NETWORK SECURITY
CS G513

BITS Pilani
K K Birla Goa Campus

PROJECT WRITE UP

To: Prof. Sougata Sen

From: Pratik Pawar, Akshata Rane, Aditya Agarwal

Date Apr 14, 2025

Subject: Write up for project – Custom Secure VPN implementation – a secure proxy tunnel

1. INTRODUCTION

In an increasingly interconnected digital landscape, the secure transmission of data across networks remains a paramount concern. Understanding the fundamental principles and practical implementations of network security is crucial for developers, administrators, and users alike. This document details the design, implementation, and analysis of a functional prototype of a simple, secure proxy tunnel, developed using Python on the Windows operating system. The primary objective of this project is not to create a production-grade VPN replacement, but rather to serve as a practical exposition of core network security concepts through hands-on development.

The system establishes a secure, encrypted communication channel between a designated client machine and a server machine. By leveraging this channel, the client can selectively route network traffic – specifically web browsing traffic in this implementation – through the trusted server. This architecture provides two key demonstrable benefits: firstly, it offers a mechanism to bypass local network access restrictions or filtering policies that might prevent direct access to certain internet resources from the client's network environment. Secondly, it enhances user privacy relative to the destination web service by masking the client's original IP address; the destination server only observes traffic originating from the proxy server's IP address.

2. ARCHITECTURE OVERVIEW

The system consists of two primary Python scripts:

server.py: Runs on a machine with unrestricted internet access. It acts as the secure gateway and proxy endpoint.

client.py: Runs on the user's machine (potentially behind a restrictive network). It acts as a local proxy server for the user's browser and as a client to the remote server.py.

Data Flow:

- Browser Request: The user's browser, configured to use client.py as its proxy, sends a standard HTTP/HTTPS request to 127.0.0.1:LOCAL_PROXY_PORT.
- Client Script Processing: client.py receives the browser request.
- Encryption (Client): client.py extracts the target URL/host, encrypts this information using a pre-shared symmetric key (AES via Fernet).
- Tunneling (Client -> Server): client.py establishes a TCP connection to the server.py (running on REMOTE_SERVER_IP:REMOTE_SERVER_PORT) and sends the encrypted data packet.
- Decryption (Server): server.py receives the encrypted data and decrypts it using the same pre-shared key.
- Internet Request (Server): server.py acts as a standard internet client, making the actual HTTP/HTTPS request to the target URL decrypted in the previous step.
- Internet Response (Server): server.py receives the plain web response (HTML, images, etc.) from the target web server.
- Encryption (Server): server.py encrypts this entire web response using the pre-shared key.
- Tunneling (Server -> Client): server.py sends the encrypted response back to client.py over the established TCP connection.
- Decryption (Client): client.py receives the encrypted response and decrypts it.
- Proxy Response (Client): client.py sends the plain, decrypted web response back to the waiting browser.

3. CORE LOGIC AND WORKFLOW

3.1 Initialization

Shared Key: Both client and server load the identical pre-shared secret key (SHARED_SECRET_KEY). This key is essential for the cryptography.fernet.Fernet cipher suite initialization.

Server Socket: server.py creates a TCP socket (socket.AF_INET, socket.SOCK_STREAM), binds it to a specified IP (LISTEN_IP, e.g., 0.0.0.0) and port (LISTEN_PORT, e.g., 4433), and enters a listening state (socket.listen()). It uses threading (threading.Thread) to handle multiple client connections concurrently (though the simple example primarily tests one).

Client Local Proxy Socket: client.py creates a TCP socket, binds it locally (LOCAL_PROXY_IP = '127.0.0.1', LOCAL_PROXY_PORT = 8080), and listens for incoming connections from the browser. It also uses threading to handle concurrent browser requests.

3.2 Client-side process (client.py)

Accept Browser Connection: The main thread accepts a new TCP connection from the browser on 127.0.0.1:8080.

Receive Browser Request: Reads the HTTP/HTTPS request data sent by the browser. A simple parsing attempt is made to extract the target URL or host:port information (e.g., from the GET or CONNECT line).

Establish Server Connection: Creates a new TCP socket connection to the remote server (REMOTE_SERVER_IP, REMOTE_SERVER_PORT).

Encrypt Target: Encodes the extracted target URL/host string to bytes and encrypts it using the initialized Fernet cipher (cipher_suite.encrypt()).

