



# Taraxa Ficus Root Bridge Smart Contracts

Security Assessment (Summary Report)

August 16, 2024

*Prepared for:*

**Előd Varga**

Taraxa

*Prepared by:* **Alexander Remie and Justin Jacob**

# About Trail of Bits

---

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at <https://github.com/trailofbits/publications>, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow [@trailofbits](#) on Twitter and explore our public repositories at <https://github.com/trailofbits>. To engage us directly, visit our "Contact" page at <https://www.trailofbits.com/contact>, or email us at [info@trailofbits.com](mailto:info@trailofbits.com).

## **Trail of Bits, Inc.**

497 Carroll St., Space 71, Seventh Floor  
Brooklyn, NY 11215

<https://www.trailofbits.com>

[info@trailofbits.com](mailto:info@trailofbits.com)

# Notices and Remarks

---

## Copyright and Distribution

© 2024 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Taraxa under the terms of the project statement of work and has been made public at Taraxa's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

The sole canonical source for Trail of Bits publications is the [Trail of Bits Publications page](#). Reports accessed through any source other than that page may have been modified and should not be considered authentic.

## Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

# Table of Contents

---

<b>About Trail of Bits</b>	<b>1</b>
<b>Notices and Remarks</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>Project Summary</b>	<b>4</b>
<b>Project Targets</b>	<b>5</b>
<b>Executive Summary</b>	<b>6</b>
<b>Summary of Findings</b>	<b>8</b>
<b>Detailed Findings</b>	<b>9</b>
1. Lack of safeTransfer usage for ERC20	9
2. The add() function can revert	11
3. G1 and G2 from() method lack field point validation	12
4. Missing validation allows signatures to be duplicated to finalize any PillarBlock	13
5. Incorrect mapping key used in validation inside registerContract	15
6. Reentrancy in applyState can lead to breaking the contract and stealing hook-enabled tokens	17
7. Confusing application of settlementFee to locking native assets	20
<b>A. Vulnerability Categories</b>	<b>21</b>
<b>B. Code Quality Recommendations</b>	<b>23</b>
<b>C. Fix Review Results</b>	<b>26</b>
Detailed Fix Review Results	27

# Project Summary

---

## Contact Information

The following project manager was associated with this project:

**Sam Greenup**, Project Manager  
[sam.greenup@trailofbits.com](mailto:sam.greenup@trailofbits.com)

The following engineering director was associated with this project:

**Josselin Feist**, Engineering Director, Blockchain  
[josselin.feist@trailofbits.com](mailto:josselin.feist@trailofbits.com)

The following consultant was associated with this project:

**Alexander Remie**, Consultant      **Justin Jacob**, Consultant  
[alexander.remie@trailofbits.com](mailto:alexander.remie@trailofbits.com)      [justin.jacob@trailofbits.com](mailto:justin.jacob@trailofbits.com)

## Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
July 2, 2024	Pre-project kickoff call
July 15, 2024	Delivery of report draft; report readout meeting
August 16, 2024	Delivery of summary report with fix review

# Project Targets

---

The engagement involved a review and testing of the following target.

## bridge

Repository	<a href="https://github.com/Taraxa-project/bridge">https://github.com/Taraxa-project/bridge</a>
Version	f82dd87c7e23358c74f7dc37451ca61110b60942
Type	Solidity
Platform	EVM

## beacon-light-client

Repository	<a href="https://github.com/Taraxa-project/beacon-light-client">https://github.com/Taraxa-project/beacon-light-client</a>
Version	b2eafadf37e466f7cbcd3c5b1ca0a917f152137a
Type	Solidity
Platform	EVM

# Executive Summary

---

## Engagement Overview

Taraxa engaged Trail of Bits to continue a review of the security of its bridge smart contracts at commit f82dd87 and to conduct a best-effort review of the Beacon chain light client at commit b2eafad. The contracts within the bridge repo facilitate the bridging of tokens between the Taraxa and Ethereum chains. The Beacon chain light client facilitates light client proofs of a beacon chain (which is used by the bridge contracts).

