

Chainlink - Custom Senders and Receivers

Security Assessment

February 25, 2025

Prepared for:
Lee Poor
Chainlink Labs

Prepared by: **Benjamin Samuels and Bo Henderson**

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.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
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.

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

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources 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

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Project Summary **4**
Executive Summary **5**
Project Goals **8**
Project Targets **9**
Project Coverage **10**
Automated Testing **12**
Codebase Maturity Evaluation **15**
Summary of Findings **18**
Detailed Findings **19**
1. CustomSender only supports sending Wrapped Ether 19
2. Lack of zero-value checks 22
3. OraclePool has insolvency risks when LSTs are slashed 24
4. Swapping at Chainlink prices will lead to CSR insolvency due to oracle timing discrepancy 26
5. Price oracle fails to verify sequencer liveness 28
6. Eigenpie deposit flow may be vulnerable to sandwiching attacks 30
7. The CustomSender contract does not enforce minimum gas amounts when bridging 32
A. Vulnerability Categories **34**
B. Code Maturity Categories **36**
C. Code Quality Recommendations **38**
D. Automated Analysis Tool Configuration **40**
E. Mutation Testing **41**
F. Incident Response Recommendations **43**
G. Fix Review Results **45**
Detailed Fix Review Results 46
H. Fix Review Status Categories **47**
I. Subsequent PR Review **48**
Detailed Subsequent PR Review Results 48

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Project Timeline

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

Date	Event
August 7, 2024	Pre-project kickoff call
August 16, 2024	Status update meeting #1
August 26, 2024	Delivery of report draft
August 26, 2024	Report readout meeting
February 25, 2025	Subsequent PR review

Executive Summary

Engagement Overview

Chainlink Labs engaged Trail of Bits to review the security of CSR (Custom Senders & Receivers) smart contracts. These contracts are used to allow users on optimistic rollups to stake tokens on L1-based protocols using a dedicated liquidity pool. Multiple stakeable tokens and protocols are supported, including Lido, Eigenpie, and Frax.

A team of two consultants conducted the review from August 12 to August 23, for a total of four engineer-weeks of effort. Our testing efforts focused on pricing oracle implementation and cross-protocol interactions with CCIP and each supported staking protocol. With full access to source code and documentation, we performed static and dynamic testing of the CSR smart contracts, using automated and manual processes.

Observations and Impact

The CSR contracts are well-organized and relatively simple. However, a large surface area for failure exists in its interactions with a wide variety of external services. For example, the `OraclePool` contract risks insolvency if upstream LSTs are slashed (TOB-CHAINLINK-CSR-3) or due to a discrepancy between the price oracle heartbeat and Lido rebase timings (TOB-CHAINLINK-CSR-4). In addition, the system may be vulnerable to sandwich attack risks due to a lack of slippage protection while interacting with the Eigenpie protocol (TOB-CHAINLINK-CSR-6).

Although the code features thorough in-line comments, higher-level documentation, such as the runbook, was missing. As a result, we identified one potential deviation between the expected functionality and implementation capabilities that materially limit the types of staking protocols and underlying stakeable token that can be used (TOB-CHAINLINK-CSR-1).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Chainlink 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.
- **Expand documentation.** Although the code is well commented, high-level documentation is necessary to ensure end users can safely deploy and interact with CSR components safely.
    - **Divide the technical specifications into discrete milestones.** The provided specification provides an excellent overview of required features.

However, because these contracts are upgradable and additional features are planned for the future, this document ought to clarify which features are required at each stage of the project's life cycle. In earlier stages—such as when only WETH is supported by the `CustomSender` contract—documentation should describe not only the requirements and use-cases for this stage, but also provide an explicit list of features that are not included, as well as additions that are not strictly necessary but will aid the system's transition to its next phase. This will clarify the responsibilities of each upgrade and help prevent mistakes associated with half-complete features.

  - **Create a runbook for protocol operators.** Entities outside of Chainlink will run the CSR contracts, and they may not be familiar with how the contracts work. A runbook should be established that provides operators with guidance including, but not limited to:
    - How to upgrade the contracts
    - How the operator may detect a potential security incident
    - How to freeze the contracts during a security incident
    - Any routine maintenance that needs to be performed (refilling liquidity, updating the Chainlink keeper, etc.)
  - **Create an incident response plan**. See Appendix F for guidance.

- **Expand the test suite.** Although `forge coverage` reports 100% test coverage, some gaps in the smart contract test coverage were identified by mutation tests. Write additional tests that assert the expected behavior of every branch of the business logic, and then use code coverage assessments to confirm that gaps in test coverage have been closed. See Appendix E for details.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	1
Medium	2
Low	0
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	Count
Data Validation	5
Timing	1
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the CSR contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is there a way to obtain staked tokens at a large discount compared to their real value?
- Do the contract's receivers call into each staking protocol correctly?
- Is there a way to "drain" liquidity from the protocol so that the system becomes insolvent?
- Are there any incorrect or error-prone steps in the deployment and initialization of the system's smart contracts?
- Can any movement of funds be diverted to an attacker or frozen in an invalid destination?
- Are any access controls missing, or do any roles have unsuitable permissions?
- Can a message failure be retried or recovered from in a way that is robust to out-of-gas errors and replay attacks?
- Is the CSR system robust against failures in third-party integrations, such as L2 sequencer outages, stale oracle prices, or slashed stakes?

