Task 1: Security Review of ScanCraft Application

1. Description of the Application

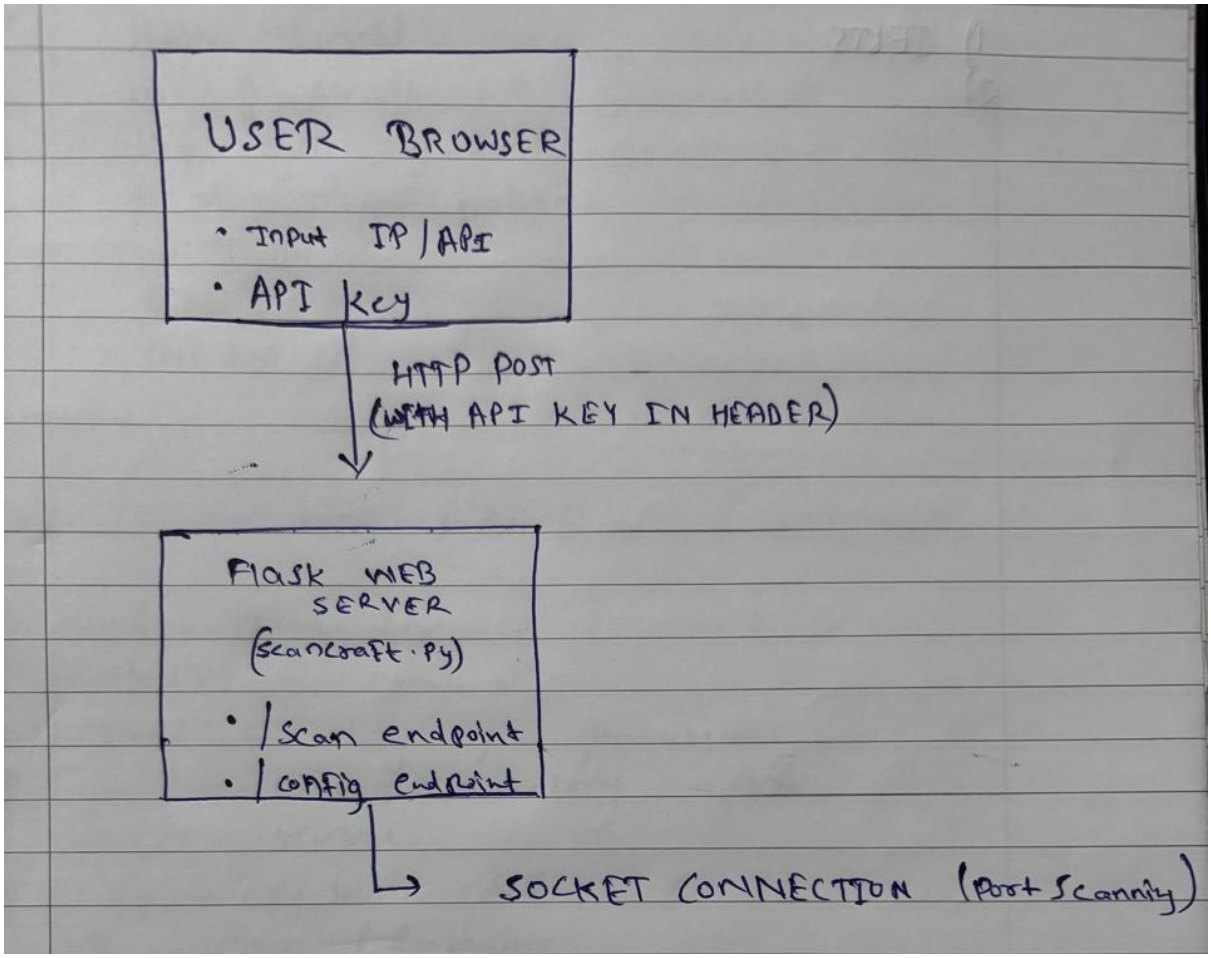
What the Application Does:

ScanCraft is a web-based network port scanner that allows users to:

- Scan network ports on specified IP addresses or hostnames
- Check which ports are open/available on target systems
- The application is designed to diagnose connectivity issues and check port availability on servers.