Although this was meant to be a continuation of the review of the previous bridge review (commit ec89224), the provided commit for this audit (f82dd87) included an almost complete rewrite of the bridge smart contracts. Therefore, we spent a considerable amount of time on re-reviewing the entire implementation, instead of purely focusing on the parts not covered during the original audit. This limited the depth of our review (as well as our best-effort review of the Beacon chain light client).

A team of two consultants conducted the review from July 8 to July 14, 2024, for a total of 1.6 engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

## Observations and Impact

We identified seven issues during this short engagement, three of which are high severity. The first issue (**TOB-TARA-1**) shows how not using a “safe” ERC20 transfer function can lead to loss of funds for users when dealing with particular non-standard ERC20 tokens.

The second issue (**TOB-TARA-4**) allows anyone to arbitrarily inflate the votes for a given PillarBlock during finalization; i.e., it is possible to finalize PillarBlocks that did not receive enough votes from validators to pass the threshold. This issue highlights the importance of validating all inputs and correctly handling all edge-cases.

The third high-severity issue (**TOB-TARA-6**) shows a reentrancy-related vulnerability that allows hook-enabled tokens (ERC777) to be drained from a ERC20 locking connector. Additionally, whenever this reentrancy is performed, it will corrupt the internal state; this would break the contract and require an upgrade to fix. This issue highlights the danger of not adhering to the Checks-Effects-Interactions pattern and/or not using reentrancy guards. Additionally, this issue highlights the usefulness of using **Slither** to perform static analysis on Solidity smart contracts, as Slither detects this issue.

We also want to highlight that the inline documentation could be significantly improved. This would have likely uncovered **TOB-TARA-5**, and overall would make the implementation

easier to understand. In particular, many if-statements are hard to understand since there is no comment explaining why the conditions are defined as they are.

Our review of the `beacon-light-client` repo was a best-effort review that uncovered one finding ([TOB-TARA-3](#)). A full review of this repository could uncover more issues and is recommended.

## Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Taraxa Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Improve the inline documentation.** Writing inline comments may cause issues to become obvious. Additionally, this would make it easier to understand the implementation, especially the various if-statements it contains.
- **Perform a full audit of the `beacon-light-client` repository.** During this short review, we performed only a best-effort review of the `beacon-light-client` repo. Additionally, as stated in the Engagement Overview, the changing of commits required us to re-review the entire bridge repo; as a result, we had very limited time to review the `beacon-light-client`.
- **Perform another audit of the bridge repository once it is finalized.** As stated in the Engagement Overview section, due to changing of commits, the level of depth we achieved during this short review was insufficient for us to be reasonably confident that no more bugs are present. Performing another audit would increase confidence in the security of the implementation and would likely uncover more issues. Additionally, examination of the amount of changes between the previous and current commits of our reviews suggests that the bridge repo is not yet in a finished state. We recommend reviewing it once it has been finalized and no more changes are planned.



## Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of safeTransfer usage for ERC20	Data Validation	High
2	The add() function can revert	Denial of Service	Informational
3	G1 and G2 from() method lack field point validation	Data Validation	Informational
4	Missing validation allows signatures to be duplicated to finalize any PillarBlock	Data Validation	High
5	Incorrect mapping key used in validation inside registerContract	Undefined Behavior	Informational
6	Reentrancy in applyState can lead to breaking the contract and stealing hook-enabled tokens	Undefined Behavior	High
7	Confusing application of settlementFee to locking native assets	Arithmetic	Informational

