The Cork

SIRS MEIC-A Group 6

Afonso Lopes	Eduardo Claudino	Tomás Pereira
95526	95567	95682

A project made for the course of
Network and Computer Security

Instituto Superior Técnico
Portugal

I. Business Context

In this project, we will be developing a secure implementation of a service denominated **TheCork**. TheCork is a restaurant reservation service that enables coordination between restaurant schedules and an easy to use mobile application for users.

The app allows customers to book tables for a specific number of people if the restaurant they ask for has enough vacancies. Restaurants are able to manage their schedule and to approve and deny customer reservations through a back-office controller. TheCork also offers a website interface with the exact same functionality as the app and also a discount card service integration system for customers to obtain discounts in their restaurant bookings.

For simplification purposes, since the focus of the project is the security measures utilized in the service and not the functionality, the client interface (mobile app/website) will actually just be implemented as a terminal application (CLI). Our system itself will also only handle simple restaurant reservations, the access management between customer and staff and the chosen security challenge.

II. Infrastructure Overview

II.1 Brief Description

For the chosen scenario, we have decided that the system will be separated into 4 Virtual Machines:

- VM-DB: The Database machine, running **MySQL**
- VM-API: The API machine, running a **spring-boot** application on top of **Java**
- VM-Router: The machine that will be acting as a router and **firewall** for the remaining machines, utilizing **UFW**
- VM-Client: The client machine, which will have a CLI application that will connect to the service API through **HTTPS**

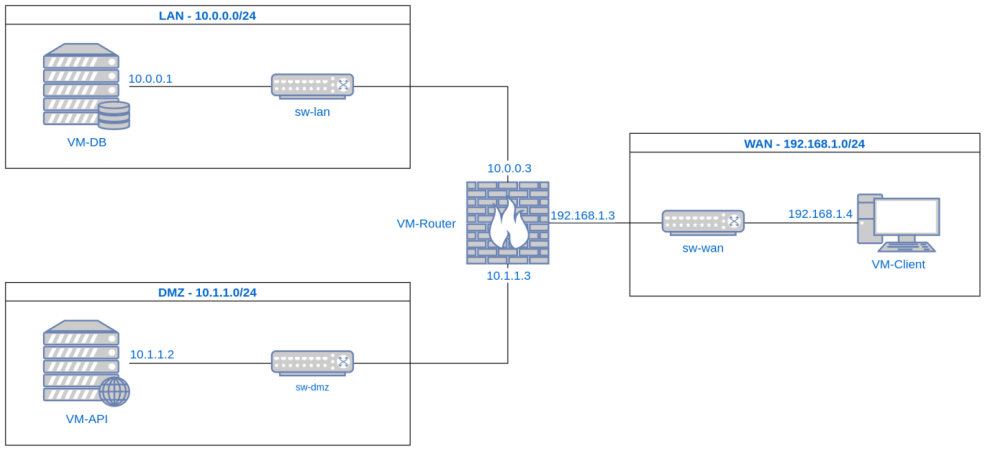
All machines are connected to VM-Router:

- VM-DB is connected through a switch denominated as **sw-lan**;
- VM-API is connected through a switch denominated as **sw-dmz**;
- VM-Client is connected through a switch denominated as **sw-wan**;

This way we can simulate not only real life routing situations but also create the designated sub-nets found in real network infrastructures.

II.2 Network Diagram

We can now define the **network diagram** as such:



II.3 Technology Choices

We can then define the VM network **interfaces** as such:

- ***VM-DB:***
  - `enp0s3`: connected to `sw-lan`
- ***VM-API:***
  - `enp0s3`: connected to `sw-dmz`
- ***VM-Router:***
  - `enp0s3`: connected to `sw-lan`
  - `enp0s8`: connected to `sw-dmz`
  - `enp0s9`: connected to `sw-wan`
  - `enp0s10`: connected to host network
- ***VM-Client:***
  - `enp0s8`: connected to `sw-wan`
  - `enp0s9`: connected to host network

Thus, we can say that VM-Router enables net access to VM-DB and VM-API, while also enabling VM-Client to connect to the service's internal server. It acts as the system's **firewall** with the following logic:

- It forwards all Client requests received on **enp0s9** with destination **192.168.1.3:8443** towards VM-API's internal address: **10.1.1.2**;
- It routes all API requests received on **enp0s8** with destination **10.0.0.1:3306** towards VM-DB's IP, if source IP is **10.1.1.2**;
- It routes all API and DB net access requests towards VM-Router's **NAT** interface, masquerading LAN network addresses for communication with public network addresses;
- Denies all incoming and routing requests otherwise.

