Identity Enabled Distribution Control System

Universidade de Aveiro

Diogo Silva 60337
Tânia Alves 60340

Identity Enabled Distribution Control System

2nd Project

Segurança

Universidade de Aveiro

Diogo Silva 60337 Tânia Alves 60340

December 29, 2015

Contents

Context

This project was done for Security, for the 2015/2016 lective year. It aims to create an end to end secure digital rights management system to handle the distribution of video files, music files or books.

Introduction

Our project is made of two main components: the player and the server. The server is in charge of controlling the user access to the protected files. The player requests and plays the files from the server. In order for the user to have access to the titles he/she wants, we also have a web application where the user can buy the titles to play later. To reach the goal of this project, we also needed a database that helped manage the user and file related information.

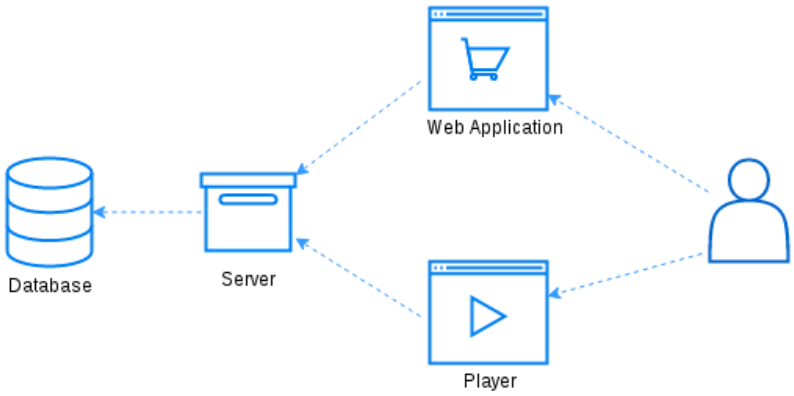

Figure 1: Component overview

Chapter 1

System components

This chapter includes information about the database we used to support the server. We stored information about the users, files, players and devices that were later used.

1.1 Database

For the database we thought of this layout:

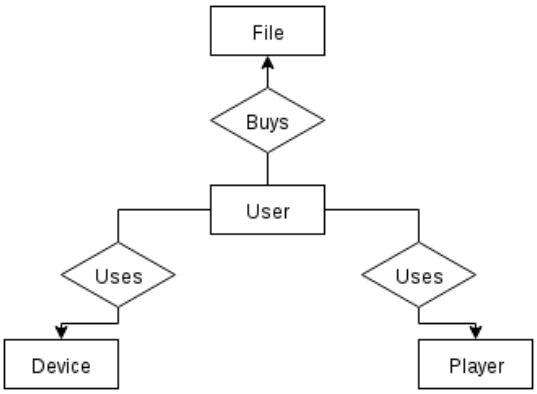

Figure 1.1: DB Schema

We need to save information about the users that belong to the system and buy the files. The files that are stored in the server and are then sent to the players. The players that will interact with the server and play the files. And the devices where the users play the files.

1.1.1 Database tables

Using the information presented, we derived the tables for the database, with all the attributes necessary for our implementation of the system.

