in collaboration with

Softwarica College of IT & E-commerce | Coventry University

Enhanced Honeytoken Authentication System

Dynamic Credential Traps, Intrusion Detection and Two-Factor Authentication

BSc. (Hons) Ethical Hacking and Cybersecurity

E39A

ST4017CMD Introduction to Programming

Submission Date: March 1, 2026

Submitted by:
Name: riggyan parajuli
Coventry ID: 17107444
Student ID: 250325

submitted to:
Abishek Bimali

Abstract

This project, Enhanced Honeytoken Authentication System, is a Python-based desktop application developed to strengthen authentication security through the integration of honeytoken credential traps, two-factor authentication (2FA) using Time-based One-Time Passwords (TOTP), session management, and intrusion detection. The system is built with a graphical user interface (GUI) using Tkinter and employs symmetric encryption via the Fernet algorithm from the cryptography library (Python Cryptographic Authority, 2023) to protect stored user data. When an attacker attempts to log in using a honeytoken credential, the system immediately detects and logs the intrusion with a timestamp and IP address. The program heavily relies on core Python concepts including classes and objects, dictionaries, lists, string operations, file I/O, exception handling, regular expressions, and cryptographic functions. The application also provides login history tracking, statistics dashboards, configurable session timeouts, and CSV report generation, making it a comprehensive security-oriented tool that is both functional and educational.

Table of Contents

List of Figures

1. Introduction

1.1 Background: Nature of Honeytoken Systems

Cybersecurity threats have grown considerably in scale and sophistication, with attackers frequently targeting authentication systems to gain unauthorized access to sensitive data. Among the most effective defensive techniques used in modern security is the concept of a honeytoken — a fabricated credential designed to look legitimate but which, when used, immediately signals an intrusion attempt (Cains et al., 2022). Unlike traditional authentication systems that only reject unauthorized credentials, a honeytoken system turns the attacker's own action into evidence, logging the event and alerting administrators in real time (Spitzner, 2003).

Password-based authentication remains the most common method of access control in digital systems, yet it is also one of the most vulnerable. Attackers employ dictionary attacks, brute force techniques, credential stuffing, and rainbow table attacks to gain entry (Hanna, 2025). Even when passwords are hashed and salted, weak or commonly used passwords can be compromised quickly. Adding a second layer of authentication, such as a time-based one-time password (TOTP), significantly reduces this risk (NIST, 2017).

A significant amount of personal information is publicly available through social media and can be used to guess passwords. Users frequently include names, birthdays, and references to personal interests — all of which are easily harvested by attackers (Kaspersky, 2020). This makes both password complexity and deception-based detection equally important components of a layered security strategy (Conteh & Schmick, 2016).

1.2 Problem Statement and Scope

Modern applications rely on username and password combinations that are often insufficiently protected. A large proportion of data breaches occur due to weak, repeated, or stolen credentials (Verizon, 2023). Traditional login forms give attackers unlimited attempts or provide minimal feedback when intrusions occur. There is also a general lack of affordable, educational tools that demonstrate how honeytoken-based intrusion detection works in practice (ICLG, 2025).

The scope of this project is to build a Python desktop application that:

- Allows real users to register and log in securely using password hashing and TOTP-based 2FA
- Enables administrators to define honeytoken credentials that, when used, trigger security alerts
- Logs all intrusion attempts with timestamps and local IP addresses
- Provides session management, login history, statistics, and exportable reports
- Encrypts all stored data using the Fernet symmetric encryption scheme

1.3 Major Objectives and Goals

1. To design and implement a GUI-based authentication system using Python and Tkinter
2. To implement honeytoken credential traps that detect and log unauthorized access attempts
3. To provide two-factor authentication using TOTP generated via HMAC-SHA1
4. To encrypt all stored user and configuration data using the Fernet encryption algorithm
5. To demonstrate core Python programming concepts including OOP, file handling, and exception handling

2. Literature Review

Honeytokens as a cybersecurity concept were first popularized by Spitzner (2003) (Spitzner, 2003), who described them as digital traps designed to detect unauthorized activity. Unlike honeypots, which simulate entire systems, honeytokens are individual data artifacts — such as fake usernames, files, or API keys — that have no legitimate use. When accessed, they provide a reliable signal that an intrusion has occurred. Spitzner's foundational work established that deception-based detection is effective precisely because it requires no prior knowledge of the attack vector.

Research exploring honeytoken deployment within enterprise identity and access management frameworks showed that honeytokens placed within credential stores successfully detected simulated insider threats and external attacks with a false positive rate near zero (Cains et al., 2022). This finding supports the use of honeytokens as a complementary layer alongside traditional authentication controls.

The use of time-based one-time passwords (TOTP) as a second authentication factor is standardized under RFC 6238 (M'Raihi et al., 2011), which extends the earlier HOTP standard. NIST Special Publication 800-63B recommends TOTP-based authentication as an effective method for multi-factor authentication, citing its resistance to phishing and replay attacks (NIST, 2017). TOTP generates short-lived code derived from a shared secret and the current Unix time, making it impossible to reuse.

Password hashing using PBKDF2 is recommended by security standards organizations as a secure method for storing user credentials (Kaliski, 2000). The function applies a pseudorandom function iteratively, which significantly increases the time required to conduct brute force or dictionary attacks against a compromised password database (OWASP, 2021).

