Peer Review – Security Programming

Group: Project Group 12

Reviewer: Aditya Sajeesh

1. Executive Summary

Your repository implements the SOCP chat system, using RSA-OAEP to encrypt direct messages and RSA-PSS for the transport signatures. Two intentionally planted, non-destructive backdoors exist and are documented in the BACKDOOR_README file. These backdoors materially weaken the cryptographic assurances by using RSA-1024 key generation, and bypass signature verification during inter-server USER_ADVERTISE gossip.

Beyond implementing the SOCP and 2 intentional backdoors, the code demonstrates a sensible structure and contains multiple security checks (message freshness, dedup, PSS verification). However, there are some protocol and defensive gaps (tolerating missing transport signatures in client code, storing/accepting remote pubkeys without size/format enforcement). As a result, attackers who choose to exploit these vulnerabilities may potentially be able to impersonate users, inject false USER_ADVERTISE or cause clients to use insecure, weak keys, thereby enabling offline key recovery or MitM attacks. Because the backdoors are environment-variable gated, accidental exposure (such as running in development mode with the aforementioned env vars set) or a mis-merged branch would enable exploitation.

2. Identified Vulnerabilities

2.1 Intentional Backdoors

Section 2.1 identifies the 2 intentional backdoors which are present in the submitted code. The backdoors are mapped within the project's BACKDOOR_README.md: a weak-key backdoor (RSA-1024) and the acceptance of invalid/unsigned USER_ADVERTISE. The impacts of these vulnerabilities will be discussed in Section 2.3.

2.2 Protocol Compliance Issues

Section 2.2 highlights areas within the submitted code which are in violation of the Secure Chat Overlay Protocol (SOCP) which was agreed upon by the whole cohort.

SOCP Section 18 (Compliance Checklist) specifies that the keys must be RSA-4096 keys; at current, the keys used are RSA-1024. Further, `server.py` and `datavault.py` currently accept and store public keys without validating the modulus size or algorithm metadata; `datavault.users.pubkey` stores `pubkey_b64u` values as-is in `register_user`. This is in direct violation of Section 18, as RSA-OAEP (SHA-256) must be used for encryption of public keys.

Within `client.py`, `verify_transport_sig` returns True if `msg.get("sig")` is absent. Conversely, the server's `bootstrap_with_introducer` appropriately verifies the Introducer's signature, however the presence of BACKDOOR_TRUST_GOSSIP indicates that the server can be run to accept unsigned advertisements (identified previously in 2.1). This is in violation of SOCP's Processing rules, which states that your implementation must "Verify sig using from [sic] server's public key." The acceptance of unsigned advertisements is also in violation of Section 4, which states that "All payloads MUST be signed using RSASSA-PSS (SHA-256)."

Finally, your server advertisement flows accept `pubkey_b64u` and then converts via `der_b64url_to_public_pem` even though the `keys.is_uuid_v4` function exists. The gen_introducer_keys.py function, alongside introducers.yaml are used for bootstrapping but no strict parsing/enforcement is visible. While this may work in theory, performing such a roundabout and convoluted identifier conversion can cause ambiguity or collisions when attempting to communicate with other servers.

2.3 Security Vulnerabilities

Section 2.3 explains the vulnerabilities which arise from each of the aforementioned backdoors and protocol compliance issues.

Weak-key generation and acceptance: Attackers will easily be able to perform offline private-key recovery due to the weak key encryption. Within a live environment, attackers may be able to perform a Man-in-the-Middle (MitM) attack or even decrypt messages sent between users in a public or private setting.

Signature verification bypass during inter-server gossip: As a result of running your server with the BACKDOOR_TRUST_GOSSIP flag, attackers will be able to impersonate legitimate users, inject malicious advertisements or even redirect messages between two legitimate users to the attacker's nodes.

Permissive error handling, dev tolerances: As a result of incorrect handling of missing signatures in verify_transport_sig (along with dummy privilege placeholders in your register_user's server registration flow), it is possible for attackers to easily exploit or bypass other security measures which have been implemented elsewhere in your application.

Beyond the backdoors and security vulnerabilities, there is a very serious issue of note: your repository history shows sensitive artifacts being committed to the repo root: .keys files and data_vault.sqlite (along with their WAL/SHM) are present – an attacker viewing your repo commit history can identify that these files contain private keys and DB snippets respectively, immediately compromising your application.

3. Code Quality Assessment

Your code is structured clearly. Each of your files are named appropriately and correspond to their designated function within your application. Cryptography primitives RSA-OAEP and RSASSA-PSS are used for content and transport protection respectively (except specific instances as identified in 2.2, 2.3). Further, you have complied with some of the SOCP's mandates within your implementation: freshness checks, de-duplication and server bootstrap signature verification path are some of your application's positive implementations stemming from the SOCP. You have also helpfully provided transparent documentation within BACKDOOR_README.md to assist in easily identifying the backdoors present in your app.

Despite these strengths, the overall security policy enforcement within your app is inconsistent: the key-size checks and signature enforcement are lax in some areas and gated by environment variables in others. Your app tolerates missing transport signatures by returning True in verify_transport_sig. The repository's secrets (.keys and data_vault.sqlite) are not scrubbed from the repo tree, enabling attackers a direct path to compromise the integrity of your server. Error handling is optimistic at best, and silently ignores critical failures at worst. Finally, there is no evidence of testing (manual or automated) within your repository.

Overall, your code is readable and pragmatic, but security enforcement must be hardened and operational hygiene improved in order to fully secure your application.

4. Recommendations

This section will provide recommendations to improve your application. The recommendations are provided in order of highest priority to lowest.

1. **Enforce minimum key sizes and reject weak keys at import**: A check should be added in `keys.der_b64url_to_public_pem`, or a new validation wrapper to reject RSA keys where the `pub.key_size < 2048` (preferably 4096, to keep in line with SOCP specification).
2. **Remove environment-gated signature bypasses**: Eliminate your second intentional backdoor (or default it to disabled) and refuse unsigned/invalid `USER_ADVERTISE` messages. It is better to fail closed rather than fail open.
3. **Fix client-side signature tolerance**: Change `client.py.verify_transport_sign` to return False on missing signatures, add explicit handling/logging to help debugging without accepting unsigned frames silently.
4. **Remove committed secrets**: Remove `.keys` and `data_vault.sqlite` from the repository and its history (use BFG or git filter-repo) and rotate all the keys and secrets. You should add `.gitignore` entries for `.keys` and `data_vault.sqlite` so that Git does not include these within future commits.
5. **Add strict wire-format validation**: Canonical JSON should be validated for signed payloads. UUIDs should be lowercased and canonical. Check the `pubkey_b64u` length.
6. **Integrate replay and rate-limiting protections**: Limit the number of advertisements accepted per remote server per minute. Add replay windows and nonce checks where appropriate.
7. **Use mutual TLS for server-to-server transport in addition to signatures**: This reduces the impact of a compromised signalling channel and adds defense-in-depth
8. **Add automated tests, evidence of manual testing**: Less of a security feature, but would be great to demonstrate evidence of testing in your final submission
9. **Document secure deployment of client**: After following the steps above, update your README with secure defaults and remove documentation of backdoor code + the code itself prior to resubmitting your “production-ready” code