# Detailed Findings

## 1. Lack of safeTransfer usage for ERC20

Severity: High

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-TARA-1

Target: bridge/src/connectors/ERC20LockingConnectorLogic.sol

### Description

The applyState function in the ERC20LockingConnectorLogic contract uses the transfer function instead of the safeTransfer function provided by the SafeERC20 library. This can cause tokens whose transfer function does not conform to the ERC20 specification to behave incorrectly. In particular, it could result in no tokens being transferred to a recipient while the contract behaves as though the tokens did get transferred and does not revert, leading to loss of funds.

```
22     function applyState(bytes calldata _state) public virtual override onlyBridge
{
23         Transfer[] memory transfers = decodeTransfers(_state);
24         uint256 transfersLength = transfers.length;
25         for (uint256 i = 0; i < transfersLength; i) {
26             IERC20(token).transfer(transfers[i].account, transfers[i].amount);
27             unchecked {
28                 ++i;
29             }
30         }
31     }
```

Figure 1.1: The use of the ERC20 transfer function in *ERC20LockingConnectorLogic.sol#L22-L31*

### Exploit Scenario

Alice, a user of the Taraxa bridge, wants to transfer her USDT to the Taraxa chain. However, the transfer unexpectedly reverts. Because the error is uncaught, Alice loses her USDT tokens.

### Recommendations

Short term, use the safeTransfer function of the SafeERC20 library.

Long term, keep up to date with the usage of third-party libraries and ensure that they are used appropriately throughout the codebase.

## 2. The add function can revert

Severity: Informational

Difficulty: High

Type: Denial of Service

Finding ID: TOB-TARA-2

Target: lib/Maths.sol

### Description

The add function in the Maths library fails to account for an arithmetic edge case involving the negation of a signed integer. In particular, the negation of `int256.min` will revert, as it does not have an appropriate two's complement representation.

```
5    function add(uint256 a, int256 b) internal pure returns (uint256) {
6        if (b < 0) {
7            return a - uint256(-b);
8        }
9        return a + uint256(b);
10    }
```

Figure 2.1: The add function in *Maths.sol*#L5-L10

### Recommendations

Short term, ensure that the edge case in the library is handled appropriately by using an unchecked block for a b value of `int256.min`.

Long term, improve unit testing to uncover edge cases and ensure intended behavior throughout the system.

### 3. G1 and G2 from method lack field point validation

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-TARA-3

Target: beacon-light-client/src/bls12381/{G1,G2}.sol

#### Description

Both the G1.sol and G2.sol files' from method lack validation that the BLS12Fp points used to create G1 and G2 subgroup elements are valid field elements. In particular, they do not check that each of the BLS12Fp field elements fit into the modulus. This could lead to multiple issues and undefined behavior downstream when using various elliptic curve point functionalities, including pairings.

```
93    /// @dev Derive Bls12G1 from uint256[4].
94    /// @param x uint256[4].
95    /// @return Bls12G1.
96    function from(uint256[4] memory x) internal pure returns (Bls12G1 memory) {
97        return Bls12G1(Bls12Fp(x[0], x[1]), Bls12Fp(x[2], x[3]));
98    }
```

Figure 3.1: The from method in G1.sol#L93-L98

```
120    /// @dev Derive Bls12G1 from uint256[8].
121    /// @param x uint256[4].
122    /// @return Bls12G2.
123    function from(uint256[8] memory x) internal pure returns (Bls12G2 memory) {
124        return Bls12G2(
125            Bls12Fp2(Bls12Fp(x[0], x[1]), Bls12Fp(x[2], x[3])),
126            Bls12Fp2(Bls12Fp(x[4], x[5]), Bls12Fp(x[6], x[7]))
127        );
128    }
```

Figure 3.2: The from method in G2.sol#L120-L127

#### Recommendations

Short term, call `is_valid` on each of the BLS12Fp points used to create G1 and G2 group elements.

Long term, improve unit testing to uncover edge cases and ensure intended behavior throughout the system.

