BOB Onramp Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, carrotsmuggler, SpicyMeatball

April 19th 2024 - April 22th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **bob-collective/bob-onramp** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About BOB Onramp

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. The liquidity provider (LP) first locks that token in Onramp.sol as well as some small amount of ETH to allow that user to do some swaps on BOB. The LP receives Bitcoin to their specified address and can re-balance by converting that to the wrapped token and re-depositing.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes - 9bf1966f9469d2b13d426402da8adb6059a30096

fixes review commit hashes - 403389e4fae9d4a0dd59fa9e419cb50442cd33f3

Scope

The following smart contracts were in scope of the audit:

- `OnrampFactory`
- `Onramp`

7. Executive Summary

Over the course of the security review, 0xunforgiven, carrotsmuggler, SpicyMeatball engaged with BOB to review BOB Onramp. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	BOB Onramp
Repository	https://github.com/bob-collective/bob-onramp
Date	April 19th 2024 - April 22th 2024
Protocol Type	Hybrid Layer 2

Findings Count

Severity	**Amount**
Medium	5
Low	5
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[M-01]	getTxOutputValue() doesn't sum up all the values	Medium	Resolved
[M-02]	BTC transactions can have multiple inputs and outputs	Medium	Resolved
[M-03]	Race condition for bridged BTC	Medium	Acknowledged
[M-04]	Malicious recipient can gas grief the relayer	Medium	Acknowledged
[M-05]	Multiple onramps can have the same scriptPubKeyHash	Medium	Acknowledged
[L-01]	Tokens with decimals <8 not supported	Low	Resolved
[L-02]	Bitcoin transactions can have large outputVector	Low	Acknowledged
[L-03]	validateProof uses encodePacked	Low	Acknowledged
[L-04]	LP params changed while users are waiting	Low	Acknowledged
[L-05]	Replaying the same BTC transactions on different LPs	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] `getTxOutputValue()` doesn't sum up all the values

Severity

Impact: High, user bridged funds will be lost.

Likelihood: Low, a user should repeat target address multiple times in `outputVector`

Description

To calculate the bridged amount code calls `getTxOutputValue()` to find the transferred amount to the onramp address. That function only took the first value of the target address in `outputVector` list. The issue is that the same address may repeat multiple times in the `outputVector` and the code should sum all of them to calculate the total amount transferred to the target address.

Recommendations

Sum all the values for `scriptPubKeyHash` in `outputVector` list.

[M-02] BTC transactions can have multiple inputs and outputs

Severity

Impact: Medium

Likelihood: Medium

Description

BTC txs can have multiple inputs and outputs and the relayer chooses the `reciever` and `onramp` contract based on the BTC transaction after users make the BTC transactions. The issue is that it's not clear which input address would be chosen as a token receiver and which output address is going to be chosen as an onramp contract. Funds may go to a different receiver address or the relayer may use the wrong onramp contract. If the relayer handles only specific BTC txs users should be informed about relayer limitations.

Recommendations

Ensure the relayer uses the correct recipient address.

[M-03] Race condition for bridged BTC

Severity

Impact: High, users would lose bridged BTC.

Likelihood: Low, requires special conditions.

Description

To bridge BTC users need to transfer their funds to the onramp address (`scriptPubKeyHash`) and then the relayer would call `proveBtcTransfer()` and the target onramp would transfer the bridged token to the user. The issue is that multiple users may have transferred their BTC to that address at the same time and the onramp may not have enough tokens to handle all those orders so some of the users would lose their funds.

Recommendations

Add queue mechanism for off-chain relayer and handle users one by one. Or don't ask users to transfer BTC directly to the onramp contract's `scriptPubKeyHash`. gather BTC in the relayer BTC address and when the bridge was successful in the BOB then transfer onramp's BTC.

[M-04] Malicious recipient can gas grief the relayer

Severity

Impact: High

Likelihood: Low

Description

When a user sends BTC on the Bitcoin mainnet, the relayer attempts to execute the swap on BoB network with onramp contract

```
function proveBtcTransfer(
        BitcoinTx.Info memory _txInfo,
        BitcoinTx.Proof memory _txProof,
        address payable _recipient,
        Onramp _onramp
    ) external onlyOwner {
        require(getOnramp[address(_onramp)], "Onramp does not exist");

        bytes32 txHash = relay.validateProof
          (txProofDifficultyFactor, _txInfo, _txProof);

        uint256 outputValueSat = BitcoinTx.getTxOutputValue
          (_onramp.scriptPubKeyHash(), _txInfo.outputVector);

        _onramp.executeSwap(txHash, outputValueSat, _recipient);
    }
```

The on-ramp contract transfers 1:1 amounts of wrapped BTC and some extra ETH tokens as gratuity.

```
function executeSwap(
      bytes32_txHash,
      uint256_outputValueSat,
      addresspayable_recipient
    ) external onlyFactory {
         ---SNIP---

        uint256 amount = calculateAmount(_outputValueSat - feeSat);

        emit ExecuteSwap(_recipient, _outputValueSat, feeSat, amount, gratuity);

        // transfer token
        token.safeTransfer(_recipient, amount);

        // slither-disable-next-line arbitrary-send-eth,low-level-calls
>>      (bool sent,) = _recipient.call{value: gratuity}("");
        require(sent, "Could not transfer ETH");
    }
```

A malicious recipient contract can take advantage of this, by implementing a `receive` function in which it spends all the transaction gas, reverting the function to grief the relayer. As there is no minimum limit on the BTC amount, except for the `dustThreshold` which can be relatively small, this attack can be repeatedly executed with small amounts of tokens, wasting the relayer's gas. In the worst-case scenario, this could lead to a Denial of Service (DoS) of the protocol.

