Cyber Security: Privacy-Preserving Machine Learning: Secure Client-Server Communication using AES Encryption.

Calvin Ssendawula
Adams State University
208 Edgemont Blvd. Unit 890
Alamosa. Colorado 81101
ssendawulac@adams.edu

***Abstract*—The transmission of sensitive data over insecure networks poses a significant challenge in the modern digital era, with threats such as data breaches and eavesdropping becoming increasingly prevalent. To address this issue, this project implements a lightweight, custom system for secure client-server communication using AES encryption. AES, a symmetric encryption algorithm, was chosen for its efficiency, security, and suitability for real-time data protection.**

Testing and implementation confirmed the systems' ability to securely transmit and decrypt sensitive client data without reliance on external tools or HTTPS. The results demonstrate the practical application of AES encryption in resource-constrained environments, making it an effective solution for secure communication over insecure networks. This project serves as a foundation for future enhancements, such as integrating HTTPS, dynamic salts, or broader encryption use cases.

***Keywords*— AES, encryption, confidentiality, secure communication, flask, python**

I. INTRODUCTION

I an era of increasing digital dependency, the security of transmitted data has become a critical concern. Data breaches and eavesdropping incidents are on the rise, compromising sensitive information and eroding trust in digital platforms. Whether in financial transactions, medical records, or prevent unauthorized access and ensure privacy. Modern networked applications must address these challenges by adopting robust mechanisms for secure communication, especially in environments where traditional transport-layer security like HTTPS may not be feasible. Symmetric encryption is a widely used technique for securing data, where the same key is used for both encryption and decryption. Among various symmetric encryption algorithms, Advanced Encryption Standard (AES) stands out due to its proven security, speed, and efficiency. AES supports multiple key lengths (128, 192, or 256 bits), providing flexibility and enhanced security for various applications. Its compatibility with diverse platforms and resistance to most known cryptographic attacks make it a popular choice for securing sensitive data in both small-scale and large-scale systems.

This project aims to design and implement a lightweight, custom system for securely transmitting sensitive data over an insecure network. By leveraging AES encryption and a Flask-based server-client architecture, the solution demonstrates how sensitive information can be encrypted on the client side, securely transmitted to the server, and decrypted for further use—all without relying on external debugging tools or traditional HTTPS infrastructure. This approach highlights the practical implementation of strong encryption techniques in resource-constrained or controlled environments.

II. RELATED WORK

In Secure communication is a cornerstone of modern networked systems, with established protocols like HTTPS, TLS, and RSA providing robust methods to ensure data confidentiality, integrity, and authenticity. HTTPS, for example, secures communication by leveraging TLS (Transport Layer Security), which combines symmetric encryption for data protection and asymmetric methods like RSA for secure key exchange. While widely adopted, HTTPS requires certificate management and introduces overhead, making it less ideal for lightweight or localized applications. Similarly, TLS and SSL protocols are powerful but resource-intensive due to their reliance on trusted Certificate Authorities (CAs) and the computational cost of asymmetric encryption during handshakes. RSA, though highly secure for key exchanges and digital signatures, is computationally expensive and less suited for encrypting large data payloads.

In specific scenarios, lightweight encryption methods like AES (Advanced Encryption Standard) can be more practical and efficient. Controlled environments, such as IoT systems or intranet-based applications, often prioritize speed and simplicity over external certificate validation. For example, AES encryption with pre-shared secrets eliminates the need for complex public key infrastructure, making it ideal for secure communication in resource-constrained or localized setups. Additionally, AES is particularly well-suited for devices with limited computational power due to its efficiency and minimal overhead, as opposed to the resource demands of RSA or TLS.

PSEUDO CODE: CLIENT SIDE

```
# Step 1: Generate encryption key
key = PBKDF2HMAC(password, salt)

# Step 2: Encrypt data
iv = generate_random_iv()
ciphertext = AES_encrypt(key, plaintext, iv)

# Step 3: Transmit data
send_json = {
   "iv": iv.hex(),
   "ciphertext": ciphertext.hex()
}
```

```
response = send_to_server(send_json)

# Print server's response
print(response)
```

PSEUDO CODE: SERVER SIDE

```
# Step 1: Extract data
iv = get_iv_from_request()  # Extract IV (hex) and convert to bytes
ciphertext = get_ciphertext_from_request()  # Extract ciphertext (hex) and convert to bytes

# Step 2: Derive key and decrypt data
key = PBKDF2HMAC(password, salt)
plaintext = AES_decrypt(key, iv, ciphertext)

# Step 3: Return decrypted data
response = {
   "message": "Decryption successful",
   "data": plaintext
}
return response
```