Send Encrypted Request: Sends the encrypted target information through the socket connected to the remote server.

Receive Encrypted Response: Waits to receive the encrypted response data from the server through the same socket. Reads data in chunks until the server closes the connection or data stops arriving.

Decrypt Response: Decrypts the received ciphertext using cipher_suite.decrypt().

Forward to Browser: Sends the resulting plaintext (the actual web content) back to the browser through the original socket connection established in step 1.

Close Sockets: Closes the connection to the browser and the connection to the remote server.

3.3 Server-Side Process (server.py)

Accept Client Connection: The main thread accepts a new TCP connection from a client.py instance on the configured LISTEN_PORT.

Receive Encrypted Request: Reads the encrypted data sent by the client script.

Decrypt Request: Decrypts the received ciphertext using cipher_suite.decrypt(). This yields the target URL/host requested by the original browser.

Fetch from Internet: Uses the requests.get() function to make an HTTP/HTTPS request to the decrypted target URL. It handles potential redirects and fetches the content. Basic error handling for request failures is included.

Encrypt Response: Encrypts the entire content received from the target web server (HTML, headers potentially included by requests handling) using cipher_suite.encrypt(). If fetching failed, an error message might be encrypted instead.

Send Encrypted Response: Sends the encrypted web content (or error) back to the client.py instance through the socket connection established in step 1.

Close Socket: Closes the connection to the client script.

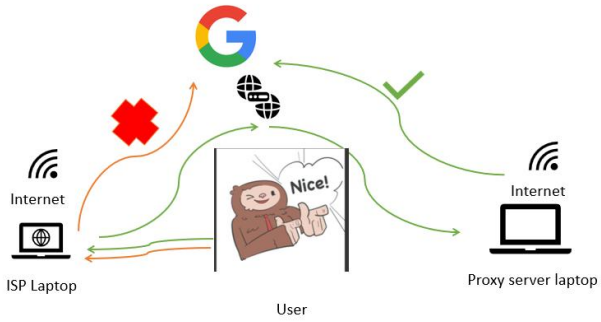

Figure 1. Work flow of the proxy tunnel

4. NETWORK SECURITY CONCEPTS IMPLEMENTED

Confidentiality:

Mechanism: Achieved using symmetric encryption with AES-128 in CBC mode provided by the cryptography.fernet implementation.

Protection: Ensures that anyone eavesdropping on the network segment between the client machine and the server machine cannot read the actual data being exchanged (target URLs requested by the client, web content returned by the server). They would only see unintelligible ciphertext.

Authentication (Basic):

Mechanism: Relies solely on the possession of the pre-shared secret key (SHARED_SECRET_KEY).

Protection: Only a client and server pair that share the identical secret key can successfully decrypt messages encrypted by the other. This prevents unauthorized clients from using the server and prevents the client from connecting to an impostor server (assuming the key is kept secret). This is a basic form; it lacks robustness against key compromise and doesn't verify identities beyond key possession.

Data Integrity:

Mechanism: Fernet tokens include a SHA256 HMAC signature calculated using the encryption key.

Protection: Upon decryption (cipher_suite.decrypt()), Fernet automatically verifies this HMAC signature. If the ciphertext was altered in any way during transit between the client and server (e.g., by a man-in-the-middle attacker), the signature verification will fail, and the decryption call will raise an InvalidToken exception. This prevents the application from acting on corrupted or maliciously modified data within the tunnel.

Secure Tunneling (Proxy Method):

Mechanism: The client script acts as a local proxy. All traffic configured to use this proxy is forced into a single TCP connection pathway to the server. The data within this pathway is protected by the encryption and integrity mechanisms.

Protection: The client's original IP address is hidden from the final destination web server, which only sees the IP address of the server.py machine. It creates a secure "pipe" for the proxied application's traffic through potentially untrusted intermediate networks between the client and the server.

5. TECHNOLOGY STACK

Language: Python 3

Core Modules:

socket: Low-level network interface for TCP connections.

threading: Handling concurrent connections from the browser (client-side) and potentially multiple clients (server-side).

cryptography: High-level library providing the Fernet (AES + HMAC) implementation for secure symmetric encryption.

requests: User-friendly library on the server for making HTTP/HTTPS requests to the target websites.