Project Targets

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

Chainlink Custom Sender-Receiver

Repository	github.com/Aphyla/chainlink-csr
Version	`8e08c07ec55bbe32f447b450d915f6813eac6cc7`
Type	Solidity

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Upgradeable contracts**. Most of the contracts in the system are upgradeable. We reviewed these contracts to assess whether there is a possibility of storage collisions, and checked for cases where a logic contract constructor sets a storage variable in the logic contract instead of the proxy contract.

- **Receiver contracts**. These contracts are deployed to the L1 and used to deposit stakeable tokens into the relevant staking protocol. These contracts were reviewed to determine if they are safely interacting with each staking protocol using their intended entry points, along with correct use of approvals.

- **Bridge adapter contracts**. These contracts are used to bridge funds between the L1 and L2, and vice versa. Various bridges are used for different types of tokens. Each bridge adapter contract was reviewed to assess if it uses each bridge correctly and pays the required amount of fees. Some of these contracts need to handle a callback from the bridge (e.g., `ccipReceive`). These contracts were reviewed to determine if they correctly check the sender and original origin of the message.

- **Pooling and oracle contracts**. These contracts are used on the L2 to facilitate `fastStake` calls. These contracts were reviewed to determine if they are querying Chainlink oracles correctly (checking the age of the data, etc.) and to assess whether the sequencer uptime oracle is used. The configuration for each oracle, and the oracle configurations themselves, were reviewed to determine how large a pricing discrepancy may occur at any one time.

- **Sender & sync contracts**. These contracts were reviewed to assess adequate support for each stakeable token defined in the system specifications and correct integration with the Chainlink Keeper system.

- **Subsequent PR review**. After delivery of the initial report and the fix review, Trail of Bits performed a limited review of PR #3, which adds referral support to the `CustomSender` contract. This review was scoped to the code changes in that pull request only and is not a comprehensive re-assessment. See Appendix I for details.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- In TOB-CHAINLINK-CSR-6, we identify the possibility of sandwiching attacks against sync transactions when they are run on the L1. We were only able to partially verify this issue due to the size and complexity of Eigenpie's codebase.
- There may be sandwiching risks present in other restaking protocols; only the staking interface for each protocol was reviewed.
- Given the contract only supports WETH as a stakeable token, there may be security issues introduced by adding support for non-WETH stakeable tokens, as is defined in the specification document.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix D
`slither-mutate`	A deterministic mutation generator that detects gaps in test coverage	Appendix E

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- Common issues and anti-patterns in Solidity
- Gaps in test coverage

Test Results

The results of this focused testing are detailed below.

`slither-mutate`: The following table displays the portion of each type of mutant for which all unit tests passed. The presence of valid mutants indicates that there are gaps in test coverage because the test suite did not catch the introduced change.

- Uncaught revert mutants replace a given expression with a `revert` statement and indicate that the line is not executed during testing.
- Uncaught comment mutants comment out a given expression and indicate that the effects of this line are not checked by any assertions.
- Uncaught tweak mutants indicate that the expression being executed features edge cases that are not covered by the test suite.

The `chainlink-csr/contracts` subdirectory is the root for all target paths listed below. Contracts supporting tests or that produced zero analyzed mutants (e.g., interfaces) were omitted from mutation testing analysis.

Target	Uncaught Reverts	Uncaught Comments	Uncaught Tweaks
adapters/ArbitrumLegacyAdapterL1toL2.sol	0%	17%	5%
adapters/BaseAdapterL1toL2.sol	0%	14%	0%
adapters/BridgeAdapter.sol	0%	0%	17%
adapters/CCIPAdapter.sol	0%	25%	0%
adapters/FraxFerryAdapterL1toL2.sol	0%	17%	11%
adapters/OptimismLegacyAdapterL1toL2.sol	0%	14%	0%
automations/SyncAutomation.sol	0%	16%	3%
ccip/CCIPBaseUpgradeable.sol	0%	0%	0%
ccip/CCIPDefensiveReceiverUpgradeable.sol	0%	35%	17%
ccip/CCIPSenderUpgradeable.sol	0%	14%	0%
ccip/CCIPTrustedSenderUpgradeable.sol	0%	14%	0%
libraries/FeeCodec.sol	0%	0%	6%
libraries/TokenHelper.sol	0%	0%	14%
receivers/CustomReceiver.sol	0%	18%	9%
receivers/EigenpieCustomReceiver.sol	0%	12.5%	12.5%
receivers/FraxCustomReceiver.sol	0%	14%	0%
receivers/LidoCustomReceiver.sol	0%	43%	0%

senders/CustomSender.sol	0%	12%	5%
utils/OraclePool.sol	0%	23%	3%
utils/PriceConverterOracle.sol	0%	25%	0%
utils/PriceOracle.sol	0%	14%	0%

