Peer Review Report: Code Testing and Security Analysis (Group: 100)

Introudaction

This report provides constructive and evidence-based feedback on the Secure Overlay Chat System, focusing on both its strengths and areas where security and reliability can be further improved. The testing approach combines static analysis tools (Bandit and Snyk) with dynamic fuzzing (Atheris) to capture a wide view of the system's behavior, from design-level patterns to real execution outcomes. The project already demonstrates a solid understanding of secure communication, particularly through its use of RSA for key exchange, AES-CBC encryption for message confidentiality, and careful thread management using locks to avoid race conditions. Building on this foundation, the analysis aims to highlight how targeted refinements, such as strengthening cryptographic practices, improving input validation, and reducing unnecessary exposure, can make the implementation even more resilient.

Strengths in Implementation

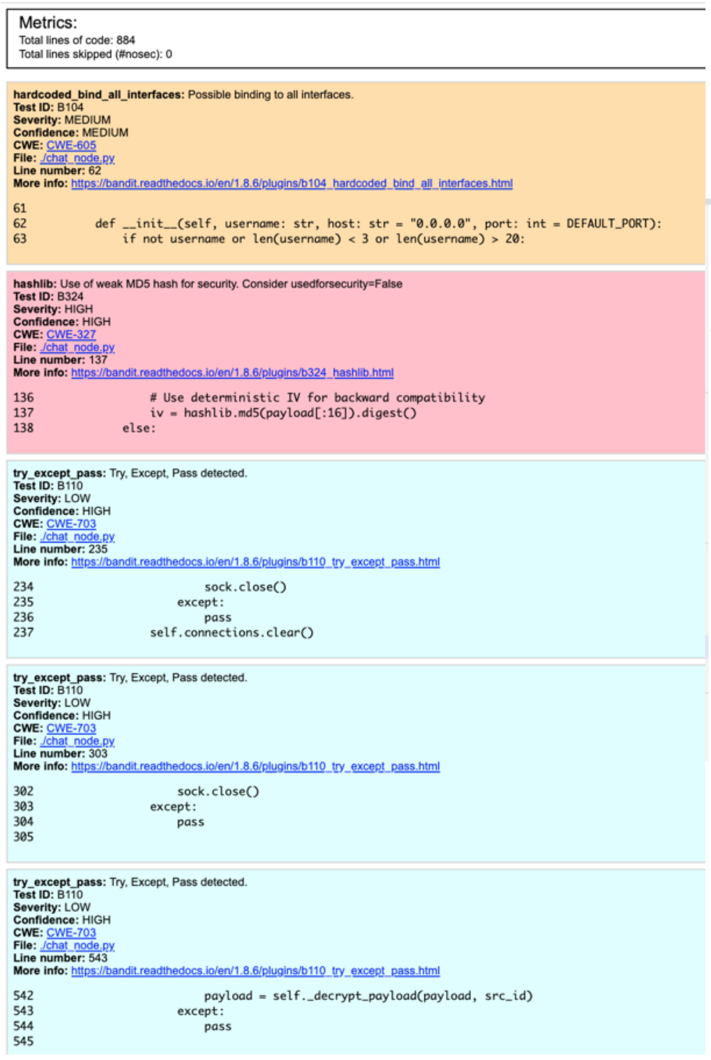
The implementation demonstrates several strong design and security practices that contribute to the system's overall robustness and reliability. The use of RSA for key exchange and AES-CBC for payload encryption reflects a solid understanding of secure communication principles, ensuring that message confidentiality is maintained during peer-to-peer exchanges. The design also incorporates threaded handling of multiple concurrent connections, which enables efficient resource utilization and smooth operation under load. The inclusion of structured message headers containing timestamps and integrity information enhances traceability and supports message validation, helping to prevent replay or tampering. Furthermore, the use of lock mechanisms such as RLock to control access to shared data demonstrates careful attention to thread safety and race condition prevention. Each peer's session key establishment adds an additional layer of security by isolating communication channels, thereby reducing the risk of key reuse or cross-session compromise. Collectively, these elements indicate a well-considered approach to secure and reliable distributed system design.

