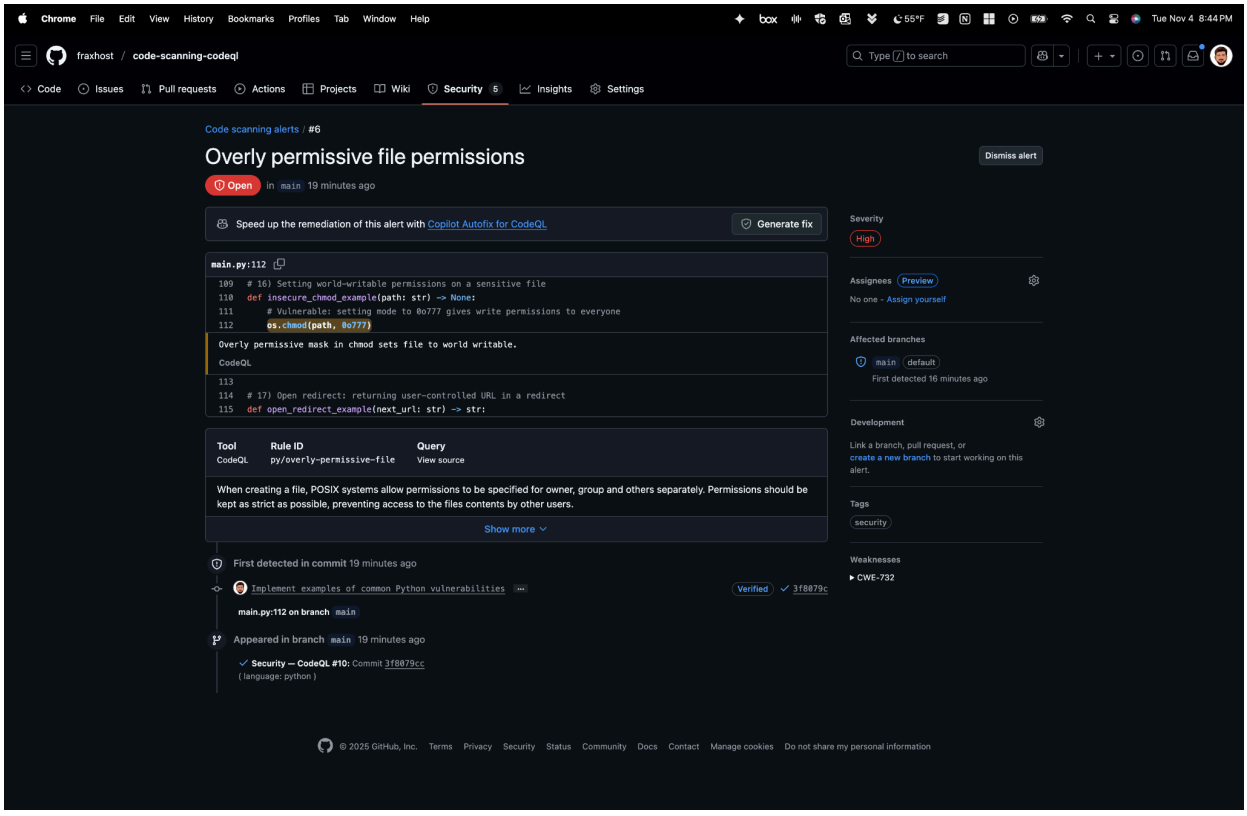
(1) Overly Permissive File Permissions

- **Overly permissive file permissions** occur when a file or directory is created with access rights that are too broad (e.g., anyone can read/write/execute it).
- This allows **unauthorized users or processes** to modify, delete, or steal sensitive data.
- In Linux, using permissions like `chmod 777` or Python's `os.chmod(path, 0o777)` is a common example.
- Attackers can exploit this to replace config files, inject malicious code, or escalate privileges.
- Secure practice: grant the **minimum necessary access** (e.g., `600` or `700`) and avoid world-writable directories unless explicitly required.