III. APPROACH/ALGORITHM

In focuses on implementing secure client-server communication using Advanced Encryption Standard (AES) in Cipher Feedback (CFB) mode. The design leverages symmetric encryption for its efficiency and practicality, especially in controlled environments. The process is divided into two main sections: client-side and server-side operations. Each step emphasizes simplicity, security, and adaptability to ensure sensitive data is transmitted securely over an insecure network. On the client side, the first step involves generating a robust encryption key. To achieve this, the PBKDF2HMAC algorithm is used to derive a cryptographic key from a pre-shared password and a salt. This process enhances security by ensuring the key is both strong and unique to the session. The use of a high iteration count in PBKDF2HMAC adds computational complexity, making it significantly harder for attackers to brute force the key. After the key is generated, the plaintext data is encrypted using AES in CFB mode. This encryption mode is particularly well-suited for streaming data or variable-length messages as it avoids the need for padding. A unique Initialization Vector (IV) is generated randomly for each encryption session. The IV is essential for maintaining security because it ensures that identical plaintexts encrypted with the same key produce different ciphertexts. This randomness helps protect the data from pattern analysis. The encrypted data, along with the IV, is then formatted into a JSON object for transmission. The IV and ciphertext are converted to hexadecimal strings to ensure compatibility with the HTTP protocol and JSON format. The JSON object is sent to the server using an HTTP POST request. This straightforward approach simplifies the client-server interaction while ensuring the secure delivery of encrypted data. On the server side, the process begins with receiving the JSON payload. The server extracts the IV and ciphertext from the payload and then independently derives the symmetric encryption key using the same PBKDF2HMAC process. Since the server and client share the same password and salt, they can generate identical keys without exchanging them over the network, reducing the risk of key compromise. Decryption is performed using AES in CFB mode. The server decrypts the ciphertext using the derived key and the IV extracted from the payload. This process reverses the encryption performed on the client side, converting the data back to its original plaintext form. The server ensures the decrypted data matches the expected format and content to validate its integrity. This verification step is critical for ensuring the communication process was not tampered with during transmission. To enhance reliability, the server sends a response back to the client confirming the successful decryption of the data. This response typically includes the decrypted plaintext, allowing the client to verify the accuracy of the operation. This feedback mechanism provides a layer of assurance that the transmission was both secure and successful. The choice of AES in CFB mode simplifies implementation while maintaining a high level of security. Unlike other modes, CFB avoids padding-related complexities and is resilient to certain types of attacks, making it an ideal choice for this project. Additionally, the use of a random IV adds a dynamic layer to the encryption process, further strengthening security.

IV. RESULTS

The results of this project demonstrate the effectiveness of AES encryption in securely transmitting data between a client and a server over an insecure network. During the testing phase, plaintext messages were encrypted on the client side, transmitted securely, and successfully decrypted on the server side. For example, a plaintext message like *"Secure Communication Test"* was encrypted into a ciphertext such as *"a3b8c1..."* (hexadecimal format) and then accurately decrypted back into the original plaintext on the server.

Testing encompassed a variety of plaintext lengths to ensure the system's robustness. Short messages, like single words, and longer text bodies, such as paragraphs or JSON-formatted strings, were transmitted without issues. The system successfully encrypted and decrypted both cases, proving its capability to handle varying data sizes. For example, a 500-character message was encrypted and transmitted without noticeable latency, ensuring the solution is practical for moderate-sized payloads. Edge case testing included handling plaintext containing special characters, numbers, and mixed encoding formats, such as Unicode. Messages like *"P@$$w0rd#123 🚀"* were encrypted and decrypted correctly, showing the system's ability to manage complex character sets. This testing validated the solution's versatility and reliability in real-world scenarios where user input or sensitive data might contain special symbols. The random generation of an Initialization Vector (IV) for each encryption session ensured that even identical plaintext messages resulted in unique ciphertexts. For instance, encrypting *"Hello World"* multiple times produced different encrypted outputs, verifying that the IV-based approach prevents patterns that could be exploited by attackers. This feature strengthens the overall

security of the system by making pattern analysis infeasible. Performance testing revealed minimal overhead during encryption and decryption, even with longer plaintexts or high iteration counts in the key derivation process. The use of AES in Cipher Feedback (CFB) mode proved advantageous for streaming data and variable-length messages, eliminating the need for padding and maintaining efficient encryption and decryption operations.

```
PS C:\Users\mukas\OneDrive\Desktop\CYBERSECURITY CLASS FILES\SECURE DATA TRANSMISSION> python server.p
y
 * Serving Flask app 'server'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
 server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 173-047-739
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\client.py', reloading
127.0.0.1 - - [11/Dec/2024 23:17:21] "POST /receive HTTP/1.1" 200 -
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 173-047-739
```

```
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\kdf\\pbkdf2.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\kdf\\__init__.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\__init__.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\hashes.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\backends\\__init__.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\ciphers\\__init__.py', reloadin
g
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\ciphers\\base.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\ciphers\\modes.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\_cipheralgorithm.py', reloading
 * Detected change in 'C:\\Users\\mukas\\OneDrive\\Desktop\\CYBERSECURITY CLASS FILES\\SECURE DATA TRA
NSMISSION\\venv\\Lib\\site-packages\\cryptography\\hazmat\\primitives\\ciphers\\algorithms.py', reload
ing
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 173-047-739
```

```
.... .... 1... = Push: Set
.... .... .0.. = Reset: Not set
.... .... ..0. = Syn: Not set
.... .... ...0 = Fin: Not set
TCP Flags: ·······AP···
Window: 501
Calculated window size: 64128
Window size scaling factor: 128
Checksum: 0x7242 [unverified]
Checksum Status: Unverified
Urgent Pointer: 0
Timestamps
Time since first frame in this TCP stream: 0.056294000 seconds
Time since previous frame in this TCP stream: 0.000000000 seconds
SEQ/ACK analysis
iRTT: 0.027309000 seconds
Bytes in flight: 2772
Bytes sent since last PSH flag: 2772
TCP payload (1386 bytes)
TCP segment data (1386 bytes)
```

