A. INTRODUCTION

1. **Team Name:** Infinite Void
2. **Team Members:**
    - Galen Chang
    - Josiah Kila
    - Chris Dang
    - Michelle Ho
3. **Application Details:**
    - **Title:** Store Transaction Tracker
    - **Description:** A simple command line-based application that is used to keep track of your spending. Users can securely input their purchases and view their total spending.
    - **Functional Requirements:**
        - A secure database of purchases
        - Purchase description and cost
        - Total spending over some period of time (day, week, month, year)
        - Functions on Windows devices
        - Utilizes secure connections (SSH)
        - STRETCH GOAL: User interface
4. **Type of program:**
    - Command Line that connects to a server via ssh to retrieve and update transactions database
5. **Development Tools:**
    - Python
    - SQL
    - VS Code or IntelliJ
    - Github

B. REQUIREMENTS

1. **Security & Privacy Requirements:**

    1. Introduction to Privacy and Security in Software Development

        In the development of the Store Transaction Tracker, it is imperative to integrate security and privacy measures from the outset. The application, which is likely to handle sensitive information, requires a structured approach to safeguarding data. Adopting principles from the SDL (Security Development Lifecycle) Privacy Bug Bar provides a framework for categorizing and addressing potential privacy issues. This framework is essential for developing an application that not only functions efficiently but also maintains the trust and confidentiality of its users.

    2. End-User Scenarios: Critical Privacy Issues

        The first layer of the SDL Privacy Bug Bar emphasizes critical privacy issues in end-user scenarios. These include the lack of notice and consent for transferring sensitive PII (Personally Identifiable Information), insufficient user controls for managing personal data, inadequate data protection measures, and non-compliance with child protection laws. In the context of the Store Transaction Tracker, these scenarios underline the necessity of implementing explicit user consent mechanisms, robust data protection protocols, and specific measures for handling children's data in compliance with legal requirements.

    3. Importance of User Controls and Data Protection

        An essential aspect of the Store Transaction Tracker's development involves providing users with control over their data. This includes the ability to opt-in or opt-out of data collection and sharing, as well as access to personal data for review and correction. Moreover, data protection should be a cornerstone of the application's architecture, with encryption of PII both in transit and at rest. This ensures that user data is protected against unauthorized access and breaches.

    4. Addressing Moderate and Important Privacy Concerns

Beyond critical issues, the SDL Privacy Bug Bar also identifies moderate and important privacy concerns. These include the handling of non-sensitive PII, data minimization principles, and the appropriate use of cookies. For the Store Transaction Tracker, it is crucial to minimize the collection of unnecessary data, use cookies judiciously and securely, and ensure that any non-sensitive PII is handled with the same level of care as sensitive information.

5. Enterprise Administration Scenarios

In scenarios involving enterprise administrators, the SDL Privacy Bug Bar highlights the need for specific controls and privacy disclosures. It is crucial for the Store Transaction Tracker to have mechanisms that allow enterprise administrators to control the flow of sensitive data and be clearly informed about how and where the data is stored, processed, and transferred. This is particularly important in the context of automated data transfers and legal compliance.

6. Definitions and Compliance with Privacy Terms

Understanding and complying with the definitions of terms like “PII,” “sensitive PII,” “explicit consent,” and “anonymous data” are fundamental to the application’s privacy strategy. The Store Transaction Tracker must be designed in a way that these terms are not only understood but are also reflected in the application's functionality, user interface, and documentation. This ensures that all stakeholders, including end-users and enterprise administrators, are on the same page regarding privacy expectations and realities.

7. Implementing SDL Privacy Bug Bar Guidelines

In conclusion, integrating the SDL Privacy Bug Bar guidelines into the development of the Store Transaction Tracker provides a comprehensive approach to addressing privacy concerns. This involves creating clear and user-friendly consent mechanisms, robust data protection systems, and specific controls for different user scenarios, including enterprise administrators. By prioritizing these aspects, the project team can ensure that the application is not only functional and user-friendly but also adheres to high standards of privacy and security, thereby building trust and compliance.

2. **Quality Gates (or Bug Bars):**

   Issues that must be addressed now in preparation for what to do in those scenarios.

   The End-User and Enterprise Administration Scenarios (Privacy):

   End-User Scenarios:

   - Critical - The privacy of users has been compromised.
     - Lack of control: The UI lacks features that should be available to users for privacy reasons.
       - EX: No option to directly delete all of the stored information.
     - Lack of child protection: Age is not asked before the use of the product.
     - Lack of data protection: The data is stored in a standard database with little to no security.
       - EX: Other than having to sign into an admin account, the database can be accessed and modified.
   - Important - The privacy of users may be compromised if nothing is done in response.
     - Lack of consent: The UI will not confirm with the user before taking action regarding their information.
       - EX: Does not ask for consent before transferring their newly added transactions to the database from their local machine.
   - Moderate - The privacy of users is susceptible to compromise.
     - Lack of internal data management and control:
       - EX: The data is stored at an organization that does not have a retention policy and therefore the client's information will be on record for a very long while.

   Enterprise Administration Scenarios:

- Critical - The privacy of clients is compromised by the administration.
    - Lack of enterprise controls: Almost no constraints to what an administrator can do to other accounts.
        - EX: Admins are not prompted with consent to modify a user’s information.
    - Lack of data protection: Client data is not protected from the admin view.
        - EX: Admins can access a user’s information through a simple SQL query.
- Important - The privacy of clients has a high chance of being compromised.
    - Lack of consent: The UI will not confirm with the admin user for major actions regarding other user accounts.
        - EX: Does not ask for consent before executing a SQL command that exposes a user’s data.
- Moderate - The privacy of admins may be harmful to the data of clients.
    - Lack of enterprise controls: Unable to track the admin account that made changes.
        - EX: No mechanism for tracking the name of the admin that made severe changes to another user’s account.