Filename: main.py
Line No: 112

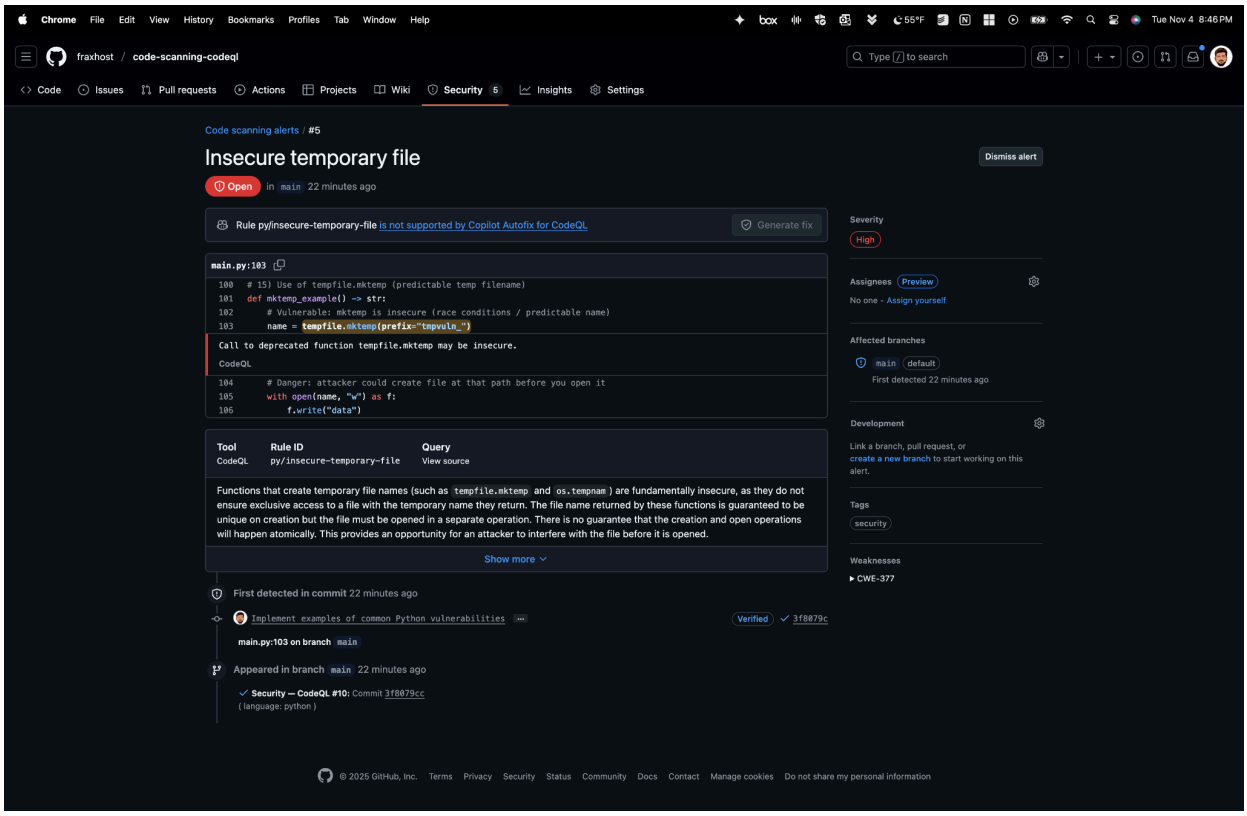


Screenshot: Overly Permissive File Permissions

(2) Insecure Temporary File

- An **insecure temporary file** is created in a way that allows another program or attacker to access or modify it before the legitimate program uses it.
- If a temp file is created with predictable names (like `/tmp/data.txt`), an attacker can create or replace it beforehand.
- This can cause data leaks, unauthorized file modification, or even **code execution** if the program later runs data from that file.
- Temp directories like `/tmp` are shared, so files placed there need secure random names and proper permissions.
- Safe practice: use secure temp functions (e.g., `tempfile.NamedTemporaryFile()` in Python) to avoid guessing and tampering.