The Fernet symmetric encryption scheme, provided by Python's cryptography library, uses AES-128-CBC with HMAC-SHA256 for authenticated encryption. Fernet guarantees that a message encrypted using a given key cannot be decrypted or tampered with without that key,

making it well suited for protecting application data files stored on disk (Python Cryptographic Authority, 2023).

Research into GUI-based security tools has shown that usability plays a critical role in adoption. A tool that is difficult to understand or operate is often abandoned regardless of its technical capability (Yee, 2004). The Tkinter framework, as Python's built-in GUI toolkit, provides a pragmatic solution for building accessible interfaces for educational security applications without additional dependencies.

Session management vulnerabilities, including session hijacking and fixation, are consistently listed among the OWASP Top Ten risks (OWASP, 2021). Implementing automatic session timeouts after a configurable period of inactivity is a recognized countermeasure that limits the window of opportunity for attackers who gain physical or remote access to an active session. Usability in this context is equally important — users are more likely to adopt secure practices when tools are well designed (Furnell & Clarke, 2012).

3. Methods Applied

3.1 Algorithm of the Program

The program is structured around a single Python class, EnhancedHoneytokenAuthTool, which encapsulates all functionality. The following pseudocode outlines the main algorithmic flow:

Step 1: Application Initialization

1. Load or create the Fernet master encryption key from disk
2. Initialize the Fernet cipher using the loaded key
3. Load encrypted data files: users, honeytokens, login history, and settings
4. Set session variables (current_user, session_start_time) to None
5. Call _create_ui() to launch the Tkinter GUI

Step 2: User Registration (_handle_register)

1. Accept username, password, and confirm password from entry fields
2. Validate that all fields are filled and passwords match
3. Check username does not already exist in users or honeytokens
4. Evaluate password strength score using _check_password_strength()
5. Generate a random OTP secret using secrets.token_hex(20)
6. Hash password using PBKDF2-SHA256 with a random salt via _hash_password()
7. Store user data dictionary and save to encrypted file

Step 3: Login and Honeytoken Check (_handle_login)

1. Accept username and password from the login form
2. Check if username exists in the honeytokens dictionary
3. If found and password matches, call _log_intrusion() and show security alert
4. If username is not in legitimate users, log failed attempt and show error
5. Verify password using _verify_password() against stored PBKDF2 hash
6. If valid, set current_user and proceed to the OTP verification screen

Step 4: OTP Generation and Verification

The _generate_otp() function implements a simplified TOTP algorithm (M'Raihi et al., 2011). It divides the current Unix timestamp by 30 to obtain a time counter, encodes it as an 8-byte big-endian integer, and applies HMAC-SHA1 with the user's stored secret. A dynamic truncation extracts a 4-byte segment from the hash output and reduces it modulo 1,000,000 to produce a 6-digit code. During verification, both the current counter and the previous counter are checked to allow for minor clock drift.

Step 5: Session Management and Timeout

After successful login, _start_session_timeout() schedules a callback using Tkinter's after() method. If the user does not interact within the configured timeout period (default 15 minutes), _handle_session_timeout() is triggered, which logs the user out automatically. This approach follows OWASP session management best practices (OWASP, 2021).

Step 6: Intrusion Logging (_log_intrusion)

When a honeytoken credential is used, the function records the timestamp, the honeytoken username, and the local IP address obtained via socket.gethostbyname(). The log entry is appended to a plain text file (intrusion_log.txt) and the intrusion counter in the statistics dictionary is incremented.

3.2 Procedure Explanation

Encryption and Data Storage

All application data — including user credentials, honeytoken details, login history, and settings — is stored in encrypted binary files using Fernet symmetric encryption (Python Cryptographic Authority, 2023). On the first run, the application generates a 256-bit key and writes it to master.key. This key is loaded on every subsequent run to initialize the Fernet cipher. Data is serialized to JSON before encryption and deserialized after decryption.

```python
    def _generate_otp(self, secret, counter=None):
        """Generate TOTP (Time-based OTP)"""
        if counter is None:
            counter = int(time.time() / 30)

        counter_bytes = counter.to_bytes(8, byteorder='big')
        hmac_hash = hmac.new(secret.encode(), counter_bytes, hashlib.sha1).digest()

        offset = hmac_hash[-1] & 0x0f
        code = (
            (hmac_hash[offset] & 0x7f) << 24 |
            (hmac_hash[offset + 1] & 0xff) << 16 |
            (hmac_hash[offset + 2] & 0xff) << 8 |
            (hmac_hash[offset + 3] & 0xff)
        )

        otp = str(code % 1000000).zfill(6)
        return otp
```

```python
    def _hash_password(self, password):
        """Hash password with salt"""
        salt = secrets.token_hex(16)
        pwdhash = hashlib.pbkdf2_hmac('sha256', password.encode(), salt.encode(), 100000)
        return f"{salt}${pwdhash.hex()}"

    def _verify_password(self, stored_password, provided_password):
        """Verify password against hash"""
        salt, pwdhash = stored_password.split('$')
        test_hash = hashlib.pbkdf2_hmac('sha256', provided_password.encode(), salt.encode(), 100000)
        return test_hash.hex() == pwdhash
```

Figure 1&2: VS Code — _generate_otp and _hash_password Functions

The _hash_password() function generates a 16-byte random salt using secrets.token_hex(16), then applies PBKDF2-HMAC with SHA-256 and 100,000 iterations. The resulting hash and salt are stored together in the format salt$hash, which allows the verification function to reconstruct the hash for comparison.