Testing Approaches and Tools Used

I used Bandit as a static analysis tool to automatically scan the source for insecure patterns (hashlib.md5, hardcoded bindings, broad exception handlers). For dynamic testing I executed controlled runtime probes (non-destructive socket-based fuzzing and connectivity checks) inside an isolated workspace to observe how the node responds to malformed headers and diagnostic requests. These automated scans were complemented by a focused manual code review targeting cryptographic routines and diagnostic handlers to verify findings and trace exploitation paths.

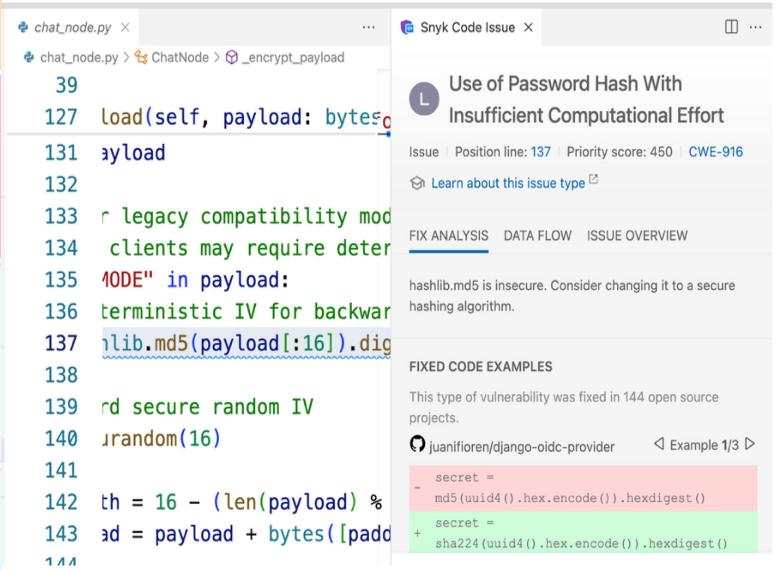
1. Static analysis (Bandit & Snyk):

The static analysis tool **Bandit** was used to scan the project code for potential security issues by analyzing the Python source files without executing them. The scan identified several findings, including a medium-severity issue where the server is configured to bind to all network interfaces (hardcoded_bind_all_interfaces) in chat_node.py at line 62, a high-severity issue involving the use of the weak MD5 hash function (hashlib) at line 137, and multiple low-severity issues related to generic exception handling using try-except-pass statements in various parts of the code, as partially shown in *Figure 1*. Bandit was executed via the command line on the project directory, and its reports were used to locate vulnerabilities, assess their severity, and reference the corresponding CWE identifiers. Additionally, the **Snyk** tool was used to perform a complementary static scan focusing on code dependencies, identifying a high-priority vulnerability in the MD5-based hashing at line 137, as shown in *Figure 2*. Together, these tools provided a comprehensive static analysis of the project code.



```
Metrics:
Total lines of code: 884
Total lines skipped (#nosec): 0

hardcoded_bind_all_interfaces: Possible binding to all interfaces.
Test ID: B104
Severity: MEDIUM
Confidence: MEDIUM
CWE: CWE-605
File: ./chat_node.py
Line number: 62
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b104_hardcoded_bind_all_interfaces.html

61
62        def __init__(self, username: str, host: str = "0.0.0.0", port: int = DEFAULT_PORT):
63            if not username or len(username) < 3 or len(username) > 20:

hashlib: Use of weak MD5 hash for security. Consider usedforsecurity=False
Test ID: B324
Severity: HIGH
Confidence: HIGH
CWE: CWE-327
File: ./chat_node.py
Line number: 137
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b324_hashlib.html

136                # Use deterministic IV for backward compatibility
137                iv = hashlib.md5(payload[:16]).digest()
138            else:

try_except_pass: Try, Except, Pass detected.
Test ID: B110
Severity: LOW
Confidence: HIGH
CWE: CWE-703
File: ./chat_node.py
Line number: 235
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b110_try_except_pass.html

234                        sock.close()
235                    except:
236                        pass
237                self.connections.clear()

try_except_pass: Try, Except, Pass detected.
Test ID: B110
Severity: LOW
Confidence: HIGH
CWE: CWE-703
File: ./chat_node.py
Line number: 303
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b110_try_except_pass.html

302                    sock.close()
303                except:
304                    pass
305

try_except_pass: Try, Except, Pass detected.
Test ID: B110
Severity: LOW
Confidence: HIGH
CWE: CWE-703
File: ./chat_node.py
Line number: 543
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b110_try_except_pass.html

542                        payload = self._decrypt_payload(payload, src_id)
543                    except:
544                        pass
545
```

