**## Project 1: Interactive Honeypot & Threat Dashboard 🍯**

This project is a journey into defensive security and threat intelligence. You'll set up a trap and then build the tools to watch what gets caught.

**Project Roadmap**

**Phase 1: Set Up the Decoy (The Honeypot)**

1. **Get a Server:** Rent the cheapest Linux Virtual Machine (VM) you can find. A $4-6/month "droplet" from **DigitalOcean** or a **Linode "Nanode"** is perfect. Use a standard Ubuntu 22.04 image. This VM will be your sacrificial honeypot, so don't use it for anything else.
2. **Install the Honeypot Software:** We'll use **Cowrie**, an excellent medium-interaction SSH and Telnet honeypot.
   * SSH into your new server.
   * Follow the official Cowrie installation guide. It's well-documented and involves installing some dependencies (like Python) and then cloning their repository from GitHub.
   * **Crucial Configuration:** In the cowrie.cfg file, find the output section and enable JSON logging (jsonlog\_enabled = true). This makes the logs machine-readable, which is essential for your dashboard later.
3. **Run it and Gather Data:** Start Cowrie. Within hours, you will see automated bots from around the world trying to log in. Let it run for a day or two to collect some interesting initial data. You can view the live logs in the log/cowrie.json file.

**Phase 2: Build the Backend and Data Processor**

This part of the project will fetch, parse, and store the attack data. You can run this on a separate, free-tier service like **Heroku** or **Render**, or even on the same VM to save money initially.

1. **Choose a Framework:** **Python with Flask** is a fantastic choice. It's simple and powerful.
2. **Set Up a Database:** Start with **SQLite**. It's a single-file database that's built into Python, so there's no complex setup.
3. **Create a Data Model:** Define a database table to store the attacks. It might look like this:
   * id (Primary Key)
   * timestamp (When the attack happened)
   * source\_ip (The attacker's IP address)
   * source\_country (You'll look this up later)
   * username (The username they tried)
   * password (The password they tried)
   * event\_type (e.g., 'login\_failed', 'command\_input')
4. **Write a Parser Script:** Create a Python script that:
   * Securely copies the cowrie.json log file from your honeypot VM to your backend server (using a library like paramiko for SFTP).
   * Reads the new log entries.
   * For each entry, it parses the JSON, looks up the IP address using a free geolocation service (like ip-api.com), and inserts the structured data into your SQLite database.
   * You can run this script automatically every 5-10 minutes using a cron job.

**Phase 3: Create the Frontend Dashboard**

This is the visual part where you bring your data to life.

1. **Build API Endpoints:** In your Flask app, create a few simple API routes:
   * /api/latest\_attacks: Returns the last 20 attacks as JSON.
   * /api/map\_data: Returns a list of all unique IP addresses and the number of times they've appeared.
   * /api/stats: Returns JSON data for charts, like the Top 10 most common passwords and usernames.
2. **Design the UI:** Use simple HTML, CSS, and JavaScript.
   * **Live Attack Feed:** A simple table that fetches data from /api/latest\_attacks every 10 seconds and updates the rows.
   * **Attack Map:** Use **Leaflet.js** (a free, open-source mapping library). Your JavaScript will fetch data from /api/map\_data, and for each IP, add a circle or marker on the map at its geographic coordinates.
   * **Charts:** Use **Chart.js** to create bar charts for your /api/stats endpoint. It's incredibly easy to get started with.

By breaking it down like this, you go from an abstract idea to a concrete, buildable system.

**## Project 4: End-to-End Encrypted File Sharing Service 🔐**

This project is a deep dive into modern web technologies and applied cryptography. The elegance here is in making the server "dumb" – it only stores encrypted blobs of data it cannot read.

**Project Roadmap**

**Phase 1: The Client-Side Crypto Core (All in the Browser)**

This is the most critical part. You'll write all of this in JavaScript.

1. **The UI:** Create a very simple HTML page with two main sections, one for uploading and one for downloading. You can use CSS to hide the download section initially.
2. **The Encryption Logic:**
   * Use an <input type="file"> element to let the user select a file.
   * Use the **Web Crypto API** (window.crypto.subtle), which is built into all modern browsers.
   * When a user selects a file, your JavaScript will:
     1. Read the file's content as an ArrayBuffer.
     2. Generate a secure, random encryption key: await crypto.subtle.generateKey({name: 'AES-GCM', length: 256}, true, ['encrypt', 'decrypt']);
     3. Encrypt the file's ArrayBuffer with this key.
     4. Export the key to a URL-safe string (e.g., using Base64 encoding).
3. **The Upload:** After encryption, POST the **encrypted data** (not the original file!) to your backend. The backend will save it and return a unique ID (e.g., a7b2x9).
4. **Generate the Share Link:** Your JavaScript will then construct the final link: https://your-site.com/?id=a7b2x9#your-base64-encoded-key. The key is placed in the **hash fragment** (#...), which is never sent to the server.

**Phase 2: The "Zero-Knowledge" Backend**

The backend is simple by design. Its only job is to store and retrieve opaque blobs of data.

1. **Choose a Platform:** **Serverless functions** are perfect for this. You can use Vercel Functions, Netlify Functions, or Cloudflare Workers. They are free to start with and scale automatically.
2. **Create Two API Endpoints:**
   * POST /api/upload: Receives the raw encrypted file data. It generates a random ID, stores the data in a cloud storage bucket, and returns the ID as JSON.
   * GET /api/download?id=...: Reads the ID from the query parameter, fetches the corresponding encrypted file from the storage bucket, and sends it back to the client.
3. **Choose a Storage Provider:** Use an object storage service. **Backblaze B2** or **Cloudflare R2** have extremely generous free tiers that are perfect for a project like this. AWS S3 is another standard choice. Your serverless function will use the provider's API key to read/write files.

**Phase 3: The Decryption and Download Flow**

1. **Check the URL:** When your page loads, your JavaScript must immediately check if window.location.hash exists.
2. **If the hash exists:**
   1. Hide the "upload" section and show the "download" section.
   2. Grab the key from the hash and the file ID from the query parameter (?id=...).
   3. Decode the Base64 key and import it back into a usable format with the Web Crypto API.
   4. Make a GET request to your /api/download?id=... endpoint to fetch the encrypted file data.
   5. Use the imported key to decrypt the data.
   6. If successful, convert the decrypted ArrayBuffer into a Blob and create a temporary download link (<a href="..." download>) so the user can save the original file.