Password Strength Checker

The _check_password_strength() function evaluates a password against five criteria: minimum length of 8 characters, presence of lowercase letters, uppercase letters, digits, and special characters. Each satisfied criterion adds one point to a score out of five. The score maps to a label ranging from Very Weak to Very Strong, and a list of unmet requirements is returned as feedback. In the registration form, this feedback updates in real time as the user types (Hanna, 2025).

```python
    def _check_password_strength(self, password):
        """Check password strength and return score and feedback"""
        score = 0
        feedback = []

        if len(password) >= 8:
            score += 1
        else:
            feedback.append("At least 8 characters")

        if re.search(r"[a-z]", password):
            score += 1
        else:
            feedback.append("Lowercase letter")

        if re.search(r"[A-Z]", password):
            score += 1
        else:
            feedback.append("Uppercase letter")

        if re.search(r"\d", password):
            score += 1
        else:
            feedback.append("Number")

        if re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
            score += 1
        else:
            feedback.append("Special character")

        strength = ["Very Weak", "Weak", "Fair", "Good", "Strong", "Very Strong"]
        return score, strength[score], feedback

```

Figure 3: VS Code — _check_password_strength and evaluate_password Functions

Honeytoken Management

The honeytoken manager screen allows the user to define fake credentials by entering a fake username and password. These are stored in the same encrypted format as legitimate users, except in a separate honeytokens dictionary. During login, the system checks the honeytokens dictionary before the user's dictionary. If a match is found, an intrusion is logged immediately, regardless of whether the attacker knows the correct password or not (Spitzner, 2003).

```python
    def _log_intrusion(self, username, ip=None):
        """Log intrusion attempt with enhanced details"""
        if ip is None:
            ip = self._get_local_ip()

        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_entry = f"[{timestamp}] ⚠ INTRUSION DETECTED - Honeytoken: '{username}' | IP: {ip} | Status: BLOCKED\n"

        with open(self.intrusion_log_file, "a", encoding="utf-8") as f:
            f.write(log_entry)

        self.stats["intrusions_detected"] += 1
        return log_entry
```

```python
    def _log_login_attempt(self, username, success=True, reason=""):
        """Log all login attempts"""
        entry = {
            "username": username,
            "timestamp": datetime.now().isoformat(),
            "success": success,
            "ip": self._get_local_ip(),
            "reason": reason
        }
        self.login_history.append(entry)
        self._save_login_history()

        if success:
            self.stats["total_logins"] += 1
        else:
            self.stats["failed_attempts"] += 1

```

Figure 4&5: VS Code — _log_intrusion and _handle_login Functions

Login History and Reporting

Every login attempt, whether successful or failed, is recorded to an encrypted login history list. Each entry stores the username, timestamp, success status, IP address, and reason for failure if applicable. This history can be exported to CSV format from the export menu. A full system report can also be generated, summarizing total users, honeytokens, login counts, and timestamps.

Tools and Technologies Used

- Python 3.x — Core programming language
- Tkinter — Built-in Python GUI framework for the graphical interface
- cryptography (Fernet) — Symmetric encryption of stored data files
- hashlib / hmac — PBKDF2-SHA256 password hashing and TOTP generation
- secrets — Cryptographically secure random number generation
- re — Regular expressions for password pattern checking
- socket — Local IP address retrieval for intrusion logs

- csv / json — Data serialization and export
- VS Code — Code editing and testing environment

4. Analysis of Results Obtained

Result 1: Login Screen and Application Launch

When the application is launched, the main login screen is displayed. It shows input fields for username and password, buttons for login and registration, and additional options for managing honeytokens, viewing statistics, exporting reports, and adjusting settings. The system IP address is shown at the bottom, which serves as useful contextual information for administrators.