V. CONCLUSION

This project addressed the critical issue of secure data transmission over inherently insecure networks, where sensitive information is often at risk of being intercepted or manipulated. The primary objective was to create a lightweight and efficient system for secure communication, leveraging the Advanced Encryption Standard (AES) to ensure confidentiality and data integrity. By focusing on a controlled client-server setup, the project offered a practical solution for secure data exchange in scenarios where traditional methods like HTTPS may be excessive or impractical. The implemented solution combined AES encryption in Cipher Feedback (CFB) mode with Python's robust cryptography libraries and a Flask-based server for data transmission. The client securely encrypted plaintext data using AES with a randomly generated Initialization Vector (IV) and transmitted the ciphertext alongside the IV to the server. On the server side, the system decrypted the data and validated its integrity before sending a response. This end-to-end process successfully ensured that data remained protected throughout the transmission.

The project demonstrated tangible results by encrypting and decrypting diverse types of data, including short messages, long strings, and special characters, without compromising security or performance. The use of PBKDF2HMAC for key derivation ensured that the encryption keys were derived securely from passwords and salts, further enhancing the system's resilience against brute force attacks. Testing confirmed that the system could handle a variety of edge cases while maintaining reliability and efficiency. One of the key achievements of this project was its ability to balance security with simplicity. By avoiding the complexities associated with certificate management and relying on pre-shared secrets for key generation, the system highlighted the feasibility of lightweight encryption methods in controlled environments. This makes the solution particularly suitable for applications such as IoT devices, localized systems, or prototypes where implementing traditional security protocols may not be practical. The successful implementation of AES encryption also highlighted its practical applications for modern secure communication needs. The use of Flask as the server framework and Python as the programming language facilitated rapid development and integration, showcasing the flexibility of these tools in building secure systems. The project stands as a testament to the effectiveness of combining well-established cryptographic principles with accessible development frameworks. While the project achieved its goals, it also identified areas for future enhancement. For example, integrating dynamic salts for key derivation, adding support for file encryption, or incorporating additional authentication layers would further strengthen the system's security and expand its potential applications. These improvements could make the system more robust and adaptable to a broader range of use cases.

REFERENCES:

1. **NIST**: "Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publication 197 (FIPS 197)*, U.S. Department of Commerce/National Institute of Standards and Technology, 2001. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.197

2. **Python Cryptography Library Documentation**: "cryptography 41.0.1: Encrypting with AES," *Cryptography.io*, Python Software Foundation. [Online]. Available: https://cryptography.io/en/latest/

3. **Flask Documentation**: "Flask: Full-featured Python web framework," Pallets Projects. [Online]. Available: https://flask.palletsprojects.com/

4. **RFC 2898**: "PKCS #5: Password-Based Cryptography Specification Version 2.0," B. Kaliski, RSA Laboratories, September 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2898

5. **Daemen, J., & Rijmen, V.**: *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin, Germany: Springer-Verlag, 2002.

6. **Katz, J., & Lindell, Y.**: *Introduction to Modern Cryptography*, 2nd ed. Boca Raton, FL, USA: CRC Press, 2014.

7. **O'Reilly Media**: *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*. Python for practical cryptography applications.

8. **OWASP Foundation**: "Introduction to Cryptography," OWASP Cheatsheets. [Online]. Available: https://cheatsheetseries.owasp.org

9. **Stack Overflow Discussions**: Practical challenges in implementing AES encryption in Python. [Online]. Available: https://stackoverflow.com/questions

10. **IBM Developer Blog**: "AES Encryption Implementation: Security Best Practices," IBM. [Online]. Available: https://developer.ibm.com/

11. S. Guha, S. S. Yau and A. B. Buduru, "Attack Detection in Cloud Infrastructures Using Artificial Neural Network with Genetic Feature Selection," 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), Auckland, New Zealand, 2016, pp. 414-419, doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.32. keywords: {Feature extraction;Cloud computing;Testing;Genetic algorithms;Training;Training data;Multi-layer neural network;Cloud infrastructures;cyber-attack detection;artificial neural network;feature selection;genetic algorithm},

12. Y. Pu, J. Luo, Y. Wang, C. Hu, Y. Huo and J. Zhang, "Privacy Preserving Scheme for Location Based Services Using Cryptographic Approach," 2018 IEEE Symposium on Privacy-Aware Computing (PAC), Washington, DC,

USA, 2018, pp. 125-126, doi: 10.1109/PAC.2018.00022. keywords: {Servers;Privacy;Encryption;Data privacy;Resistance;Privacy Preservation, Location Based Services, Identity-Based Encryption},