Technologies Used:

- Backend Framework: Flask (Python web framework)
- Network Operations: Python socket library
- Concurrency: ThreadPoolExecutor for parallel port scanning
- Frontend: HTML/JavaScript with inline templates
- System Commands: subprocess module for command execution



2. Identified Security Risks

1) Hardcoded Secrets

What the Issue Is:

The application contains hardcoded sensitive credentials directly in the source code.

API key:

API_KEY = "Eric" (Line 11)

Admin password:

ADMIN_PASSWORD = "ERIC123" (Line 12)

The same API key is also hardcoded in the client-side JavaScript code (Line 54)

Why It's a Risk:

Hardcoded secrets in source code create multiple security vulnerabilities:

1) Source Code Exposure: Anyone with access to the codebase can see the secrets
2) Version Control Leakage: If code is committed to Git/GitHub, secrets remain in history forever, even if later deleted
3) Client-Side Exposure: The API key is visible in browser JavaScript - anyone can view page source and see it
4) Cannot Rotate: Changing the key requires code modification and redeployment to all environments
5) Shared Across Environments: Same key used in development, testing, and production
6) Developer Access: All developers, contractors, and anyone with repository access can see secrets

What Could Happen If This Isn't Fixed

If secrets remain hardcoded in the ScanCraft application, anyone with access to the source code or repository can easily obtain API keys or passwords. Once exposed, attackers can gain unauthorized access to the scanning functionality, abuse the service, or reuse the credentials in other systems if they are shared. Hardcoded secrets also make it difficult to rotate or revoke compromised keys, increasing the risk of long-term unauthorized access.

Mitigation:

Instead of hardcoding sensitive information in the source code, can store it securely using environment variables. For better security, will use dedicated secret management services such as AWS Secrets Manager, HashiCorp Vault, Azure Key Vault, or Google Cloud Secret Manager to manage and rotate secrets safely. Must make sure API keys and credentials are never exposed in client-side code, where anyone can view them. Regularly rotate API keys to reduce the risk if a key is ever leaked. Additionally, use a .gitignore file to ensure secret files and configuration data are not accidentally committed to version control.

1. Use Environment Variables:

import os

```
API_KEY = os.environ.get("SCANNER_API_KEY")
if not API_KEY:
   raise RuntimeError("SCANNER_API_KEY environment variable not set")
ADMIN_PASSWORD = os.environ.get("ADMIN_PASSWORD")
if not ADMIN_PASSWORD:
   raise RuntimeError("ADMIN_PASSWORD environment variable not set")
```

2. Implement API Key Rotation:

```
# Generate unique keys per user
import secrets
def generate_api_key():
   return secrets.token_urlsafe(32)
```

2) Weak Authentication

What the Issue Is: The application uses a single, shared API key for authentication across all users and requests. The authentication mechanism has multiple critical weaknesses:

1. **Single Shared Secret**: One API key (Eric) used by everyone
2. **No User Identification**: Cannot tell which user made which request
3. **No Rate Limiting**: Unlimited requests with valid key
4. **Simple Header Check**: Just checks if header matches hardcoded value
5. **No Token Expiration**: API key never expires

```
api_key = request.headers.get("X-API-KEY")
   if not api_key or api_key != API_KEY:
      return jsonify({"error": "Unauthorized"}), 401
```

Why It's a Risk:

Weak authentication creates a situation where all users share the same API key, resulting in no individual accountability. It also introduces an all-or-nothing access model. Additionally, weak authentication provides no protection against abuse. Without rate limiting or per-user controls, a single key can be used to perform unlimited scans, making the application vulnerable to misuse and denial-of-service attacks.

What could happen if it's not fixed?