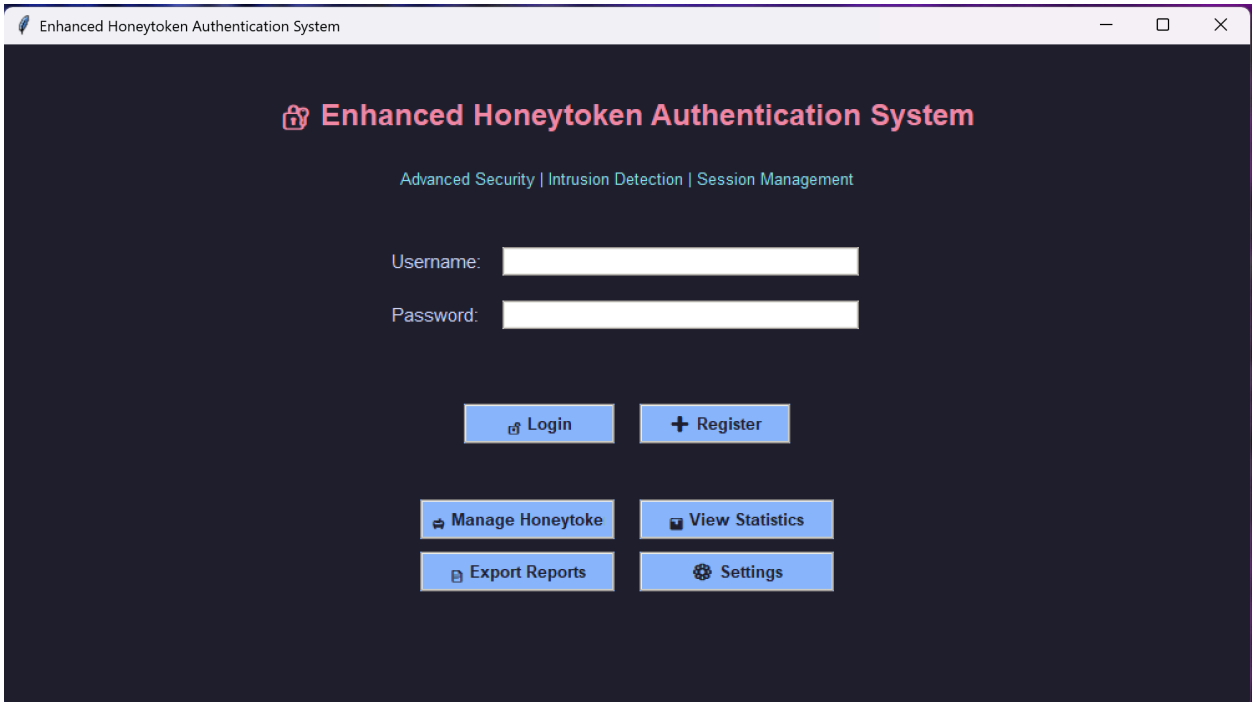


Figure 6: Login Screen of the Application

Result 2: User Registration with Password Strength Feedback

The registration screen allows new users to create an account. As the user types their password, the strength indicator updates in real time, showing a progress bar and a label ranging from Very Weak to Very Strong, along with a list of missing requirements (Hanna, 2025). If the password scores below 3 out of 5, the system prompts the user to confirm, encouraging stronger choices.

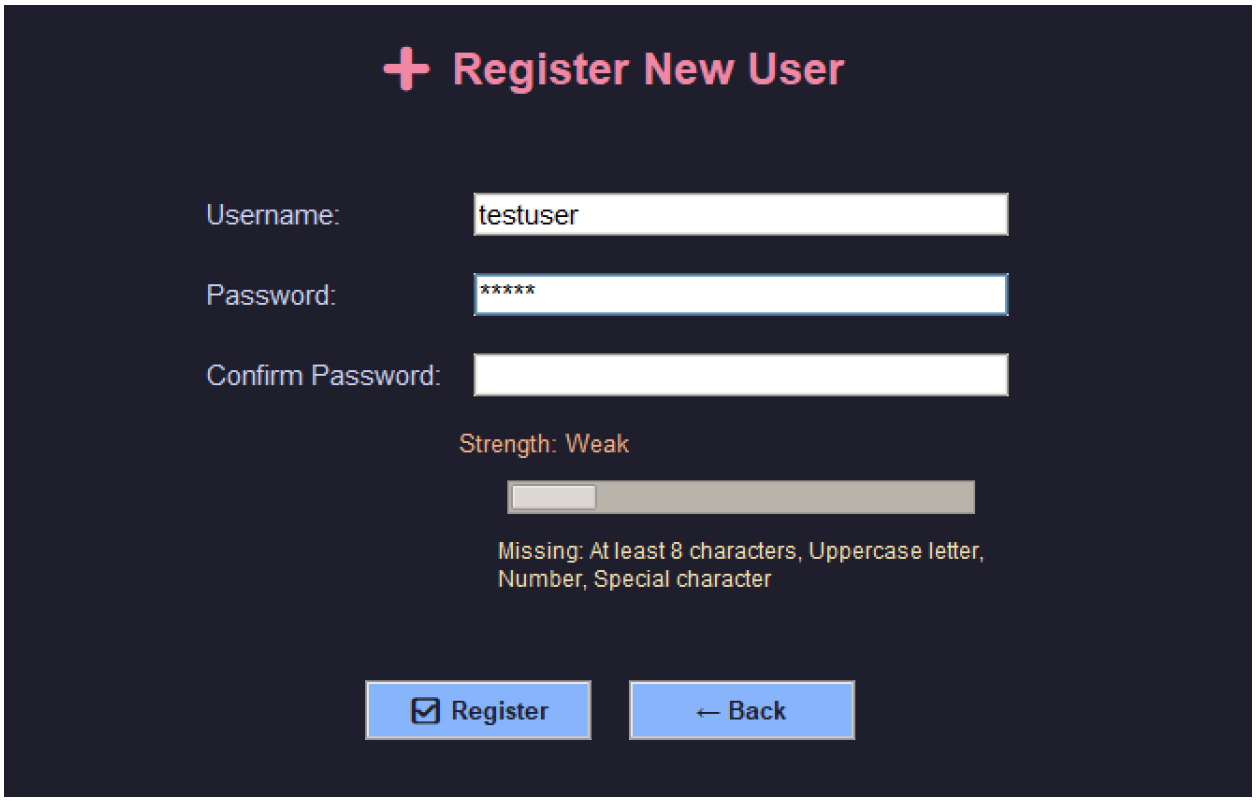
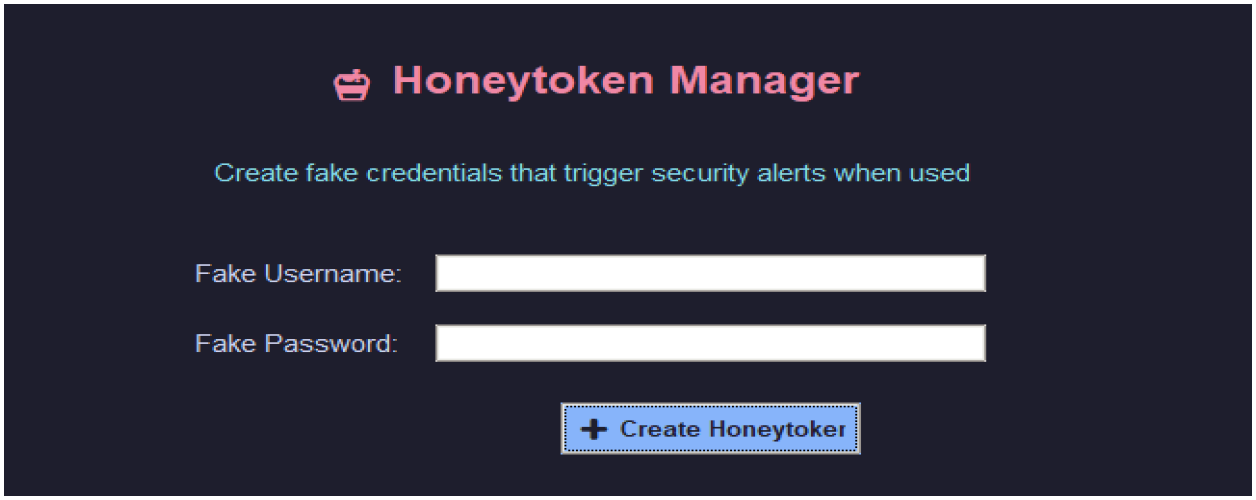