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The system's arithmetic is primarily used to standardize prices returned by Chainlink oracles. None of this arithmetic lacks checks for overflows. Division is used sparingly and only in calculations that are not sensitive to rounding errors. Although the arithmetic involved is relatively straightforward, calculations are not generally specified with in-line comments nor other documentation.	**Satisfactory**
Auditing	The system emits sufficient events during state-changing or other critical operations to support off-chain monitoring. Events lack in-line comments describing their implications and the emission of events, and their parameters are insufficiently validated by the test suite. No incident response plan or documentation of off-chain monitoring strategies was provided.	**Moderate**
Authentication / Access Controls	Access controls are differentiated in accordance with the principle of least privilege and we did not identify any missing access controls. Authorization checks are well tested, however we identified a few instances where test coverage does not cover important access controls. A summary of roles and their privileges is inadequately documented.	**Moderate**
Complexity Management	Each smart contract has a well-defined responsibility and their methods and state variables follow clear and consistent naming conventions. No functions are excessively long or have unnecessarily high cyclomatic complexity. We did not identify any instances of duplicated code.	**Satisfactory**

Cryptography and Key Management	The only cryptographic primitives this project uses are hash functions, and key management was out of scope for this engagement.	**Not Applicable**
Decentralization	Most contracts are designed to be implementations behind an owned proxy or are directly owned. The governance process surrounding ownership was not in scope for this review.	**Not Considered**
Documentation	Detailed technical specifications were provided, and all functions have associated in-line comments describing the function at a high level along with execution requirements and expected event emissions. However, in-line method documentation often lacks sufficient descriptions of all function parameters. Because these contracts will be operated by third parties, it is essential to provide a runbook with clear, detailed documentation describing how to operate and interact with them safely. For more information, see the long-term strategic recommendations.	**Moderate**
Low-Level Manipulation	Assembly is only used for manual storage access in accordance with the EIP-7201 standard. The `TokenHelper` contract uses a low-level call to transfer native tokens using all available gas. Low-level manipulation in this system is simple, and we did not find any associated issues.	**Satisfactory**
Testing and Verification	Integration tests use multiple forked testnets to verify cross-chain interactions. Unit tests are thorough and `forge` reports 100% coverage. However, our mutation analysis revealed gaps in test coverage, in particular regarding verification that events are emitted and contain expected data. See Appendix E for details.	**Satisfactory**
Transaction Ordering	Some issues related to transaction timing were identified, including a discrepancy between the update times of Lido's rebase oracle and Chainlink's price oracles (TOB-CHAINLINK-CSR-4). Usage of off-chain data while swapping eliminates many common sources of front-running risks. However, some frontrunning risks may be present in the case of	**Moderate**

	sequencer outages (TOB-CHAINLINK-CSR-5)	

Summary of Findings

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

ID	Title	Type	Severity
1	CustomSender only supports sending Wrapped Ether	Undefined Behavior	Informational
2	Lack of zero-value checks	Data Validation	Informational
3	OraclePool has insolvency risks when LSTs are slashed	Data Validation	High
4	Swapping at Chainlink prices will lead to CSR insolvency due to oracle timing discrepancy	Timing	Medium
5	Price oracle fails to verify sequencer liveness	Data Validation	Medium
6	Eigenpie deposit flow may be vulnerable to sandwiching attacks	Data Validation	Undetermined
7	The CustomSender contract does not enforce minimum gas amounts when bridging	Data Validation	Informational

Detailed Findings

1. CustomSender only supports sending Wrapped Ether	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHAINLINK-CSR-1
Target: `contracts/senders/CustomSender.sol`	

Description

The `CustomSender` contract does not implement or support the sending of tokens other than `WETH`, despite the specification and sample use flows indicating it should support other types of tokens.

There are several barriers to using the `CustomSender` contract for sending tokens other than wrapped ether.

The first barrier is that the `CustomSender` contract conflates the fee token type with the token type being transferred in `_ccipBuildAndSend`, as shown in figure 1.1.

```
function _ccipBuildAndSend(
    [...]
) internal virtual returns (bytes32 messageId) {
    (uint256 maxFeeOtoD, bool payInLinkOtoD, uint256 gasLimitOtoD) =
FeeCodec.decodeCCIP(feeOtoD);
    uint256 feeAmountDtoO = FeeCodec.decodeFee(feeDtoO);

    _wrapNative(feeAmountDtoO);

    bytes memory packedData = FeeCodec.encodePackedData(recipient, amount, feeDtoO);

    messageId = _ccipSend(
        destChainSelector, token, amount + feeAmountDtoO, payInLinkOtoD, maxFeeOtoD,
gasLimitOtoD, packedData
    );
}
```

Figure 1.1: Fee token amounts are being directly added with transferred token amounts. (chainlink-csr/contracts/senders/CustomSender.sol#257–275)

In `_ccipBuildAndSend`, the code wraps `CustomSender`'s native balance into a wrapped token and then adds the fee amount to the token amount being transferred over CCIP. If the token being transferred over CCIP is not the wrapped native token, `_ccipSend` will try

to send an incorrect token amount—likely insufficient to pay for the destination-to-origin `ccipSend` call.

The second barrier is that if the extra amount sent to the L1 is intended to be swapped for ETH/LINK to pay for the destination-to-origin fee, that swap needs to have a price limit set to ensure it is executed at a fair price. The current implementation of `CustomSender` does not have support for passing this information to `CustomReceiver`.

