Pashov Audit Group

BOB Gateway Security Review

August 27th 2025 - August 30th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About BOB Gateway

BOB is a hybrid Layer-2 powered by Bitcoin and Ethereum. The design is such that Bitcoin users can easily onboard to the BOB L2 without previously holding any Ethereum assets. The user coordinates with the trusted relayer to reserve some of the available liquidity, sends BTC on the Bitcoin mainnet and then the relayer can provide a merkle proof to execute a swap on BOB for an ERC20 token.

BOB Gateway is a bridge solution that enables Bitcoin users to seamlessly onboard to the BOB L2 by swapping BTC for Ethereum-based assets (like wBTC or tBTC) via a trusted relayer and smart contracts, without requiring pre-existing ETH holdings.

5. Executive Summary

A time-boxed security review of the **bob-collective/bob-gateway** repository was done by Pashov Audit Group, during which **Newspace**, **LordAlive**, **DadeKuma** engaged to review **BOB Gateway**. A total of **4** issues were uncovered.

Protocol Summary	
Project Name	BOB Gateway
Protocol Type	Hybrid Bitcoin Layer 2
Timeline	August 27th 2025 - August 30th 2025

Review commit hash:
- 6ad820b234dee5fb8f5c72d3e2b58bc7ed1997b9 (bob-collective/bob-gateway)

Fixes review commit hash:
- 4622efb85313fec1ff764861675a54c89a4290f6 (bob-collective/bob-gateway)

Scope

`api_client.rs` `app.rs` `error.rs` `main.rs` `models.rs` `scan_and_process_orders.rs` `utils.rs`

6. Findings

Findings count

Severity	Amount
High	1
Medium	1
Low	2
Total findings	**4**

Summary of findings

ID	Title	Severity	Status
[H-01]	A lack of a recovery mode can result in locked funds for users	High	Resolved
[M-01]	Users always pay max fees when an order is accepted	Medium	Resolved
[L-01]	Inconsistent error classification and data leakage risk	Low	Resolved
[L-02]	Incorrect error log	Low	Resolved

High findings

[H-01] A lack of a recovery mode can result in locked funds for users

Severity

Impact: High

Likelihood: Medium

Description

The solver is a backend program that continuously polls the L2 chain to identify orders that can be accepted, enabling funds to be bridged to BTC.

The order fulfillment process involves the following sequential steps:

1. Update BTC balance.
2. For each new order, verify its status is valid.
3. For each active order, execute `accept_and_send_btc_tx`:

3.1. Check order rate limits.

3.2. Verify sufficient balance to fulfill the order.

3.3. Create/sign the BTC transaction.

3.4. Validate all required BTC addresses.

3.5. Accept the order on L2 (modifies state).

3.6. Update the database with BTC transaction information (modifies state).

3.7. Send the BTC transaction (modifies state).

The core issue is that some of these steps are not idempotent. State is persisted after each individual step rather than within a transactional boundary.

This means that if the program is halted at any point during the process (e.g., due to a power outage, application panic, or unexpected termination), an order can become stuck in an inconsistent state, resulting in locked funds.

For example, if step 3.5 (accepting the order on L2) succeeds, but the server stops before step 3.6 (updating the database), the order is marked as "accepted" on-chain, but funds are never bridged.

When the server restarts, the order will not be reprocessed because it is already considered accepted, leaving the user's funds locked indefinitely.

Recommendations

Consider implementing a recovery mode for the process.

An example implementation could include the following steps:

1. Record the intent to accept an order in the database before accepting the order.
2. Modify the order processing loop to check both all active orders and accepted orders that have not yet been bridged (saved on intent DB but not bridged yet).
3. Use event logs from the `acceptOrder` function to verify that an order is accepted by the legit solver to prevent double-spending.

Medium findings

[M-01] Users always pay max fees when an order is accepted

Severity

Impact: Medium

Likelihood: Medium

Description

When a user submits an order, they specify the maximum amount of fees they are willing to pay (`order.satFeesMax`). The solver then calculates the expected fees, which include both the inclusion fee and the protocol fee.

The issue is that the system consistently charges the user the full maximum amount (`order.satFeesMax`) rather than the actual calculated fee amount.

This results in users always paying the maximum possible fees even when the actual cost of the operation is significantly lower, analogous to setting a gas limit and always being charged the full amount regardless of actual gas used.

Recommendations

Consider the following changes:

```
    let amount_to_send = Amount::from_sat(
-       (order.satAmountLocked - order.satFeesMax)
+       (order.satAmountLocked - overall_expected_fees)
            .try_into()
            .map_err(|_| Error::ConversionError)?,
    );
```

Low findings

[L-01] Inconsistent error classification and data leakage risk

The error-handling flow currently mixes transient (retryable) issues with permanent failures and logs raw error messages directly into tracing or external systems such as Sentry. This can cause:
- Temporary network issues are being escalated as critical errors.
- Business logic errors (e.g., rate-limit exceeded) are being retried indefinitely.
- Sensitive request/response details are being leaked into monitoring systems.

Recommendation - Define a clear error taxonomy separating transient, permanent, and critical errors.
- Sanitize error messages before logging or sending to Sentry, removing sensitive data.
- Restrict retry logic to transient errors only, while surfacing permanent ones immediately.

[L-02] Incorrect error log

The following error is thrown in `verify_order_details()` when order has insufficient fees to cover protocol and inclusion fees:

```
    // Ensure the order has sufficient fees to cover protocol and inclusion fees
    if fees_in_sats < overall_expected_fees {
        return Err(Error::FeesTooLowToAcceptOrder(
            params.id.to_string(),
            overall_expected_fees,
            fees_in_sats,
        ));
    }
```

However, the `FeesTooLowToAcceptOrder` is defined as follows in `errors.rs`:

```
>>  #[error("Cannot process order with ID {0}: as expected fees {1} given fees {1}.")]
    FeesTooLowToAcceptOrder(String, BigDecimal, BigDecimal),
```

As seen, no matter how correct the parameters are provided, the message remains distorted.

Also in `start_services()`, `offramp_registry_address` is logged as offramp owner in the error message as follows:

```
    let receiver =
        offramp_registry.solvers(offramp_owner_address, token_address).call().await?.recipient;

    if receiver == Address::ZERO {
        eyre::bail!(
            "offramp with owner address {} and token address {} is not registered on-chain, or
the receiver is not set.",
>>          offramp_registry_address,
```

```
            token_address,
        );
    }
```

The actual owner address is `offramp_owner_address` as used to derive the `receiver` and as such the message is incorrect.

Recommendations

```
-   #[error("Cannot process order with ID {0}: as expected fees {1} given fees {1}.")]
+   #[error("Cannot process order with ID {0}: as expected fees {1} given fees {2}.")]
    FeesTooLowToAcceptOrder(String, BigDecimal, BigDecimal),
```

and

```
    if receiver == Address::ZERO {
        eyre::bail!(
            "offramp with owner address {} and token address {} is not registered on-chain, or
the receiver is not set.",
-           offramp_registry_address,
+           offramp_owner_address,
            token_address,
        );
    }
```