- socket Module: Provides the standard Python interface to the low-level Berkeley Sockets API, used to create, manage, and communicate over the fundamental TCP network connections (client connect, server bind/listen/accept, send/receive) forming the project's network tunnel foundation.
- threading Module: Enables basic concurrency, allowing the server to handle multiple client connections and the client to handle multiple browser requests simultaneously without blocking, improving overall responsiveness for network I/O operations.
- cryptography Library (Fernet): Supplies high-level, secure symmetric encryption using the Fernet recipe (AES-128-CBC + HMAC-SHA256). It directly implements the tunnel's Confidentiality and Data Integrity by encrypting/decrypting data and verifying its authenticity based on the shared secret key.
- requests Library: A user-friendly HTTP/HTTPS client library utilized server-side to easily make outbound web requests to the target URLs specified by the client. It handles complexities like redirects, connection pooling, and SSL/TLS verification, enabling the server to fulfill the proxied requests.

ngrok is a powerful tool used in this project primarily to overcome the NAT traversal problem, making the server.py script, running on a machine potentially behind a restrictive firewall or router (like a home network or mobile hotspot), accessible from the public internet without requiring manual port forwarding or complex network configuration.

The Problem:

- Most devices on local networks (home, office, mobile hotspot) are assigned private IP addresses (e.g., 192.168.x.x).
- These devices sit behind a NAT (Network Address Translation) device (the router/phone).
- The NAT device has a single public IP address facing the internet.
- Incoming connection requests from the public internet to the public IP address don't automatically know which internal device to forward the request to.

- Manually configuring port forwarding on the router is often needed but may not be possible (e.g., on mobile hotspots, university networks).

ngrok's Solution (Simplified):

1. Outbound Tunnel: The ngrok client executable running on the server machine establishes a persistent, outbound TLS connection to the secure ngrok cloud service. Outbound connections are typically permitted by firewalls.
2. Public Endpoint: The ngrok cloud service assigns a unique public URL (e.g., tcp://0.tcp.ngrok.io:12345) to this tunnel.
3. Traffic Forwarding: When an external client (like client.py) connects to this public ngrok URL:
    i. The connection hits the ngrok cloud servers.
    ii. The ngrok service routes this incoming traffic through the established secure tunnel back down to the ngrok agent running on the server machine.
    iii. The ngrok agent then forwards this traffic locally to the specified port where server.py is listening (e.g., localhost:4433).
4. Response Path: The response from server.py travels back through the ngrok agent, up the tunnel to the ngrok cloud, and finally back to the originating client.

In this project: ngrok acts as a readily available "public front door" for the server.py script. The REMOTE_SERVER_IP and REMOTE_SERVER_PORT in client.py are set to the public hostname and port provided by the ngrok service, abstracting away the complexities of the server's actual network location and configuration.

To effectively demonstrate the tunnel's capability to circumvent access restrictions, a controlled blocking mechanism was simulated locally on the client machine. This was achieved by modifying the Windows hosts file, a standard operating system component used for static hostname-to-IP address mapping. This method provides a reliable and easily reversible way to make specific websites inaccessible directly from the client system, thereby creating a clear "before" scenario for testing the proxy tunnel.

The hosts file, typically located at C:\Windows\System32\drivers\etc\hosts, is consulted by the Windows networking stack before initiating a standard DNS query during hostname resolution. By adding an entry that maps a target website's hostname (e.g., www.example-blocked-site.com) to the loopback IP address (127.0.0.1) or the null route address (0.0.0.0), we ensure that any direct connection attempt from the client machine to this hostname fails. For instance, the entry 127.0.0.1 www.example-blocked-site.com redirects any browser request for this site to the local machine itself, where no corresponding web server is running, resulting in a connection failure.

This local block serves as the baseline for demonstrating the proxy tunnel's effectiveness. When the tunnel is inactive, accessing the modified hostname in a browser confirms its inaccessibility. Subsequently, when the browser is configured to use the local proxy (client.py listening on 127.0.0.1:8080), the workflow changes significantly. The browser directs its request to the client.py script. Crucially, the client.py script does not perform hostname resolution itself; instead, it extracts the target hostname (www.example-blocked-site.com), encrypts it, and transmits it through the secure tunnel to the server.py script.

