Segurança

Mestrado Integrado em Engenharia de Computadores e Telemática

# Project Report

# Identity Enabled Distribution Control System


Authors

Daniel Silva - 51908

Tiago Oliveira - 51687

Professor

João Paulo Barraca


Aveiro

December 29th, 2015

## Studies and decisions

After reading all the criteria and components to implement on this project we begun searching for a framework to develop our Identity Enabled Distribution Control System.

First we decided to implement the IEDCS on Python. We started using Django Framework, but because we didn't know nothing about Django, the learning curve was somewhat high, we changed to CherryPy for its simplicity. Later we realised that the second phase of the project, especially the smart card implementation, would give us a lot of difficulties if we coded in Python. So we search for a framework to develop in Java. Which led us to Spring Framework. Spring Framework was awesome, but only in the beginning. We were struggling to do basic stuff. So we've decided to start from scratch and went back to Django and not CherryPy because of its limitations. Django offered more in security technologies as shown ahead in this document.

For our IEDCS, we've chosen images (manga illustrations) over the proposed ebooks as our "product to sell". We thought that would be as easy as ebooks (to cipher/decipher, exchange between client and server) and more fun to do it.

For all the libraries that our project would need, we've created a virtual environment and installed packages there, like a container. That gave us the flexibility of managing better all the packages installed without interfering with the main OS. But were some minor disadvantages. For instance, we had some problems using *opencv* and *PIL*, that would give us the possibility of never store any non-ciphered image to the hard drive of the client. That force us to use the imagemagick package, installed outside the virtual environment. With this approach, we had to store momentarily non-ciphered images. Momentarily in this case means that in the process of decrypting a ciphered image, we've to write to the disk to use the "display" command and 0.1 seconds later we delete the image.

On the encryption side, pycrypto and *m2crytpo* were used. We begun using pycrypto because we thought that was the standard library to use and because was the most well documented. Later on we were forced to use m2crypto to extract certificates from the smart card (pteid). Nevertheless, on that process we used both libraries to extract the certificate and then use the certificate to extract the public key (authentication public key from pteid). On the asymmetric keys, we used RSA with PKCS1_OAEP padding, because it's the standard. On the symmetric keys, we used the AES.MODE_CFB. This cipher reduces patterns, doesn't need padding because

it's a continuous cipher and supports random access. On the hashing, we used SHA256 because SHA is the standard digest function for creating hashes and 256 bits because is adequate for the hashes we need.

To create a unique hardware fingerprint, to assist the verification and authentication of the computer used by the client, we used the following formula: the hostname, the size of the RAM, CPU name and most importantly, because it's really unique and of difficult forgery, the serial number of the first disk (hard drive or SSD) on the partition table.

We have chosen Virtual Block Device instead of ENCFS, because is more flexible, more modular and increases security comparatively to ENCFS.

One more problem emerged. How to "hide" the code of the produced application for the client. In Python it's almost impossible to hide any code, even from the ".pyc" files, it's very easy to decompile. But then our Professor told us about Nuitka. Nuitka is a Python compiler that translates the Python into a C++ program. We had a few problems using Nuitka after implementing SSL on the server and on the player because of the libraries. Also we had problems with m2crypto. The solution was to "recurse" less libraries. At the end, Nuitka solved us two problems. Hiding all the code and merging all the Python files developed to the application to a single file, making it more simple to create a "download file" to the client. One file, instead of six.