The Server and Client (Security):

Server:

- Critical - When the server has been taken over and cannot take back control.
    - Elevation of privilege: The ability to either execute arbitrary code or obtain more privilege than authorized.
        - EX: unauthorized file system access and SQL injection (for code execution) that will result in a compromise of all information in the database.
- Important - Scenarios where it may elevate to a Critical level if not addressed promptly.

    - Elevation of privilege: The ability to either execute arbitrary code or obtain more privilege than authorized.
        - EX: unauthorized file system access and SQL injection (for code execution) done by a user on file.
    - Information disclosure: The ability to locate and read information about another user or system information.
        - EX: User A spies on User B’s transactions from 1 week ago without having to log into User A’s account.
    - Bypassing the provided security features.
        - Skipping over the authentication procedure (login with a username and password).
- Moderate - Not a threat to the program itself or users and their private information.
    - Denial of service - The Python command is not able to run properly.
        - EX: Access is denied because User A opens the command line without the proper privileges.
    - Spoofing - The ability to create a visually identical UI that targeted users are “accustomed to trust.”
        - EX: An attacker creating another command line Python program that uses the same command and the same text-based menu format.
- Low - Issues that may be a hindrance to usage but are completely avoidable.
    - Denial of service - The Python command is not able to run properly.
        - EX: User is unaware of the need to download Python or forgets the syntax for the initial running command.
    - Loss of data - Losing data amid saving.

        - EX: User enters transaction details but their laptop ran out of battery so that information is lost before getting stored in our database.

Client:

- Critical - When the client accounts have been taken over and cannot take back control.
    - Elevation of privilege: The ability to either execute arbitrary code or obtain more privilege than authorized.
        - EX: unauthorized file system access and SQL injection (for code execution) that will result in modifying other client information.
- Same stuff

3. **Risk Assessment Plan for Security & Privacy**
    1. **Introduction**
        - Objective: To assess and mitigate risks in the Store Transaction Tracker, focusing on privacy and security from the development phase through deployment and maintenance.
    2. Risk Identification
        - A. Poor Input Validation
            - Risk: SQL injection, path traversals, and remote command execution.
            - Mitigation Strategy: Implement rigorous input validation, parameterized queries, and regularly update application security practices.
        - B. Database Integrity and Security
            - Risk: Database corruption, unauthorized alterations, and data breaches.
            - Mitigation Strategy: Use strong encryption algorithms, implement hash checks for database integrity, and regularly audit database security.

- C. Privilege Management
    - Risk: Exploitation due to over privileged service accounts.
    - Mitigation Strategy: Employ the principle of least privilege, restrict file system access, and conduct regular privilege audits.
- D. Encrypted Communications
    - Risk: Data interception through unencrypted communications.
    - Mitigation Strategy: Implement end-to-end encryption for all data transmissions and use secure protocols like TLS.
- E. Session Management
    - Risk: Unauthorized access due to inactive user sessions.
    - Mitigation Strategy: Develop an automatic logout feature after a period of inactivity and educate users about the importance of signing out.
- F. Phishing Attacks
    - Risk: Account compromise through phishing.
    - Mitigation Strategy: Implement multi-factor authentication and conduct user education programs on recognizing phishing attempts.

3. Risk Analysis and Evaluation
    - Assess the Likelihood: Evaluate the probability of each risk occurrence based on current security and privacy measures.
    - Assess the Impact: Determine the potential impact of each risk on users, the organization, and compliance obligations.
4. Risk Mitigation Implementation
    - A. Input Validation
        - Develop and enforce strict input validation protocols.
        - Regularly update the application to protect against new vulnerabilities.
    - B. Database Security
        - Encrypt database contents using industry-standard algorithms.
        - Implement routine checks to ensure database integrity.

- C. Privilege Management
    - Conduct a thorough review of service account privileges.
    - Implement access control measures to restrict file system access.
- D. Encrypted Communications
    - Enforce HTTPS for all data in transit.
    - Regularly update encryption protocols to address emerging threats.
- E. Session Management
    - Develop an automatic logout mechanism based on inactivity.
    - User education on the importance of logging out after use.
- F. Phishing Protection
    - Implement and maintain robust multi-factor authentication.
    - Conduct regular user training sessions on cybersecurity awareness.

5. Monitoring and Continuous Improvement
    - A. Security Audits and Testing
        - Regularly perform security audits and penetration testing.
        - Continuous monitoring for any unusual activities or potential breaches.
    - B. User Feedback and Incident Reporting
        - Establish a system for users to report security concerns.
        - Implement an incident response plan to address reported vulnerabilities quickly.
6. Compliance and Documentation
    - A. Compliance with Standards
        - Ensure the application complies with relevant data protection laws and industry standards.
        - Maintain thorough documentation of all security measures and audits.
7. Training and Awareness
    - A. Staff Training
        - Regular training for staff on the latest security practices and threat awareness.

        - Specialized training for staff handling sensitive information.
    - B. User Education
        - Develop user guides and resources on secure usage of the application.
        - Regular communication with users about the importance of security practices like multi-factor authentication.

C. DESIGN

1. Design Requirements

1. Solid Database Design: Central to the Store Transaction Tracker is its database which requires a strong and safe design. The use of an encrypted SQL database is essential for safeguarding transaction information. This coding not just protects information when it is still, but also strengthens it from unauthorized entry or violations. Moreover, using parameterized requests is key to stop SQL injection assaults, a usual weakness where attackers play with SQL inquiries to reach or damage the database. Further, the structure of the database has to allow transaction rollbacks and atomic operations. It maintains data integrity by enabling the system to go back to a stable state if there is any failure or error in a transaction, this way it stops partial updates or corruption of data.
2. Protection and Encryption of Data: Keeping confidential data secure is very important in the Store Transaction Tracker. By using SSL/TLS protocols to encrypt data while it's being transferred, we can make sure that all information sent between client and server is safe from unwanted listening or interception. The encryption of data at rest is also very important as it safeguards against unauthorized access when the data is kept safe. It's necessary to keep updating encryption algorithms regularly in order to protect from cyber threats that continue changing, and for keeping security at an optimum level. Moreover, putting in place safe backup procedures lessens the danger of losing data. It includes making extra data storage options, which make recovery possible if there is data corruption or loss.
3. For the purpose of maintaining pure data in Store Transaction Tracker, it is important to have strict validation and sanitization processes for input. The act of validating inputs stops harmful insertion of data like code injections or incorrect formats that could possibly interfere with database functions or create safety risks. To clean user inputs, we take out or make harmless any dangerous parts in the data before it is dealt with or kept. This method of having two steps uses both client-side and server-side checking to improve our app's protection from bad inputs which helps us handle data safely and dependably.