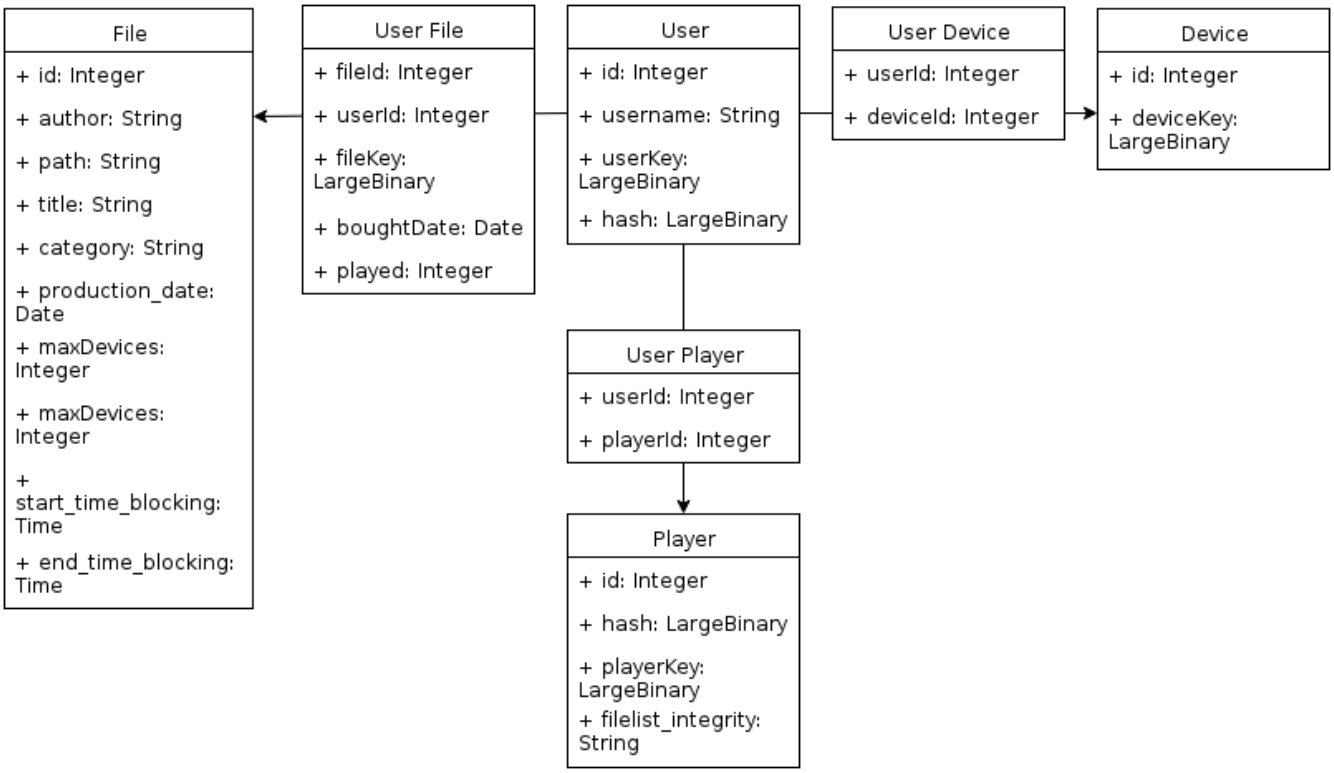

Figure 1.2: DB Tables

Each "main" entities (User, Device, File and Player) will have an id that identifies the entity in the database. We then have the required key for each one of them as well.
In the User table we have added the username field that identifies the each user. For the second part of the project, the username is literally the full name present in the authentication certificate extracted from the user's citizen card. Also, for this part of the project, we added an extra field that holds the hash of the user's authentication certificate.
The File table also has some extra fields that provide information to the user about the file he is playing or buying, such as, the title of the file, the author, the category and the date of the title's production.
The actions presented in Figure 1.1 had to be converted to extra tables since, in each case we had a many-to-many relationship. So, we have here the extra tables that store some information of the interactions, like specific keys and adittional information like the date that when the user bought the file.
Besides the tables presented before, we needed a few more tables to help implement the authorization policies:

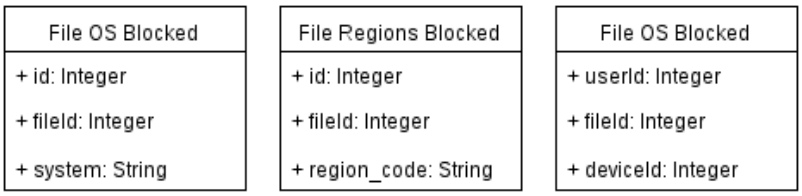

Figure 1.3: DB Auxiliary Tables

1.1.2 Technologies used

For the database we used:

PostgreSQL Open source, object relational database management system. We chose this over SQLite for example because it has a better support for storing secure data.

SQLAlchemy Open source SQL toolkit. Works as an object-relation mapper for Python. This allowed us to create database scripts that created the tables and populated the database.

1.2 Server

The server is meant to interact with the database and control the access of the users and players to the files. This interaction is done over a *HTTP Rest* interface.

1.2.1 Structure

The server is composed of several files:

Server file This is the main file that holds the code to execute the server. When we want to start the server, it must be with *sudo* because the server is using port 443 to host (which requires previligies).

Cipher file This file holds auxiliary cryptographic functions that are used by the server.

Checker file Holds auxiliary functions that make certain verifications before each request to the *HTTP Rest API*, such as making sure the user is logged in before the player tries to download a file, or check if there's a device key reported when player logged in, etc.

1.2.2 Implementation

The server communicates with the player through *SSL*. We did it like this so that we could focus on other parts of the system instead of implementing it from scratch. For this, we needed to create certificates for both server and players.
CherryPy does not run SSL communications natively so, we needed to use Apache to create a proxy for the CherryPy server. The requests are caught by the Apache server and sent to CherryPy to be proccess. The responses to these requests are then sent back to Apache and returned to the client.

1.2.3 Technologies used

Apache Free, open source web server that contains modules for TLS, SSL, proxy, ... Which is exactly what we need to have a fully functional HTTPS proxy.

CherryPy Object oriented web application framework for Python. We chose this because it is built for rapid development of web applications and offers the basic configurations, that is what were looking for.

psycopg2 package This is a Python-PostgreSQL database adapter that manages context, diagnosis errors and more.

OpenSSL Toolkit that implements SSL and TLS protocols with cryptographic support. We used this so that we could create a certain level of abstraction on how the communications were made, and like that, we could focus in the rest of the system.

1.3 Web page

The web page is where the user can buy the titles. It requires the user to log in with the citizen card before proceeding with the titles purchase. The same API that the player uses is used here.