Figure 1: Bandit Report



```
 39
127  load(self, payload: bytes
131  ayload
132
133  r legacy compatibility mod
134   clients may require deter
135  MODE" in payload:
136  terministic IV for backwar
137  hlib.md5(payload[:16]).dig
138
139  rd secure random IV
140  urandom(16)
141
142  th = 16 - (len(payload) %
143  ad = payload + bytes([padd
```

Use of Password Hash With Insufficient Computational Effort

Issue | Position line: 137 | Priority score: 450 | CWE-916

Learn about this issue type

hashlib.md5 is insecure. Consider changing it to a secure hashing algorithm.

FIXED CODE EXAMPLES

This type of vulnerability was fixed in 144 open source projects.

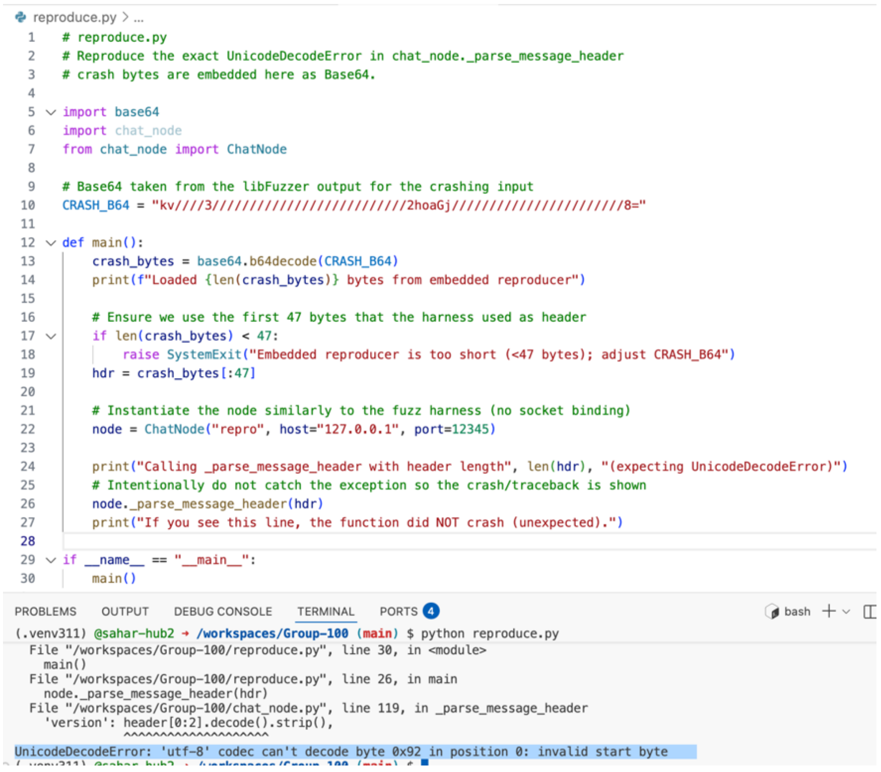
juanifioren/django-oidc-provider   Example 1/3

```
-  secret =
   md5(uuid4().hex.encode()).hexdigest()
+  secret =
   sha224(uuid4().hex.encode()).hexdigest()
```