4. User Agreement and Privacy Adjustments: In tune with privacy regulations and what users expect, the Shop Transaction Monitor needs to have transparent and simple-to-find privacy rules. These rules need to explain how we gather, handle, and keep user information. Mechanisms of explicit consent from user are very important for any activities that collect and process data. This makes sure users completely understand and accept how their details are utilized. Additionally, the application should give users capability to see, change and remove their saved data. This promotes openness but also provides control for users on their own details in line with privacy rights and rules.
5. Access and Controls for Administrators: The blueprint must incorporate strong controls that allow administrators to handle the application in a secure manner. Admin rights should be limited according to the least privilege principle, verifying that administrators get only essential access permissions needed to execute their responsibilities. This reduces the danger of unpermitted access to data or alterations in the system. Keeping track through audit trails and logs is crucial for supervising admin tasks, which allows control and responsibility. Also, by adding strong access controls to admin interfaces like multi-factor authentication and IP whitelisting, the system gets more protection against unapproved entry and possible internal risks.
6. User Interface and Experience: User interface is a very important part of the Store Transaction Tracker. A clear command-line interface is required for simple navigation and usage, making sure that users can manage their transactions effectively without any misunderstanding or annoyance. For an extra aim, we could create a basic visual user interface (GUI) to improve the experience of users and make our application easier for more people to use. This GUI ought to be made in such a way that it is simple for everyone, with straightforward instructions and responses. It should ensure even those who are not tech-savvy can engage well with the app comfortably as well as efficiently.
7. Compliance and Law Following: Store Transaction Tracker has to follow data protection laws like GDPR, CCPA, and others. It cannot be avoided or negotiated against. The design of the application should be made in a way that follows these

rules which keep changing over time. Also, regular updating is needed for this application to stay compliant with such regulations. This is about providing precise explanations and following rules associated with terms such as "Personally Identifiable Information" (PII) and "explicit consent." Making sure the compliance protects both users of the application also safeguards the company from any legal issues. This improves trustworthiness and reliability of software in the marketplace.

8. Scalability and Performance: The Store Transaction Tracker should have capacity to grow and work effectively even if transactions or users increase. This needs planning of the application along with its foundational database in a way that can manage more load but still not affect performance. Making database requests better for performance is very important so that data can be quickly fetched and stored, which will make the user's experience better. Also, making use of caching methods where it fits best can greatly enhance response times and lessen the pressure on the database, adding to how well an application works overall.
9. Handling Mistakes and Recording: Handling mistakes is important to avoid application failures and loss of data. The system must have the ability to manage unexpected situations smoothly, with methods for recording errors and informing administrators when necessary. Logging in detail is crucial not just for debugging, but it also serves a significant function in security auditing because it gives system activities and anomalies record. Moreover, giving simple to understand error messages aids users with solving problems without revealing sensitive information about the system while keeping usability and safety balanced.

2. Attack Surface Analysis & Reduction

1. Poor input validation

    If the user’s input is not validated properly, this could lead to a series of vulnerabilities. Attacks similar to an SQL injection could occur, where the attacker is able to have the application return information they should not be able to query or modify/delete entries in the database. The attacker might be able to conduct path traversals, where they are able to “break out” of the application and query the server’s file structure. More sinister attacks could be running commands on the server, which leaves it vulnerable to exploitation. An attacker might be able to reach out to a server, download malware, and execute it, giving them the ability to have remote access where they can then further compromise the system and put the users at risk.

2. Corrupting or altering database

   The database should be properly encrypted with a secure algorithm that hasn't been compromised. Not doing so could result in the attacker corrupting the database, where the attacker could change the amount spent and description of the purchase, completely delete or make the database unrecoverable, or even replace the database with malicious code that is read into the program and executed (circles back to point 1). There should also be a way to validate that the database has not been tampered each time it is accessed by the user. One way to do this could be getting a hash of the user's database after the client closes the connection and storing it. When the user accesses the database again, it is checked against that hash.

3. Improper run level (should run as a controlled, locked down account)

   The application should never run as an over privileged user. The service account should be very locked down and only given permissions it explicitly needs. It would be absolutely disastrous if the account is given System, Root, or administrator privileges. The account should not be able to read or write to directories anywhere other than its own directory tree. This might help prevent file system traversals.

4. Encrypted communications

   Communications between the client and server must be encrypted. Failure to do so could subject the user to a man in the middle attack or the attacker being able to use something like WireShark to capture packets and view the user's confidential information.