It should be noted that this only impacts PTT flows where the token being transferred to the L1 is not WETH, *and* the method of bridging back to the L2 is CCIP. Using the L2's native token bridge does not typically require any special fees, eliminating the need for such fees in the system's logic.

Exploit Scenario

A `CustomSender` is created for re-staking LST tokens from the L2 to EigenLayer LRT tokens. The EigenLayer LRT tokens are bridged back to the L2 using CCIP. A user tries to use the `CustomSender` to re-stake 10000 wei of wstETH, for which there will be a 100 wei of ETH destination-to-origin fee.

If the user gives the `CustomSender` contract an infinite approval for wstETH, it will then send 10000+100 wei of wstETH to mainnet through `ccipSend`. On the L1, the `CustomReceiver` contract will have to swap 100 wei of wstETH to ETH, instead of the correct amount of wstETH required to swap for ETH. This may negatively impact the amount of tokens the user ultimately receives, or prevent the destination-to-origin bridge transaction from succeeding at all.

Recommendations

Short term, two sets of options can be pursued:

1. Do not use CCIP for destination-to-origin transactions where the token being staked is not ETH/WETH. If the token being staked is not ETH/WETH, the L2's native bridge should be used for the destination-to-origin transaction. L2 native bridges do not impose any fees, so the `CustomReceiver` will not have to worry about swapping tokens to ETH to pay for fees.
2. Add the ability for `CustomSender` to send both the token to be staked, and the WETH required to pay for the destination-to-origin `ccipSend`. This approach is more complicated because the amount of WETH sent to mainnet may not exactly match the fees required to pay for the destination-to-origin `ccipSend`. If there is not enough WETH, the transaction will revert and become stuck. If there is too much WETH, then a mechanism should be added to refund that WETH to the user on the L2, costing even more gas.

Long term, it would make sense for CCIP to implement some kind of round-trip transaction mechanism to simplify this type of use case and allow an optimized way to perform this

flow. This would allow CCIP to earn fees on the destination-to-origin transaction and simplify user implementations so they do not have to interact with multiple types of bridges.

2. Lack of zero-value checks	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHAINLINK-CSR-2
Target: Various	

Description

Several functions fail to validate incoming arguments, so callers of these functions may mistakenly provide value-transferring methods a zero-address destination, thereby causing the loss of funds.

```
function recoverTokens(Client.Any2EVMMessage calldata message, address to) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    _verifyAndMarkFailedMessage(message);

    uint256 length = message.destTokenAmounts.length;
    for (uint256 i; i < length; ++i) {
        Client.EVMTokenAmount calldata tokenAmount = message.destTokenAmounts[i];
        IERC20(tokenAmount.token).safeTransfer(to, tokenAmount.amount);
    }

    emit TokensRecovered(message.messageId);
}
```

Figure 2.1: The `recoverTokens` method of `contracts/ccip/CCIPDefensiveReceiverUpgradeable.sol#L147–L156`

Other instances of this issue include immutable addresses assigned during contract construction. The following state variables are addresses that receive funds and which are not checked to be non-zero during assignment:

- `ArbitrumLegacyAdapterL1toL2.L1_GATEWAY_ROUTER`
- `EigenpieCustomReceiver.EGETH_STAKING`
- `FraxCustomReceiver.FRXETH_MINTER`
- `CustomSender.WNATIVE`

Exploit Scenario

Bob attempts to send a cross-chain message but the transaction fails, leaving funds stranded in the `CCIPDefensiveReceiverUpgradeable` contract. Alice, the administrator of this contract, calls the `recoverTokens` method to recover these trapped funds.

However, she accidentally provides a zero address as the `to` argument. As a result, these funds will be lost forever.

Recommendations

Short term, add zero-value checks to all function arguments that are addresses that receive value transfers. If providing a zero-address is impossible (e.g., if an internal method is only provided `msg.sender` as an argument), clearly document this fact to alert future developers and auditors so that dangerous changes to the code may be detected and remedied in a timely manner.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts. Slither-action is a popular tool for integrating slither into GitHub CI actions.

3. OraclePool has insolvency risks when LSTs are slashed	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHAINLINK-CSR-3
Target: `OraclePool`	

Description

The `OraclePool` contract executes `fastStake` swaps using Chainlink oracles to set the price, allowing the system to become insolvent when an LST is slashed and its price per token goes down.

CSR is intended to be a closed-loop system. The system is provided with an initial amount of stakeable tokens, which are sync'ed to provide LST liquidity on the L2. Users then use `fastStake` to pay stakeable tokens in exchange for LST. When liquidity is low, the system is sync'ed again, ad infinitum.

When sync is called, the CSR creates a virtual "debt": the stakeable tokens are going to be converted to LSTs on the L1, but the CSR contract still "owes" the stakeable tokens to the system operator. As users call `fastStake`, the CSR is paid stakeable tokens in exchange for LSTs, thus obtaining the ability to repay its debt to the operator.

Under almost all circumstances, the price of a LST token in stakeable token terms only goes up. However, when the LST provider is slashed, its price can decrease. Once the Chainlink oracle updates with the new, lower price, the CSR system will begin selling LST tokens at a discounted price compared to the price they were acquired for.

