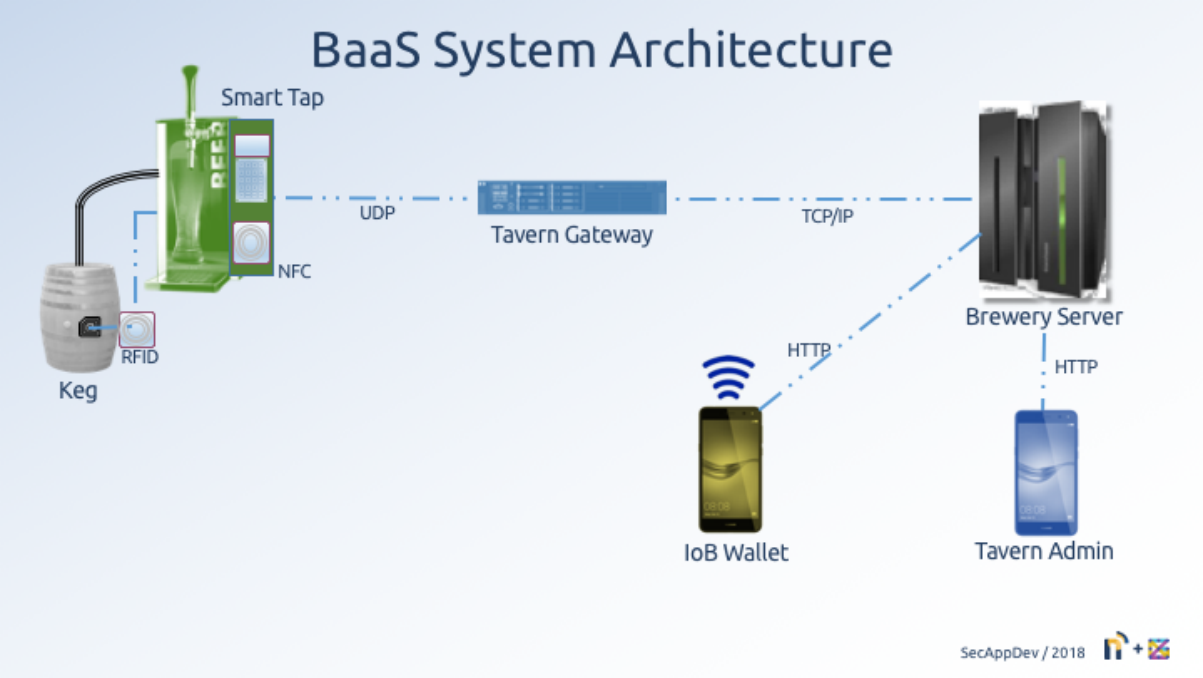
Exercises

Version 1

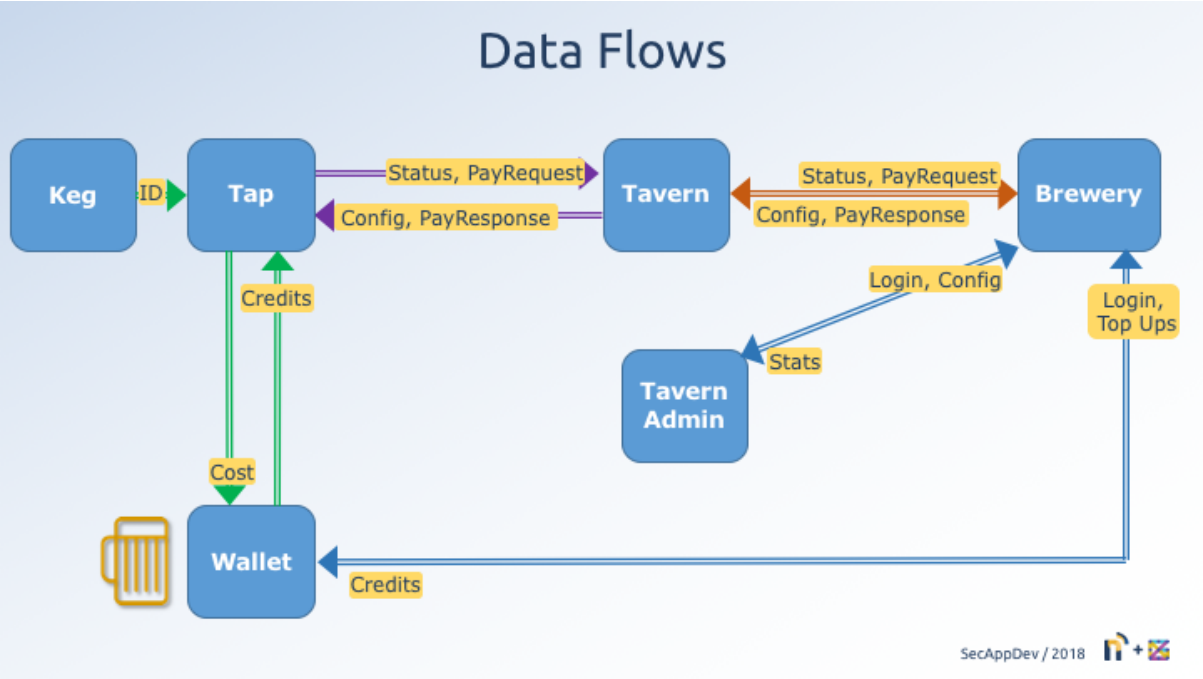
21/02/2018

1. Quick and Dirty Threat Modelling

System Architecture



Data Flows



Threat Agents

Use the following table to annotate threat-agents and their presumed motivations.