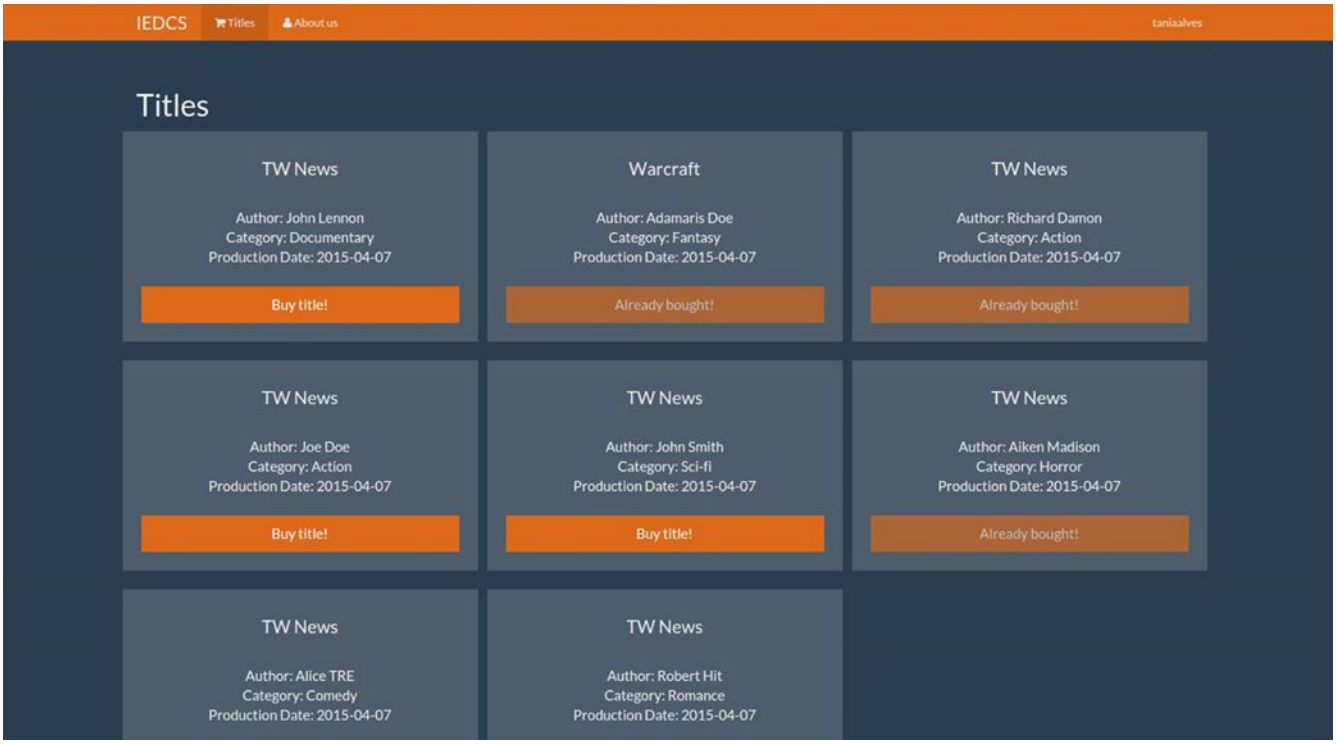

Figure 1.4: Web page interface

It is a simple interface that has a button to buy a specific title.

1.4 Player

The player allows the user to play the files he/she has bought previously through the WebPage.
In this implementation we used video files in order to prove that our implementation can handle big files.

1.4.1 Structure

The player implementation we have is made of 5 main files.

Player file This file contains the mainloop for the player's graphical interface. It is here where most of the requests to the server and main operations are made.

Playback file The playback file contains the code that actually plays the file. The decryption is done here, block by block and fed to the thread that runs a Mplayer2 instance.

My List file This file is an auxiliary file that defines the structure of the list that the player displays to the user containing the titles the users owns. This is only to improve the appeareance of the list.

videos folder This folder is where the player stores the encrypted videos. Inside, there is another folder for each user that uses the player named after his/her username. This method was chosen so that it is easy for each player to go and grab the files that belong to him/her if he/she chooses to move the files and use another player or device.

CC utils file This auxiliary files contains methods that allow interact with the citizen card.

1.4.2 Implementation

The player garantees to the server that the its code was not altered by the user and the server knows how it is implemented. For that the integrity of the player is validated. This validation consists in a exchange of messages between server and player, in section 3.2 the details of this validation are presented with more detail. After the user starts the player application, he/she must log in.

Figure 1.5: Login page for the player

For the user to login, the citizen card must be inserted in the card slot and the user must insert the pin code that corresponds to the authentication action. At this point, besides the user authentication certificate, we also send the Device Key (item 1) to the server alongside the Player certificate (chapter 4).

After the user logs in and it is confirmed by the server, a list of titles is displayed so that the user can choose which one he/she wants to reproduce.

Figure 1.6: Page that lists the files for the user

When the user selects the file that he/she wants to play, a thread running the Mplayer2 is started. The thread reads blocks of the encrypted file, decrypts it and sends it to the Mplayer2 buffer. The player gets the file through a stream instead of a simple *GET* request, so that large files can be played immediately. At this point, several other validations are performed: policies, player certificate validation, player integrity (from before), and Device Key (also from before - This service is referenced in more detail at subsection 3.5.1).
The buffer is managed by the Mplayer2 player and is customizable (the user can give the buffer the size it wants by changing the value on the Mplayer2 settings).

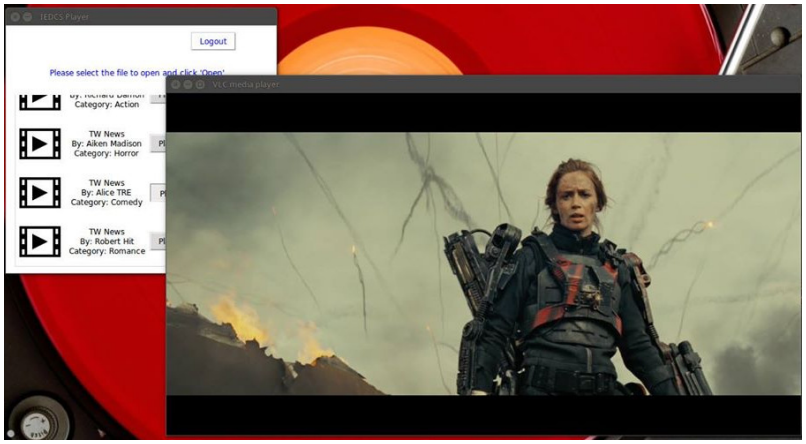

Figure 1.7: Video playing in the Mplayer2 thread

All the requests done to the server are performed over the server's *HTTP Rest* API that was presented earlier.

1.4.3 Technologies used

Tkinter We built the graphical interface with Tkinter that is the Python graphic user interface framework.

Mplayer To make things a bit easier when handling video files, we used the Mplayer2 player that takes care of the video format encoding.

Python requests package This package enabled us to communicate with the server producing requests and receiving responses.

PyCrypto package This is the package that helps handle encryptions and decryptions.