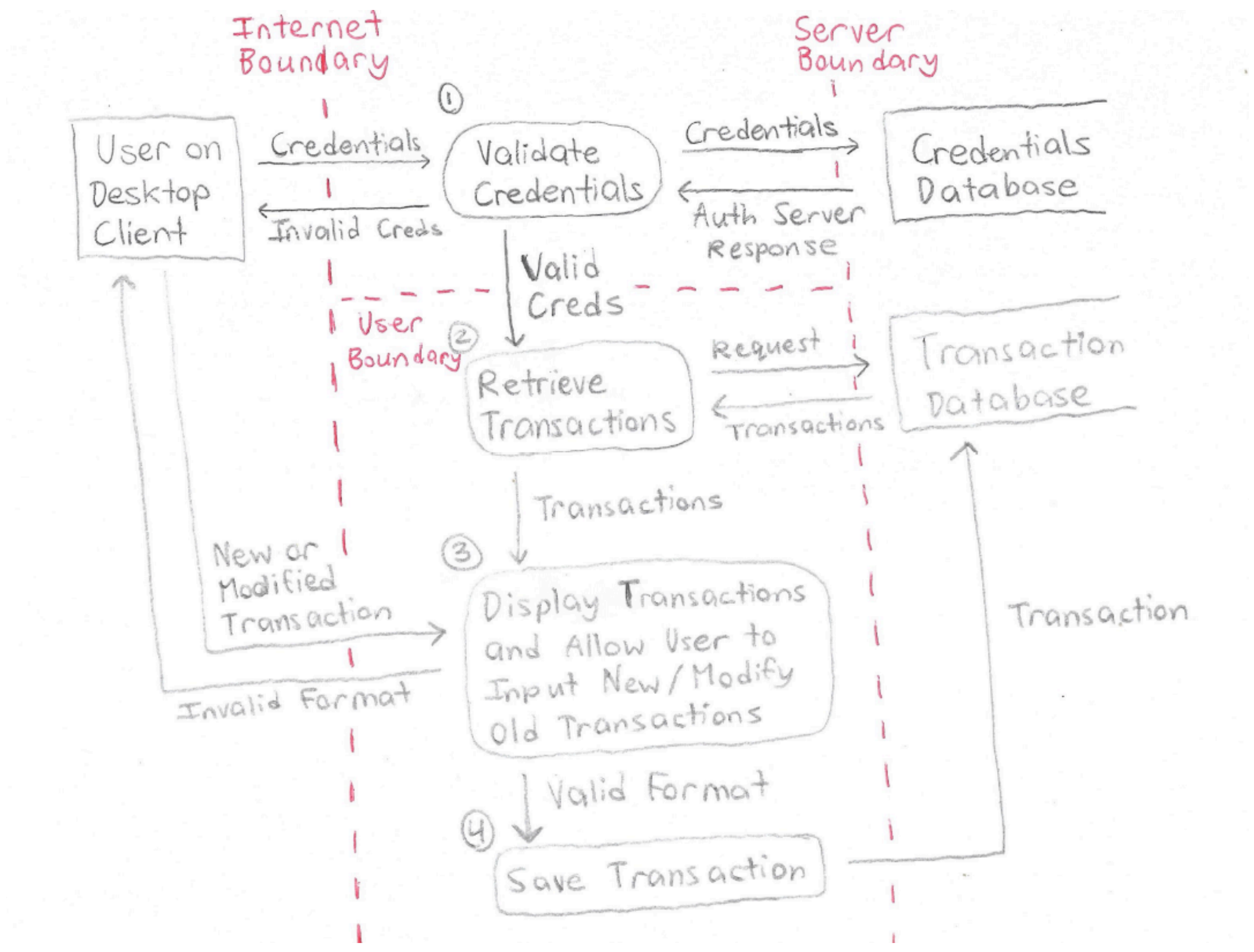
5. User forgetting to close app or sign out

    If a user forgets to sign out of the application or close it, it leaves them vulnerable to someone sitting down at their device and browsing their transactions. Implementing an automatic logout based on a certain amount of time of inactivity (similar to Laulima) would help deal with this issue.

6. Phishing

    One of the simplest and most effective ways attackers get into someone's personal information is through phishing. Multi Factor authentication would be useful here, as this would not only prevent an attacker from logging in whenever they want, but the multi factor authenticator would preferably one that might offer a second chance for the user to recognize they were phished, such as where the account is logging in from.

3. Threat Modeling



1. Internet Boundary Threats and Categorization
   - DDoS - Denial of Service
   - Authentication Brute Force - Spoofing
   - Man in the Middle - Information Disclosure
   - Packet Sniffing - Information Disclosure
   - Poor Input Validation - Tampering, elevation of privilege
2. Server Boundary Threats and Categorization
   - Physical Access to Server - Elevation of privilege, tampering, information disclosure, denial of service
   - Theft of Credential Database and Password Cracking - Spoofing, information disclosure
   - Theft of Transaction Database - Information disclosure, denial of service

- Obtaining Server User, Root, or Administrator Credentials - spoofing, tampering, denial of service, elevation of privilege
- Exploitation of Program - Denial of service, elevation of privilege

3. User Boundary Threats and Categorization
    - Data Modification - Tampering, repudiation
    - Access to User Account Information - Spoofing, information Disclosure, Repudiation
    - Poor Input Validation - Tampering, elevation of privilege

Implementation:

1. **Approved tools:**
   - Command Line
     1. Not currently necessary as we are using Visual Studio Code as our IDE. Once released, this will be the primary way to run the program.
   - Visual Studio Code v1.86.2
     1. IDE we are using.
   - Python v3.11.8
     1. Language we are using.
   - Python (extension for Visual Studio Code) v2024.0.1
     1. Extension that enables Python usage in Visual Studio Code
     2. Allows for additional extension integration
        1. Pylance, linting, debugging, etc
   - Pylance (extension for Visual Studio Code) v2024.2.2
     1. A suite of quality of life additions
        1. Auto-imports, static code analysis, static type checking, etc
   - Python Debugger (extension for Visual Studio Code) v2024.0.0
     1. Debugging tools
        1. Set breakpoints, step through code, variable inspection, etc
   - Pylint (extension for Visual Studio Code) v2023.10.1
     1. Customizable Python linter that provides immediate feedback
   - SQLite (extension for Visual Studio Code) v0.14.1
     1. Makes querying and viewing SQLite3 databases easy
   - SQLite3
     1. Currently being used as the database for user accounts.