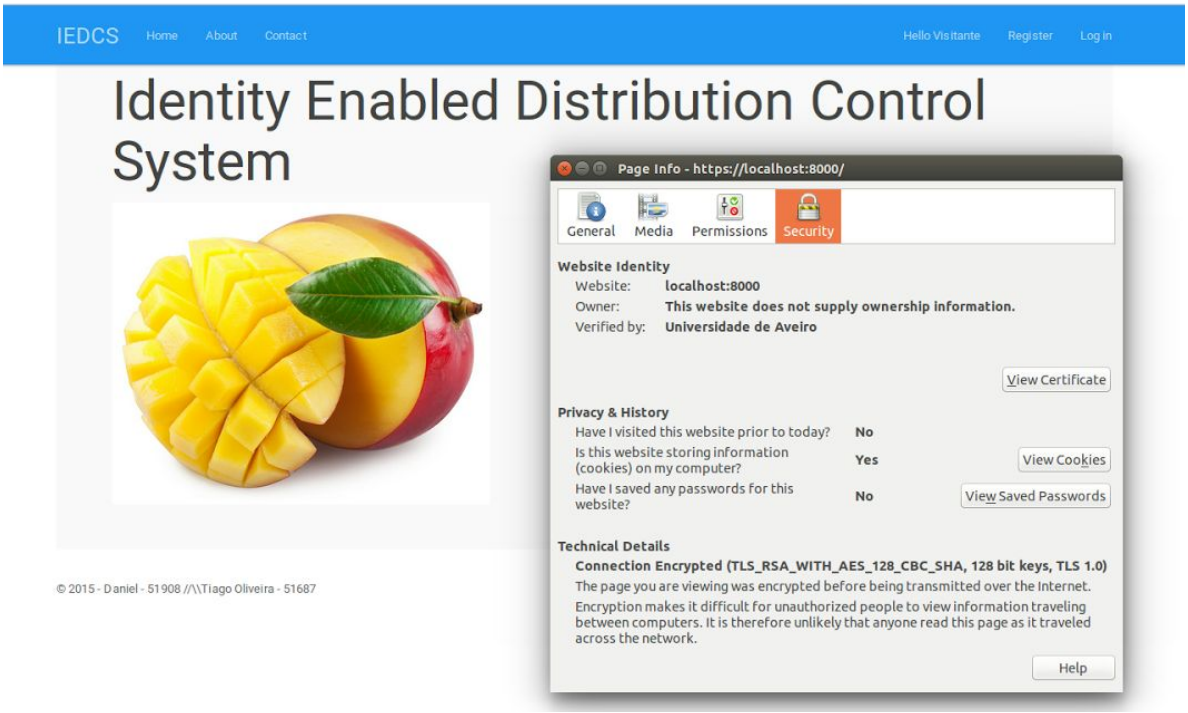
## Functionalities Implemented

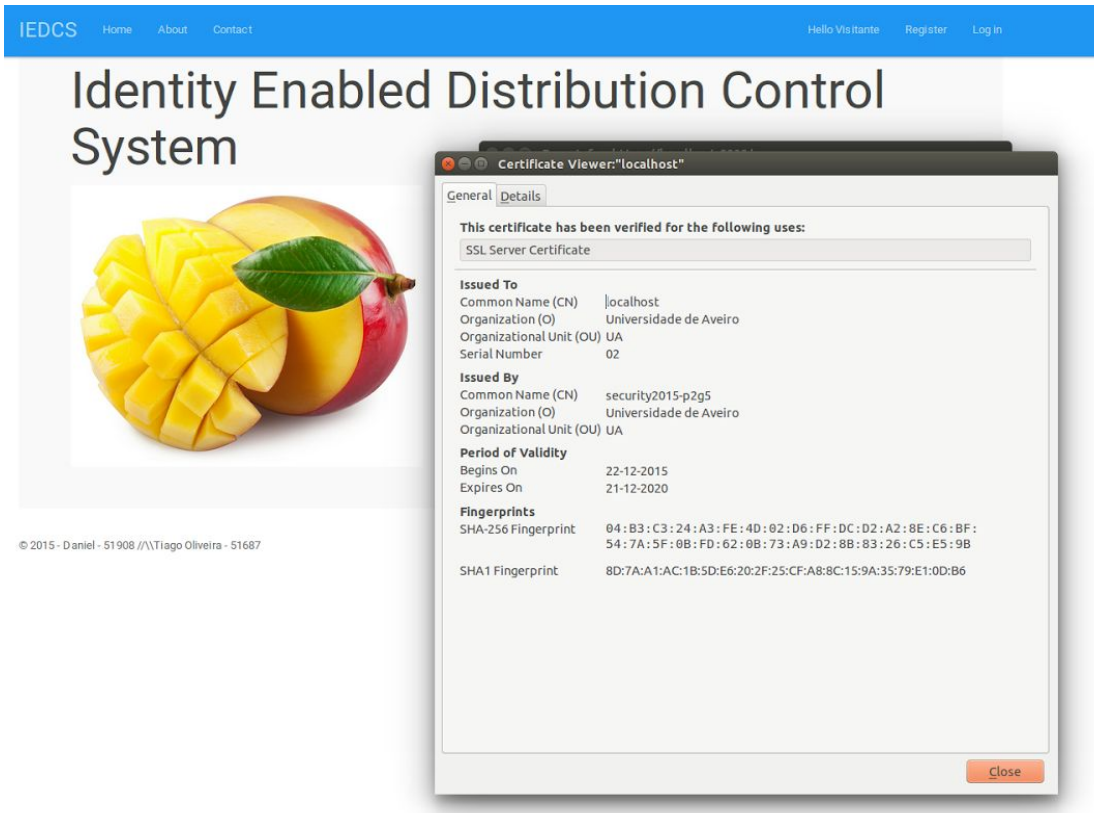
We developed a Player, a Server and a Web Page. The Player is used by the client to access the purchased mangas. The Web Page is used for registering a user, buy content and download the custom made Player for the user. The Server manages all the necessary data, requests from the Player, Web Page and database.