These firewalls rules are configured using *Uncomplicated Firewall*, **UFW** for short. The resulting configurations used in this setup result in the following status log:

```
Default: deny (incoming), allow (outgoing), deny (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
10.0.0.1 3306 on enp0s3    ALLOW FWD   10.1.1.2 on enp0s8
Anywhere on enp0s10        ALLOW FWD   Anywhere on enp0s3
Anywhere on enp0s10        ALLOW FWD   Anywhere on enp0s8
10.1.1.2 8443 on enp0s8    ALLOW FWD   Anywhere on enp0s9
Anywhere (v6) on enp0s10   ALLOW FWD   Anywhere (v6) on enp0s3
Anywhere (v6) on enp0s10   ALLOW FWD   Anywhere (v6) on enp0s8
```

To complete the routing logic, three chain rules were added to the nat iptable through the `/etc/ufw/before.rules` file, so that NAT and DNAT logic could be implemented:

- `-A POSTROUTING -s 10.0.0.1 -o enp0s10 -j MASQUERADE` , for DB masquerading.
- `-A POSTROUTING -s 10.1.1.2 -o enp0s10 -j MASQUERADE` , for API masquerading.
- `-A PREROUTING -i enp0s9 -p tcp --dst 192.168.1.3 -j DNAT --to-destination 10.1.1.2:8443`, so that TheCork's API server could be accessed externally by clients through the 192.168.1.3 IP address. This is also known as **port forwarding**.

In terms of technologies used, the **API** will be developed in **Java** and the **Client App** will be developed in **Python**. In terms of communications, the API will implement a **REST** interface for client-server communications, built with the **Spring Boot** framework. For the API to communicate with the DB, the system will use the **JDBC** library.

For the remainder technologies, as said before, the firewall policies will be configured using **UFW** and the Database will be running on a **MySQL** engine.

III. Secure Communications

To secure the connections between our machines, we're employing two different security protocols, one for each pair:

- For the connection between the **Client** and the **API**, we'll be using **HTTPS**.
- For the connection between the **API** and the **DB**, we'll be using **SSL**.

This is due to the fact that the technologies we're using support each of these, respectively.

The API will have a certificate which will be signed by a self-signed root CA. The client will be configured to trust this custom CA. There will also be a certificate present in the database server, to enable MySQL over SSL.

IV Security Challenge

TheCork allows their users to buy Gift Cards, which may vary in cost. Said cards can then be redeemed by the current owner, and the respective funds will be added to the user's wallet. As expected, these cards may only be redeemed once, and the service must ensure that they can't be tampered with: users with malicious intent shouldn't be able to change their value, use them several times, etc, among other things.
Given this fact, we can define the following requirements:

- **R1:** *Gift cards must guarantee **integrity checks**.* All gift cards present in the system must have been at some point created by either an **authenticated user** or the system itself. They must also present preventive measures to avoid **brute-force guessing attacks**, **data manipulation** and overall **theft** of rightful ownership.
- **R2:** *Client side user requests must be **authenticated**.* For any kind of interaction with the API, the client application must first confirm the user's **authenticity** with something only the user should know: their **credentials**.
- **R3:** *The Database's information must be stored with full **integrity** and possible **confidentiality** in mind.* This means passwords, user wallet values and gift card ownership can only be interpreted by the back-end itself. Any attempts at tampering with this data should be perceived as data corruption and handled as such.
- **R4:** *Communications must be provided with complete **confidentiality** in regards to payload.* This is to ensure the privacy of the two most valuable pieces of information exchanged in the system: the **randomly generated nonces** for both the authentication token and gift card code.
- **R5:** *The act of gifting gift cards must be **validated** by the system.* This hinders attempts at data theft and also promotes the avoidance of less secure sharing methods, such as email.