#### 4. Missing validation allows signatures to be duplicated to finalize any PillarBlock

Severity: High

Difficulty: Low

Type: Data Validation

Finding ID: TOB-TARA-4

Target: bridge/src/eth/TaraClient.sol

#### Description

The `finalizeBlocks` function lacks proper validation of the `lastBlockSigs` argument and therefore allows any caller to arbitrarily inflate the number of signatures by duplicating the same signature in `lastBlockSigs`. This allows a `PillarBlock` that did not gain the required amount of validator votes to pass and be accepted as valid.

Figure 4.1 shows that the `lastBlockSigs` argument is not validated and passed into the `getSignaturesWeight` function.

```
81     function finalizeBlocks(PillarBlock.WithChanges[] memory blocks,  
CompactSignature[] memory lastBlockSigs) public {  
82         uint256 blocksLength = blocks.length;  
83         uint256 weightThreshold = totalWeight / 2 + 1;  
84         for (uint256 i = 0; i < blocksLength;) {  
...  
102             // skip verification for the first(genesis) block. And verify  
signatures only for the last block in the batch  
103             if (finalized.block.period != 0 && i == (blocks.length - 1)) {  
104                 uint256 weight =  
105                 getSignaturesWeight(PillarBlock.getVoteHash(blocks[i].block.period, pbh),  
lastBlockSigs);  
106                 if (weight < weightThreshold) {  
107                     revert ThresholdNotMet({threshold: weightThreshold, weight:  
weight});  
108                 }  
109             }  
...  
117         }  
118     }
```

Figure 4.1: The `finalizeBlocks` function in `TaraClient.sol`#L81-L118

Figure 4.2 shows the `getSignaturesWeight` function. This function does not in any way prevent duplicate signatures in the list to cause a revert. As long as the signature signed the `PillarBlock` hash (`pbh` variable), it is deemed valid, and the accompanying signer's

(=validator) vote count is added to the weight variable. This weight variable is returned to `finalizeBlocks` after all signatures have been processed. The `finalizeBlocks` function will then continue with validating that this weight is at least `weightThreshold` (highlighted in red in figure 4.1). By inflating the amount of signatures due to duplicate signatures, a malicious user can circumvent this check for `PillarBlocks` that lack the required amount of votes.

```
126     function getSignaturesWeight(bytes32 h, CompactSignature[] memory
signatures)
127         public
128         view
129         returns (uint256 weight)
130     {
131         uint256 signaturesLength = signatures.length;
132         for (uint256 i = 0; i < signaturesLength; i++) {
133             address signer = ECDSA.recover(h, signatures[i].r,
signatures[i].vs);
134             weight += validatorVoteCounts[signer];
135         }
136     }
```

Figure 4.2: The `getSignaturesWeight` function in `TaraClient.sol`#L126-L136

## Exploit Scenario

Eve calls the `finalizeBlocks` function with a `PillarBlock` whose last block has some votes, but not enough votes. Eve duplicates one of the signatures so many times as to pass the `weightThreshold`. The call succeeds, and a `PillarBlock` without enough votes was deemed valid and finalized.

## Recommendations

Short term, prevent duplicate signatures from being accepted inside the `getSignaturesWeight` function, and instead trigger a revert in case of duplicate signatures.

Long term, always validate inputs as much as possible. Also think of and handle edge cases such as duplicating values, passing zero values, and other ways of invalid input.

## 5. Incorrect mapping key used in validation inside registerContract

Severity: Informational

Difficulty: Low

Type: Undefined Behavior

Finding ID: TOB-TARA-5

Target: bridge/src/lib/BridgeBase.sol

### Description

A check inside the `registerContract` function uses the wrong value for the key into the `connectors` mapping, which will likely result in this check always passing. In terms of the check itself, this does not currently pose a problem due to the other validations, one of which uses the right value for this same mapping to perform the same check a couple lines further down.

The `connectors` mapping is used to look up which connector should be used for a specific source contract. As such, the mapping's key is a source contract address, and the value is a connector address. At the end of figure 5.1 a line is highlighted in orange, which shows the correct setting of a value in this mapping where the key is the `srcContract` address and the value is a connector address.