We use a Certificate created by us, to implement a SSL connection between the Web page and the Server as well as between the Player and the Server. All of the communication exchanged by the Web Page/Server and the Player/Server is done using APIs specifically made for each requisite.

## Web Page

For a user to register, he needs a smart card reader and a smart card (pteid). Because it will be necessary to extract the CC number and the authentication public key, to further save this data to the database with all the remaining data filled by the user (password, nickname, email, first name and last name). Then, a user can login and buy mangas and see account information. Below, the images show the certificate in action on the Web Page.





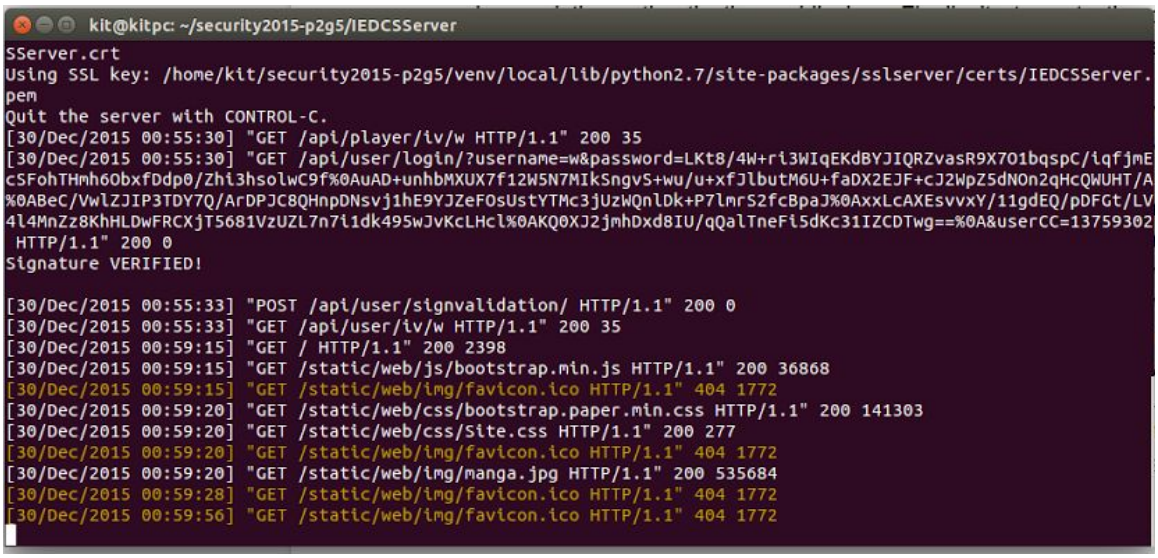

## Server

The Server manages the requests made by the Web Page and Player through a SSL connection. When a user registers in the Web Page, it automatically generates a user key, player key, hashes the password with a salt, retrieves the CC number and the authentication public key. Finally it stores to the database. The player key (asymmetric key) is stored to the database with a passphrase, that encrypts with DES-EDE3-CBC.