IV.1 Context Assumptions

To have a more focused solution design and due to time constraints, we made the following assumptions and context choices in regards to project functionality and/or extra security requirements:

- Gift card purchasing is entirely done online. Most real world systems take into account the existence of physical gift cards, but to facilitate the interpretation of our security challenge's scenario, the gift cards exist as entirely virtual entities. This means that customers buy, redeem and gift their gift cards through the app interface and those gift cards exist a priori in the database for them to be bought, as to create the effect of limited availability.
  - In theory, gift cards should run out when *Staff* members haven't created any more. However, for testing purposes and app flow, the database automatically restocks its gift card table with one of the same value whenever a gift card is purchased.
- Client application output is based on printing the contents of the response JSON. This is for easier comprehension of app flow, as users in a fully fledged app wouldn't have direct access to things such as their authentication token or status codes.
- It is up to the customer to remember their purchased gift card details. In a fully fledged client app, the purchased gift card's information would remain on screen temporarily or be exported to a file, before having it's id and nonce fully removed from client app state.
- In our project, the DB and API services have certificates issued by different root CA's. This is because the DB keys were generated automatically by a MySQL certification setup kit. In a real world system, all the certificates within the same organization would have the same root CA in their chain.

V Proposed Solution

V.1 Solution Design Introduction

To first define the security context in which our solution will be inserted in, we can say that our solution comes from one strong assumption: the system will never be fully compromised at the LAN level (DB machine). Due to a design choice that promoted easier implementation of the service itself, both the front-end and back-end of TheCork are found in the VM-API machine. The front-end is a simple *Controller-based* HTTPS interface, which forwards the requests directly to the back-end classes, which processes them by interacting with the DB machine. This means that there is a separation of responsibilities between these two, but unfortunately not a separation of context.

In a real world scenario, the back-end segment of the app would have been in the LAN layer of the system, to avoid unnecessary exposure to the outside network and thus adding a layer of security to the overall API system. The front-end would be the only service exposed to the outside network, and would have the job of sanitizing input and pre-processing commands, before sending them to the back-end.

This solution will also have a simplified cryptographic implementation of database integrity. This is achieved by using a symmetric key to encrypt critical data stored in the DB. The key will be stored at the back-end level of the API, and the IVs used in the encryption process will be stored in a DB table. Unfortunately, this means that the compromise of the API layer would also imply access to the cryptographic key used in the database. This solution was found to be unsatisfactory, so we will be explaining the correct procedures that should have been implemented to establish correct **database integrity** in the conclusion.

V.2 Trust Relationships

In terms of **data tampering**, having access to the router machine itself won't give the attacker any bigger advantage, it'll simply open the service to interference and other denial of service attacks, which are out of the scope of this security

challenge. For that reason, our solution will be focusing on reinforcing security in the communication channels and in the data end points: the database and the client.

So, for our scenario, we could say that:

- The **DB** machine is **trusted**.
- The **API** machine is **partially trusted**.
- The **Client** machine is **not trusted**.

Only communications from the client are probable to have malicious intent. However, if tampering is found, it is very possible that the API was compromised. The certification and routing systems in place guarantee that the service's internal network should be safe, and, even if not, the system has limited usability when not connected under an admin account. Our project has its database populated by a script, but it is easy to assume that in a real world context a Staff account or a root DB account could open our system to leaks and/or tampering and destruction of data. For this reason, we will be talking about a correct implementation database integrity later on.

V.3 Attacker Model

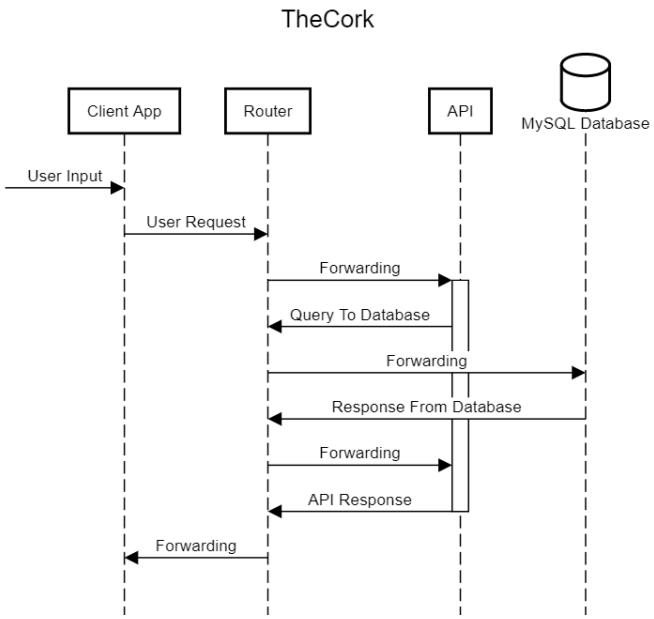
As for an **attacker model**, we can say that our attacker could attempt to:

1) Intercept and tamper with any communication in-between machines: whether to steal or tamper with generated auth tokens, purchase information or database query results;
2) Attempt to login to another user or staff account;
3) Brute-force redeem gift cards with an account;

The attacker has his own account in the service. But, we will also be saying that he has these opportunities to help out in the execution of his attacks:

4) Access to an already open user account in a client machine;
5) Access to the row information of the giftcard table in the database and a way to run UPDATE commands;

V.4 The Secure Protocol



In this solution, the communication is done linearly: The client app communicates with the API front-end, the API front-end sends the received request for processing in the API back-end, the API back-end queries and updates the MySQL Database during operations and then finally the response flows back into the client app. All involved entities possess valid certificates for secure communications, whilst the API back-end possesses a symmetric key for database encryption.

For this system to work, we will have a populated database table with "inactive" gift card codes, meaning that they have no owner. When an authenticated user buys a gift card, the API queries for the first available gift card of the desired value that has no owner, and upon discovery, the gift card found is linked to the user's account, now allowing the user to either redeem it or transfer it. Gift card sharing is through the use of an in-app transfer system, where logged in users can send currently owned gift cards to other valid users, as long as they know the gift card information.
In terms of **integrity**, our gift cards will be utilizing a **sequential ID** combined with a 32 character (128 bit) **nonce**, to prevent users from guessing gift card codes.

Users may **authenticate** into the system through a set of credentials: username and password. Login is verified by **hashing** the concatenation of the input password with the user's respective **salt** and comparing it to the stored SHA256 hash in the adequate table of the database: client for customers and staff for administrators. Password hashes are stored as 64 character long hexadecimal strings, and the salts are stored as 10 character long strings. Upon a successful login, the API creates a 32 character (128 bit) nonce for **authentication token** purposes, updating it into the user's row in the database and sending it to the client app. For testing purposes, the token has a validity of 2 minutes, meaning that afterwards the user is forced to login again.

Finally, as mentioned before, database integrity is complemented through the use of **cryptography** for the most sensitive stored data. This includes the user's wallet value, the giftcard's owner parameter and finally the giftcard's value parameter. The use of varying IVs for all encrypted rows is crucial for maintaining confidentiality between repeated values in the database as well.

In the context of programming libraries used to implement all of these criteria, we used several standard Java and Python libraries: **java.crypto**, **java.sql**, **java.security**, **python requests**, **python sslkeylog**. This is due to them being the more standardized and flexible libraries available, that go well with our chosen frameworks.

So, for a short summary, we can say that the following **security properties** are achieved:

- **Confidentiality**
  - HTTPS between Client and API communications
  - SSL between API and DB communications.
  - Encryption of sensitive database data.
  - Hashing and salting of user passwords.
  - Limited client app interaction possibilities to increase service information confinement.
- **Integrity**
  - HTTPS between Client and API communications
  - SSL between API and DB communications.
  - User authenticity through authentication tokens.
  - Secure randomly generated nonces for both tokens and giftcards.
  - Gifting of gift cards requires valid authentication.

With the full view of the Solution Proposal, we can now analyze the attacker model's effectiveness against our system. In order, we could say that their previously defined attempted attacks and opportunities would be defunct as follows:

1) The connections between client and server are secured using TLS protocol, this way, even if the attacker is able to eavesdrop packets, he won't be able to see or change the contents.

2) User and staff authentication is done using a password, so without knowing the password the attacker won't be able to do any operations. As mentioned before, the attacker isn't able to eavesdrop the password through the communication

channels.