Figure 5.1 shows that the wrong value is used for the key in the `connectors` mapping (highlighted in yellow). However, a couple of lines down, the same check is performed with the right value for the key (highlighted in blue), so this important validation is performed.

```
142     function registerContract(IBridgeConnector connector) public payable {
143         if (msg.value < registrationFee) {
144             revert InsufficientFunds(registrationFee, msg.value);
145         }
146
147         address srcContract = connector.getSourceContract();
148         address dstContract = connector.getDestinationContract();
149
150         if (connectors[address(connector)] != IBridgeConnector(address(0))) {
151             return;
152         }
153         if (srcContract == address(0)) {
154             revert ZeroAddressCannotBeRegistered();
155         }
156         if (localAddress[dstContract] != address(0) ||
157         address(connectors[srcContract]) != address(0)) {
157             revert ConnectorAlreadyRegistered({connector: address(connector),
158             token: srcContract});
159         }
```

```

158     }
159
160     address owner = OwnableUpgradeable(address(connector)).owner();
161     if (owner != address(this)) {
162         revert IncorrectOwner(owner, address(this));
163     }
164
165     connectors[srcContract] = connector;
166     localAddress[dstContract] = srcContract;
167     tokenAddresses.push(srcContract);
168     emit ConnectorRegistered(address(connector), srcContract, dstContract);
169 }

```

*Figure 5.1: The registerContract function in TaraClient.sol#L81-L118*

## Recommendations

Short term, remove lines 150, 151, and 152, as this same check is already performed on line 156.

Long term, add more inline documentation to the implementation. In particular, add comments that explain what if-statements are checking. This would have likely uncovered this incorrect mapping usage.



## 6. Reentrancy in applyState can lead to breaking the contract and stealing hook-enabled tokens

Severity: High

Difficulty: Medium

Type: Undefined Behavior

Finding ID: TOB-TARA-6

Target: bridge/src/lib/BridgeBase.sol

### Description

The lack of reentrancy guards on the `applyState` function allows the stealing of hook-supporting tokens by calling `applyState` recursively. Since no popular tokens support this standard or hooks in general, this limits the impact of this issue. However, because the epoch incorrectly increments whenever `applyState` is called through reentrancy, this will likely prevent the broken bridge contract's `applyState` function from accepting valid bridge messages, since the epoch does not match what is expected.

The `applyState` function can be called by anyone and will apply bridged messages (either from Ethereum to Taraxa, or vice versa). Inside this function, the epoch in the input arguments is validated to be one above the current epoch (called `appliedEpoch`, as highlighted in yellow in figure 6.1). At the very end of the function, the `appliedEpoch` is incremented (as highlighted in blue in figure 6.1). In the middle of the function, a loop over all of the bridged messages is performed; this will call the `applyState` function of the configured connector (as highlighted in green in figure 6.1). These will then transfer the tokens to the recipient (this could involve transferring an ERC20 token amount, minting an ERC20 token amount, or transferring an amount of the chain's native token).