6. LIMITATIONS AND SECURITY CONSIDERATIONS

Pre-Shared Key Vulnerability: The entire security rests on the secrecy of the single shared key. If compromised, all confidentiality and integrity are lost. There's no secure key exchange or rotation mechanism.

Server Trust: The server decrypts all traffic. The server operator has full visibility into the proxied web requests. This model requires complete trust in the server machine and its operator.

Metadata Visibility: While the data between client and server is encrypted, the fact that the client is connecting to the server's IP address and port (or the ngrok endpoint) is visible on the client's local network. Traffic analysis could infer activity patterns.

Simplified Proxy: The HTTP/proxy logic is basic and might fail with complex web applications or non-HTTP protocols.

7. CONCLUSION AND RESULTS

This project successfully demonstrates a basic secure proxy tunnel using Python. It effectively implements confidentiality, basic shared-key authentication, and data integrity for traffic between the client and server components. By tunneling browser traffic through an encrypted channel, it achieves IP masking and can bypass simple network restrictions, serving as a valuable educational tool for understanding these fundamental network security concepts in a practical context. However, its limitations highlight the complexities involved in building robust, production-grade VPN or secure proxy solutions.

The following images show the successful bypassing of restriction on a lightweight website and activities of client and server: We have used karl.berlin what this purpose.

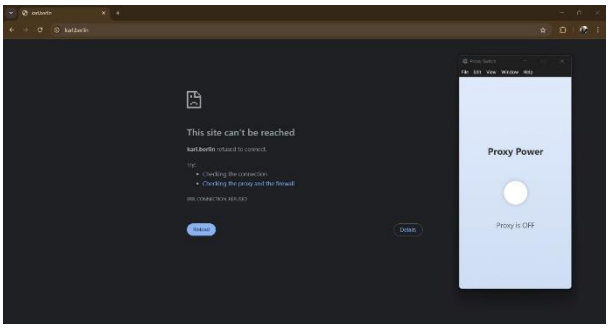

Figure 2 Blocked website for a client

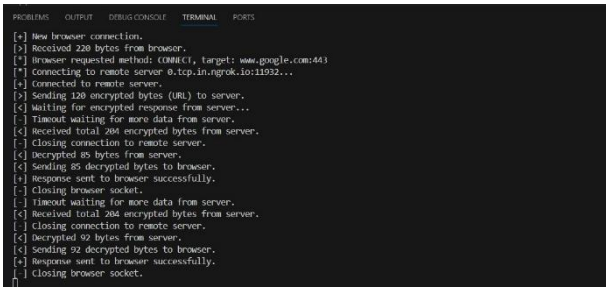

Figure 3. Client requesting webpage and getting response

```
Encryption cipher initialized successfully.
[*] Listening on 127.0.0.1:4433
[*] Accepted connection from 127.0.0.1:63568
[+] New client connection.
[>] Received 19 decrypted bytes.
[*] Client requested URL: http://karl.berlin/
[*] Forwarding request to http://karl.berlin/...
[<] Received 3109 bytes from target server.
[<] Sending 4236 encrypted bytes back to client.
```

Figure 4. Server sending target website data

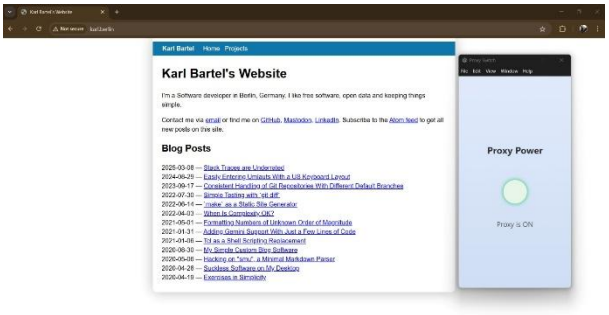

Figure 5. Website accessible after turning on proxy tunnel

GITHUB LINK

NeuroNut/Custom-VPN-implementation: Network Security Project Component

8. REFERENCES

[1] https://cryptography.io/en/latest/fernet/

[2] https://docs.python.org/3/library/socket.html.

[3] https://requests.readthedocs.io/en/latest/.

[4] Kurose, J. F., & Ross, K. W. (2021). Computer Networking: A Top-Down Approach (8th ed.). Pearson.

[5] Paar, C., & Pelzl, J. (2010). Understanding Cryptography: A Textbook for Students and Practitioners. Springer.