Figure 7: User Registration Screen with Password Strength Indicator

Result 3: Honeytoken Manager

The honeytoken manager allows administrators to create fake credentials by specifying a username and password. Any credential entered here becomes a trap. The list of active honeytokens is displayed with their creation dates and trigger counts (Spitzner, 2003). Honeytokens can be deleted from this screen. This feature is accessible without logging in, making it easy to configure traps before deployment.



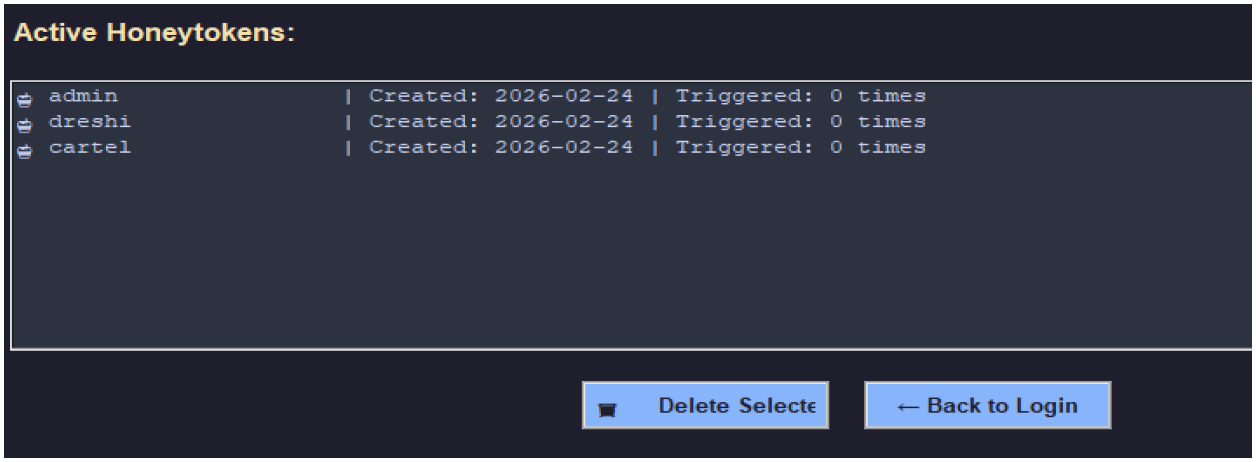


Figure 8&9: Honeytoken Manager Screen

Result 4: Two-Factor Authentication Screen

After entering valid credentials, the user is directed to the OTP verification screen. Clicking the Show My OTP Code button reveals a 6-digit code that refreshes every 30 seconds, with a countdown timer. Because the code is generated locally using a shared secret and the current time, it cannot be intercepted or reused (M'Raihi et al., 2011).