When the Player boots up, sends to the Server data signed with the smart card (authentication key from pteid) to further be validated by the Server with the previously public key extracted in the registering process.

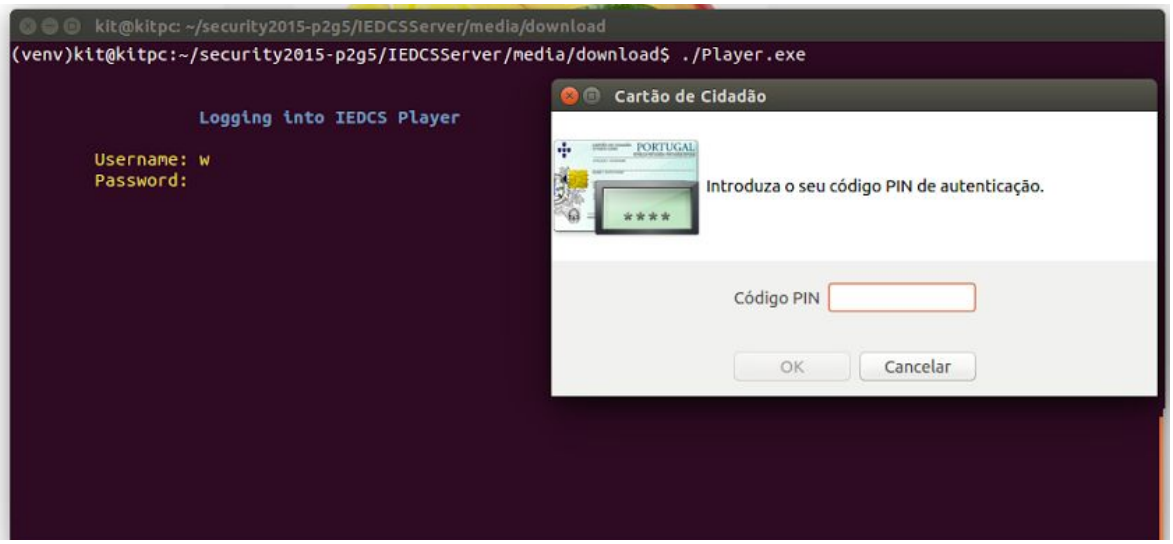
When the Player asks for content to watch, the server will “send” (files in the same area, not really send them) a ciphered file. For each image it creates a file key, ciphers the image with the file key and writes a “magic key” on the beginning of the ciphered file. The “magic key” written on the beginning of the file is ciphered with the device public key. Then the Player will remove the header (“magic key”) and decipher with device key. Then it will cipher it with the player public key and send it to the Server through an API. The server will decipher the “magic key” with public key, verifies if it’s correct and generate the “aux key” with the “magic key”. Then ciphers the “aux key” with device public key and send it as a reply through the same API. Now the Player can fully recreate the file key, decipher the image and display.

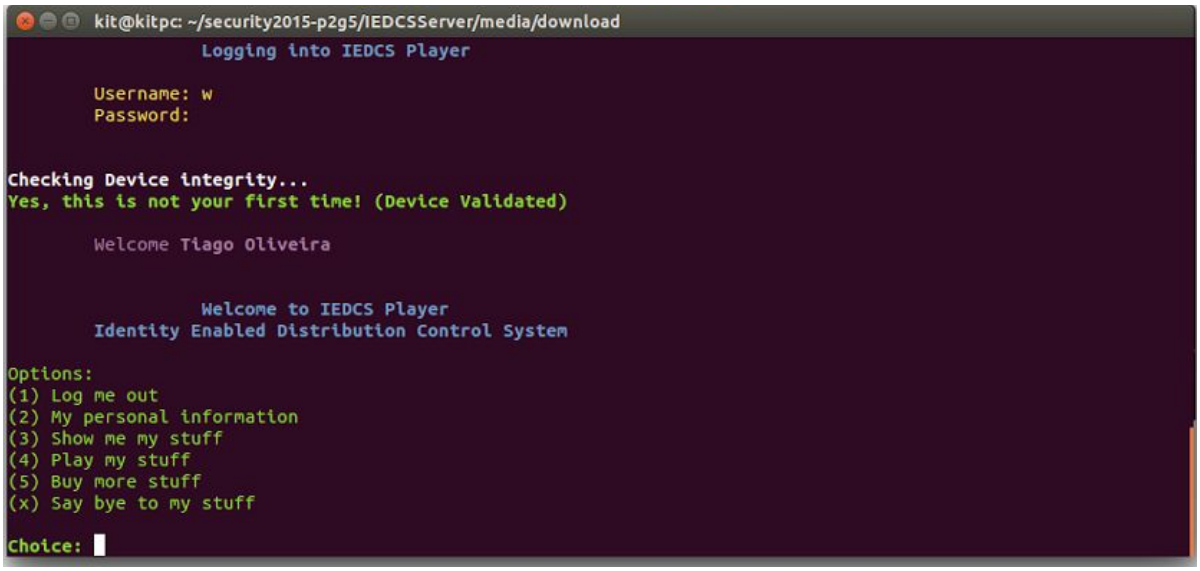
Below an image showing the requests made to the Server, verifying the signature, login from a user, etc.