In the case of an ERC20 transfer in which the token to be transferred is an ERC777 (or an otherwise hook-supporting token) contract, the recipient can use reentrancy to call back into the `BridgeBase.applyState` function with the same arguments. This works since `appliedEpoch` is incremented only at the end of the function.

```
175     function applyState(SharedStructs.StateWithProof calldata state_with_proof)
public {
176         uint256 gasleftbefore = gasleft();
177         // get bridge root from light client and compare it (it should be proved
there)
178         if (
179             SharedStructs.getBridgeRoot(state_with_proof.state.epoch,
state_with_proof.state_hashes)
180             !=
lightClient.getFinalizedBridgeRoot(state_with_proof.state.epoch)
```

```

181     ) {
182         revert StateNotMatchingBridgeRoot({
183             stateRoot:
SharedStructs.getBridgeRoot(state_with_proof.state.epoch,
state_with_proof.state_hashes),
184             bridgeRoot:
lightClient.getFinalizedBridgeRoot(state_with_proof.state.epoch)
185         });
186     }
187     if (state_with_proof.state.epoch != appliedEpoch + 1) {
188         revert NotSuccessiveEpochs({epoch: appliedEpoch, nextEpoch:
state_with_proof.state.epoch});
189     }
190     uint256 statesLength = state_with_proof.state.states.length;
191     uint256 idx = 0;
192     while (idx < statesLength) {
193         SharedStructs.ContractStateHash calldata proofStateHash =
state_with_proof.state_hashes[idx];
194         SharedStructs.StateWithAddress calldata state =
state_with_proof.state.states[idx];
195         if (localAddress[proofStateHash.contractAddress] == address(0)) {
196             unchecked {
197                 ++idx;
198             }
199             continue;
200         }
201         bytes32 stateHash = keccak256(state.state);
202         if (stateHash != proofStateHash.stateHash) {
203             unchecked {
204                 ++idx;
205             }
206             revert InvalidStateHash(stateHash, proofStateHash.stateHash);
207         }
208         if
(isContract(address(connectors[localAddress[proofStateHash.contractAddress]]))) {
209             try
connectors[localAddress[proofStateHash.contractAddress]].applyState(state.state) {}
catch {}
210         }
211         unchecked {
212             ++idx;
213         }
214     }
215     uint256 used = (gasleftbefore - gasleft()) * tx.gasprice;
216     uint256 payout = used * feeMultiplier / 100;
217     if (address(this).balance >= payout) {
218         (bool success,) = payable(msg.sender).call{value: payout}("");
219         if (!success) {
220             revert TransferFailed(msg.sender, payout);
221         }
222     }
223     ++appliedEpoch;
224 }

```

Figure 6.1: The `applyState` function in `BridgeBase.sol` #L175-L224

### Exploit Scenario

Eve holds 1,000 of an ERC777 token called TokenX in a contract she has deployed (called AttackerContract). Eve bridges these 1,000 TokenX from Ethereum to Taraxa. Eve's 1,000 tokens are now locked inside the configured ERC20LockingConnector on Ethereum. After bridging to Taraxa, she immediately initiates a bridge of 1,000 TokenX from Taraxa back to Ethereum.

She now monitors the Ethereum chain to spot the first time anyone tries to call the `EthBridge.applyState` function to bridge messages, which includes her bridging message. Once she spots it, she front-runs the transaction and calls `EthBridge.applyState` with the same arguments.

The message handling loop will now call the `TokenX.transfer` function to transfer the 1,000 tokens to AttackerContract, after which the `ERC777_callTokensToReceived` hook will call `AttackerContract`. Inside the called `AttackerContract` function, a call is made back to `EthBridge.applyState` with the same arguments as the original call. This will lead to a loop where each time `EthBridge.applyState` is called, the AttackerContract will receive 1,000 TokenX tokens.

The only limitations of this attack are gas, the amount of TokenX inside the ERC20LockingConnector, and the failure of other bridging messages due to lack of tokens or multiple executions.

### Recommendations

Short term, add reentrancy guards to the `BridgeBase.applyState` function. This will prevent the aforementioned issue.

Long term:

- Use `Slither` to detect this issue, and integrate it into the CI using `slither-action`.
- Review all non-view functions and consider adding reentrancy guards to each of these functions.
- Update the implementation to follow the `Checks-Effects-Interactions` pattern. This pattern is considered a best practice and structures the code in a manner that helps prevent reentrancy vulnerabilities.

## 7. Confusing application of settlementFee to locking native assets

Severity: Informational

Difficulty: Low

Type: Arithmetic

Finding ID: TOB-TARA-7

Target: bridge/src/lib/BridgeBase.sol

### Description

The process of deducting the fee from the passed in `msg.value` is confusing and makes it difficult for callers to transfer a specific amount of native assets.