2. **Deprecated/Unsafe Functions** :
   - Python Deprecated Functions:
     1. time.clock(): Deprecated since Python 3.3, removed in Python 3.8. Use time.perf_counter() or time.process_time() instead for high-resolution timing.

        ii. os.popen(): Considered less secure than subprocess. Use subprocess.Popen() for more reliable and secure subprocess management.
        iii. urllib.urlopen(): Deprecated since Python 3. Use urllib.request.urlopen() for opening URLs.
- Pylint Deprecated Checks:
    i. For any deprecated Pylint checks, ensure to update the .pylintrc configuration file to the latest standards. Pylint is regularly updated, so it's crucial to keep its configuration and usage up to date.
- SQLite3 Deprecated Features:
    i. Direct String Concatenation in Queries: Directly concatenating strings to construct SQL queries can lead to SQL injection vulnerabilities.
        1. Alternative: Use parameterized queries to safely create SQL queries. This approach uses placeholders for parameters that are then bound to query parameters at execution time, preventing SQL injection.
- Visual Studio Code Extensions and Python Practices
    i. Python 2 Support in Extensions: With Python 2 reaching end-of-life, some features or support in extensions might be deprecated.
        1. Focus on Python 3: Ensure all development and extensions are fully compatible with Python 3, particularly the version you're using (v3.11.8).
- Deprecated API Usage in Extensions: Extensions like Pylance or the Python extension itself may deprecate certain APIs or features as they update.
    i. Best Practice: Regularly update extensions to the latest version and follow the extension documentation for updated features and best practices.
- General Best Practices
    i. Regularly Update Your Tools: Keeping your development tools, languages, and extensions up to date is crucial for accessing the latest features and security updates.

ii. Consult Official Documentation: When in doubt about the deprecation status of a function or feature, consult the official documentation for the most accurate and up-to-date information.
iii. Security Practices: Always prioritize secure coding practices, especially when dealing with databases, executing commands, or handling user input.

3. **Static Analysis:**

Pylint (extension for Visual Studio Code) v2023.10.1

The Python extension for Visual Studio Code comes with Pylance, which has a rather poor static code analysis tool built in. It placed red squiggly lines under syntactic errors, such as when you missed a colon or your indentation is incorrect. It also dulled the colors of variables and imports that were not being used. Referencing methods and variables were also marked with yellow squiggly underlines. Helpful, but extremely basic. It did not find errors that would cause issues during run time. If the code looked like it would work, Pylance wouldn't detect the errors.

For instance, we incorrectly wrote "delete_record(sql_connection, cursor)" as one of the options the user could select. What we meant to write was "delete_account(sql_connection, cursor)". Another error we wrote was when we were adding a function to the Transaction class. We wanted to have a function return the "cursor" attribute. But when trying to define the "cursor" function, we had it return "self.cursor.connect_to_db('database.db')". When we tried to test the program, we ran into issues that Pylance did not catch.

After some quick searching online, we decided to install Pylint, and all the errors in our code (including some that we didn't find while testing) were clearly marked with red squiggly lines. Unused imports and variables were much more clearly marked with a yellow squiggly line, and style errors like not camel casing correctly or no docstring were underlined with a blue squiggly line.

Going back to the two aforementioned errors, Pylint had them marked with the red squiggly lines. Highlighting the bits of code in question displayed a box telling us exactly what was wrong. The mistyped "delete_record" was marked with "Too many positional arguments for function call". After seeing this, we immediately recognized that we needed to change the function call to "delete_account" like we originally intended. For the second error, Pylint told us that the "method 'cursor' has no 'connect_to_db' member". We realized that we were trying to write a function called "cursor" that would return itself as if it were an attribute. A very careless mistake, but one that we would not have caught had it not been for Pylint.

The only issue we are currently facing with Pylint is that we haven't found a way to ignore errors. Our code is covered in blue squiggly lines due to no docstrings. It is not something that is significant right now, but there may be instances where we may want to intentionally ignore some warnings Pylint is showing us (and also because seeing blue squiggly lines everywhere makes it seem like there are problems with our code when there really isn't).

QR Authenticator Integration

The QR Authenticator steps into the Store Transaction Tracker application marks a significant enhancement in our security protocol, specifically in the authentication process for user login. This update necessitates a revision of our verification report to include the analysis of this new feature's implementation and security implications.

Verification

The verification process has been expanded to encompass the newly integrated QR Authenticator feature, ensuring that the application not only continues to meet our security and functionality standards but also leverages enhanced user authentication mechanisms.

Dynamic Analysis

Expanded Tool Use:

- Alongside Pytest and Coverage.py, we incorporated additional testing to specifically address the QR Authenticator functionality. This involved creating mock scenarios to simulate the generation of QR codes, scanning through an emulator, and verifying the authentication process with generated codes from a Google Authenticator app emulator.

Experiences with QR Authenticator Integration:

- Challenges: The QR code uploads to the project folder which is not what you want for a full blown application. Additionally, on every login in attempt, you will have to rescan the QR code with the Google Authentication App.
- Successes: Despite initial challenges, our dynamic analysis successfully validated the QR code generation and authentication flow. We were able to automate the verification of correct and incorrect codes, ensuring that the login process works as expected with this added layer of security. This has significantly enhanced our confidence in the application's authentication mechanisms.

Attack Surface Review

- The introduction of QR Authenticator functionality prompted a focused review of the libraries and APIs involved in generating and processing QR codes and integrating with authentication systems like Google Authenticator.
- External Dependencies: The reliance on external apps (e.g., Google Authenticator) was considered. While we cannot control third-party app security, ensuring our QR codes are correctly formatted and securely generated falls within our responsibility.
- The transaction SQLite3 database is currently in cleartext. This has obvious ramifications for privacy and security, but plans are in place to encrypt all the entries. Next update will have encrypted tables.

Conclusion: The comprehensive attack surface review underscored that the incorporation of QR Authenticator functionality, while introducing new components into the system, does not significantly elevate the project's vulnerability profile, assuming that all external dependencies are maintained securely and updated regularly. The findings support the continued integration of QR Authenticator, with a commitment to periodic security reviews and updates as part of our ongoing effort to safeguard the application against emerging threats and vulnerabilities. This proactive stance ensures the resilience of the QR Authenticator feature and the security of the overall system. The problem with the cleartext transaction database is known and will be fixed in the next update

Assignment 4

Fuzz Testing

As part of our software development lifecycle (SDLC), we conducted a series of verification processes to ensure the robustness and security of our application, specifically focusing on fuzz testing. Fuzz testing, or fuzzing, is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The primary goal is to uncover bugs, security vulnerabilities, or crashes by monitoring the program's behavior under abnormal conditions.

