LINK: <https://github.com/dmclange/csbp1>

FLAW 1: Broken Authentication

Broken authentication is the security flaw where an unauthorized user can gain access to an authorized user account. This can be achieved through a number of different ways such as but not limited to: credential stuffing, the use of known username/password lists; weak or wellknown passwords, such as admin/admin or Password1; missing or ineffective multi-factor authentication; Session IDs are not invalidated.

This can be mitigated by not shipping or deploying with admin credentials, limiting or delaying repeated login attempts implementing multi-factor authentication. Log authentication failures and alert administrators when attacks are detected.

The flaw I chose to include was generating a SQLite database with the known username/password combination of admin/admin. With this flaw the attacker gains access to all information stored application. (Additionally should this site store for example medical data and/or credit card data this is also a sensitive data exposure.)

The fix for this flaw is to delete the current database, generate a new database and a new superuser which ideally would not be named admin.

FLAW 2: Cross-Site Scripting (XSS)

Cross-Sitte Scripting is the flaw where an attacker gets the application to execute code not native to the application. The three forms of this are Reflected XSS, where the attacker gives the victim a malicious url and if the victim visits the site malicious code is executed. Second is Stored XSS where the malicious code is stored on the server and does not require the victim to visit a malicious url in the application. Finally there is DOM XSS where the application uses for example unsafe javascript frameworks.

This flaw can be mitigated by using frameworks that automatically escape XSS by design, manually escaping untrusted HTTP request data and applying context-sensitive encoding.

The flaw I chose to introduce is a simple proof-of-concept piece of code stored in polls/templates/polls/search.html where a reflected XSS attack is possible.

To fix this issue simply remove the search.html and remove the line of code pointing to it in polls/urls.py.

FLAW 3: Security Misconfiguration

Security Misconfiguration is the flaw where an otherwise possibly secure application is by either the administrator or programmer is rendered insecure through human intervention. Such an attack vector may be present through for example not removing sample data from a production system, default accounts, unused files/features and many more.

To mitigate this kind of flaw a repeatable hardening process working on a minimal platform that is periodically checked by an automated process to verify the effectivity would be ideal. The mitigation strategy may also be called principle of least privilege when speaking of operating systems.

The flaw I chose to implement is leaving DEBUG options turned on. This allows an attacker to receive overly informative error messages and stack traces. Such a flaw can then be utilized to system data, functionality or complete system compromise.

To patch this flaw simply set DEBUG to false and allowed hosts to 127.0.0.1.

FLAW 4: Using Components with Known Vulnerabilities

Using Components with Known Vulnerabilities is the flaw where, while the application itself may be secured the required components for it have unpatched known security vulnerabilities.

One of the most well-known examples of this is OpenSSL's Heartbleed bug which requires the library to be version 1.0.1g or above to patch this vulnerability.

This is the flaw I chose to include. The user installs the application on a long unpatched system that has a vulnerable OpenSSL library and thus is vulnerable to Sensitive Data Exposure (which is detailed below). To fix this issue implement a patching policy and don't run Internet facing machines with obsolete software.

FLAW 5: Sensitive Data Exposure

Sensitive data exposure is the security flaw where data, that is required to be kept secret, is exposed to an attacker. This can either be data required by law to be kept secret, such as personal information, credit card numbers or health records or sensitive business data. Some of the laws that specify how certain kinds of data are required to be protected are the EU's General Data Protection Act (GDPR), the United States' Health Insurance Portability and Accountability Act (HIPAA) and the Payment Card Industry Data Security Standard (PCI-DSS).

This flaw can be exploited by, for example, transmitting data in clear text, old or weak cryptographic algorithms or default cryptographic keys.

Such a flaw can be mitigated by, for example, properly classifying data according to their required privacy laws, regulatory requirements or business needs, not storing sensitive data unnecessarily. When data needs to be stored encrypt it at rest with up-to-date and strong cryptographic algorithms and perhaps most importantly independently verify the effectiveness and settings.

The flaw I chose to include is sharing the SECRET\_KEY in a public repository thereby rendering it unsuitable as it is used as a seed for hashing and signing sessions. While the app as-is does not contain any sensitive data, using it for example as a business internal questionaire the user could add such.

To fix this issue the SECRET\_KEY in the settings needs to be changed. To do this you could generate a second project and simply copy the new key from that project over however the django project recommends not storing it in the settings file and keeping it instead in the environment variables.