## Player

The Player will always need the smart card reader and the correct smart card inserted to work. First the user inserts the username and password. Then, on the first run, it will sign some data and send to the Server as explained before. And because of that, it will ask to insert the PIN. If everything checks out, it will be able to decipher the user file (.pkl) and decipher the player public key (.pub). Also, on the first run it will generate the device key, send the device public key to the server. Then it will cipher the device key and write it to a file (.priv). Every action, see account information, see bought mangas, etc, it will be first checked if the smart card is present and if the CC number of that card matches with the CC registered by the user. Below we can see the process of the login on the Player.






## Problems and Deficiencies

Our main problem is the possibility of any user, copying the download URL, change the user id number and use that URL to download a player that belongs to another user. For instance, "/get/download34.zip" where 34 is the user id, if it changes to 33, he will be able to download the Player made for that user 33. But he will need the smart card, username and password of that user to access that Player.

For some reason, a last minute error with Nuikta and M2Crypto force us to send with the Player, the M2Crypto library. Nuitka for some reason, can't "recurse" the M2Crypto and the Player crashes because it cannot found the M2Crypto.

## Security Technologies

### Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input.

### Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.
Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate.

CSRF protection works by checking for a nonce in each POST request. This ensures that a malicious user cannot simply "replay" a form POST to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the nonce, which is user specific (using a cookie).

## SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

By using Django's querysets, the resulting SQL will be properly escaped by the underlying database driver.

## Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

Django contains clickjacking protection in the form of the X-Frame-Options middleware which in a supporting browser can prevent a site from being rendered inside a frame.

## Session security

Subdomains within a site are able to set cookies on the client for the whole domain. This makes session fixation possible if cookies are permitted from subdomains not controlled by trusted users.

## SESSION_COOKIE_SECURE

Whether to use a secure cookie for the session cookie, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection.

Since it's trivial for a packet sniffer (e.g. Firesheep) to hijack a user's session if the session cookie is sent unencrypted. It will prevent you from using sessions on insecure requests.

## CSRF_COOKIE_SECURE

Same as Session Cookie Secure but for the CSRF cookie. The cookie will be marked as “secure,” which means browsers may ensure that the cookie is only sent with an HTTPS connection.

## SESSION_COOKIE_HTTPONLY

Whether to use HTTPOnly flag on the session cookie. If this is set to True, client-side JavaScript will not to be able to access the session cookie.

HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the RFC 2109 standard for cookies, and it isn’t honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.

This makes it less trivial for an attacker to escalate a cross-site scripting vulnerability into full hijacking of a user’s session.

## SECURE_SSL_REDIRECT

Redirect all HTTP connections to HTTPS.

## HTTP Strict Transport Security (HSTS)

For sites that should only be accessed over HTTPS, you can instruct modern browsers to refuse to connect to your domain name via an insecure connection (for a given period of time).

This reduces your exposure to some SSL-stripping man-in-the-middle (MITM) attacks.

## Host header validation

Django uses the Host header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake Host value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails.

Because even seemingly-secure web server configurations are susceptible to fake Host headers, Django validates Host headers against the ALLOWED_HOSTS setting in the django.http.HttpRequest.get_host() method.