Filename: main.py
Line No: 103

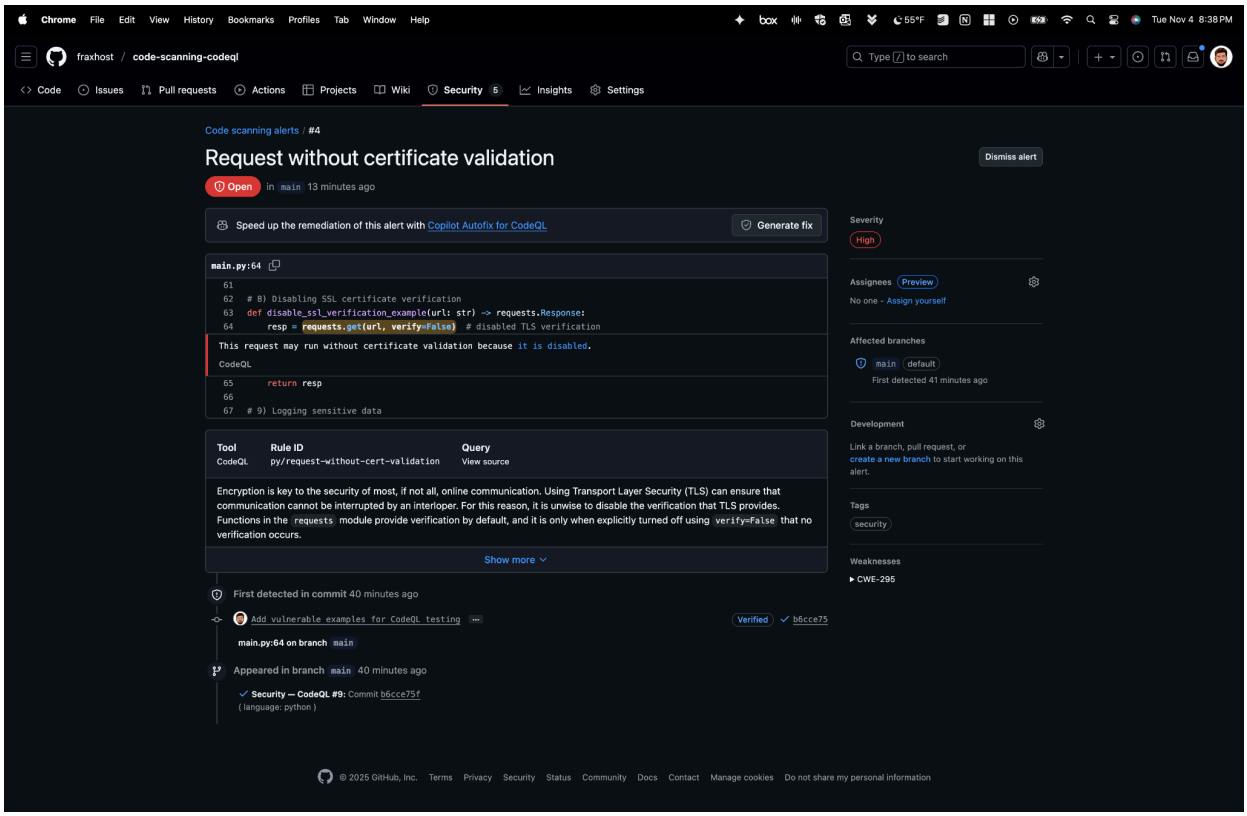


Screenshot: Insecure Temporary File

(3) Request without Certificate Validation

- **Request without certificate validation** means a program makes an HTTPS request but does **not verify the server's SSL certificate**.
- This skips the security check that ensures the server is **authentic** and not an attacker pretending to be the server.
- Without validation, the connection can be intercepted using a **man-in-the-middle attack**.
- Attackers can steal sensitive data (passwords, API keys) or inject malicious responses.
- It is often seen in code like `verify=False` in Python `requests`, and should be avoided except in controlled testing environments.