PCKS11 package This package was used to get information from the citizen card.

Chapter 2

System keys

2.1 Player Key

This is a key that must be generated after each code is evaluated by the company that wants to implement the system. This evaluation checks if the player, and the respective code, meet the security requirements. Taking into consideration when it is generated and the process behind it, this key ensures that the player works accordingly to the security policies.
At this stage, the player was generated from a random array of bytes and stored in the database, on the server side, since we are the ones that are devoloping the player. In the player, the Player Key is hardcoded.
This key is present in the server and in the player. Besides that, a certificate for the player is generated so that we can guarantee its authenticity and we can also associate the player that the user is using to the Player Key. As such, there is no way to reproduce files without the player having a valid certificate.

2.2 Device Key

The Device Key is a key that identifies the device where the player is being executed. This key should be the same even if the we have more than one player running in a device. In the Device Key generation, there were some dificulties in finding the best solution.
We wanted to use the processor serial number because we know that the number we get is already unique. However, this is an information that is hard to get. We came across the command *cpuid* that is used in linux systems, but the Python wrapper available, *Pycpuid*, requires an option to be enabled on BIOS to check the CPU UUID (not everyone has it enabled). We also thought about the MAC Address but it was automatically discarded since MAC Address can be modified easily.
We then turned to the command *dmidecode* that gives us a list of device

informations that could be used. Yet again, we found an obstacle. To access the processor serial number, we needed to run the player with *sudo* which goes against one of the principles of security ("No entity should have greater permissions in the system other than basic permissions it needs to perform its tasks"). So, with some research we found that the information the *dmidecode* command provides is stored in some system files, specifically, in the */sys/devices/virtual/dmi/id/* directory. We then browsed all the files present in the directory and came across the one file that didn't require *sudo* to read. This file was the *modalias*, that contained all the information gathered by *dmidecode* that didn't require superuser permissions.
So, for the device key generation, we read the file and hashed the entire content with SHA256, using its digest.
In our solution, this key is generated when the user logs in the player and is sent to the server at this stage. So, the Device Key is present both in the player and in the server.

2.3 User Key

The User Key is a key that identifies the user in the server. In our implementation, this key is generated when the user registers in the web page and is then stored in the database along with the rest of the user information gathered. This key is only ever present in the server, and, as we will see, this will be part of the step that forces the player to communicate with the server each time a file is played.

2.4 File Key

This key identifies the file that the user bought and is independent of the device and player. When compared to the rest of the keys, this one requires a bit more work to derive.

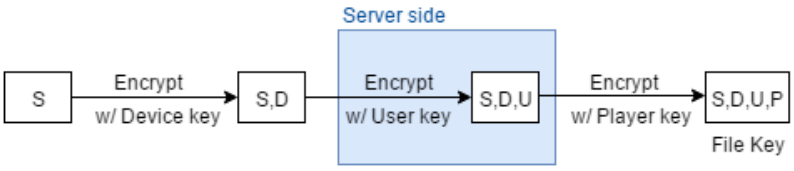

Figure 2.1: Derivation process for the File Key

As we can see from the previous diagram, the File Key is derived from the previous keys presented by encrypting them in a chain of operations. We used the AES algorithm for this.
In the server side, if it is the first file encryption that the server performs for that file and user, we have the following steps:

1. The server generates a random start seed and a random initilization vector since we are going to use CBC to decrypt the file.
2. It then performs the encryption chain presented above to get the resulting File Key.
3. The server then encrypts the file with the result.
4. The encrypted file and the start seed are sent to the player so that it can decrypt the file.
5. The server stores the File Key and the initilization vector (IV) in the database.

If the file has already been sent at least one time, the steps change a bit:

1. The server gets the File Key stored in the database.
2. It then performs the chain of encryption in reverse, decrypting the File Key with the rest of the keys that it has access at that time.
3. The result will be the seed that was used to derive that File Key, and so, it is sent to the player.

This chain allows the player to derive the File Key, even if the device or/and player changes, because the information that the server sends to the player is the starting seed.
In the player side, whatever the File Key is, the operation's chain is the same. However, the blue square present in the diagram above represents a step in which the player must include the server because it does not have access to the User Key.

1. The player gets the seed (and IV used to decrypt file later) from the server and encrypts it with the Device Key.
2. It then sends the result to the server and the server will encrypt it with the User Key, which the player does not have access to (to do this user must be correctly identified, in this case there must be a login associated to the session)
3. The server sends the result back and the player will then encrypt it with the Player Key, resulting in the File Key.

Chapter 3

Server-Client interactions (REST API detailed)

3.1 Summary

All pre-validations of the services are in the *checker.py* file that implements a wrapper that tells CherryPy what are the necessary verifications that it has to perform before each service (eg. @require(logged(), device_key(), is_player(), player_integrity()) for the download. You can find this annotation in the top of the *GET* request for the download operation).

3.1.1 Player - Server

The server-player interaction takes advantage of 7 different types of requests to the server:

1. GET /api/user/loginchallenge allows the player to get a challenge to initiate a session in the server, logging in the user. The server generates a random byte array and returns it to the user to be signed, confirming the user's identity. It requires the portuguese citizen to be able to use this function.

2. POST /api/user/loginchallenge is sent with the following payload in json format: portuguese citizen auhentication certificate in PEM format, the challenge signed with the user's private key, authentication certificate common name and certificate root common name. After the certificate is validated, this information will be associated with the current user session_id, allowing us to keep a record of the login during the whole interaction/session. Besides these parameters, the player also sends the Device Key that it calculated.
   If the user fails to validate the challenge, it will need to request a new challenge.

3. POST /api/user/logout that removes any association in the server to the session_id (this operation requires a previous login, that is, the username must be associated with a session_id before hand).