Fuzz Testing Methodology

For our application, we employed a custom fuzzing script designed to interact with the application's command-line interface. The application in question, account_database_function.py, is a Python script managing user accounts, including functionalities for account creation, login, and two-factor authentication.

Tools and Techniques

- Fuzzing Tool: Custom Python Fuzzing Script
- Target: account_database_function.py
- Techniques: Random String Generation for Command-Line Input
- Duration of Each Fuzzing Session: Approximately 10 seconds per iteration
- Total Iterations: 100

Process

Setup: We prepared a controlled environment to ensure the fuzzing process did not interfere with any production databases or services.
Input Generation: Our fuzzing script generated a wide range of random strings, simulating potential user inputs. These inputs were designed to test the application's ability to handle unexpected or malicious data.

Execution and Monitoring: We ran the script, passing the generated inputs to the application, and monitored its behavior for any signs of crashes, errors, or vulnerabilities.
Analysis: After each iteration, we analyzed the outcome, focusing on any failures or unexpected behaviors triggered by the fuzz inputs.

Results

- Successes: The majority of fuzz inputs resulted in "Success" messages, indicating that the application handled unexpected inputs without crashing.
- No Direct Crashes: No inputs directly caused the application to crash, which suggests a basic level of input handling resilience.

Analysis of Findings

- Handling of Unexpected Inputs: The application appears to robustly handle a broad range of input types without crashing, indicating effective basic input validation.

Mitigation and Enhancements

Following the results, we focused on areas identified as potential concerns:

Improved Input Validation: We reviewed and enhanced input validation measures to ensure that all inputs are efficiently processed or rejected if inappropriate.
Handling Timeouts: We introduced additional checks and balances to handle operations that could lead to excessive processing times, ensuring that the application remains responsive under all conditions.
Code Optimization: We optimized several parts of the codebase to improve overall efficiency and reduce the risk of hanging or slow processing.

The fuzz testing campaign provided valuable insights into the resilience of our application against unexpected and potentially malicious inputs. While no significant vulnerabilities were uncovered during this phase, the process highlighted the importance of continuous testing and optimization. Moving forward, we plan to integrate more sophisticated fuzzing techniques and extend our testing to cover more complex user interaction scenarios.

John The Ripper

John The Ripper is a password recovery tool capable of cracking passwords hashed/encrypted by many popular algorithms. By utilizing dictionaries of common passwords, John The Ripper applies these algorithms to the dictionary in an attempt to find a match with the encrypted password that you want to recover. If the dictionaries do not work, John The Ripper is also capable of testing common alterations to the dictionaries (for example, if "password" is one of the words, then "Password" might be tested), as well as a brute force approach.

Testing Methodology

Our program utilizes SQLite3 to store sha256 encrypted usernames and passwords. A commonly used username (admin) and password (password) combination was tested alongside a standard firstname.lastname<random number> username and a password with standard complexity requirements (12 character minimum with 3 capital, lowercase, and special characters). John was used with the default dictionary found on Kali Linux to test both test cases, as well as the brute force approach.

Tools and Techniques

- Cracking Tool: John The Ripper
- Target: userdatabase.db
- Attack Types: Default dictionary attack and brute force attack

Process

Setup: We prepared a controlled environment to ensure the fuzzing process did not interfere with any production databases or services. The previously mentioned accounts (see "Testing Methodology" above) were created and encrypted in the user database. The encrypted usernames and passwords were extracted from the SQLite3 file and placed into a flat .txt file.

Attack Types: John The Ripper runs with a default dictionary if one is not supplied to it. This is the dictionary that was used for testing. The brute force attack was allowed to run for 1 hour.

Execution and Monitoring: John The Ripper was fed the .txt file and allowed to run in both dictionary and brute force attack modes and left to run until either the username/password was found or the 1 hour time limit ran out.

Results

- The common username/password was quickly found by the dictionary attack.
- The complex username/password was not cracked by either the dictionary or the brute force attack after 1 hour.

Analysis of Findings

- Addition of username and password complexity requirements greatly increases the security of the user accounts. As expected, common usernames and passwords are cracked easily and should not be used under any circumstances.

Mitigation and Enhancements

Following the results, we focused on areas identified as potential concerns:

Username and Password Complexity: Requiring complex usernames and passwords should be implemented, as the amount of time it takes for John The Ripper (and potentially all other password cracking tools) makes it extremely difficult for attackers to gain access to confidential data.

Testing our encryption choice of SHA 256 against John The Ripper proves the industry standard of requiring complex usernames and passwords as a strong defense against password cracking tools. Currently, there are no complexity requirements, posing a severe security risk to the application. By the next update, we plan to implement a complexity requirement.

SQL Injection

SQL Injection attacks primarily work because most applications don't have a mechanism to ensure that any input data will be in a form that behaves properly. This is mostly due to input data that includes special characters that can be read as code such as quotation marks and semicolons followed by malicious SQL commands. These types of attacks have a wide variety of results since any SQL commands can be executed once the attacker finds out that a certain character can be used to inject these commands.

SQL Injection Methodology

To test our application against SQL injection attacks, we wrote up a few attempts and tested it each time user input was being asked. Those attempts included special characters (quotation marks and semicolons).

Tools and Techniques

- SQL Injection Tool: Write the various attacking attempts as the input data when asked for user input.
- Target: Deleting all data tables and displaying all tables (all the stored information).
- Techniques: Writing two different SQL commands with a different special characters preceding to break out of the user input section.
- Duration of Each SQL Injection Session: A few seconds per attempt.
- Total Iterations: 20.

Process

Setup: No extra setup environment was needed as SQL Injection attacks are done on the application through the input fields.