Filename: main.py
Line No: 64

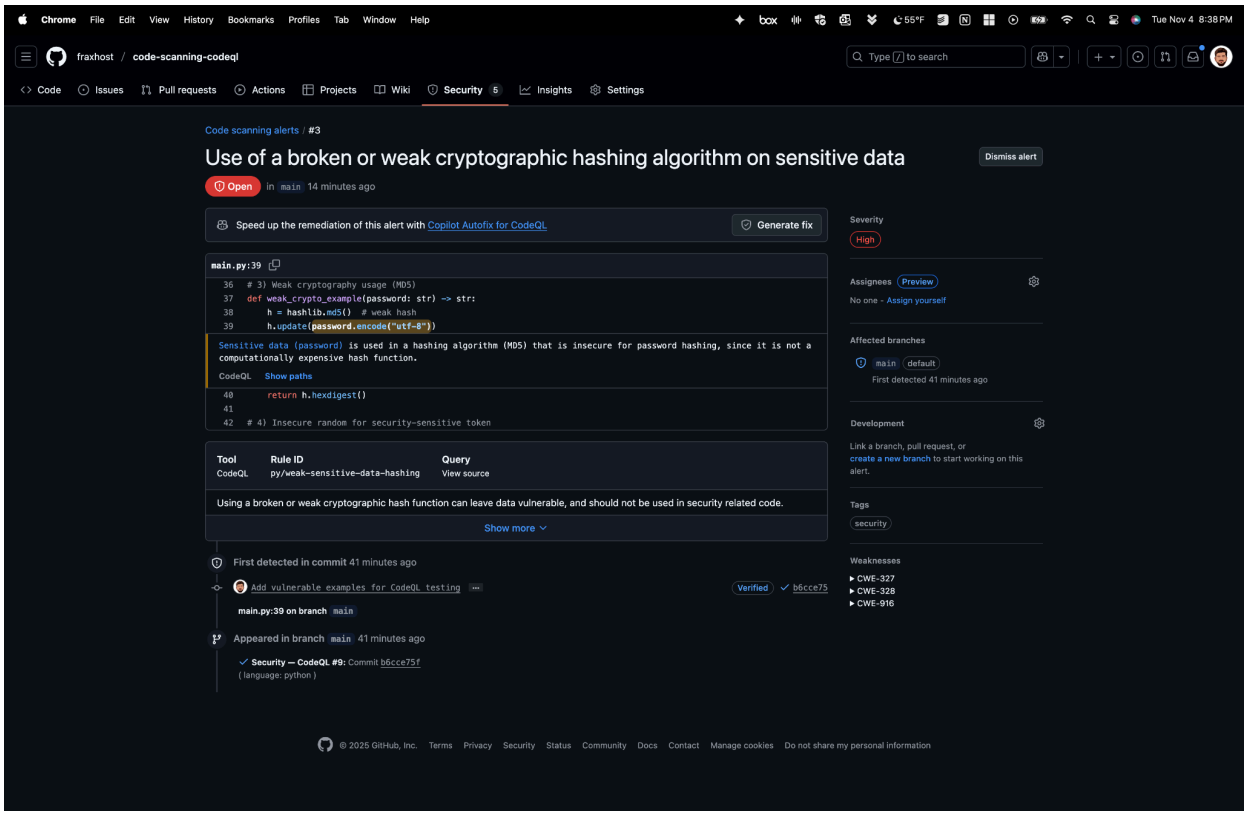


Screenshot: Request without Certificate Validation

(4) Use of a Broken or Weak Cryptographic Hashing Algorithm on Sensitive Data

- **Weak cryptographic hashes** like MD5 or SHA-1 are no longer secure for protecting sensitive data such as passwords or tokens.
- They are vulnerable to **collision attacks**, where two different inputs produce the same hash, compromising data integrity.
- Using them on passwords allows attackers to **precompute hash tables (rainbow tables)** to reverse them easily.
- Sensitive data hashed with weak algorithms can be **easily cracked or tampered**, leading to breaches or impersonation.
- Secure practice: use modern, strong hashing algorithms like **SHA-256** with salts or **password-specific hashing functions** like `bcrypt`, `scrypt`, or `Argon2`.