4. GET /api/title/user/ that lists all the titles that the user purchased earlier (this operation also and only requires that the user is logged).

5. GET /api/valplayer/ that returns a SALT which allows the server to perform the player's integrity validation.

6. POST /api/valplayer/<hash> using the previous returned SALT, the player must send the locally calculated HASH and then the server will confirm if the result is the expected result or not (sort of a challenge).

7. GET /api/title/<pk> receives a stream of the file's content (where <pk> is the title identifier of the file that the user wants to download). This is a critical operation in terms of system security because it can't be used if one of the following conditions fails: Existance of player's certificate, previously performed player integrity, the user is logged in, the player has already sent the Device Key.

8. POST /api/title/validate/<hash> which it is used by the player to send to the server the HASH that it has at that moment. The server will then encrypt the HASH with the User Key and sends the result back to the player (this operation requires the same conditions presented in the previous item).

3.1.2 Webpage - Server

The server-web page interaction is based in 5 different requests to the server:

1. POST /api/user/login where the param is the device key. On the server side, the server will try to obtain the certificate used for the SSL communications that should be the citizen card certificate.

2. POST /api/user/logout that removes any association in the server to the session_id (this operation requires a previous login, that is, the username must be associated with a session_id before hand).

3. GET /api/title/all that lists all the titles available for purchase (does not require login).

4. GET /api/title/user/all that lists all the titles available to the user, with more detail so that the user can purchase or just view them (this requires login).

5. POST /api/title/<pk> where <pk> is the id of the title that will be purchased by the user (this operation also requires previous login).

3.2 Checking player integrity

Every time a player executes, an operation to check its integrity is performed. This falls into the authorization headers, however, we deal with it right off the start.

We get the files that belong to the player implementation and create a hash of each of the files. We then concatenate the hashes of each file and once more hash the result with a random generated salt and we get the final result.

The server generates a SALT and stores it to the session temporarily (SALT is discarded when the routine to validate is called or when requested to generate a new SALT), and sending it to the player. Player will execute the sequence of operations described before to generate the hash. The result will be compared with the information stored in the server, and if they are equal, the player's integrity will be confirmed, otherwise, we assume someone has altered the code and the player is rejected.
Two services are invoked:

1. GET /api/valplayer/ that returns a SALT that allows the server to validate the integrity of the player.

2. POST /api/valplayer/<hash> that submits the result of the challenge (if it fails, the SALT is discarded and a new one must be generated).

3.3 Logging in

To be able to perform any action, the user must first login. This is mandatory in both web page and player.
For this, the server offers a login operation through the *Rest API*

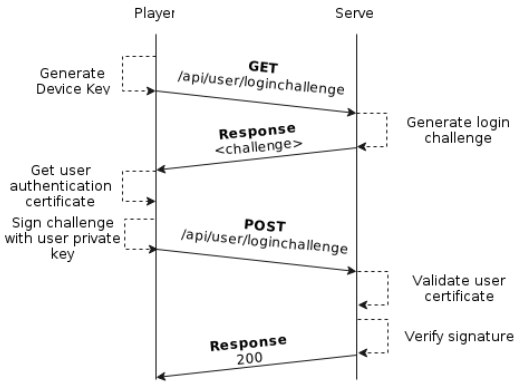

Figure 3.1: Login interaction (player - server)

As we can see from the diagram above:

1. The player starts by generating the Device Key for the device.
2. The player requests a challenge to the server.
3. The player sends the response to the server containing generated Device Key, the user authentication certificate, the certificates common name, the certificate's root common name and the data received earlier from the server signed with the user's private key.
4. The server checks the certificate that the user sent, verifies the signed challenge and accesses the database to verify the information it received associating the Device Key to the user and the player to the user if he used a player to log in.
5. The server sends the response that signals the operation was completed successfully.

The communication between player and server is done using sessions. Before the player starts communicating with the server, a session is created with the *requests* package for Python. From now on, all communications will be performed over this session.

3.4 Purchasing the file on the web page

This operation requires the user to be logged in already and is only available in the web page.

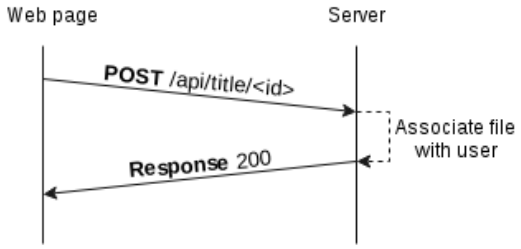

Figure 3.2: Purchase interaction (player - server)

As we can see, the operation isn't complex and the only thing done here is the association of the purchased file with the user. This association translates in a new entry in the UserFile table in the database.

3.5 Downloading the file from the player

In this interaction, we have two distinct situations:

1. Full file download
2. Cryptographic information download

3.5.1 Download requisites (validations)

So that the download of a file is possible, a few criteria must be met:

1. The user must be logged in.
2. The Device Key must be reported to the server on the login operation.
3. The player's authenticity must be validated (with the help of the certificate chapter 4, which will allow the server to associate the Player Key to session).
4. The player's integrity must be validated.
5. The user must have purchased the file previously through the web page.
6. The file for the user must meet all the policies imposed.

The criteria verification has the order presented.

3.5.2 Full file download

The full file download situation means that this is the first time the player downloads the file the user selected. And so, the server must send back the full encrypted file along with the cryptographic information that will allow the player to decrypt the file.
To decrypt/encrypt the key, we used CBC, so, an IV is part of the cryptographic headers as well, even though we used streams to download the file (enabling large files to reproduce as well without the processing time of the encryptino being noticeable). We chose CBC because CherryPy's stream is done over TCP, that means that all data is guaranteed to treach the player. If it was done over UDP, we would probably choose CTR because there is no dependency between blocks in the encryption operation and we have random access to the file.