Input Generation: The following inputs were used at each section of user input:

- “John Doe”;DROP TABLE *;”
- “John Doe”DROP TABLE *;”
- “John Doe;”DROP TABLE *;”
- “First”;SELECT *;”

- “First”SELECT *”
- “First “;SELECT *”

Execution and Monitoring: Each of the above inputs were tried as input when the user was prompted to put in their information. We monitored how our application would respond to the inputs, looking for any changes in our database during the session (when using the delete command) and any information of our database being revealed in the terminal (when using the select all command).

Results

These commands didn’t reveal or delete any of the existing databases and stored information. The application instead just used the whole input as the user’s responses.

Analysis of Findings

We believe that the application didn’t break with the SQL Injection attack attempts because all of the user inputs are hashed before any other database-related commands are executed. However, had we not filtered (hashing allows for removing unwanted special characters to be put into our own SQL commands) and verified user inputs, SQL Injection attacks would probably work and our system could be easily breached since there are multiple opportunities for user input.

Mitigation and Enhancements

Since we know that we must filter and verify the user inputs, we will need to enhance this portion of the application. It may be wise to add a feature to stop the entire transaction if any of the user inputs contains any special characters. Allowing a message to notify the user not to use these sepcial characters will educate the user as well so they can create answers that don’t include them.

Static Analysis

Following the implementation of additional error handling mechanisms as highlighted in our previous phase of development, we conducted a further round of static analysis using Pylint. This review aims to assess the impact of these enhancements on code quality and reliability, and to verify the effectiveness of the newly introduced error handling in our Python application, particularly within the account_database_function.py module.

Tool Utilization

Pylint

- Version: 2023.10.1
- Configuration: Default settings with attempts to modify behavior for ignoring trivial warnings (e.g., missing docstrings)

Methodology

The analysis focused on evaluating the changes made to error handling mechanisms and overall code improvements since the last review. Specific attention was paid to areas previously identified as problematic or lacking robustness.

Key Findings and Actions

Enhanced Error Handling

- Before: Initial rounds of static analysis and manual testing revealed several crucial gaps in error handling, particularly around database operations and user input processing.
- After: Enhanced error handling was implemented, including more comprehensive try-except blocks and validation checks before proceeding with database operations.

Specific Improvements and Issues

Database Connection and Cursor Handling

- Previous Issue: Lack of error handling around database connections and cursor creation.
- Current Status: Improved with robust try-except blocks. Pylint confirmed no errors related to these operations.
- Action Taken: Implemented explicit error handling to catch and log database connection issues, preventing the application from crashing and informing the user appropriately.

Function Misnaming and Misuse

    - Previous Issue: Functions like delete_record were incorrectly named and used, causing runtime errors that were not caught by less sophisticated tools.
    - Current Status: Renamed accurately to delete_account, with Pylint highlighting the issue before runtime.
    - Action Taken: Corrected function names and usage in the code, improving the code's readability and functionality.

- Improper Method Calls
    - Previous Issue: Incorrect method calls like self.cursor.connect_to_db('database.db') were identified.
    - Current Status: Errors were corrected, and method usage was clarified and fixed.
    - Action Taken: Refactored erroneous method calls to ensure that all functions and methods are utilized correctly.

Code Quality

- Unused Variables and Imports: Pylint effectively identified unused variables and imports that were not highlighted in earlier reviews. These were cleaned up to improve code efficiency and readability.
- Code Style Issues: Non-camel case naming and missing docstrings were flagged. While not critical, these issues were partially addressed to comply with Python coding standards, enhancing maintainability and readability.

Challenges

- Ignoring Trivial Warnings: We encountered difficulties in configuring Pylint to ignore non-critical warnings such as missing docstrings. This resulted in numerous blue squiggly lines throughout the code, which could obscure more significant issues.
- Resolution: We are exploring Pylint's configuration options to better tailor the tool's behavior to our needs, potentially editing the .pylintrc file to adjust thresholds and disable certain warnings.

The additional error handling and corrections made since the last static analysis have significantly improved the stability and reliability of the application. Pylint has proven indispensable in identifying both major and minor issues, allowing for timely and effective corrections. Moving forward, we will continue to refine our static analysis process, aiming to balance strict code quality enforcement with practical development needs. Further adjustments to our Pylint configuration will seek to minimize distractions from non-critical warnings, focusing our attention on issues that impact the application's functionality and security.

Dynamic Analysis

Current Tool Use and Methodology

Tools Used:

- Pytest: For running dynamic tests on the application.
- Coverage.py: To assess the coverage of tests across the codebase.
- Emulators: Used to simulate interactions with the Google Authenticator app for QR code authentication.

Testing Scope:

- QR Code Generation: Ensuring QR codes are generated correctly within the application.
- Authentication Process: Testing the integration of the QR Authenticator within the user authentication workflow.

Current Findings

QR Code Storage Issue

- Description: QR codes are currently stored directly in the project folder, posing potential security risks due to unauthorized access and cluttering the workspace.
- Impact: Increased risk of sensitive data exposure and reduced efficiency in managing storage resources.
- Planned Improvement: Implement storage of QR codes in a secure, temporary directory that is cleared after each session.

Login Process Usability

- Description: Users are currently required to rescan the QR code with every login attempt, which has been identified as a significant usability issue.
- Impact: Potential decrease in user satisfaction and efficiency due to the cumbersome login process.
- Planned Improvement: Adjust the login flow to require QR code scanning only upon new device setup or when security protocols explicitly require it.

Experiences with QR Authenticator Integration

Challenges

- QR Code Uploads: The handling of QR code uploads to the project folder is scheduled for a security overhaul to prevent unauthorized access and ensure data privacy.
- User Experience in Login Attempts: The current requirement for frequent rescanning of QR codes diminishes the user experience and efficiency.

Successes

- Authentication Validation: Despite the noted challenges, our dynamic testing successfully validated the QR code generation and authentication processes. The system reliably differentiates between correct and incorrect authentication attempts, upholding the integrity of the login process.