Figure 2: Snyk Report

2. Dynamic analysis (Fuzzing):

To complement static analysis, I applied dynamic testing using the coverage-guided Python fuzzer **Atheris** to evaluate the robustness of the chat_node module against malformed inputs. I created a focused fuzzing harness (fuzz_target.py) that exercised internal routines, *_parse_message_header*, *_handle_route_update, _handle_diagnostic*, and the *encrypt/decrypt* functions, by feeding a wide variety of byte sequences and monitoring for crashes or abnormal behavior. Running this harness (python fuzz_target.py) reliably produced a runtime crash: a *UnicodeDecodeError* in *_parse_message_header* when non-UTF-8 header bytes were encountered. Atheris/libFuzzer automatically logged the failure, generated a corpus of interesting inputs, and saved the crashing input (crash-*) for reproducibility. To make it easier for peers to test and understand the issue without re-running the fuzzer, I embedded this exact input (Base64) into a standalone reproducer script (*reproduce.py*), which deterministically reproduces the same crash. This approach demonstrates the value of fuzz testing in uncovering real runtime vulnerabilities, such as insufficient input validation and weak error handling, which can lead to denial-of-service conditions. By examining this case, we can appreciate the importance of combining static and dynamic testing: static tools like Bandit identify potentially risky code patterns, while fuzzing shows whether these patterns can actually be exploited at runtime. *Figure 3,* showing the crash input in *reproduce.py*, provides concrete evidence of the issue. Applying similar fuzzing techniques to your own code can help detect subtle runtime bugs early and strengthen overall resilience, ultimately making software safer and more reliable.



reproduce.py > ...

```python
# reproduce.py
# Reproduce the exact UnicodeDecodeError in chat_node._parse_message_header
# crash bytes are embedded here as Base64.

import base64
import chat_node
from chat_node import ChatNode

# Base64 taken from the libFuzzer output for the crashing input
CRASH_B64 = "kv////3//////////////////////////2hoaGj///////////////////////8="

def main():
    crash_bytes = base64.b64decode(CRASH_B64)
    print(f"Loaded {len(crash_bytes)} bytes from embedded reproducer")

    # Ensure we use the first 47 bytes that the harness used as header
    if len(crash_bytes) < 47:
        raise SystemExit("Embedded reproducer is too short (<47 bytes); adjust CRASH_B64")
    hdr = crash_bytes[:47]

    # Instantiate the node similarly to the fuzz harness (no socket binding)
    node = ChatNode("repro", host="127.0.0.1", port=12345)

    print("Calling _parse_message_header with header length", len(hdr), "(expecting UnicodeDecodeError)")
    # Intentionally do not catch the exception so the crash/traceback is shown
    node._parse_message_header(hdr)
    print("If you see this line, the function did NOT crash (unexpected).")

if __name__ == "__main__":
    main()
```

```
(.venv311) @sahar-hub2 → /workspaces/Group-100 (main) $ python reproduce.py
  File "/workspaces/Group-100/reproduce.py", line 30, in <module>
    main()
  File "/workspaces/Group-100/reproduce.py", line 26, in main
    node._parse_message_header(hdr)
  File "/workspaces/Group-100/chat_node.py", line 119, in _parse_message_header
    'version': header[0:2].decode().strip(),
               ^^^^^^^^^^^^^^^^^^^^
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x92 in position 0: invalid start byte
```

