Online Retailer Design

Introduction and Background

In an era of rapid digital transformation, online retailers have become integral to modern commerce, providing a platform for businesses to reach global audiences with ease. This project aims to develop a secure online retailer application using Python to enhance the user experience and the security mechanisms. We aim to create a command-line interface where a user can create an account, login, and input their card details in order to purchase a product. The login and card identifier are stored in a database, which is only accessible via an admin account.

Rationale

The rationale behind selecting this use-case is that the context of an online retailer interface is ubiquitous within day-to-day life.

Project Assumptions

There will be two types of users: normal users (customers) and administrator users. Administrators will have full access.

System requirements and Specifications

1. Functional Requirements

a. Users:

i. Account creation/update/deletion

ii. Credit card info. addition/update/removal

iii. Product purchase

b. Admins:

i. Account creation/update/deletion

ii. Ability to promote a user account to an admin account.

iii. Product addition/update/view/removal

iv. Ability to view user details.

2. Tools and Technologies

a. Programming Language: Python

b. Version Control: Git

c. IDE: VS Code, PyCharm etc.

d. Linters and Static Analysis: PyLint

3. Libraries for encryption and testing

a. Encryption: hashlib

b. Testing: Pytest

Methodology

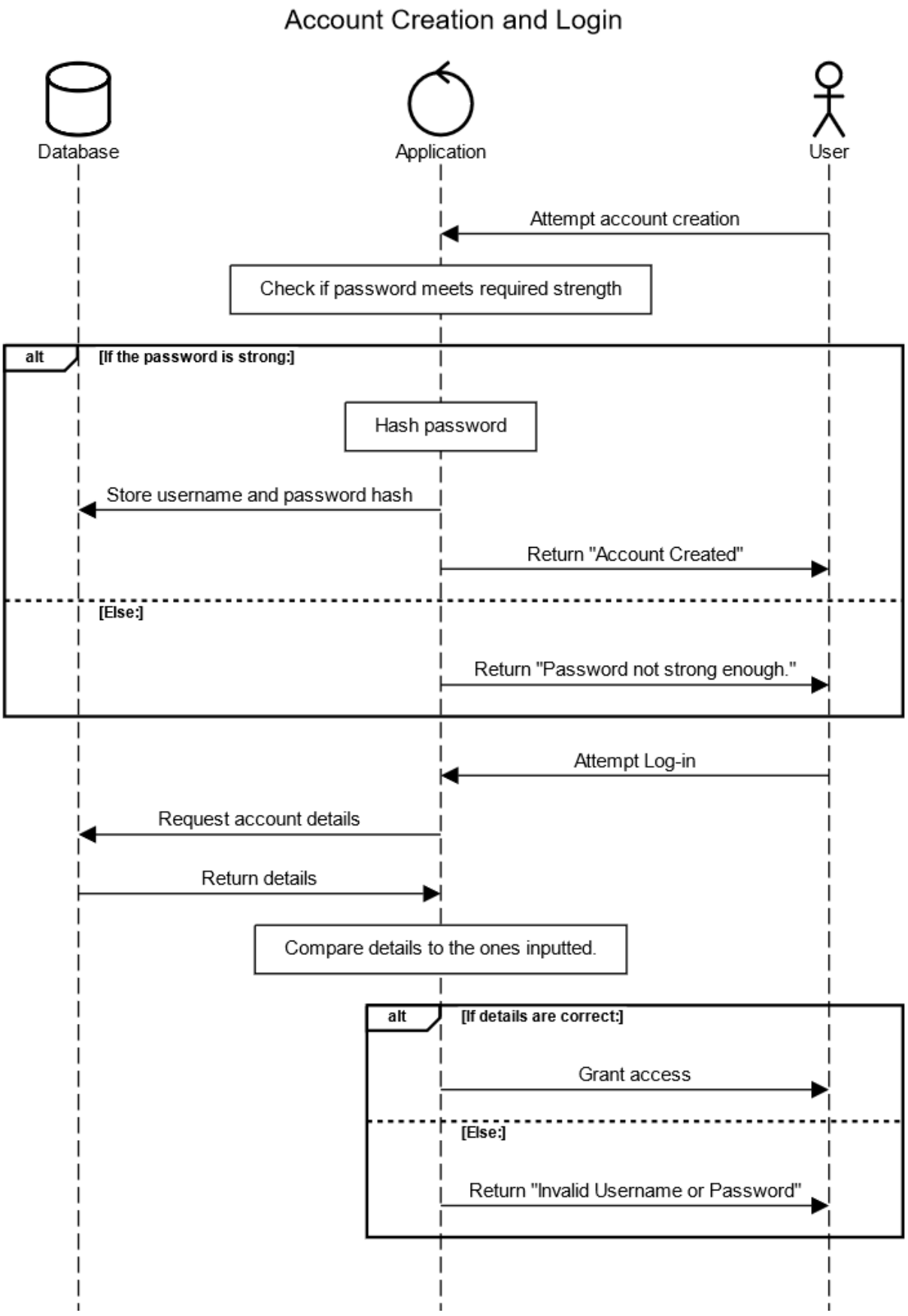
The chosen methodology is the waterfall model, due to its linear nature (Lutkevich & Lewis, N.D.). As the requirements are set in stone and clearly outlined, iteration on them is not required, thus a model such as a waterfall is ideal.