Recommendations

Create a pull mechanism, instead of sending ETH, let recipients claim it.

[M-05] Multiple onramps can have the same scriptPubKeyHash

Severity

Impact: High

Likelihood: Low

Description

Each onramp contract has `scriptPubKeyHash` address which users deposit to those addresses to bridge their BTC, the issue is that the code doesn't check to make sure different onramp contracts have the different `scriptPubKeyHash` address. This would create issues and have multiple impacts based on the off-chain relay implementation:

1. Users can deposit and receive tokens multiple times from onramp contracts that have the same `scriptPubKeyHash`.
2. LP provider can set duplicate `scriptPubKeyHash` for his onramp contracts and front-run other users BTC transfers self-bridge his tokens and receive his funds in onramp and cause loss for users.
3. LP provider can set duplicate `scriptPubKeyHash` for different onramp contracts and cause users to lose funds because of different fees in those onramps.

Recommendations

Contract shouldn't allow different onramp contracts to have the same `scriptPubKeyHash` address.

8.2. Low Findings

[L-01] Tokens with decimals <8 not supported

Creating an on-ramp contract for tokens with fewer than 8 decimal places is impossible due to underflow in the constructor.

```
constructor(
        address _owner,
        ERC20 _token,
        bytes32 _scriptPubKeyHash,
        uint64 _feeDivisor,
        // should be a very small amount
        uint64 _gratuity
    ) Ownable() {
        transferOwnership(_owner);
        factory = msg.sender;
        token = _token;
>>      multiplier = 10 ** (token.decimals() - 8);
```

It may not be a problem for wBTC and tBTC tokens, but it could be an issue if a bridged BTC token with fewer than 8 decimals is created in the future. Consider refactoring `calculateAmount` function:

```
function calculateAmount
      (uint256 _outputValueSat) public view virtual returns (uint256) {
+       uint8 decimals = token.decimals();
+       uint256 amountSubFee = decimals >= 8 ? _outputValueSat * 10 **
+ (decimals - 8) : _outputValueSat / 10 ** (8 - decimals);
-       uint256 amountSubFee = _outputValueSat * multiplier;
        return amountSubFee;
    }
```

[L-02] Bitcoin transactions can have large `outputVector`

When a user wants to bridge their tokens, the relayer calls `proveBtcTransfer()` to finalize the transfer. The issue is that the code calls `getTxOutputValue()` to find the bridged amount and that function loops through all the `outputVector` of the bitcoin tx. The size of `getTxOutputValue` can be up to 400K and 3000 output so this may cause OOG for `proveBtcTransfer()` calls and users' bridged transactions can't be finalised.

[L-03] `validateProof` uses `encodePacked`

The function `validateProof` calculates the transaction hash.

```
txHash =
            abi.encodePacked(
              txInfo.version,
              txInfo.inputVector,
              txInfo.outputVector,
              txInfo.locktime
            ).hash256View(
```

The issue is that `txInfo.inputVector` and `txInfo.outputVector` are both variable-length bytes objects. This is unsafe when used with `encodePacked`. This is because `encodePacked` does not take into consideration the individual lengths and just puts them together, and thus can lead to hash collisions with different inputs.

As an example, the hash of `encodePacked([a,b],[c])` is identical to the hash of `encodePacked([a],[b,c])`. This also extends to bytes objects, which are also variable length.

This isn't exploitable in the current codebase but is still inadvisable. Consider using `abi.encode()` instead.

More info can be found here: https://swcregistry.io/docs/SWC-133/

[L-04] LP params changed while users are waiting

Block confirmations on the Bitcoin network can take up to 10 minutes. For multiple confirmations, users can wait for 20-30 minutes before the relayer initiates the transaction to mint tokens on BOB.

The issue is that there is no `deadline` parameter that users can pass in. So users can be made to wait a long time, and the state of the LP can change in this waiting time period.

Liquidity providers can change their parameters after calling `startUpdate` and waiting 6 hours. Within this time period, users are still able to utilize that pool for carrying out swaps. However, when this waiting period is about to end, there can be users waiting for block confirmations on the Bitcoin network who have already initiated their transactions. The pool owners can then change the parameters and update their fees, and the transaction would then take place and cost the user more fees than they imagined.

The main issue is that bridging off of bitcoin can take a long time, and has a variance of 10 minutes depending on when the last block was produced. On EVM-based chains with MEV, a lot can happen in these 10 minutes, especially if this time window coincides with the unlock time of a pool configuration.

Pools can force users to pay extreme fees (up to 50%), which they did not sign up for when they initiated the transaction.

Consider either adding a `slippage` param, which will force a certain amount of tokens out, or a `deadline` param which will prevent their swap from going through if it is too late.

[L-05] Replaying the same BTC transactions on different LPs

When a BTC bridging event is started, the relayer picks up the relevant transaction info and proof and submits it on the EVM chain (BOB) to mint out the respective tokens from an onramp. The system is designed to have multiple onramps, so the user can choose any ramp they wish.

The issue is that once the BTC transaction is used to do a mint, it must be negated, so it can never be used again. This is a very core design of every bridge and should be followed.

The issue here is that this transaction negation in the current design happens at the individual ramp level.

```
// Onramp.sol
spent[_txHash] = true;
```

The different ramps don't share this state. So even if a transaction is negated on Ramp A, it can be triggered again with Ramp B, since on that ramp the BTC deposit transaction has not been negated yet.

Thus, the relayer has the ability to use a single proof to mint out assets from every ramp. Even though the relayer is controlled by a trusted party, it should not have the power to re-mint using already used-up transactions.

The contracts in the current state essentially let the relayer double-spend the same BTC tokens on every ramp.

The transaction hash negation should be moved to the central `OnrampFactory` contract. This way the same txHash cannot be spent on two different ramps.