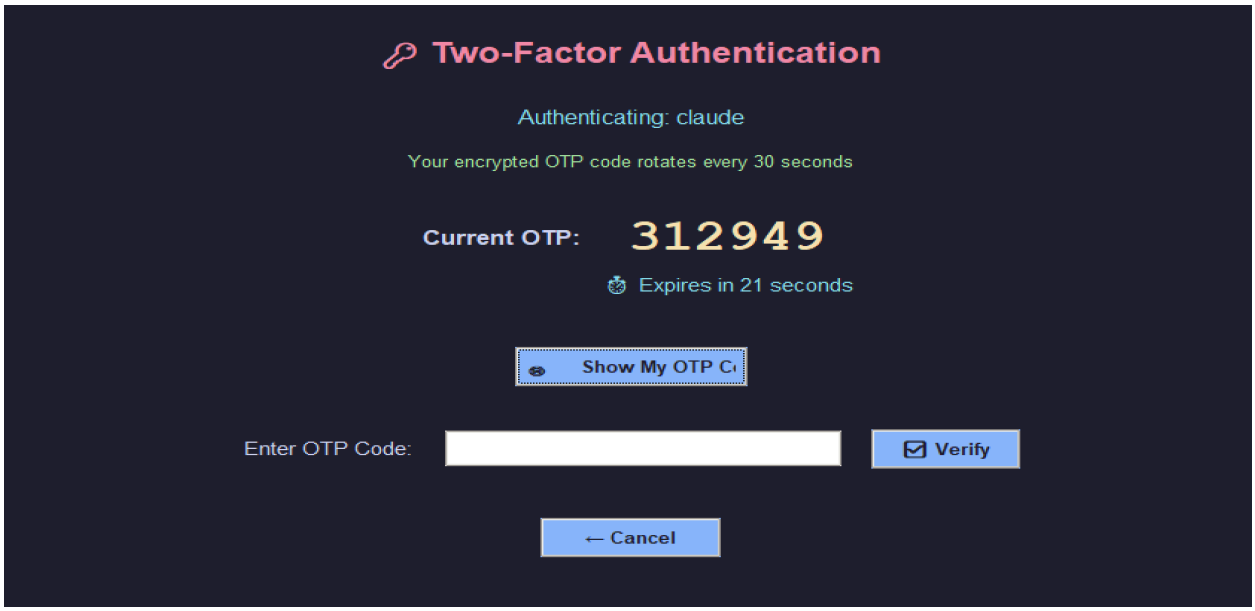


Figure 10: Two-Factor Authentication (OTP) Screen

Result 5: User Dashboard

Upon successful OTP verification, the user is taken to their personal dashboard. The dashboard displays account statistics including creation date, total login count, and last login time. Buttons provide access to the user's profile, login history, intrusion log, system statistics, password change dialog, and logout. The session start time and auto-logout countdown are shown at the top.

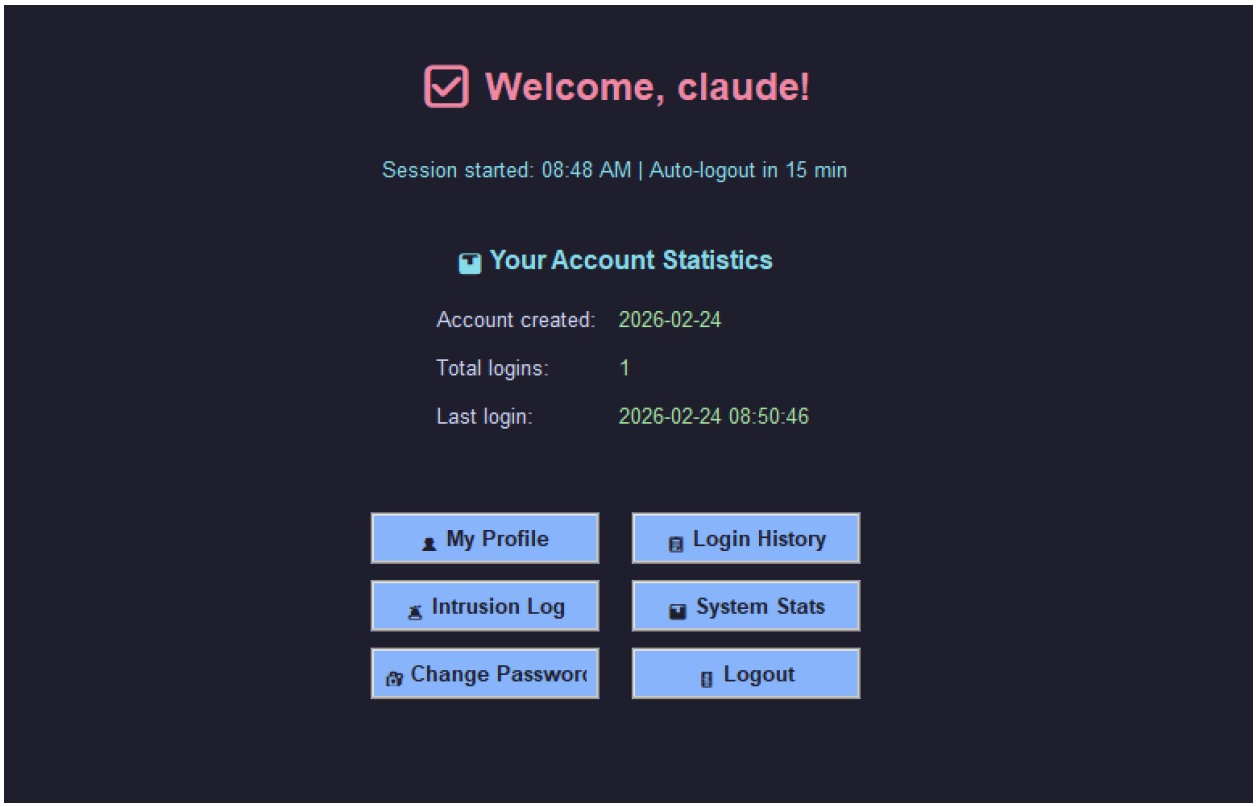


Figure 11: User Dashboard after Successful Login

Result 6: Honeytoken Triggered — Intrusion Alert

When a user attempts to log in with a honeytoken username and its corresponding password, the system immediately detects the intrusion. A security alert dialog is shown, displaying the timestamp, the honeytoken username, and the IP address of the device. The event is written to the intrusion log file. This confirms the honeytoken detection mechanism works in real time (Spitzner, 2003).