Exploit Scenario

A CSR system is deployed for wstETH, with initial sync of 1 ETH of liquidity from its operator. After the sync, the CSR contract owes its operator 1 ETH, and has 1 wstETH of liquidity available for staking.

Lido is then slashed for 5% of its total staked balance, and the Chainlink oracle for ETH:wstETH changes from 1:1 to 0.95:1. As users continue to call `fastStake`, the CSR contract will sell its 1 wstETH balance for 0.95 ETH, incurring a 0.05 ETH insolvency for the contract operator.

Recommendations

Short term, add a price guard to `OraclePool` to ensure it never sells tokens for less than the price they were acquired for. When sync is called, the `CustomSender` should observe the latest wstETH/ETH oracle price and set this value in the `OraclePool` contract as the

price the liquidity was purchased at. When a user calls `fastStake` and their tokens are swapped, `OraclePool` should compare the observed Chainlink oracle price with the purchase price and use the higher number to facilitate the swap.

Long term, documentation should be included with the CSR system on how operators should handle a potential slashing.

4. Swapping at Chainlink prices will lead to CSR insolvency due to oracle timing discrepancy

Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-CHAINLINK-CSR-4
Target: Chainlink wstETH-stETH Oracle, Accounting	

Description

`OraclePool` performs Lido swaps using the wstETH-stETH exchange rate oracle, which has an exceptionally long heartbeat of 24 hours and typically triggers at 10pm UTC every day. Lido performs rebases once a day at 12pm UTC, meaning the wstETH-stETH price oracle has an incorrect, lower-than-expected price for 14 hours each day.

In essence, this means that whenever the CSR contract is sync'ed, it will execute `fastStake` transactions at a price that will make the contract insolvent for up to 14 hours after the sync, then at a solvent price afterward until the next poorly timed sync.

Over a long timeframe, this will cause the CSR contract to become insolvent. However, if the price discount plus fees for wstETH on a DEX is less than the price discrepancy, a risk-free arbitrage opportunity is created that may cause the CSR contract to become insolvent much faster.

Exploit Scenario

The Arbitrum wstETH/ETH oracle returns a price that is 0.027% off of the actual price on mainnet, and a DEX is available on mainnet with a wstETH/ETH pool fee of 0.005%, with an exchange rate equivalent to 1 stETH = 0.99995998850055 ETH after accounting for swap fees.

A malicious user on Arbitrum starts with 1 ETH, and stakes it for wstETH using the CSR contract. They receive an amount of wstETH equal to 1.00022 stETH on mainnet due to the pricing discrepancy. The malicious user can then swap their 1.00027 mainnet-equivalent stETH equivalent for 1.0002299776974451485 ETH on mainnet, thus observing an arbitrage profit. Swapping on mainnet may incur additional fees due to bridging costs; however, this technique should also work exclusively on Arbitrum if there is a DEX with adequate liquidity/fees.

While these numbers may appear contrived, the 0.027% price skew is taken from the largest price skew observed in the past 30 days, the 0.005% swap fee is taken from Balancer's wstETH/ETH pool, and the exchange rate is from Curve Finance's concentrated stETH/ETH pool on mainnet.

Recommendations

Short term, Chainlink should update the deviation threshold for all Lido-related oracles to 0%, with a maximum heartbeat of 24 hours 30 minutes. This will force all wstETH oracles to update when Lido performs their daily rebase.

Long term, this fix may not be adequate because it may still leave several minutes where the issue is exploitable, and it is uncertain whether Chainlink can update the oracle configuration to use a deviation threshold of 0%.

So in addition to the above, we recommend incorporating this finding's fix into `fastStake`'s fee model. This finding is only exploitable under very precise conditions: when the oracle's price discrepancy exceeds the difference between the stETH discount and the DEX's swap fees. If `fastStake` implements a fee that ensures the inequality is never broken, then the finding is mitigated. A `fastStake` fee of as little as 0.022% would prevent the exploit scenario from being viable.

5. Price oracle fails to verify sequencer liveness	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHAINLINK-CSR-5
Target: `PriceOracle.sol`	

Description

The `PriceOracle` contract fails to verify whether the L2's sequencer is up or recently started, potentially allowing swaps to execute at stale prices.

L2 optimistic rollups use centralized sequencers to post transaction batches to mainnet. When these sequencers go down, transactions submitted over the L2 peer-to-peer layer are queued until the sequencer is alive again.

If an oracle price update is submitted to the network during a sequencer outage, it will be executed as soon as the sequencer comes back up and will use `block.timestamp` to record an `updatedAt` equal to the time the transaction was executed rather than the time the transaction was generated. This could cause a stale price to be treated as valid, potentially allowing users to perform swaps at an invalid exchange rate.

Exploit Scenario

Consider a deployment where the wstETH:ETH oracle currently returns a 1:1 price ratio.

1. The sequencer for the L2 goes down.
2. The Chainlink oracle's heartbeat elapses and a transaction is sent refreshing the exchange rate to 1:1. This transaction waits in the mempool for the sequencer to come back online.
3. Lido's validators build and propose an unusually large number of high-value blocks, pushing the exchange rate up to 1 wstETH per 1.05 ETH.
4. Chainlink creates and sends a volatility-based price update to reflect the new exchange rate. This transaction is also pending in the mempool.
5. Later, the sequencer comes back online and starts processing the mempool backlog.
6. The 1:1 heartbeat-based price update is processed first. Although this report is now stale, the associated timestamp is set by the `block.timestamp` so the `OraclePool` contract considers it valid.