Figure 7.1 shows the `lock` function, which deducts the `settlementFee` from the amount to be bridged. A more user-friendly way of doing this would be to add an argument to the `lock` function called `amount` that indicates the exact amount to bridge. In the body of the function, a check is then performed that ensures that the difference between `msg.value` and `amount` is exactly the `settlementFee`.

```
37  function lock() public payable {
38      uint256 fee = bridge.settlementFee();
39      uint256 lockingValue = msg.value;
40
41      // Charge the fee only if the user has no balance in current state
42      if (!state.hasBalance(msg.sender)) {
43          if (msg.value < fee) {
44              revert InsufficientFunds(fee, lockingValue);
45          }
46          lockingValue -= fee;
47      }
48
49      if (lockingValue == 0) {
50          revert ZeroValueCall();
51      }
52      state.addAmount(msg.sender, lockingValue);
53      emit Locked(msg.sender, lockingValue);
54  }
```

Figure 7.1: The `lock` function in `NativeConnectorLogic.sol#L37-L54`

### Recommendations

Short term, replace the existing `deduct-fee-from-msg.value` with an `amount` argument in the `lock` function and a check: `require(msg.value - amount == settlementFee)`.

Long term, design functions so that they are simple and easy to use for external parties. This improves the usability and prevents confusion for integrators.

## A. Vulnerability Categories

---

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

## B. Code Quality Recommendations

The following recommendations are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Remove the integer increment before the revert call.** There is no added benefit to doing an integer increment if the operation after it is always a revert. Instead, save some gas by skipping the increment and simply executing the revert.

```
202     if (stateHash != proofStateHash.stateHash) {
203         unchecked {
204             ++idx;
205         }
206         revert InvalidStateHash(stateHash, proofStateHash.stateHash);
207     }
```

Figure B.1: Excerpt from the `applyState` function in `BridgeBase.sol#L202–L207`

- **Use `address.code.length` instead of the custom `isContract` function.** Since Solidity 0.8.0, the `address.code` member is available. To get the size of the code at an address, use `address.code.length`. Since Solidity 0.8.1, the `address.code.length` operation uses `extcodesize` under the hood instead of copying all the code before checking the length, which is very gas inefficient. We therefore recommend directly using `address.code.length` and removing the `isContract` function.

```
130     function isContract(address addr) internal view returns (bool) {
131         uint256 size;
132         assembly {
133             size := extcodesize(addr)
134         }
135         return size > 0;
136     }
```

Figure B.2: The `isContract` function in `BridgeBase.sol#L130–L136`

- **Normalize the `EthBridge` and `TaraBridge` files.** Both of these files contain `initialize` functions and they both serve the same purpose, although on a different chain. We recommend normalizing the initialization function setup (i.e., also using an `unchained` function in the `TaraBridge` contract).

```
8     contract EthBridge is BridgeBase {
9         function initialize(
10             IBridgeLightClient _lightClient,
11             uint256 _finalizationInterval,
12             uint256 _feeMultiplier,
```



```

13         uint256 _registrationFee,
14         uint256 _settlementFee
15     ) public initializer {
16         __initialize_EthBridge_unchained(
17             _lightClient, _finalizationInterval, _feeMultiplier,
18             _registrationFee, _settlementFee
19         );
20     }
21     function __initialize_EthBridge_unchained(
22         IBridgeLightClient _lightClient,
23         uint256 _finalizationInterval,
24         uint256 _feeMultiplier,
25         uint256 _registrationFee,
26         uint256 _settlementFee
27     ) internal onlyInitializing {
28         __BridgeBase_init(_lightClient, _finalizationInterval,
29             _feeMultiplier, _registrationFee, _settlementFee);
30     }