Figure 3: reproduce.py

Analysis of Test Results

Static scans flagged risky patterns: binding to 0.0.0.0 (exposes the node to the network), use of hashlib.md5 for cryptographic purposes (weak/deterministic), and several broad try/except: pass blocks (silenced errors). These issues create real risks: an attacker who can reach the service (because it listens on all interfaces) can probe it and deliver crafted inputs; deterministic IVs from MD5 make AES-CBC ciphertexts linkable or vulnerable to chosen-plaintext correlation attacks (an attacker who can observe multiple messages with repeated prefixes may deduce relationships or, with additional weaknesses, recover plaintext); and swallowed exceptions can let attacks silently corrupt state or bypass defenses. Concretely, an adversary could remotely scan for nodes, send repeated messages with known prefixes to detect reuse patterns in ciphertexts, and exploit hidden failures to escalate an otherwise minor bug into a persistent compromise. To reduce these risks, make the bind address configurable (and default to localhost for development), replace MD5/CBC patterns with authenticated encryption (e.g., AES-GCM or high-level APIs in cryptography), and replace broad exception swallowing with targeted handlers and logging so suspicious inputs trigger observable alerts rather than silent state corruption.

Moreover, Fuzzing produced a deterministic crash in _parse_message_header when non-UTF-8 or malformed header bytes were supplied, demonstrating a reliable remote denial-of-service if the node accepts untrusted bytes. In practice an attacker could automate sending malformed headers (simple network script) to repeatedly crash one or many nodes (amplified across the overlay), causing service outages or forcing restarts that open windows for further attacks. Moreover, this crash can be combined with other issues we found: an attacker may first use a DoS to force reconnection flows and then trigger an unauthenticated diagnostic (admin_debug or DIAG_REQUEST) to exfiltrate private keys, or use predictable IVs to correlate ciphertexts observed during reconnection. High-value mitigations are straightforward: validate header size and expected byte ranges before decoding, guard decode calls with explicit error handling (decode(..., errors='ignore'/'replace') or conditional checks), and remove or strongly authenticate any debug/diagnostic entry points so they cannot be invoked by arbitrary peers. These focused fixes stop simple exploit scripts and make the system much harder to compromise in chained attacks.

Vulnerabilities

Vulnerability: Malformed header crash (Denial of Service)
Title: Crash on malformed message header during header parsing (UnicodeDecodeError)
Severity: High (Denial of Service)
Location: chat_node.py → _parse_message_header (header decode at ~line 119)
Type: Uncaught exception / input validation flaw
CWE: CWE-20 (Improper Input Validation); related CWE-693/CWE-703 (missing defensive error handling)
Summary: Fuzzing produced a header input that causes _parse_message_header to attempt to decode non-UTF-8 bytes as UTF-8, raising a UnicodeDecodeError and terminating the process. This is a runtime robustness issue: untrusted bytes can crash the node.

Evidence / How found: Atheris/libFuzzer run generated a crashing input and stack trace showing the exception at header[0:2].decode(); libFuzzer wrote a crash artifact. Reproducer (reproduce.py) with embedded Base64 reproduces the crash deterministically.
Reproduction (commands):

1. Create venv / activate (Python 3.11): such as > python3.11 -m venv .venv311 && source .venv311/bin/activate
2. Install deps: python -m pip install atheris cryptography
3. Run harness: run python reproduce.py with embedded crash bytes

Impact / Exploitation: Remote Denial-of-Service: an attacker who can send raw bytes to the node can repeatedly crash it, causing availability loss. Combined with other flaws (e.g., debug paths), DoS can be escalated into further compromise.
Mitigation: Validate header length and contents before decoding; perform boundary checks and explicit byte-range validation; use decode(..., errors='replace') or better, avoid decoding until format is verified; add narrow try/except that logs and ignores malformed headers. Add fuzzing into CI to catch regressions.
Confidence: High (deterministic fuzzer crash + reproducer).