7. Users race the second price update and perform large, profitable `fastStake` function calls, potentially draining the `OraclePool` contract of wstETH.
8. The second price update is processed, bringing the exchange rate back in line with market conditions.

Recommendations

Short term, ensure this finding is well-documented for CSR operators. It is likely not worth the gas to check the sequencer uptime oracle if such arbitrage opportunities are unlikely to become available during outages, especially if the following long term recommendation will be implemented in the near future.

Long term, update the `OffchainAggregator` contract to set the `updatedAt` timestamp according to when the report was generated instead of when the report was processed by the L2 chain. It may also be possible to address this issue on the Chainlink side by replacing transactions in the mempool when updating an oracle to a stale price.

6. Eigenpie deposit flow may be vulnerable to sandwiching attacks

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-CHAINLINK-CSR-6
Target: `EigenpieCustomReceiver`	

Description

The `EigenpieCustomReceiver` contract calls `depositAsset` using a value of "0" for the minimum amount out, potentially exposing the system to sandwiching attacks. The code that calls `depositAsset` is shown in figure 6.1.

```
function _depositNative(uint256 amount) internal override returns (uint256) {
    uint256 balance = IERC20(EGETH).balanceOf(address(this));

    IEigenpieStaking(EGETH_STAKING).depositAsset{value:
amount}(PLATFORM_TOKEN_ADDRESS, amount, 0, address(0));

    return IERC20(EGETH).balanceOf(address(this)) - balance;
}
```

Figure 6.1: `depositAsset` is called with a `minimumAmountOut` of zero. (chainlink-csr/contracts/receivers/EigenpieCustomReceiver.sol#50–56)

A `minimumAmountOut` parameter usually indicates that a function involves some kind of price-sensitive swap that may be vulnerable to sandwiching attacks.

If that is the case for Eigenpie, then providing a `minimumAmountOut` of zero creates a potential sandwiching attack that may cause fewer tokens to be minted as intended. Since Chainlink CCIP transactions are posted to the public mempool, it is trivial for MEV searchers to identify and sandwich vulnerable transactions.

Unfortunately, Eigenpie does not have adequate documentation to determine the purpose of `minimumAmountOut`. We made an effort to investigate Eigenpie's source to determine whether it uses `minimumAmountOut` to prevent potential MEV attacks, but were unable to form a conclusive opinion in the limited time allocated.

Exploit Scenario

A malicious user is able to tamper with Eigenpie's ETH/wstETH exchange rate using precisely timed stake/unstake transactions.

A sync transaction is created on the L2 for Eigenpie, and is sent to CCIP for bridging.

The malicious user observes the CCIP offramp transaction in the mempool and sandwiches it to force it to execute at a poor price for CSR. After the CCIP offramp transaction, the malicious user clears their position and harvests the profit.

Recommendations

Short term, verify whether Eigenpie staking calls are vulnerable to sandwiching-based MEV. If so, use a `minimumAmountOut` value that ensures a fair execution price. This value may be passed in from the L2->L1 CCIP transaction, or alternatively obtained from a Chainlink oracle on mainnet.

Long term, ensure adequate guards are in place to prevent MEV from being extracted from staking protocol calls. Any function called by CSV that includes a `minimumAmountOut` parameter should be heavily scrutinized for potential MEV vectors.

7. The CustomSender contract does not enforce minimum gas amounts when bridging

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-CHAINLINK-CSR-7
Target: `CustomSender`, `CustomReceiver`	

Description
The `CustomSender` contract does not enforce a minimum amount of gas when calling `ccipSend` to the receiver contracts. This may cause unexpected behavior such as L1 transaction reverts when calling out to the staking protocol, or reverts that are not captured within the `CustomReceiver`'s `failedHashes` feature.

Although the amount of gas required to call each LST protocol is not known at this time, a minimum amount of gas to allow the `failedHashes` to operate can still be determined to partially fix this issue.

Exploit Scenario
We establish two exploit scenarios:

1. A user calls `slowStake` with a very small amount of gas for the L1 transaction. When CCIP calls the receiver contract, the receiver contract reverts with "out of gas" before the LST protocol is even called. This will cause confusion for the user because their transaction will not show up in `failedHashes`; their transaction will have to be re-tried using CCIP itself.
2. A user calls `slowStake` with a moderate amount of gas, but not enough to complete the LST call. In this scenario, the LST call will revert with "out of gas," but the try/catch in `ccipReceive` will not be able to add the transaction to the `failedHashes`, as shown in figure 7.1.

```
function ccipReceive(Client.Any2EVMMessage calldata message) external override
onlyCCIPRouter nonReentrant {
    _checkSender(message.sourceChainSelector, message.sender);

    try this.processMessage(message) {
        emit MessageSucceeded(message.messageId);
    } catch (bytes memory error) {
        _getCCIPDefensiveReceiverStorage().failedHashes[message.messageId] =
keccak256(abi.encode(message));
```

```
        emit MessageFailed(message, error);
    }
}
```

Figure 7.1: The highlighted section of code will not run if `processMessage` runs out of gas. (chainlink-csr/contracts/ccip/CCIPDefensiveReceiverUpgradeable.sol#183–192)

Recommendations

Short term, determine the amount of gas required to call the `CustomReceiver` and execute all code up to the `processMessage` call; assume this value is called `RECEIVE_PRE_PROCESS_GAS`. Then, determine the minimum amount of gas required to handle the catch{} statement if the `processMessage` call fails; assume this value is called `PROCESSMESSAGE_REVERT_GAS`.