Security Requirements

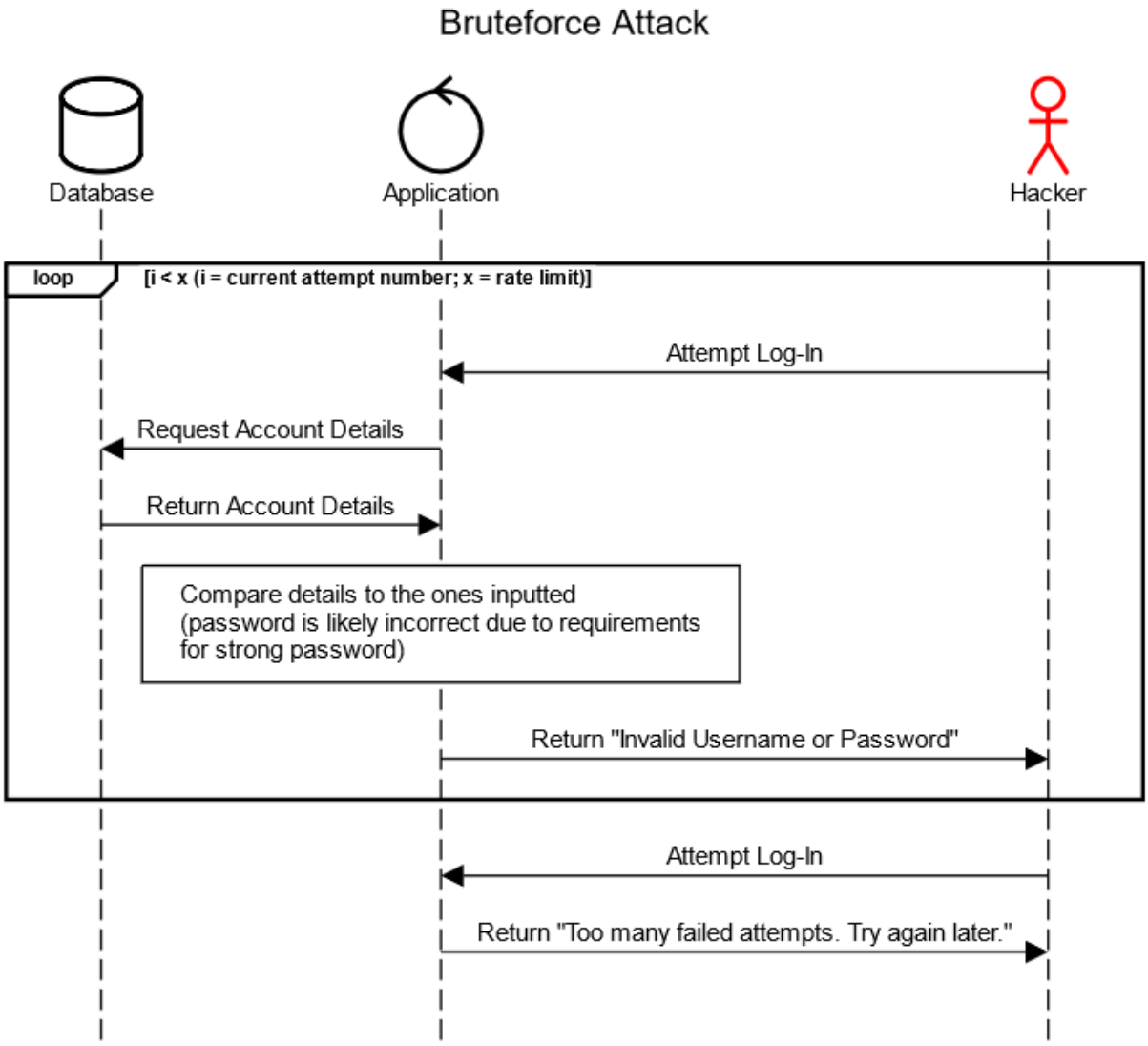
1. Refrain from storing passwords. Instead, maintain hashes and perform comparisons during the login process.
2. Mandate the use of robust passwords for both users and administrators.
3. Encrypt card details prior to their storage.
4. Protect against Brute Force attacks by requiring strong passwords and rate limiting failed login attempts.
5. Prevent denial of service attack by rate limiting the API, API injections by sanitizing user-provided input, buffer overflow by using tested external libraries as Python is already fairly secured against this.
6. Add logs for significant events in the application e.g. Password change, Successful and failed logins, Successful purchases, etc.