Weak authentication can allow unauthorized users to access the application and misuse its functionality. If credentials are exposed or reused, attackers can repeatedly access the system without detection. Since all users share the same authentication mechanism, actions cannot be traced to specific individuals, making monitoring, accountability, and incident investigation difficult.

Here I mentioned Mitigation:

Implement strong authentication mechanisms that uniquely identify each user instead of using a shared API key. Introduce role-based access control to limit what actions users can perform based on their permissions. Apply rate limiting and request throttling to prevent abuse and reduce the risk of denial-of-service attacks. Maintain proper logging and audit trails to track user activity and support incident investigation.

- Add Rate Limiting:

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["100 per day", "10 per minute"]
)
@app.route("/scan", methods=["POST"])
@limiter.limit("5 per minute")  # Max 5 scans per minute per IP
def scan():
    pass
```

3) **Lack of Input Validation**

   What the Issue Is: The application accepts user input (IP addresses, port numbers) without proper validation or sanitization. Critical inputs are used directly without checking:

   - IP Address: No format validation, no type checking (Line 61)
   - Port Numbers: Weak validation, no range limits

```
# VULNERABILITY 3: Lack of Input Validation
# No IP address format validation
ip = data.get("ipAddress")
start_port = data.get("startPort")
end_port = data.get("endPort")
# Minimal validation - accepts almost anything
if not ip:
    return jsonify({"error": "IP address required"}), 400
```

```
# VULNERABILITY 3: No proper validation on port numbers
   # Not added range limits, no type checking
   try:
      start = int(start_port)
      end = int(end_port)
   except:
return jsonify({"error": "Invalid port numbers"}), 400
```

Why It's a Risk:

1) Invalid IP Addresses
   - Accepts any string as IP (including malicious payloads)
   - No check for private vs public IPs
   - Could accept hostnames with special characters
   - No validation for loopback addresses (127.0.0.1)
   - Enables scanning of internal networks

2) Port Range Issues
   - No minimum/maximum port validation
   - Accepts negative numbers
   - No check if start > end
   - No limit on range size (could scan all 65,535 ports)
   - Enables denial-of-service attacks

3) Resource Exhaustion
   - Large port ranges consume CPU and memory
   - Creates thousands of threads simultaneously
   - Can crash the server or make it unresponsive

What Could Happen If Not Fixed:

If user input is not properly validated, the application becomes easy to misuse and unstable. An attacker could supply crafted or unexpected values to scan internal network ports that were never meant to be exposed, potentially revealing sensitive services running inside the network.

Without validation limits, the application may also be forced to scan extremely large or invalid port ranges, which can consume excessive system resources and lead to denial-of-service conditions. This could make the application slow, unresponsive, or completely unavailable to legitimate users.

Mitigation:

I will validate IP addresses to ensure only properly legit IP address are accepted. I will validate port numbers to allow only numeric values within the valid range and prevent excessive scan requests. Additionally, add input sanitization to reject malformed or unexpected input before it is processed by the application.

Validate Port Number:

```
def validate_ports(start, end):
    """Validate port number ranges"""
    # Type checking
    try:
        start = int(start)
        end = int(end)
    except (ValueError, TypeError):
        return False, "Port numbers must be valid integers"

    # Range validation
    if start < 1 or start > 65535:
        return False, "Start port must be between 1 and 65535"

    if end < 1 or end > 65535:
        return False, "End port must be between 1 and 65535"

    # Logical validation
    if start > end:
        return False, "Start port cannot be greater than end port"

    # Prevent excessive ranges (DoS protection)
    MAX_RANGE = 1000
    if (end - start + 1) > MAX_RANGE:
        return False, f"Port range too large. Maximum allowed: {MAX_RANGE} ports"

    return True, (start, end)

# Usage:
is_valid, result = validate_ports(start_port, end_port)
if not is_valid:
    return jsonify({"error": result}), 400
start, end = result
```

Add Input Sanitization:

```
import html
import re

def sanitize_string(input_str, max_length=100):
    """Sanitize string inputs"""
    if not input_str:
        return ""