```

Figure B.3: The EthBridge contract in *EthBridge.sol*#L8-L30

```

7     contract TaraBridge is BridgeBase {
8         function initialize(
9             IBridgeLightClient _lightClient,
10            uint256 _finalizationInterval,
11            uint256 _feeMultiplier,
12            uint256 _registrationFee,
13            uint256 _settlementFee
14        ) public initializer {
15            __BridgeBase_init(_lightClient, _finalizationInterval,
16                _feeMultiplier, _registrationFee, _settlementFee);
17        }

```

Figure B.4: The TaraBridge contract in *TaraBridge.sol*#L7-L17

- **Fix incorrect comment in BridgeBase.** The connectors mapping is a mapping from *source* (or *destination*, depending on how you look at it when bridging) addresses to connector addresses.

```

25     /// Mapping of connectors to their source and destination addresses
26     mapping(address => IBridgeConnector) public connectors;

```

Figure B.5: The connectors variable in *BridgeBase.sol*#L25-L26

- **Remove the variable and directly return false.** This code could be simplified by removing the `shouldFinalize` variable and directly returning `false` after the loop. Alternatively, since the default value of a Boolean is `false`, this could be done by altogether skipping the `return false`, which would implicitly return `false`.

```

231     function shouldFinalizeEpoch() public view returns (bool) {
232         bool shouldFinalize = false;
233         for (uint256 i = 0; i < tokenAddresses.length; i++) {
234             if (!connectors[tokenAddresses[i]].isStateEmpty()) {
235                 return true;
236             }
237         }
238         return shouldFinalize;
239     }

```

Figure B.6: The `shouldFinalizeEpoch` function in *BridgeBase.sol*#L231-L239

- **Rename the `registerContract` function to `registerConnector`.** This name more clearly describes what the function does.

```

142     function registerContract(IBridgeConnector connector) public payable {

```

Figure B.7: The signature of the `registerContract` function in *BridgeBase.sol*#L142

## C. Fix Review Results

---

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On August 9, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Taraxa team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Taraxa has resolved all seven issues.

ID	Title	Status
1	Lack of safeTransfer usage for ERC20	Resolved
2	The add() function can revert	Resolved
3	G1 and G2 from() method lack field point validation	Resolved
4	Missing validation allows signatures to be duplicated to finalize any PillarBlock	Resolved
5	Incorrect mapping key used in validation inside registerContract	Resolved
6	Reentrancy in applyState can lead to breaking the contract and stealing hook-enabled tokens	Resolved
7	Confusing application of settlementFee to locking native assets	Resolved

## Detailed Fix Review Results

### **TOB-TARA-1: lack of safeTransfer usage for ERC20**

Resolved in commit [8897359](#). The transfer call has been replaced with a safeTransfer call.

### **TOB-TARA-2: The add() function can revert**

Resolved in commit [c905118](#). A special case was added to handle `type(int256).min`. Additionally, tests were added for the Maths.sol library.

### **TOB-TARA-3: G1 and G2 from() method lacks field point validation**

Resolved in commit [034d004](#). The implementation was updated to call `is_valid` on each of the BLS12Fp points.

### **TOB-TARA-4: Missing validation allows signatures to be duplicated to finalize any PillarBlock**

Resolved in commit [dd50260](#). The implementation has been updated to enforce passing in the signatures (`CompactSignature[ ]`) sorted descending based on the `CompactSignature.r` value. This prevents duplicate signatures from being allowed.

### **TOB-TARA-5: Incorrect mapping key used in validation inside registerContract**

Resolved in commit [43292b0](#). The incorrect check has been removed from the implementation. No new check needed to be added since the correct check already was present further down inside the same function.

### **TOB-TARA-6: Reentrancy in applystate can lead to breaking the contract and stealing hook-enabled tokens**

Resolved in commit [df9e7f3](#). The `appliedEpoch` increment has been moved before the external call. Therefore, the described reentrancy issue is no longer possible.

### **TOB-TARA-7: Confusing application of settlementFee to locking native assets**

Resolved in commit [cc6b08f](#). An `amount` argument was added to the function to allow the caller to exactly specify which amount he wants to bridge.