Threat Agent	Motivation

Threats

Use the following table to annotate the most relevant potential threats, targets and agents.

Threat	Target	Threat-Agent	Description
T-01			
T-02			
T-03			
T-04			
T-05			
T-06			
T-07			
T-08			
T-09			
T-10			

2. Mapping Secure Services onto Threats

For each the identified threats, determine the secure services, mechanisms and techniques that shall be used as defense against this threat.

Threat	Target	Secure-Services	Description
T-01			
T-02			
T-03			
T-04			
T-05			
T-06			
T-07			
T-08			
T-09			
T-10			

3. Implementation Exercises

During the following implementation exercises, you may use the Linux virtual machine which contains a set of SDKs and IDEs for different languages, along with skeleton code in each language.

Base path: ~/Projects/secappdev-workshop/

Target	Languages	Description
Node.js	Typescript Javascript	./smart-tap/src/user-agent
Android	Java	./smart-tap/user-android/wallet/
Embedded ESP32	C	./tap-esp32-mos/

A demo server shall be used for testing purposes, and it implements a number of HTTP+JSON endpoints. The demo server's IP and port shall be informed at the beginning of the exercises. The skeleton code implements basic async JSON GET/POST functionality that can be used during the exercises.

However, if you already have an IDE or wish to use a command-line approach, a tool called Postman is already installed, and is able to send requests via HTTP as required, as well as build and parse JSON strings.

3.1 Login to server using a sealed datagram construction

Each group will be assigned a different username and given a pre-generated password.

The task is to log-in to the server, creating a new authenticated session, using the libsodium sealed-box construction to protect the credentials. The server has a precomputed password-hash string stored in the database, calculated via libsodium and using current best-practices.

A sealed-box can be created using only the server's public-key, whose value (in hexadecimal) is:

`PublicKey`	`3e2e3ae53d747be56c75f5a969ab4a0b` `e3c2fb7610a7886b934fa9fc8f08426c`

The server login api-endpoint is '/login/<username>', where <username> is the login-name of the desired user. The login is performed in two steps:

1. GET /login/<username>
   This retrieves a JSON containing a not-yet-authenticated ID for the session (sessionID) and a random challenge to be used for the second step.

2. POST /login/<username>
   The second step involves the submission of JSON data containing two attributes, the previously received sessionID and a sealed data envelope containing the protected credentials. Should the server be able to correctly open the sealed envelope, its contents are extracted and used to verify the user's password.

   The data to be protected by the sealed envelope is JSON, before being converted to a UTF-8 byte-string, and should contain the following three attributes:

   ```
   {
       ”userID”:
       ”password”:
       ”challenge”:
   }
   ```

3. Should the submitted userID, password and challenge be correct, the sets the session state to authenticated and responds with JSON. After this, the session can be used for further requests.

Note: the sealed data is a raw buffer and must be converted to BASE64 before submission. The POST JSON should therefore be:

```
{
    “sessionID”: <received sessionID>,
    “sealedBox”: <base64 encoded result from lib-sodium>
}
```

3.2 Login to server using a signed-datagram construction

The task is to log-in to the server, creating a new authenticated session, using the libsodium crypto-sign construction. In this case, the password hashing will be performed by the client application, and the resultant salted “hash” used to derive a keypair. The secret key shall be used to sign the received challenge. The server already has the public-key generated from the transformed password, as well as a salt value which is returned in the initial request.

The same server login api-endpoint shall be used, where <username> is the login-name of the desired user. The login is performed in two steps:

1. GET /login/<username>
   This retrieves a JSON containing a not-yet-authenticated ID for the session (sessionID), a random challenge to be signed, and the salt value to be used for the salted-password-based hashing.

2. POST /login/<username>
   The second step involves the submission of JSON data containing two attributes, the previously received sessionID and a signed data envelope over the userID and the challenge.

The signed data, JSON before being converted to a UTF-8 byte-string, should contain the following two attributes:

```
{
        ”userID”:
        ”challenge”:
}
```

3. Should the submitted userID and challenge be correctly signed, the server considered that as proof that the user has entered the correct password, sets the session state to authenticated and responds with JSON. After this, the session can be used for further requests.

3.3 Send enveloped messages to the server, using a wrapped datagram construction

Continuing on from the signed login, the task is to send an encrypted and authenticated message to the server, using libsodium's public-key envelopes (crypto-box) functions. Firstly, a separate public-key pair shall be generated, in this exercise, this key shall also be derived from the password, as per the previous task, but the salted-hash shall be expanded to encompass the size of both keys.

The same server login api-endpoint shall be used, where <username> is the login-name of the desired user.

1. POST /wallet/<username>
   Submit JSON data with three attributes, the previously authenticated sessionID, a random nonce value and a crypto-box data envelope containing an RPC structure. This structure, before being converted to a UTF-8 byte-string and encapsulated inside the crypto-box, shall be:

```
{
        ”method”: “greeting”,
        ”param”: “Hello”
}
```

2. Should the session be authenticated and server able to correctly open the crypto-box, a return message is created and dispatched. This is a server-generated crypto-box, whose content you should open and display.

3. Note: As the crypto-box function requires a non-repeating nonce, this must be generated and submitted as base-64 to the server in the POST data.