses == other.addresses)

except AttributeError:

return NotImplemented

def \_\_ne\_\_(self, other):

return not self.\_\_eq\_\_(other)

def \_calculate\_hash(self):

self.\_hash = hash((self.target,

self.action,

self.subnetwork,

self.timestamp))

@property

def target(self):

return self.\_target

@target.setter

def target(self, value):

self.\_target = value

@property

def action(self):

return self.\_action

@action.setter

def action(self, value):

self.\_action = value

@property

def timestamp(self):

return self.\_timestamp

@timestamp.setter

def timestamp(self, value):

self.\_timestamp = value

@property

def subnetwork(self):

return self.\_subnetwork

@subnetwork.setter

def subnetwork(self, value):

self.\_subnetwork = value

@property

def addresses(self):

return self.\_addresses

@addresses.setter

def addresses(self, value):

self.\_addresses = value

class AttackReporting:

def \_\_init\_\_(self, config):

self.\_config = config

self.\_logger = Logger("Pollen")

self.\_last\_report\_timestamp = datetime.now()

self.\_last\_attack\_reports = []

def process(self, attack\_reports):

if not attack\_reports:

raise AttackReportingException('No attack reports provided.')

current\_timestamp = datetime.now()

timespan\_since\_last\_report = (

current\_timestamp - self.\_last\_report\_timestamp

).total\_seconds()

max\_interval = self.\_config['INTERVAL']['MAX\_REPORT\_SECONDS']

min\_interval = self.\_config['INTERVAL']['MIN\_REPORT\_SECONDS']

if timespan\_since\_last\_report > max\_interval:

self.\_last\_attack\_reports = []

if timespan\_since\_last\_report >= min\_interval:

self.\_last\_report\_timestamp = current\_timestamp

filtered\_reports = []

if not self.\_last\_attack\_reports:

self.\_last\_attack\_reports = filtered\_reports = attack\_reports

elif self.\_last\_attack\_reports == attack\_reports:

raise AttackReportingException('Reports already submitted.')

else:

for report in attack\_reports:

exclude = False

for last\_report in self.\_last\_attack\_reports:

if (report.target == last\_report.target and

report.subnetwork == last\_report.subnetwork):

old\_attackers = (report.addresses

& last\_report.addresses)

if old\_attackers == report.addresses:

exclude = True

if not exclude:

filtered\_reports.append(report)

self.\_last\_attack\_reports = attack\_reports

else:

raise AttackReportingException('Reporting frequency too high.')

return filtered\_reports

def parse\_attack\_report\_from\_node(self, message):

target = message['target']

action = message['action']

timestamp = message['timestamp']

subnetwork = str(message['subnetwork'])

addresses = set(message['addresses'])

hash = message['hash']

return AttackReport(target=target,

action=action,

timestamp=timestamp,

subnetwork=subnetwork,

addresses=addresses,

hash=hash)

def parse\_attack\_report\_message(self, message):

message\_keys = ["target", "action", "timestamp",

"subnetwork", "addresses", "hash"]

if any(key not in message for key in message\_keys):

raise AttackReportingException('Attack report message malformed.')

if type(message) != dict:

try:

message = json.loads(message)

except:

raise AttackReportingException('Cannot convert to dictionary')

timestamp\_format = self.\_config['DEFAULT']['TIMESTAMP\_FORMAT']

target = action = timestamp = subnetwork = addresses = hash = None

for key, value in message.iteritems():

if key == "target":

target = value

elif key == "action":

action = value

elif key == "timestamp":

timestamp = datetime.strptime(value, timestamp\_format)

current\_timestamp = datetime.now()

delta\_timestamp\_seconds = (current\_timestamp - timestamp).total\_seconds()

if (delta\_timestamp\_seconds >= self.\_config['INTERVAL']['MESSAGE\_LIFETIME\_SECONDS']):

return None

elif key == "subnetwork":

subnetwork = str(value)

elif key == "addresses":

addresses = set(value)

elif key == "hash":

hash = value

return AttackReport(target=target,

action=action,

timestamp=timestamp.strftime(timestamp\_format),

subnetwork=subnetwork,

addresses=addresses,

hash=hash)

from datetime import datetime

from influxdb import InfluxDBClient

class PollenDatabase:

def \_\_init\_\_(self, config):

self.\_config = config

self.\_client = InfluxDBClient(self.\_config['DATABASE']['HOST'],

self.\_config['DATABASE']['PORT'],

self.\_config['DATABASE']['USER'],

self.\_config['DATABASE']['PASSWORD'],

self.\_config['DATABASE']['NAME'])

self.\_client.drop\_database(self.\_config['DATABASE']['NAME'])

self.\_client.create\_database(self.\_config['DATABASE']['NAME'])