Vulnerability: Hardcoded bind to all interfaces (Network exposure)
Title: Server bound to all interfaces (0.0.0.0) — excessive network exposure
Severity: Medium
Location: chat_node.py → ChatNode.__init__ default host="0.0.0.0" (constructor; bind at start() around line ~200)
Type: Misconfiguration / exposure of attack surface
CWE: CWE-605 (Multiple Bind to All Interfaces)
Summary: The node defaults to binding to 0.0.0.0, which listens on all network interfaces. This increases the attack surface by allowing remote attackers on the network to reach the service.
Evidence / How found: Bandit static analysis flagged hardcoded_bind_all_interfaces (B104). Manual code inspection confirmed host="0.0.0.0" default and bind((self.host, self.port)) in start().
Reproduction / Detection steps: Run grep -n "0.0.0.0" chat_node.py or bandit -r . to see the B104 finding. Start node and check ss/netstat to see service listening on 0.0.0.0.
Impact / Exploitation: Remote attackers on any reachable network can connect and probe the node, increasing risk that other vulnerabilities (e.g., header parsing crash, diagnostic triggers) will be exploited. In multi-tenant or public networks, this could enable wide DoS or information exposure.
Mitigation: Make bind address configurable and default to 127.0.0.1 for development; require explicit opt-in for public binds; document and validate intended deployment mode; add network access controls (firewall, ACLs).
Confidence: Medium (clear static finding, high practical risk).

Vulnerability: Weak MD5-derived IV / Legacy compatibility path (Cryptography weakness)
Title: Deterministic IV derived from MD5 for AES-CBC in legacy mode
Severity: High (cryptographic weakness / confidentiality risk)

Location: chat_node.py → _encrypt_payload (legacy branch where b"LEGACY_MODE" in payload leads to iv = hashlib.md5(payload[:16]).digest() around line ~135)
Type: Weak cryptographic primitive / predictable IV usage
CWE: CWE-327 (Use of a Broken or Risky Cryptographic Algorithm) / CWE-310 (Cryptographic Issues)
Summary: When a payload contains the LEGACY_MODE marker, the code derives an AES-CBC IV deterministically using MD5 over payload bytes. Deterministic IVs break CBC security assumptions and MD5 is cryptographically weak.
Evidence / How found: Bandit flagged MD5 usage (B324); manual review located the legacy branch and identified IV derived from hashlib.md5(payload[:16]).digest(). Snyk also flagged MD5 as insecure.
Reproduction / Detection steps: Search for LEGACY_MODE and hashlib.md5 in the source; confirm behavior by calling _encrypt_payload with b"LEGACY_MODE" + ...
Impact / Exploitation: Predictable IVs enable ciphertext correlation and can assist plaintext recovery in repeated-prefix scenarios; an active attacker who can cause repeated messages with controlled prefixes can exploit this to infer message patterns or, with additional weaknesses, recover plaintext. Overall confidentiality is weakened and replay/correlation attacks become easier.
Mitigation: Remove deterministic IV code; use secure random IVs (os.urandom(12 or 16)) per message; prefer authenticated encryption (e.g., AES-GCM) via high-level cryptography APIs; remove legacy compatibility or handle it with secure migration.
Confidence: High (clear insecure crypto pattern and corroborated by static tools).

Vulnerability: Diagnostic / admin_debug backdoor exposing private key (Sensitive data exposure)
Title: Debug/diagnostic path can leak node private key (private_key_pem)
Severity: Critical (sensitive key exposure)
Location: chat_node.py → _handle_hello sets debug_mode when username.lower() == 'admin_debug'; _handle_diagnostic may return private_key_pem in response when self.debug_mode or payload == b"DIAG_REQUEST" (around lines ~360 and ~520)
Type: Sensitive data exposure / backdoor / insufficient authentication
CWE: CWE-200 (Exposure of Sensitive Information) / CWE-307 (Improper Restriction of Excessive Authentication Attempts)
Summary: A special username (admin_debug) enables debug_mode; the diagnostic handler returns the node's private key PEM when debug mode is active or on DIAG requests. This allows any peer who triggers the debug path to obtain private keys and compromise encryption.
Evidence / How found: Manual code inspection flagged the admin_debug check and diagnostic response building private_key_pem (private key bytes) which are JSON encoded and sent over the wire. Bandit-style review highlights unprotected debug actions.
Reproduction / Detection steps: Send a HELLO with username: "admin_debug" to a running node (or call _handle_diagnosticwith payload == b"DIAG_REQUEST" in a local test); observe diagnostic response containing private_key_pem. (Do not do this on production systems.)
Impact / Exploitation: Full compromise of confidentiality and authentication: any adversary who obtains the private key can decrypt session keys, read past and future encrypted messages, and impersonate the node to other peers. This is a catastrophic security failure.