This validation only applies via get_host(); if your code accesses the Host header directly from request.META you are bypassing this security protection.

## SSL/HTTPS

It is always better for security to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases – active network attackers – to alter data that is sent in either direction.

We used this github project to guide us.

## ALLOWED_HOSTS

A list of strings representing the host/domain names that this Django site can serve. This is a security measure to prevent an attacker from poisoning caches and triggering password reset emails with links to malicious hosts by submitting requests with a fake HTTP Host header, which is possible even under many seemingly-safe web server configurations.

## Password validation for users

We used Django forms to do this and to filter and force password to get a better and stronger password given by the user. For this it was inserted a regex filter on password field form like this: **((?=.*[0-9])(?=.*[a-z])(?=.*[A-Z]).{8,20})**

**Where:**

- **(?=.*[0-9])** - must contains one digit from 0-9
- **(?=.*[a-z])** - must contains one lowercase characters
- **(?=.*[A-Z])** - must contains one uppercase characters
- **.** - match anything with previous condition checking
- **{8,20}** - length at least 8 characters and maximum of 20

## Virtual Block Device

Block devices are devices such as flash drives and hard disks, which export a block-based interface.

The purpose of a virtual block device is to map a standard file as a block device. An interesting advantage of this method is that the software mapping the file can cipher information as it is written to the loopback device.

**First it is required to create a file to be mapped (in this case with 100MB):**

- dd if=/dev/zero/ of=/path/to/secure-file bs=1M count=100

**Map the file as a block device, and set that all operations should be ciphered/deciphered:**

- sudo cryptsetup luksFormat /path/to/secure-file
    - passphrase: iedcs
- sudo cryptsetup luksOpen /path/to/secure-file secret-data

**As with all block devices, a file system must be created, and then mounted:**

- sudo mkfs.ext3 /dev/mapper/secret-data
- mkdir -p /path/to/storage-mount
- sudo mount /dev/mapper/secret-data /path/to/storage-mount

**To unmount the file system just do:**

- sudo umount /opt/storage-mount

**Now we can destroy the mapping attributed to the /dev/mapper/secret-data node by issuing:**

- sudo cryptsetup luksClose secret-data

After this step, the file is free from any binding and can be transported to another system, keeping all information secure.

# References


[XSS]
https://docs.djangoproject.com/en/1.8/topics/security/#cross-site-scripting-xss-protection

[CSRF]
https://docs.djangoproject.com/en/1.8/topics/security/#cross-site-request-forgery-csrf-protection

[SQL Injection]
https://docs.djangoproject.com/en/1.8/topics/security/#sql-injection-protection

[Clickjacking protection]
https://docs.djangoproject.com/en/1.8/topics/security/#clickjacking-protection

[Session Security]
https://docs.djangoproject.com/en/1.8/topics/security/#session-security

[Session Cookie Secure]
https://docs.djangoproject.com/en/1.8/ref/settings/#session-cookie-secure

[CSRF Cookie Secure]
https://docs.djangoproject.com/en/1.8/ref/settings/#csrf-cookie-secure

[Session Cookie HTTP only]
https://docs.djangoproject.com/en/1.8/ref/settings/#std:setting-SESSION_COOKIE_HTTPONLY

[Secure SSL Redirect]
https://docs.djangoproject.com/en/1.8/ref/settings/#std:setting-SECURE_SSL_REDIRECT

[HTTP Strict Transport Security]
https://docs.djangoproject.com/en/1.8/ref/middleware/#http-strict-transport-security

[Host Header Validation]
https://docs.djangoproject.com/en/1.8/topics/security/#host-header-validation

[SSL/HTTPS]
https://docs.djangoproject.com/en/1.8/topics/security/#ssl-https

[Allowed Hosts]
https://docs.djangoproject.com/en/1.8/ref/settings/#std:setting-ALLOWED_HOSTS

[Password Validation]
http://www.mkyong.com/regular-expressions/how-to-validate-password-with-regular-expression/

[Virtual Block Device]
http://www.joaobarraca.com/page/teaching/security/2015/p/11/e11-secure-fs.pdf