# Only keep data for 1 day

self.\_client.create\_retention\_policy(name="drop",

duration="1d",

replication="1",

database=self.\_config['DATABASE']

['NAME'])

def \_write(self, json\_body):

try:

self.\_client.write\_points(json\_body)

except:

pass

def update\_inbound\_traffic(self, datapath\_id, workload):

json\_body = [

{

"measurement": "inboundtraffic",

"tags":

{

"datapath\_id": datapath\_id

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"mbps": workload

}

}

]

self.\_write(json\_body)

def update\_outbound\_traffic(self, datapath\_id, workload):

json\_body = [

{

"measurement": "outboundtraffic",

"tags":

{

"datapath\_id": datapath\_id

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"mbps": workload

}

}

]

self.\_write(json\_body)

def update\_transit\_traffic(self, datapath\_id, workload):

json\_body = [

{

"measurement": "transittraffic",

"tags":

{

"datapath\_id": datapath\_id

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"mbps": workload

}

}

]

self.\_write(json\_body)

def update\_outbound\_intratraffic(self, host, ip\_address, workload):

json\_body = [

{

"measurement": "outboundintratraffic",

"tags":

{

"ip\_address": ip\_address

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"host": host,

"mbps": workload

}

}

]

self.\_write(json\_body)

def update\_inbound\_intratraffic(self, host, ip\_address, workload):

"Outbound traffic"

json\_body = [

{

"measurement": "inboundintratraffic",

"tags":

{

"ip\_address": ip\_address

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"host": host,

"mbps": workload

}

}

]

self.\_write(json\_body)

def update\_reported\_addresses(self, datapath\_id, count):

json\_body = [

{

"measurement": "reported\_addresses",

"tags":

{

"datapath\_id": datapath\_id

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"count": count

}

}

]

self.\_write(json\_body)

def update\_blocked\_addresses(self, datapath\_id, count):

json\_body = [

{

"measurement": "blocked\_addresses",

"tags":

{

"datapath\_id": datapath\_id

},

"time": datetime.utcnow().strftime('%Y-%m-%dT%H:%M:%SZ'),

"fields":

{

"count": count

}

}

]

self.\_write(json\_body)

import json

import ipfsapi

from ipfsapi.exceptions import ConnectionError, ProtocolError, TimeoutError

from configuration import Configuration

from logger import Logger

class PollenDatastore:

def \_\_init\_\_(self, encryption=None):

self.\_config = Configuration()

self.\_logger = Logger("Pollen")

self.connection = ipfsapi.connect(self.\_config['DATASTORE']['HOST'],

self.\_config['DATASTORE']['PORT'])

self.\_encryption = encryption

def store(self, data, to\_sign=None, serialized\_public\_key=None):

try:

if all([to\_sign, serialized\_public\_key, self.\_encryption]):

data = self.\_encryption.encrypt(to\_sign,

data,

serialized\_public\_key)

self.\_logger.info("Encrypted attack report with hash {}."

.format(to\_sign))

data = json.dumps(data)

return self.connection.add\_bytes(bytes(data))

except (ProtocolError, ConnectionError, TimeoutError) as e:

return ""

def retrieve(self, ipfs\_hash):

try:

data = self.connection.cat(ipfs\_hash)

if self.\_encryption is not None:

try:

encrypted\_payload = json.loads(data)

data = json.loads(

self.\_encryption.decrypt(

encrypted\_payload=encrypted\_payload

)