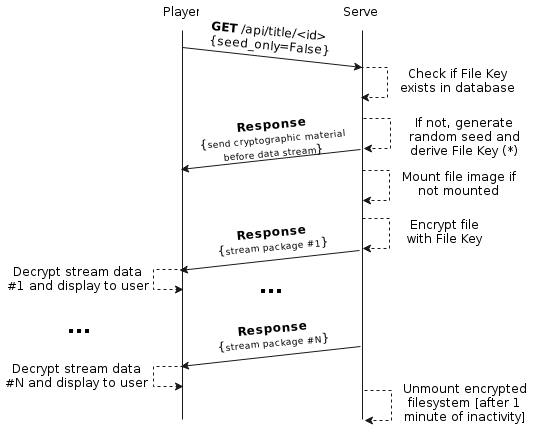

Figure 3.3: Full file download (player - server)

(*) In this operation, there can be yet another two situations:

1. The server has no File Key stored in the database, which forces the server to generate a random start seed to derive the File Key from. It then sends the seed with the encrypted file.

2. The server already has a File Key stored, and it must derive the starting cryptographic information to send to the player, along with the file.

3.5.3 Cryptographic information download

This situation only occurs if the player has already downloaded the file and has it stored on the disk.

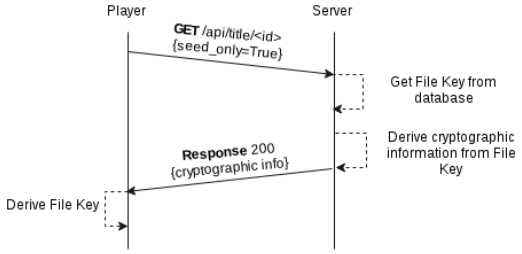

Figure 3.4: Seed only download (player - server)

3.6 Logging out from the player

Logging out is a simple operation that only implies ending the session in the server.

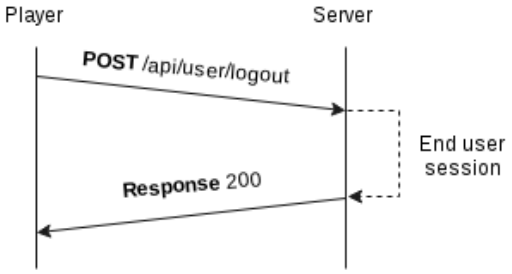

Figure 3.5: Logout interaction (Player/Webpage - Server)

Chapter 4

TLS Socket and Certificates

To guarantee that the communications channel is secure, we decided to the use a TLS Socket because it takes care of the channel encryption in a secure manner. This involves some criteria as well, such as the adequate generation of certificates.

All certificates were generated using XCA. We have generated a Root Certificate Authority (CA) that is the issuer of all the certificates presented in this report and contains the following fields:

1. CN - Security P3G1 Root
2. ST - Aveiro
3. L - Aveiro
4. O - Universidade de Aveiro
5. OU - Departamento de Electrónica, Telecomunicações e Informática

We also thought of the URI to a CRL that isn't being used at the moment, however, can be used later on to revoke certificates.

4.1 Apache Rewrite

The Apache webserver is used to validate the certificates, in our system, for the following paths:

1. /api/user/login that requests a valid citizen card authentication certificate
2. /api/valplayer that requests a valid Player certificate

All the requests validate by Apache are sent to the CherryPy server that will then verify the chain of certificates if necessary. For example, in /api/-valplayer, it verifies if the chain of certificates is in compliance with the RootCA that generated the Player certificates.
If the certificate was extracted from the citizen card, the whole chain is verified and its validation is checked.
If the path we are using is /api/user/loginchallenge, besides checking the validity of the chain of certificates, the signature of the challenge is also verified.

CherryPy is running on port 8888 which is to where the traffic is redirected from Apache to CherryPy. Even though the proxy is running in the same machine as the server, we have decided to use SSL Sockets anyway.

4.2 Citizen Authentication Certificate

We are validating the citizen authentication certificate and using it to verify the signed challenge send by the user/player. To sign the challeng requested to the server, we are using the CITIZEN AUTHENTICATION KEY (the private key) to sign the challenge on the Player side, using the citizen card signing operation. On the server side, the certificate's public key is extracted from the certificate and the signature is verified, besides, of course, validating the chain of certificates.

4.3 Apache Server SSL Certificate

For the CherryPy, a certificate issued by the Root CA had to be generated so that the chain of certificates can be guaranteed to work for any player, logged in user or browser. The CherryPy SSL adapter also had to be altered so that it supported client certificates (if present), for reasons that will be explained later.

4.4 Players SSL Certificates

We chose to generate certificates for each player so that the player validation could be performed and associated with a Player Key. This is the reason why we needed to change the builtin SSL adapter from CherryPy, supporting like this peer validation, since the default doesn't verify anything on the client side.
Being that the certificate is an optional step, if the client chooses not to use them, it cannot download files, nor request the chain of operations to derive the File Key.

4.5 Browser validation

For the browser we specified which CA of our DRM so that the browser can recognize the CA as trusted and trust all its issued certificates.

Chapter 5

Policies

The policy validation is performed when the player tries to download the file (only after ensuring that its all good with the player, this is presented with detail at subsection 3.5.1).

1. Limit the number of devices where the file can be reproduced. This information is stored in an auxiliary table in the database that associates the file with the devices allowing us to keep a track of the number.
2. File only valid for a certain region (the region detection is done based on Remote-Addr available in the HTTP request headers).
3. File only valid for a certain OS (the detection is done based in the User-Agent available in the HTTP headers, filtering the OS information).
4. Limit the time of the day the file can be reproduced (the time calculation is done server side).
5. Limit the number of times the file can be reproduced (there is a counter on the number of file plays).