 # Convert to string
  input_str = str(input_str)

    # Limit length
    input_str = input_str[:max_length]

    # Remove control characters
    input_str = re.sub(r'[\x00-\x1f\x7f-\x9f]', '', input_str)

    # HTML escape to prevent XSS
    input_str = html.escape(input_str)

    return input_str.strip()
```

Validate IP Addresses:

```
import ipaddress

def validate_ip(ip_string):
    """Validate and sanitize IP address input"""
    try:
        # Parse IP address
        ip = ipaddress.ip_address(ip_string.strip())

        # Reject private IPs (internal networks)
        if ip.is_private:
            return False, "Scanning private IP addresses is not allowed"

        # Reject loopback (localhost)
        if ip.is_loopback:
            return False, "Cannot scan localhost"

        # Reject multicast and reserved IPs
        if ip.is_multicast or ip.is_reserved:
            return False, "Invalid IP address range"

        # Reject link-local addresses
        if ip.is_link_local:
            return False, "Link-local addresses not allowed"

        return True, str(ip)

    except ValueError as e:
        return False, f"Invalid IP address format: {str(e)}"

# Usage in scan endpoint:
is_valid, result = validate_ip(ip)
if not is_valid:
    return jsonify({"error": result}), 400
ip = result  # Use validated IP
```

4) Missing HTTPS

What the Issue Is:

The application runs exclusively on HTTP (unencrypted) without any SSL/TLS encryption. All communication between the client and server is transmitted in plain text.

Why It's a Risk:

If HTTPS is not used, credentials and sensitive data can be transmitted in clear text, making them easy to intercept. This exposes user data to attackers on the network and enables man-in-the-middle attacks where communication can be read or altered. Without HTTPS, users also cannot verify the server's identity, increasing the risk of connecting to a fake or malicious server.

What Could Happen If Not Fixed:

Without HTTPS, attackers on the same network can eavesdrop on Wi-Fi traffic and perform man-in-the-middle attacks. This allows them to intercept or modify data in transit, including scan results. As a result, false information can be injected, such as showing a port as open when it is not. This can mislead users into believing a system is vulnerable, causing false alerts, incorrect reports, and unnecessary remediation actions.

Mitigate:

Enable HTTPS by configuring SSL/TLS certificates to encrypt communication between the client and server. Update the Flask application to run securely over HTTPS and ensure all client-side requests use encrypted connections. Add appropriate security headers to protect against common web attacks and maintain proper certificate management by using trusted certificates and renewing them regularly.

1. Enable HTTPS with SSL/TLS Certificates:

```
# Install certbot
sudo apt-get install certbot python3-certbot-nginx

# Get free SSL certificate
sudo certbot --nginx -d scancraft.yourdomain.com

# Certificates automatically renewed every 90 days
```

2. Update Flask Application for HTTPS:

```
from flask import Flask, request
import os

app = Flask(__name__)
```

```
# Force HTTPS in production
    if not app.debug:
    @app.before_request
     def before_request():
       if request.url.startswith('http://'):
       url = request.url.replace('http://', 'https://', 1)
       return redirect(url, code=301)

    # Set secure cookie flags
    app.config.update(
     SESSION_COOKIE_SECURE=True,      # Only send over HTTPS
    SESSION_COOKIE_HTTPONLY=True,    # No JavaScript access
    SESSION_COOKIE_SAMESITE='Strict' # CSRF protection
    )
```

3. Update Client-Side Code:

```
// Automatically use HTTPS
const API_URL = window.location.protocol === 'https:'
   ? 'https://scancraft.example.com'
   : 'http://localhost:3000';

fetch(`${API_URL}/scan`, {
   method: 'POST',
   // ... rest of code
});
```

License

For educational purposes only. Not for production use.

Created for: Jr. Cybersecurity Engineer Interview Assignment
Focus: Security vulnerability identification and analysis
Vulnerabilities: 4 critical security risks from OWASP Top 10