UML Diagrams

Log-In intended use-case example:


Account Creation and Login
Database
Application
User
Attempt account creation
Check if password meets required strength
alt
[If the password is strong:]
Hash password
Store username and password hash
Return "Account Created"
[Else:]
Return "Password not strong enough."
Attempt Log-in
Request account details
Return details
Compare details to the ones inputted.
alt
[If details are correct:]
Grant access
[Else:]
Return "Invalid Username or Password"

Log-In misuse case example:



OWASP 10 Proactive Controls

The OWASP Foundation (N.D.) has identified the “Top 10 Web Application Security Risks” of 2021 along with the relevant mitigations. In the following table, we list these risks and the mitigations that apply to our application. We also refer to the “10 proactive controls” that OWASP (N.D.) has identified that should be included in every project as they relate to the risks and mitigations.

Risk	Mitigation	Proactive Control
A01:2021-Broken Access Control	• Check that users have access to requested resources. • Rate limit the API • Add unit and integration tests to test access control.	C1: Implement Access Control
A02:2021-Cryptographic Failures	• Don’t store unused sensitive data. • Encrypt stored sensitive data. • Do not store passwords; instead, store their salted hashes.	C2: Use cryptography the proper way.
A03:2021-Injection	• Use an Object-Relational Mapping (ORM) like	C3: Validate all input and handle exceptions

	SQL Alchemy or Django to access the database (Ben Fredj et al., 2020). ● Paginate responses to avoid mass disclosure.	
A04:2021-Insecure Design	● Use a secure development lifecycle (SDL) such as Secure SCRUM. ● Refer to the security risks and mitigation user stories. ● Use unit and integration tests to check that misuse cases are mitigated. ● Associate each record with a tenant to enforce segregation.	C4: Address Security from the Start

A05:2021-Security Misconfiguration	• Use identical security configurations in all environments (e.g., development, testing, and production). • Exclude unused features and frameworks.	C5: Secure By Default Configurations
A06:2021-Vulnerable and Outdated Components	Exclude unused Python dependencies from the project.	C6: Keep your components secure.
A07:2021-Identification and Authentication Failures	• Do not give admins default credentials. • Check passwords against weak password lists. • Enforce a minimum password length. • Use the same message for	C7: Implement Digital Identity

	registration and password reset outcomes to prevent enumeration. ● Block login after repeated failed login attempts.	
A08:2021-Software and Data Integrity Failures	Use pip for dependency management.	C8: Leverage Browser Security Features
A09:2021-Security Logging and Monitoring Failures	Log login failures.	C9: Implement Security Logging and Monitoring
A10:2021-Server-Side Request Forgery	Exclude server-side requests as a feature.	C10: Stop Server Side Request Forgery

Threat Modelling

STRIDE

Threat	**Mitigation**
Spoofing	Get user from request on server side.
Tampering	Check user privileges before allowing database and filesystem writes.