Modify the `processMessage` call to include a custom amount of gas, specifically `gas left()-PROCESSMESSAGE_REVERT_GAS`. This will ensure that there is always enough gas remaining to add a failed `processMessage` transaction to the `failedHashes`, addressing exploit scenario 2.

In the `CustomSender` contract, ensure enough gas is purchased to exceed `RECEIVE_PRE_PROCESS_GAS+PROCESSMESSAGE_REVERT_GAS`. This will ensure that all messages created by `CustomSender`, when failed, will at least have their hashes added to `failedHashes`. This will address exploit scenario 1.

Long term, determine the maximum amount of gas consumed by each LST for `processMessage`, and optionally enshrine that in the contracts as well.

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 Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. 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.

- **Address repeated constants**. The constant `1e18` is repeated throughout the codebase. Consider assigning this value to a constant with a descriptive name and using this instead.
- **Avoid infinite approvals**. The `CustomSender` contract uses an infinite approval for the CCIP router, as shown in figure C.1. This behavior is not recommended, and should be changed to use manual, specific-amount approvals whenever CCIP is invoked.

  If the infinite approval is kept in place, it is recommended to document this in the protocol's runbook, and ensure the operator has a plan and provisions in place for handling a CCIP router hack.

```
function initialize(address oraclePool, address initialAdmin) public initializer {
    _grantRole(DEFAULT_ADMIN_ROLE, initialAdmin);
    _setOraclePool(oraclePool);

    IERC20(WNATIVE).forceApprove(CCIP_ROUTER, type(uint256).max);
}
```

Figure C.1: The `CustomSender` contract uses infinite approvals for CCIP. (chainlink-csr/contracts/senders/CustomSender.sol#65–70)

- **Improve event logging**. There is one event in the system that does not emit adequate information to reconstruct what is happening, making it harder to perform off-chain logging or incident triage. The event can be observed in figure C.2.

```
function _sendToken(uint64, address to, uint256 amount, bytes memory feeData)
internal override {
    [...]
    IBaseL1StandardBridge(L1_STANDARD_BRIDGE).depositERC20To(L1_TOKEN, L2_TOKEN, to,
amount, l2Gas, new bytes(0));

    emit BaseL1toL2MessageSent();
```

Figure C.2: The `BaseL1toL2MessageSent` event does not emit adequate information for off-chain logging or incident triage. (`chainlink-csr/contracts/adapters/BaseAdapterL1toL2.sol#47–56`)

- **Address unused events**. A `MessageSent` event is defined in the `ICustomSender` interface but is never used. Consider emitting this event where appropriate or removing it if it is not necessary.

D. Automated Analysis Tool Configuration

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

Slither

To install Slither, we used `pip` by running the command shown in figure D.1.

```
python3 -m pip install slither-analyzer
```

Figure D.1: Slither installation

We used Slither to detect common issues and anti-patterns in the codebase. Although Slither did not discover any severe issues during this review, some informational and code-quality issues were detected. Integrating Slither into the project's testing environment can help find other issues that may be introduced during further development and will help improve the overall quality of the smart contracts' code.

```
slither . --no-fail-pedantic --exclude='naming-convention,solc-version'
--filter-paths lib,script,test,node_modules
```

Figure D.2: An example Slither configuration

Integrating `slither-action` into the project's CI pipeline can automate this process.

E. Mutation Testing

Mutation tests make changes to each line of a target contract's source code and re-runs the test suite against each change. Changes that result in test failures indicate adequate test coverage, while changes that do not cause tests to fail indicate gaps in test coverage. Mutation testing allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it helps developers identify and add missing tests.

Slither version 0.10.2 introduced a built-in Solidity mutation testing tool with first-class support for Foundry projects. A mutation testing campaign was run against the target smart contracts using the following command:

```
slither-mutate ./contracts \
  --test-cmd='forge test' \
  --ignore-dirs='interfaces'
```

Figure E.1: A command that runs a mutation testing campaign against all smart contracts in the `protocol` repository