Chapter 6

Secure file storage

To have a secure file storage for the server media files, we decided to take the approach of the file system level security presented in the practical classes. Even though the attacker will have access to the metadata of the files and the file system structure, this information is mostly available on the WebPage for any person to browse, so, there is no need to use this step.
To achieve this secure storage feature, we used the *encfs* command that creates a mount point to where the files are sent and are then encrypted.
It was our goal to automate this whole process and so, we created a script using *expect* (tool to automate interactive applications such as telnet, ftp,...) that allowed us to simulate the interaction needed to configure *encfs*, that script is inside src/server/database/encfs.exp, it is used to create an encfs for a specific file.
We defined the structure of our file system taking into account that each user should have access what he/she needs, and nothing more.
For this, we divided the files in different folders and create a mount point for each file that we are accessing. Only the needed file is decrypted when the access to it is required, it will mount the specific file using *encfs* and after 1 minute of inactivity it will unmount by itself, allowing multiple users requesting and using the same file at the same time.
The tool *encfs* requires a password to be defined when the configurations are performed. Since each file is encrypted in a separate directory, we decided to gather the information present in the database concerning the file and add a random array that is hardcoded in the server.

Chapter 7

Confinement

Suposing that we want to protect our server/system infrastructure where our server is running from being invaded by unwanted people, we need to create a confinment for our services to run. There are 3 very important services:

1. Apache webserver (that is responsible for authentication the user with the citizen card and serves as proxy for the CherryPy requests)
2. CherryPy (our main web server where all requests are answered)
3. PostgreSQL (our data base that holds information about users, players, devices and files)

For the confinment of the services, we chose to use a Docker container. For this, we developped a Dockerfile that allows to install the whole system (not including the player since there was no need to include a player in the server side)
An important point must be enhanced: the first time that the Docker is created, the database is also created and populated and the files are stored in the .../server/media/ folder. However, these files are also deleted once they are stored in the encrypted file system. Besides the Dockerfile, we also have a script "iecds-server_run.sh" that allows to execute 4 different commands on the Docker instance:

1. start - This command starts a Docker instance in case there isn't one already running
2. stop - Stops the Docker instance if it is already running
3. status - Gives information about the Docker's instance current state
4. reset - Allows to create the Docker instance again, (here, the database is deleted and re-created once again)

Chapter 8

System vulnerabilities review

8.1 SQL Injection

QL Injection is a technique where malicious users can inject SQL commands into a SQL statement via inputs.
Using the developed player, the user can only enter his/her username and pin. Every other operation is performed with clicks, confining the user to the defined choices.
However, having the operations based on a HTTP Rest API, the user can still try to interact with the interface through command line or similar. We will then revisit the details of the REST API used:

1. /api/title/login: This operation is a POST to the server that includes parameters such as username however, no information is inserted into the database.

2. /api/title/logout: Even though this operation is also a POST, there is no information attached to it. It serves only the purpose of telling the server to close the user session.

3. /api/title/user: This is a GET that returns all the titles bought by the user. There are no parameters since the user already has a session at this stage. This represents no harm.

4. /api/valplayer/<hash>: Here, there is a POST request that takes a parameter that is the player hash that once again inserts no information in the database. Also we have a similar GET request (/api/valplayer) that takes no parameters.

5. /api/title/<id>: This a GET request that takes the title id as a parameter. A verification is made here to ensure that the parameter received is indeed an integer. If it is something else other than an integer, the server automatically returns an error code as we can see in

figure XX. We have a similar request to this one that is a POST used by the webpage, enabling the user to buy the specified title.

6. /api/title/validate/<hash>: This POST request takes in a single parameter that is the hash the player derived so far. The only thing done here to the parameter is an encryption and it is then sent back to the player. There is no database interaction directly with the database.

7. /api/title/user/all: This GET request takes no parameters as it only lists all the titles available to purchase.

There is not much room for injection in our API, however, in our case, we used the SQLAlchemy toolkit for the database interactions, as mentioned before. This toolkit automatically escapes the characters used in database queries, ONLY IF the queries aren't performed in raw form. To take advantage of this functionality we used only built in operations to INSERT, MODIFY or DELETE information from the database, never using raw built queries.
However, having the operations based on a HTTP Rest API, the user can still try to interact with the interface through command line or similar. We will then revisit the details of the REST API used: We have some examples of tests we performed to try and explore the vulnerabilities. (This was a simple python script that was changed with different values and ran)

```
security@security:~/Desktop/security-iecds-drm/src/player$ python sqlinj.py
Testing Log in:
        Username: taniaalves
        Server response: 200
security@security:~/Desktop/security-iecds-drm/src/player$ python sqlinj.py
Testing Log in:
        Username: '; create table xpto(id numeric); --
        Server response: 400
security@security:~/Desktop/security-iecds-drm/src/player$ python sqlinj.py
Testing Log in:
        Username: ' or 1 = 1; --
        Server response: 400
security@security:~/Desktop/security-iecds-drm/src/player$ python sqlinj.py
Testing Log in:
        Username: "' or 1 = 1; --
        Server response: 400
security@security:~/Desktop/security-iecds-drm/src/player$ python sqlinj.py
Testing Log in:
        Username: '" or 1 = 1; --
        Server response: 400
security@security:~/Desktop/security-iecds-drm/src/player$
```