Filename: main.py
Line No: 39

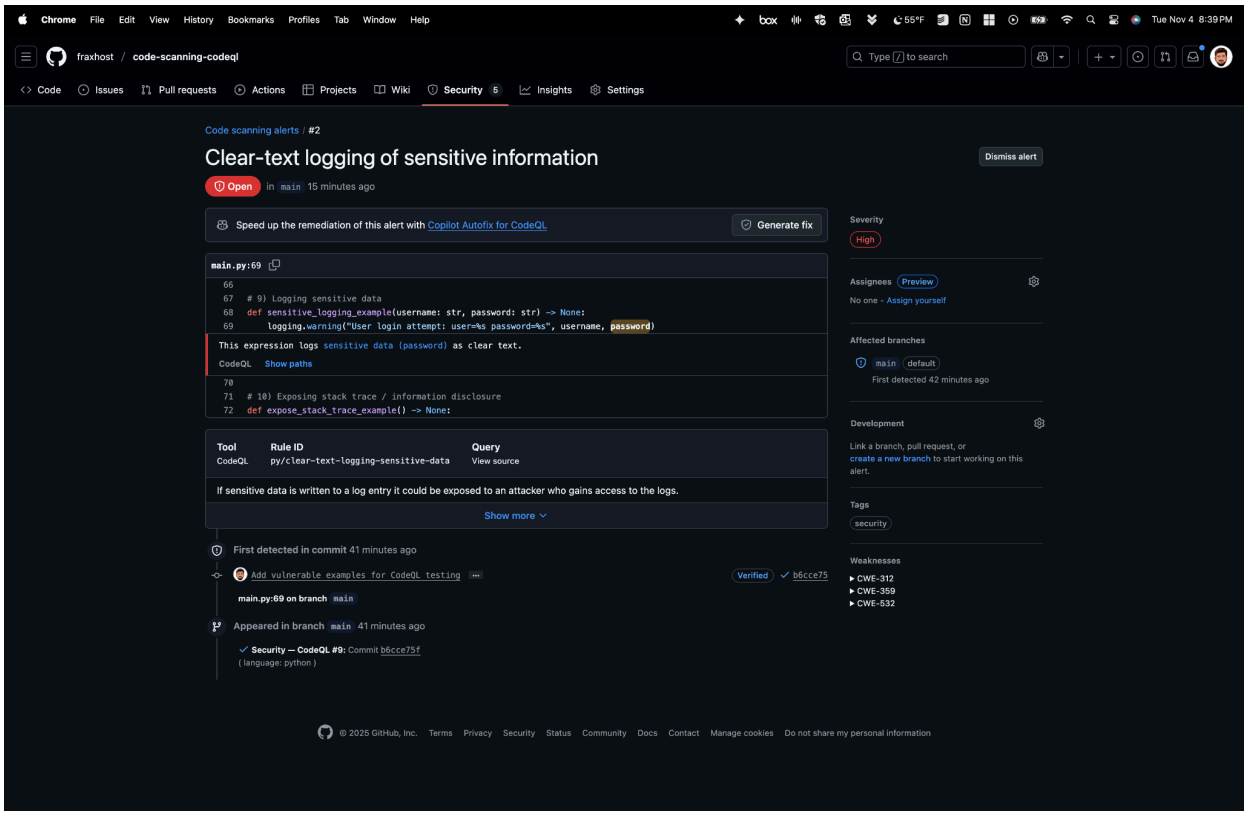


Screenshot: Use of a Broken or Weak Cryptographic Hashing Algorithm on Sensitive Data

(5) Clear-Text Logging of Sensitive Information

- **Clear-text logging** occurs when sensitive information like passwords, API keys, or personal data is written directly to logs.
- Anyone with access to the logs (developers, administrators, or attackers) can **read confidential data** easily.
- Logs may be stored long-term or sent to external systems, increasing the risk of **data leaks**.
- Logging sensitive data can lead to **compliance violations** with standards like GDPR, HIPAA, or PCI-DSS.
- Secure practice: **mask, redact, or avoid logging** sensitive information, and log only non-sensitive identifiers or anonymized data.

Filename: main.py
Line No: 69



Screenshot: Clear-Text Logging of Sensitive Information