Note that the overall runtime of this campaign is approximately 40 minutes on a consumer grade laptop. An abbreviated, illustrative example of a mutation test output file is shown in figure E.2.

```
INFO:Slither-Mutate:Mutating contract CustomReceiver
INFO:Slither-Mutate:[CR] Line 53: '__CCIPDefensiveReceiver_init()' ==>
'//__CCIPDefensiveReceiver_init()' ==> UNCAUGHT
INFO:Slither-Mutate:[CR] Line 99: 'emit AdapterSet(destChainSelector, adapter)' ==>
'//emit AdapterSet(destChainSelector, adapter)' ==> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 66: 'msg.sender != WNATIVE' ==> 'false' ==> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 113: 'message.destTokenAmounts.length != 1' ==>
'message.destTokenAmounts.length  <  1' ==> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 122: 'wnativeAmount != amount + feeAmount' ==>
'wnativeAmount  <  amount + feeAmount' ==> UNCAUGHT
INFO:Slither-Mutate:Done mutating CustomReceiver.
INFO:Slither-Mutate:Revert mutants: 0 uncaught of 10 (0.0%)
INFO:Slither-Mutate:Comment mutants: 2 uncaught of 11 (18.181818181818183%)
INFO:Slither-Mutate:Tweak mutants: 3 uncaught of 34 (8.823529411764706%)
```

Figure E.2: Abbreviated output from the mutation testing campaign on `CustomReceiver.sol`

In summary, the following features of the `CustomReceiver` contract lack sufficient tests:

- No tests validate the effects of the `__CCIPDefensiveReceiver_init` method on line 53.
- No test validates the failure case of the safeguard on line 66.

- No tests validate emission of the `AdapterSet` event nor its parameters on line 99. This type of test coverage gap occurs frequently across many contracts.
- The ROR mutants on lines 113 and 122 indicate that inequality checks are tested in one direction but not the other. For completeness, consider testing the contract's behavior if the `message.destTokenAmounts` has a length of 0 and if `wnativeAmount` is less than the sum of the `amount` and `feeAmount`.

We recommend that Chainlink Lab review the existing tests, add additional verification that would catch the aforementioned types of mutations. Then, use a script similar to that provided in figure E.1 to rerun a mutation testing campaign to ensure that the added tests provide adequate coverage.

F. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Elixir Technologies Ltd will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

G. 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 a comprehensive analysis of the system.

On September 13, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the contract author for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the seven issues described in this report, the contract author has resolved six issues, and has accepted the risk of the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	CustomSender only supports sending Wrapped Ether	Resolved
2	Lack of zero-value checks	Resolved
3	OraclePool has insolvency risks when LSTs are slashed	Resolved
4	Swapping at Chainlink prices will lead to CSR insolvency due to oracle timing discrepancy	Resolved
5	Price oracle fails to verify sequencer liveness	Undetermined
6	Eigenpie deposit flow may be vulnerable to sandwiching attacks	Resolved
7	The CustomSender contract does not enforce minimum gas amounts when bridging	Resolved

Detailed Fix Review Results

TOB-CHAINLINK-CSR-1: CustomSender only supports sending Wrapped Ether
Resolved in commit hash bc29a7. `CustomSender` logic has been updated to allow the use of non-native tokens.

TOB-CHAINLINK-CSR-2: Lack of zero-value checks
Resolved in commit hash `5f28a1`. Zero value checks have been implemented across the codebase.

TOB-CHAINLINK-CSR-3: OraclePool has insolvency risks when LSTs are slashed
Resolved in commit hash `9566ca`. `OraclePool` will now revert if the results of the latest price query indicate an LST was slashed.

TOB-CHAINLINK-CSR-4: Swapping at Chainlink prices will lead to CSR insolvency due to oracle timing discrepancy
Resolved by updating the derivation threshold for oracle price updates to the minimum allowable value. The wstETH/stETH oracles were verified to use this new derivation threshold on Arbitrum, Base, and Optimism.

TOB-CHAINLINK-CSR-5: Price oracle fails to verify sequencer liveness
Undetermined. The client has accepted the risk of this issue, choosing to document its presence in the contract's runbook in accordance with the short-term guidance.

TOB-CHAINLINK-CSR-6: Eigenpie deposit flow may be vulnerable to sandwiching attacks
Resolved. Eigenpie has confirmed that the minimum amount out value is not used to prevent sandwiching attacks.

TOB-CHAINLINK-CSR-7: The CustomSender contract does not enforce minimum gas amounts when bridging
Resolved in commit hash `f26939`. The custom sender now ensures a minimum amount of gas is provided for bridge transactions.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

I. Subsequent PR Review

When undertaking a subsequent review, Trail of Bits reviews changes made to the codebase after the delivery of the original report. This work involves a review of the specific areas of source code introduced by the change, not a comprehensive analysis of the system.

On February 25, 2025, Trail of Bits reviewed a change to the CSR system that was made after the initial engagement and was not part of its original scope. The review covered commits up to and including 495a51d. We reviewed the change to determine whether it introduces any security issues into the system. In summary, we did not identify any issues introduced by the reviewed change. For additional information, see the detailed results below.

Detailed Subsequent PR Review Results

PR #3: Add Custom Sender referral
Reviewed at merge commit eff9895. The pull request introduces a `CustomSenderReferral` wrapper that allows a user to specify a referral when fast staking through a new `fastStakeReferral` function. To support this, the visibility of `fastStake` is changed to public so that it can be invoked internally by the wrapper.

Our review confirmed the following:

- The referral event does not introduce any checks-effects-interactions (CEI) ordering issues.
- The `fastStake` function continues to revert as expected when an error occurs.
- The arguments are ordered correctly and the call is forwarded to the parent contract as intended.

We did not identify any security issues introduced by this change.