Attack Surface Review

- External Dependencies: Our reliance on third-party apps like Google Authenticator has been reviewed. While we ensure that our QR codes are generated securely, the actual security of the third-party app remains outside our direct control.
- Current Security Posture: The integration of QR Authenticator has not introduced significant new vulnerabilities according to our latest review. However, enhancements in handling and storage will further secure this feature. Transaction database is still cleartext, but we intend to encrypt it via SHA 256 like the user accounts database. No username and password complexity requirements introduces risk if the user(s) decide to use common usernames and passwords. A complexity requirement will be implemented by next update.

Release

Incident Response Plan

Escalation Manager - Chris Dang

When a potential privacy incident is identified, the escalation manager is responsible for handling communication with the reporter as well as taking the lead on appropriately resolving the issue. This individual will facilitate communication between and report progress from the legal, public relations, and security engineering teams.

Legal Representative - Galen Chang

Handle any legal matters regarding the privacy incident. Advise management on potential legal actions that may be taken against the team and provide recommendations on how to best settle the impending lawsuits. Help public relations draft the public response to avoid further legal troubles as well as assist the security engineer on the legal outcomes of the incident and what should be done to prevent further issues.

Public Relations Representative - Josiah Kila

The face of our team, this person will handle all inquiries from sources such as the media as well as advise management on the public ramifications of this incident and steps that should be taken to reduce negative impact to the team. This person will be responsible for crafting a response that will be our official stance on how this privacy incident occurred, what we are doing to fix it, and how we will assure our customers that this will never happen again.

Security Engineer - Michelle Ho

Identify the source of the breach and come up with a plan to quickly and effectively stop any further privacy breaches from occurring from this vulnerability. Additional responsibilities include handling remedial training for individuals at fault, as well as preventative training to prevent this from happening again. Work with the development team to patch the vulnerability and assist in providing security recommendations to management.

Contact Information

For any inquiries, please contact us at support@transactiontracker.com

Response Procedures

When a potential privacy incident occurs, it will be brought up to the Escalation Manager, who will then review it and speak to the Legal, Public Relations, and Security team if necessary for additional insight. If the incident has been deemed valid, all teams will immediately meet to discuss what has happened and reach out to the reporter for more information if available.

The Escalation Manager and Security Engineer will then conduct a review of the incident, to include all potential sources of the breach, how widespread and harmful the breach is, and attempt to re-create the chain of events that lead from start of breach to current situation. At this point, the Security Engineer will speak to the security and development teams to remedy the issue. Any major developments or new insights will be brought to the Escalation Manager's attention as soon as possible. The Escalation Manager will provide an estimated timeline for remediation of the incident.

In the meantime, the Legal and Public Relations teams will work together to draft an initial statement that will be released to the public, while incorporating any new information the Escalation Manager brings to them. The teams will also draft memos noting the possible impact to the company's reputation, income, and other potential legal and public fallout that will be brought up to higher management for situational awareness as well as guidance on the remediation. The Legal and Public Relations teams will also work together to deal with any lawsuits and inquiries that may occur.

Once the incident has been properly handled, all teams will meet to draft a comprehensive report on the incident, which will include how this happened, notable points of failure, ramifications to the company, as well as steps taken to remediate the situation. The teams will also create a training plan for employees to take that will provide awareness of the source of the breach and how they can help prevent future incidents from occurring.

Final Security Review

To finish up, we have tested the program's features from an end user's perspective. We were looking for SDL requirements to be met without having administration intervention or other third party contributions. After reviewing our threat model, static analysis, dynamic analysis, and quality gates/bug bars, we decided our program should receive the "Passed FSR" grade. This is because after trying out all features of the program, there have been positive responses that meet all SDL requirements. With this grade, we are confident that the program will result in a successful experience for our end users.

This decision is based on the positive outcomes observed during our exploration of all program features. We have confirmed that each feature aligns with SDL requirements, thereby assuring a secure user experience. With this grade, we express our confidence in the program's capacity to deliver a successful experience to our end users.

Certified Release and Archive Report

Release Version and Documentation

https://github.com/Team-Infinite-Void/transaction-tracker/wiki/Transaction-Tracker

Wiki page that provides a comprehensive overview of the Transaction Tracker, including installation, usage, points of contact, and more.

https://github.com/Team-Infinite-Void/transaction-tracker

The Github repository that houses the code for the Transaction Tracker. Also contains a README page that documents development progress, important links, and lessons learned.

https://github.com/Team-Infinite-Void/transaction-tracker/releases

Link to the release version of Transaction Tracker for public use.

Features and Development Plans

The current release version of Transaction Tracker is v1.0.1 and is a simple, yet advanced Python program that can not only securely store your transaction history, but provide analytics on your spending habits. By using QR authentication integrated with Google Authenticator and modern database encryption and hashing algorithms, this application ensures that the user's account information and transaction history will not be breached.

In the future, we plan to add networking capabilities, where the user is able to store their transaction and account databases securely in the cloud, allowing them to access them from any device, anywhere, at any time. We also intend to add more analytics features, such as time based filtering, more comprehensive financial reports to aid in budget planning, and tax preparation assistance. Furthermore, we plan to implement a fully functional and interactive UI, where you can visually see your spending activities in greater detail via graphs and charts. Increased security features will also be added, as well as regular updates to ensure that your data is always protected.

Technical Information and A Brief Overview

https://github.com/Team-Infinite-Void/transaction-tracker

Please visit the above link for a more detailed and complete overview of the Transaction Tracker.

Specifications for Use

- Operating System: Compatible with Windows 10 and newer.
- Python Version: Requires Python 3.8 or higher.
- Dependencies: All dependencies are listed in the requirements.txt file.

Installation Instructions

1. Clone the repository: git clone https://github.com/Team-Infinite-Void/transaction-tracker
2. Install the required packages: pip install -r requirements.txt
3. Run the application: python transaction_tracker.py

Installation Instructions

1. Delete the folder containing the Python scripts and databases.