)

if type(data) == dict:

data['signature'] = encrypted\_payload['signature']

except:

pass

return data

except (ProtocolError, ConnectionError, TimeoutError) as e:

return ""

import base64

import os

from cryptography.exceptions import InvalidSignature

from cryptography.fernet import Fernet

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives.asymmetric import rsa

import paths

from logger import Logger

class PollenEncryptionException(Exception):

def \_\_init\_\_(self, message):

self.message = message

def \_\_str\_\_(self):

return repr(self.message)

class PollenEncryption:

def \_\_init\_\_(self):

self.\_logger = Logger("Pollen")

self.\_private\_key = self.\_load\_private\_key()

if self.\_private\_key is not None:

self.\_public\_key = self.\_private\_key.public\_key()

def \_load\_private\_key(self):

private\_key = None

path = os.path.join(paths.ROOT\_DIR, "private\_key.pem")

if os.path.isfile(path):

try:

with open(path) as key\_file:

private\_key = serialization.load\_pem\_private\_key(

key\_file.read(),

password=None,

backend=default\_backend()

)

except:

self.\_logger.error("Failed to load private key from {}"

.format(path))

else:

private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048,

backend=default\_backend()

)

with open(path, "wb") as key\_file:

key\_file.write(private\_key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.TraditionalOpenSSL,

encryption\_algorithm=serialization.NoEncryption(),

))

return private\_key

def get\_serialized\_public\_key(self):

return self.\_public\_key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

def encrypt(self, to\_sign, data, serialized\_public\_key):

public\_key = serialization.load\_pem\_public\_key(

serialized\_public\_key,

backend=default\_backend()

)

symmetric\_key = Fernet.generate\_key()

encrypted\_symmetric\_key = base64.b64encode(

public\_key.encrypt(

bytes(symmetric\_key),

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

)

cipher = Fernet(symmetric\_key)

encrypted\_data = cipher.encrypt(bytes(data))

signature = base64.b64encode(

self.\_private\_key.sign(

bytes(to\_sign),

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

)

return {"encrypted\_symmetric\_key": encrypted\_symmetric\_key,

"signature": signature,

"encrypted\_data": encrypted\_data}

def decrypt(self, encrypted\_payload):

message\_keys = ["encrypted\_symmetric\_key",

"signature",

"encrypted\_data"]

if any(key not in encrypted\_payload for key in message\_keys):

raise PollenEncryptionException("Unable to decrypt malformed data")

symmetric\_key = self.\_private\_key.decrypt(

base64.b64decode(encrypted\_payload['encrypted\_symmetric\_key']),

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

try:

cipher = Fernet(symmetric\_key)

return cipher.decrypt(bytes(encrypted\_payload['encrypted\_data']))

except:

pass

return encrypted\_payload

@staticmethod

def verify(signed\_hash, signature, serialized\_public\_key):

public\_key = serialization.load\_pem\_public\_key(

serialized\_public\_key,

backend=default\_backend()

)

try:

public\_key.verify(

base64.b64decode(signature),

bytes(signed\_hash),

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

return True

except InvalidSignature as e:

return False

pragma solidity ^0.4.10;

contract AutonomousSystem {

function reportAttackers(string attackReport\_) {}

function getPublicKey() constant returns (string) {}

}

contract Relay {

mapping(string => address) subnetworkMappings;

function getContractAddress(string subnetwork\_) public returns (address) {

return subnetworkMappings[subnetwork\_];

}

function addSubnetwork(string subnetwork\_, address contractAddress\_) {

subnetworkMappings[subnetwork\_] = contractAddress\_;

}

function reportAttackers(string subnetwork\_, string attackReport\_) {

address contractAddress = subnetworkMappings[subnetwork\_];

AutonomousSystem autonomousSystem = AutonomousSystem(contractAddress);

autonomousSystem.reportAttackers(attackReport\_);

}

function getPublicKey(string subnetwork\_) public returns (string) {

address contractAddress = subnetworkMappings[subnetwork\_];

AutonomousSystem autonomousSystem = AutonomousSystem(contractAddress);

return autonomousSystem.getPublicKey();

}

}

import json

import requests

from flask import Flask, request, jsonify

from flask\_restful import abort

from configuration import Configuration

from logger import Logger

from pollen.attack\_reporting import AttackReporting

logger = Logger("Stalk")

app = Flask(\_\_name\_\_)

config = Configuration()

attack\_reporting = AttackReporting(config)

stalk\_controller = None