Mitigation: Remove unauthenticated debug backdoors. If diagnostics are needed, require strong authentication and authorization (e.g., mutual TLS, pre-shared admin credentials), and never transmit private keys — expose only non-sensitive metadata. Rotate keys on suspicion and store private keys securely (hardware or OS keystore).
Confidence: High (direct code path exposes private key).

Vulnerability: Broad try/except: pass blocks (Error swallowing / robustness)
Title: Unqualified exception handlers that swallow all exceptions (except: / pass)
Severity: Low → Medium (operational risk; can mask higher-impact issues)
Location: Multiple places (examples at ~lines 235, 303, 543, 704, 729, 760) — used when closing sockets, cleaning up peers, decrypting route updates, broadcasting, etc.
Type: Poor error handling / logging omission
CWE: CWE-703 (Improper Check or Handling of Exceptional Conditions)
Summary: The code contains several unqualified except: blocks that silently swallow all exceptions. This hides failures, reduces observability, and may leave the program in inconsistent states. Combined with other vulnerabilities it increases the chance of unnoticed exploitation.
Evidence / How found: Bandit flagged multiple try_except_pass instances (B110). Manual inspection shows many except:with pass.
Reproduction / Detection steps: Static scan with Bandit (bandit -r .) or grep for except: followed by pass. Instrument code to throw exceptions in those blocks to observe silent behavior.
Impact / Exploitation: Silent failures may allow attackers to hide the effects of malformed inputs, cause partial cleanups or resource leaks, and obscure attack traces. It can also complicate debugging and recovery after an attack. While not directly exploitable in isolation, it amplifies other vulnerabilities and complicates incident response.
Mitigation: Replace broad except: with targeted exception types (e.g., except OSError:, except ValueError:), log exception details, and where possible implement retries or safe cleanup. Ensure errors are visible to operators and tests cover error paths.
Confidence: High (clear static issue; code presence is obvious).

Short aggregated guidance :

- **Network:** Don't default to 0.0.0.0; make bind address configurable and default to localhost during development.
- **Crypto:** Remove MD5 use; always use secure random IVs and authenticated encryption (AES-GCM via cryptographyhigh-level APIs).
- **Diagnostics:** Remove unauthenticated debug/backdoor behavior; never send private keys; require strong auth for diagnostics.
- **Input handling:** Validate message headers and payloads before decoding; add defensive parsing and explicit error handling.
- **Error handling:** Replace except: pass with targeted handlers and logging to preserve observability.

Summary and Overall Feedback

The implementation demonstrates thoughtful design choices and strong technical competence, particularly through its layered security model and efficient handling of concurrent peers. The use of structured message headers, session-specific encryption keys, and synchronized data access mechanisms reflects a solid understanding of secure and reliable system architecture. However, the analysis also identifies several areas where focused improvements could further enhance the system's resilience. Replacing the MD5 hashing algorithm with modern alternatives such as SHA-256, restricting network bindings to trusted interfaces, and adopting safer decoding strategies for untrusted input would substantially reduce the risk of exploitation. These refinements not only address the identified vulnerabilities but also promote secure coding practices that improve maintainability and long-term reliability. Overall, this feedback is intended to support Group 100 development team and encourage the continued development of systems that are both robust and confidently secure.