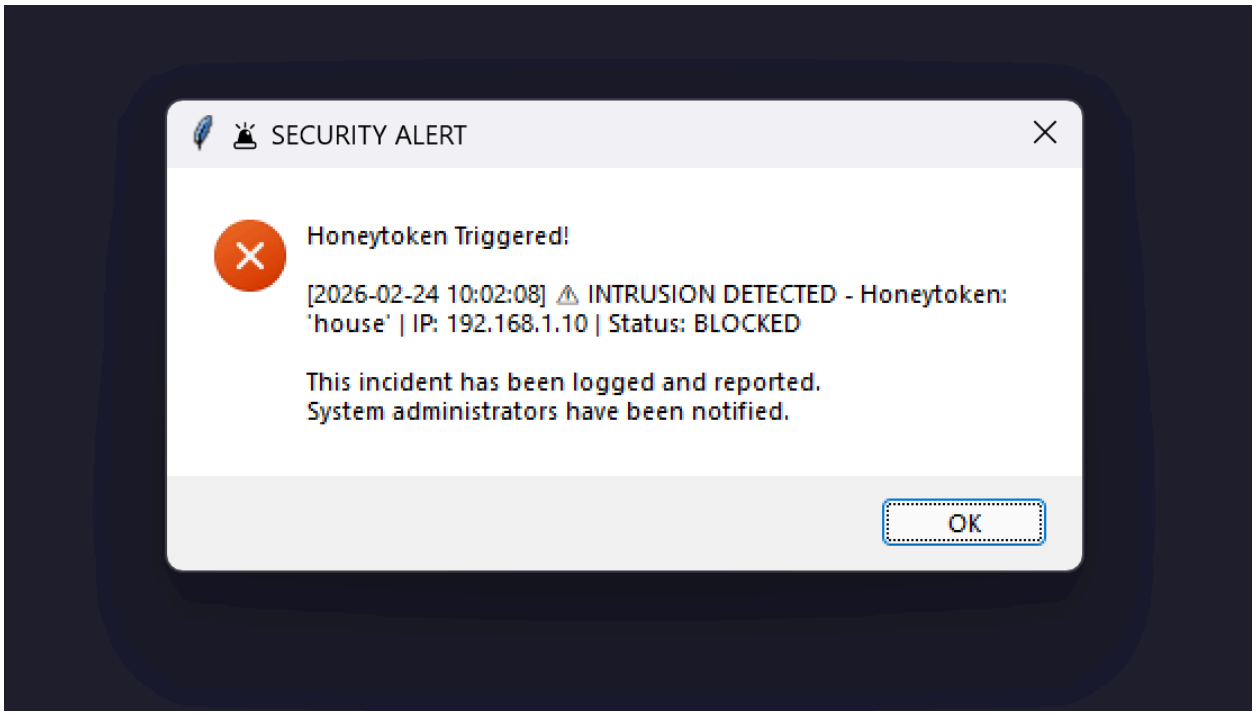


Figure 12: Intrusion Alert Triggered by Honeytoken

Result 7: Intrusion Detection Log

The intrusion log screen displays all recorded honeytoken trigger events in chronological order. Each entry includes the date, time, the honeytoken credential used, and the IP address of the attempt. This log is accessible from the user dashboard and provides a clear audit trail of suspected intrusion attempts.