Figure 8.1: Login attempt

This was a login attempt. The first result was a successful login, with an existing user, which results in a 200 response code (status OK). In the next attempt, we try to create a table by sending a CREATE SQL command. This produces a 400 status response (BAD REQUEST status), which means that the server did not find the user we tried. In the three next attempts, we try to force the login, however, the response is the same BAD REQUEST status.

```
2015-12-19 16:17:36,234 INFO sqlalchemy.engine.base.Engine SELECT "user".id AS user_id, "user".username AS user_username, "user".userk
ey AS user_userkey
FROM "user"
WHERE "user".username = %(username_1)s
2015-12-19 16:17:36,234 INFO sqlalchemy.engine.base.Engine {'username_1': u'\'; create table xpto (id numeric); --'}
2015-12-19 16:17:36,235 DEBUG sqlalchemy.engine.base.Engine Col ('user_id', 'user_username', 'user_userkey')
127.0.0.1 - - [19/Dec/2015:16:17:36] "POST /api/user/login/ HTTP/1.1" 400 513 "" "python-requests/2.4.3 CPython/2.7.9 Linux/3.16.0-4-6
86-pae"
```

Figure 8.2: SQL Injection in user search

In this figure we have the server information displayed when it tries to search for the user '; create table xpto (id numeric); – . We can see here that the character that finishes the string to start the command is escaped with a by SQLAlchemy (underlined with a small red line), and so the whole command is treated as a simple string.

Figure 8.3: Injecting in the title

In this example, we tried to inject code through the title request, considering that we are already logged in. So, for the login, we had a successful request, however, if the title is not a integer, we receive a 500 (INTERNAL ERROR) status code in the response, as it is trying to convert the string to an integer.

8.2 Buffer Overflow

Attackers can use buffer overflows to corrupt the execution stack of an application by sending custom input to the application, enabling him/her to execute code that can for example open a command shell and allow him/her to take over the machine. This attack relies on write access to particular memory addresses and the system mishandling data types. Essentially every platform is vulnerable to this attack with the exception of:

- Java / J2EE – As long as system calls or native methods are avoided
- .NET – as long as the developer stays away from unsafe or unmanaged code such as using P/Invoke or COM Interop
- PHP, Python, Perl – As long as the developer does not rely on vulnerable extensions or other vulnerable programs.

For this project we used Python alone throughout the functionalities. Python does not allow direct memory access and it is a strongly typed language (the definition for this is a fray area, however, we can that it is because the interpreter keeps track of all the variable types and so, it is the programmer's responsibility to use the variables correctly).
Even though Python is "immune" to such attacks, it settles on a C language base, having its source code written in C, which is a vulnerable language because of allowing direct memory access and not being a strongly typed language. This is a concern because we have found some problems that, having its origin in the C code, propagate to the Python language as well.

We will go through a few problems found along the way in several versions of Python: (https://lwn.net/Alerts/651990/)

- CVE–2014-1912: the function *socket.recvform_ into()*, from the Modules/socketmodule.c module, failed to check the size of the supplied buffer for the receiving socket. This led to a security flaw and was present in Python 2.5 through 2.7.7, Python 3.x before 3.3.4 and Python 3.4.x before 3.4rc1. For this project we used the Python 2.7.9 version, so, being introduced after 2.7.7, the problem was fixed by adding a simple size check of the received information. (https://yazadk.wordpress.com/2014/12/exploitable-buffer-overflow-in-python/)
- CVE-2013-1752: multiple standard library modules implementing network protocols such as httplib and smtplib failed to restrict the sizes of server responses. This CVE actually is made of several issues for httplib, and even though we use requests, these are based on httplib. We also found that all the issues belonging to this CVE were fixed for version 2.7 and 3.1.
- There are other issues that will not be addressed here that were already corrected in Python versions prior to 2.7.9.

Some of these issues, even if not solved in the versions of Python could be solved by recompiling Python's source code with canaries in the C compiler.

8.3 Cross scripting

Chapter 9

Installation

There is an install script available (install.sh) in the root folder of the repository that was done in Ubuntu 15.10 (it was tested inside a VM with 20 GB with default options).

A clone of a VM image was generated, however it is bigger than 4 GB (non zipped), with this being said, we chose not the upload it to the repository. Either way, it can be available to the teacher if problems are encountered.

After installing the whole system, if the next step is the deploy of the system, the apache2 service must be stopped if it is already running in the VM (it was installed only for tests if needed) because the Docker will expose port 443 to the outside world, enabling Docker to receive requests and send responses.

To run the Docker, there is a script iecds-server_run.sh that allows to perform start/stop/status/reset operations like we mentioned in chapter 7.

Chapter 10

Conclusions

There were some problems through out the project, like the Device Key generation, or the player validation using certificates because CherryPy doesn't support client certificate validation on its builtin adaptor (which originatesd the override of the class) on the first part, although it we changed the server on the second part for Apache2 and redirected all the traffic to the CherryPy.

However these problems were solved after some work and the final goal was reached.

Bibliography

[1] Cherrypy streaming (snipplet used to report exceptions and stream). https://cherrypy.readthedocs.org/en/3.2.6/progguide/streaming.html.

[2] Restful cherrypy presentation (based on to implement jsonify answers). https://docs.google.com/presentation/d/1pVtqh99VfmOycus9OoY19jemSYPsXbdia-v2szT7Vs0/edit#slide=id.g21ba316f_0_32.

[3] Simple authentication and access restrictions helpers (used to implement cherrypy tools). http://tools.cherrypy.org/wiki/AuthenticationAndAccessRestrictions.

[4] Xca step by step guides. http://xca.sourceforge.net/xca-14.html.

[5] The Internet Society. Pkcs #7: Cryptographic message syntax. http://www.ietf.org/rfc/rfc2315.txt.

List of Figures