3) Each gift card has a large random value (”nonce”), which is used to guarantee that even if the attacker tries to redeem random gift cards, he won't be able to do it unless he knows the specific nonce of that card. One could say that this nonce acts as the gift card ”password”. Also, each gift card is associated to a single user, so only that user, after authenticating, will be able to redeem or gift that gift card.

4) For usage of the gift card, the attacker would need knowledge of both the card's id and nonce, which isn't saved anywhere in client side app state. Not only that, eventually the authentication token will expire and will force the user to login again, ensuring recency in the session state.

5) Thanks to the cryptographic layer applied to the critical information in the database, there's no way for the attacker to interpret DB contents to gain an advantage. Usage of varying IVs also means that there are no repeating values in the rows and only through the possession of the cryptography key could the attacker decipher the information.

VI Results

After implementation of our proposed solution, we could safely evaluate the success rate of our requirements:

- **R1:** *Satisfied.* The current structural definition of our gift cards and storage methods prevents unauthorized usage of them. This makes them resilient to brute-force attacks and certain SQL injections.
- **R2:** *Satisfied.* The authentication token system proved to be fully functional and working as intended.
- **R3:** *Partially Satisfied.* Although the current implementation succeeds at both password confidentiality and data encryption, due to not having a correct method of storing and retrieving both IVs and cryptographic key, the integrity gained from implementing data encryption does not provide any additional security whatsoever in a real DB compromise situation.
- **R4:** *Satisfied.* All communication channels are secured with modern protocols that guarantee payload confidentiality.
- **R5:** *Satisfied.* Thanks to the minimal client app state, the gift card transfer system executed well at leaving no vulnerabilities in the gifting process.

All implementation choices were done, as explained previously, to maximize system data integrity and confidentiality for a fully online gift card system.

VII Conclusion

Although the solution had a lot of simplifications made to certain aspects of the given scenario, such as reduced scenario functionalities, a simplified client application and having the front-end and back-end be in the same system, the developed solution ended up guaranteeing an almost complete requirement fulfillment quota. As for security, the current working system enables very little room for attacks, and only more extreme cases could prove to be dangerous to the service's functionality, more specifically, full API or DB compromises. For this reason, we can state that the solution's **strong point** are the tightly knit **secure communications**. Due to the fact that our applications keep minimal state and rely heavily on message exchanges, having our transport layer completely secured means that very little tampering can be done from the outside. The libraries used in conjunction with the frameworks also provide a lot of resistance to most common database attacks, such as SQL injections. Finally, the heavily restrictive firewall router prevents most if not all spoofing attacks.

With the goal of a fully secure gift card system in mind, we soon realized that the service would benefit greatly from a more complex and complete infrastructure. As we added more security requirements, we came to understand the advantages and sometimes even need of several **micro-services**. An example of that would be the existence of a **database backup replication server**, which would act as a fail-safe in the case of a main database compromise. We also realized the benefits of an **Authentication Server**, which was part of security challenge (i), and which would greatly increase the robustness and confidentiality of user data. But most important of all, we realized the necessity of a **Key Management Server**.

In our attempt to add integrity to the database, we soon came to realize that the biggest challenge it brought up was key and IV storage. Database level cryptography would be completely nullified the moment the LAN layer was breached, due to the fact that essential components for encryption and decryption could be found in already compromised machines:

- The **cryptographic key** would be recovered from the **API machine**.
- The **IV data** would be recovered from the **DB machine**.

This essentially means that the integrity provided by this cryptography system would be just a small slow down for attackers that had gotten this far. Due to time constraints, we weren't able to rework this functionality of our solution. Instead, to successfully implement the **desired integrity**, we would need a fully functional and trusted **Key Management Server** to act as a mediator for API to DB encryption. The assumption is that the KMS Server is an **outside** but **trusted** and **reliable** service that the API machine could contact.

The KMS would provide the cryptography keys to the API as needed, which the API would discard after using. This would mean that even if the attacker compromised the API machine, he wouldn't find any keys, and the KMS would refuse to supply them.

Essentially, due to the fact that it is highly unlikely that both the Database and the KMS could be compromised simultaneously, the integrity established in the Database would be very effective at diminishing losses in the case of an API or DB compromise.