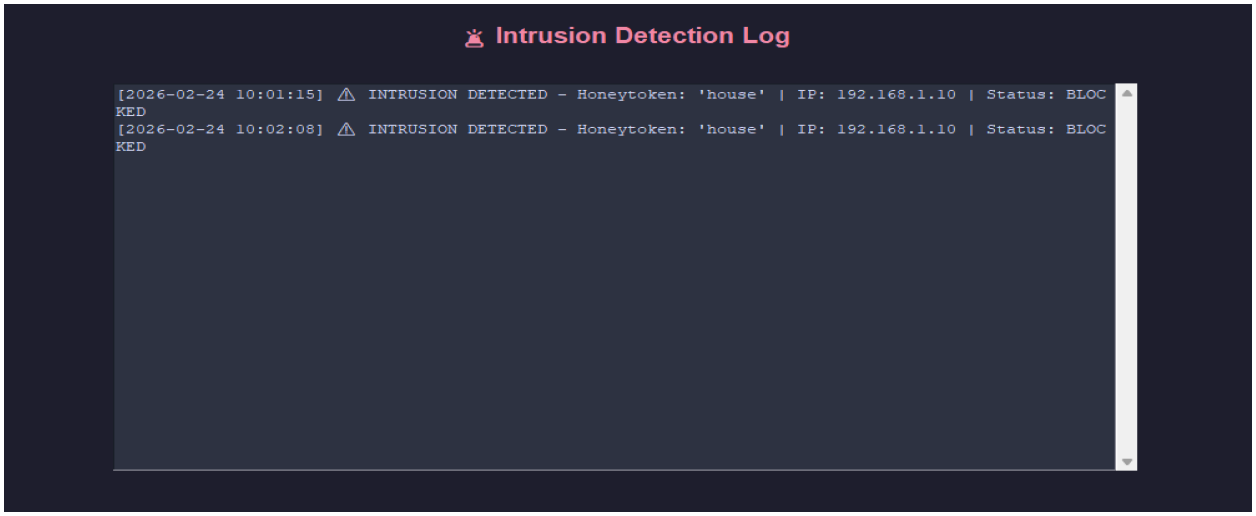


Figure 13: Intrusion Detection Log Screen

5. Discussion and Concluding Statements

The Enhanced Honeytoken Authentication System successfully demonstrates how multiple security concepts can be combined into a single, cohesive Python application. The system goes beyond basic username and password authentication by incorporating honeytoken traps that catch intrusion attempts (Spitzner, 2003), TOTP-based two-factor authentication that resists replay attacks (M'Raihi et al., 2011), and Fernet encryption that protects stored data at rest (Python Cryptographic Authority, 2023). Each of these elements addresses a distinct and realistic attack vector.

One important observation from testing is that the honeytoken detection is reliable even when an attacker guesses the correct honeytoken password. The detection occurs at the credential-lookup stage, before any session is established. This means that even a sophisticated attacker who obtains the correct password for a honeytoken account will still trigger the alert, demonstrating the strength of the deception-based approach.

The TOTP implementation, while simplified compared to a full RFC 6238-compliant authenticator application, captures the essential mechanism of time-based code generation (M'Raihi et al., 2011). By allowing for the previous 30-second window in addition to the current one, the system accommodates minor clock differences without significantly reducing security.

The application also succeeds in keeping all stored data encrypted and inaccessible without the master key. However, a limitation of the current implementation is that the master.key file is stored alongside the application. In a production deployment, this key would need to be stored more securely — for example, using hardware security modules or key management services (ICLG, 2025).

In conclusion, the project meets all its stated objectives. It provides a functional, GUI-based authentication system with honeytoken detection, two-factor authentication, encrypted storage, session management, and reporting. The application is practically relevant to the field of ethical hacking and cybersecurity, and demonstrates a clear understanding of the Python programming concepts covered in the Introduction to Programming module.

6. Further Enhancements and Recommendations

1. The honeytoken system could be extended to support email or SMS notifications when a honeytoken is triggered, alerting administrators immediately even when they are not actively monitoring the application.
2. The TOTP implementation could be upgraded to full RFC 6238 compliance and integrated with standard authenticator applications such as Google Authenticator or Authy, improving compatibility and usability.
3. The master encryption key should be stored using a more secure mechanism, such as environment variables, an operating system keychain, or a dedicated secrets management service, rather than a local file.
4. The login history and intrusion log could be connected to a database backend such as SQLite, rather than encrypted flat files, to support more efficient querying and filtering of large volumes of data.
5. A brute force protection mechanism could be implemented to temporarily lock an account after a configurable number of failed login attempts, providing an additional layer of defence against automated attacks.

References

Cains, M. G., Bird, L., Nurse, J. R. C., & Hall, A. J. (2022). Defining cyber security and cyber resilience: Unpacking the relationship between two maturing concepts. Computers & Security, 121, Article 102828. https://doi.org/10.1016/j.cose.2022.102828

Conteh, N. Y., & Schmick, P. J. (2016). Cybersecurity: Risks, vulnerabilities and countermeasures to prevent social engineering attacks. International Journal of Advanced Computer Research, 6(23), 31–38. https://doi.org/10.19101/IJACR.2016.623006

Furnell, S., & Clarke, N. (2012). Power to the people? The evolving recognition of human aspects of security. Computers & Security, 31(8), 983–988. https://doi.org/10.1016/j.cose.2012.08.004

Hanna, K. T. (2025, February 25). What is a strong password? TechTarget. https://www.techtarget.com/searchenterprisedesktop/definition/strong-password

ICLG. (2025). Cybersecurity laws and regulations Nepal 2025. International Comparative Legal Guides. https://iclg.com/practice-areas/cybersecurity-laws-and-regulations/nepal

Kaliski, B. (2000). PKCS #5: Password-based cryptography specification version 2.0 (RFC 2898). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc2898

Kaspersky. (2020). Why using personal information in your password is a risk. Kaspersky Resource Center. https://www.kaspersky.com/resource-center/threats/password-security-risks

M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., & Ranen, O. (2011). TOTP: Time-based one-time password algorithm (RFC 6238). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc6238

National Institute of Standards and Technology. (2017). Digital identity guidelines: Authentication and lifecycle management (NIST Special Publication 800-63B). U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-63b

OWASP. (2021). OWASP top ten. Open Web Application Security Project. https://owasp.org/www-project-top-ten/

Python Cryptographic Authority. (2023). Cryptography: Fernet (symmetric encryption). PyCA. https://cryptography.io/en/latest/fernet/

Spitzner, L. (2003). Honeypots: Catching the insider threat. In Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003) (pp. 170–179). IEEE. https://doi.org/10.1109/CSAC.2003.1254322

Verizon. (2023). 2023 data breach investigations report. Verizon Business. https://www.verizon.com/business/resources/reports/dbir/

Yee, K.-P. (2004). Aligning security and usability. IEEE Security & Privacy, 2(5), 48–55. https://doi.org/10.1109/MSP.2004.64

Video link :

Github link :