Repudiation	Add logging.
Information disclosure	Paginate API responses.
Denial of service	Rate limit the API.
Elevation of privilege	Add system monitoring, log events, require strong passwords, limit the number of privileged accounts, and keep software updated.

DREAD

Category	**Threats**
Damage	● Financial API credentials exposed (e.g., payment processor). ● Unbacked-up data destroyed or encrypted for ransom.
Reproducibility	Part of the system is not monitored (i.e., has no logs)
Exploitability	Published vulnerabilities in unpatched software
Affected users	● Privilege escalation allows attackers to affect other users. ● Data destroyed or encrypted for ransom. ● Exposure to payment processor credentials allows attackers to use other users' payment methods.
Discoverability	● Published vulnerabilities in unpatched software

	• Endpoints vulnerable to enumeration • Record ownership not enforced.

Legal Requirements

The application must adhere to various legal requirements to ensure compliance and avoid legal repercussions. Key legal considerations include data protection, consumer rights, and e-commerce regulations. The application must secure customer data, including personal and payment information, as mandated by the General Data Protection Regulation (GDPR) and Payment Services Regulations (PSR).

GDPR

The General Data Protection Regulation (GDPR) is a critical component of the legal framework. The GDPR mandates that businesses protect EU citizens' personal data, ensuring that it is processed lawfully, fairly, and transparently. The application must implement features that enable users to consent to data collection, access their data, and request its deletion (Voigt & Von Dem Bussche, 2017).

The Payments Services Regulations

The Payment Services Regulations (PSR) establish the legal framework for payment services within the European Economic Area (EEA). The application must comply with PSR to ensure secure and reliable payment processing. This includes implementing

strong customer authentication, ensuring payment data security, and providing clear information on fees and charges (Lynn et al., 2019).

Conclusion

The design document provides a framework for designing a secure online retail application. It applies industry best practices for ensuring all major security considerations are addressed at the application design stage. Defence mechanisms against common attack patterns are outlined. The intended user experience is described. Examples are given using UML diagrams for both an intended and a malicious use-case. Finally, the proactive controls, GPDR, and legal requirements are outlined, as compliance with them is necessary.

References

Ben Fredj, O., Cheikhrouhou, O., Krichen, M., Hamam, H. & Derhab, A. (2020) An OWASP Top Ten Driven Survey on Web Application Protection Methods, in: J. Garcia-Alfaro, J. Leneutre, N. Cuppens, and R. Yaich (eds) *Revised Selected Papers. Risks and Security of Internet and Systems*, Paris: Springer. 235–252. Available from: https://doi.org/10.1007/978-3-030-68887-5.

Lutkevich, B. & Lewis, S. (N.D.) What is the Waterfall Model? - Definition and Guide. Software Quality. Available from: https://www.techtarget.com/searchsoftwarequality/definition/waterfall-model [Accessed 5 September 2024].

Lynn, T., Mooney, J.G., Rosati, P. & Cummins, M. (eds) (2019) *Disrupting Finance: FinTech and Strategy in the 21st Century*. Cham: Springer International Publishing (Palgrave Studies in Digital Business & Enabling Technologies). Available from: https://doi.org/10.1007/978-3-030-02330-0.

OWASP Foundation (N.D.a) *OWASP Top 10*. Available from: https://owasp.org/www-project-top-ten/ [Accessed 4 August 2024].

OWASP Foundation (N.D.b) *OWASP Proactive Controls*. Available from: https://top10proactive.owasp.org/ [Accessed 31 August 2024].

Voigt, P. & Von Dem Bussche, A. (2017) *The EU General Data Protection Regulation (GDPR)*. Cham: Springer International Publishing. Available from: https://doi.org/10.1007/978-3-319-57959-7.