cLabs, Celo L2

Security Assessment

December 19, 2024

Prepared for:
Nikolaos Frestis
cLabs

Prepared by: **Anish Naik, Guillermo Larregay, and Tarun Bansal**

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

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

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@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:

Anish Naik, Consultant
anish.naik@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

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

Date	Event
October 16, 2024	Pre-project kickoff call
October 25, 2024	Status update meeting #1
November 1, 2024	Status update meeting #2
November 8, 2024	Status update meeting #3
November 19, 2024	Status update meeting #4
November 22, 2024	Delivery of report draft and report readout meeting
December 19, 2024	Delivery of final comprehensive report

Executive Summary

Engagement Overview

cLabs engaged Trail of Bits to review the security of its new Celo L2 blockchain. The rollup is a fork of the OP Stack with support for custom features such as token duality, alternative fee currencies for gas payments, and the ability to use an alternative data availability layer (alt-DA). Currently, cLabs runs the Celo L1 blockchain. The L1 blockchain will be run through a data migration process that will convert it into a rollup.

A team of three consultants conducted the review from October 17 to November 22, 2024, for a total of eight engineer-weeks of effort. Our testing efforts focused on identifying vectors that could lead to the loss/theft of funds, opportunities to bypass access controls or gas payments, vulnerabilities to denial-of-service attacks, deviations from traditional EVM- or transaction-level semantics, and deviations that could reduce backward compatibility or compatibility with the `op-geth` API. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

The code reviewed during this audit is generally of high quality. We were unable to find any vectors that could lead to a serious violation of the security of the system. However, we did identify two core patterns of vulnerabilities within the system, which are highlighted below.

Most of the issues identified during this audit pertain to the alternative fee currency implementation (TOB-CELO-L2-1, TOB-CELO-L2-4, TOB-CELO-L2-5, TOB-CELO-L2-6, and TOB-CELO-L2-7). These issues are deeply rooted within the system and cannot be triggered using the tests that come out-of-box with vanilla `op-geth` or the custom tests written by the cLabs team to test the fee currency logic. For example, the bug in the `buyGas` function (TOB-CELO-L2-1), is hard to find because traditional unit tests will test the state-transition function as a whole and will not assert on the post conditions of the `buyGas` function. Although all the identified issues surrounding the fee currency logic are generally benign, they highlight the possible benefits of directly testing these less-tested code paths.

We also identified two issues within the smart contracts that could lead to issues when the migration is complete and the contracts are upgraded (TOB-CELO-L2-8 and TOB-CELO-L2-10). During our review, we prioritized two requirements of the smart contract logic: no upgrade should cause a storage slot collision, and there must be a clear distinction between what should be callable only on L1 and only on L2. TOB-CELO-L2-8 violates the first requirement, and TOB-CELO-L2-10 violates the second. The core pattern that led to these issues is the primary reliance on manual processes for validation. Both issues were identified by scripts using the Slither API, a static analysis tool. (See the Automated Testing section for more details.) Currently, it is difficult to integrate Slither into the CI/CD pipeline because the monorepo is built using two separate compilation platforms. Compiling all the contracts using the same platform will generally allow for easier building, testing, and integrating with third-party testing frameworks.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that cLabs take the following steps prior to the L2 mainnet launch:

- **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.

- **Add testing for all `op-geth` functions that contain fee currency logic.** Adding additional unit tests that test these functions in isolation will aid in validating the fee currency behavior and is a good practice when updating critical code paths in `op-geth`.

- **Update the `celo-monorepo` project to use the same compilation platform.** Compiling, building, deploying, and testing all smart contracts using the same compilation platform will reduce the complexity of the codebase and the CI/CD pipeline and allow for easier third-party tool integrations.

- **Rerun the provided scripts and commands if additional smart contract changes are made.** If additional smart contract changes are made, especially if they update the storage layout, rerun the scripts provided in appendix D and the commands provided in appendix E.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	4
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	7
Error Reporting	1

Project Goals

The engagement was scoped to provide a security assessment of the cLabs Celo L2 blockchain. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the data migration logic correctly transform the blocks and headers from the old Celo L1 database and ensure compatibility with the Ethereum specification?
- Is the data migration logic prone to any panics or unhandled errors?
- Does the token duality logic have the necessary access controls on the transfer precompile to prevent theft of funds?
- Does the `GoldToken` smart contract have any edge cases that were introduced due to the use of the transfer precompile?
- Can a user bypass gas fees or pay less in gas fees by using an alternative fee currency?
- Does the mempool correctly order transactions across various fee currencies?
- Are there any storage layout changes in the smart contracts that could lead to storage overwrites during the L2 migration?
- Are there any opportunities to steal funds from the unreleased treasury?
- Do the epoch manager and the associated contracts correctly manage validator rewards and epoch state?
- Is it possible to call any smart contract functions that should not be callable on L2?
- Does the alt-DA layer open any denial-of-service attack vectors or cause a deviation in the expected behavior of the batcher and rollup processes?

Project Targets

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

celo-monorepo

Repository	https://github.com/celo-org/celo-monorepo
Versions	`89591e1` and PR #11261
Type	Solidity
Platform	Celo L2

op-geth

Repository	https://github.com/celo-org/op-geth
Version	`c6d7b55`
Type	Go
Platform	Linux, macOS, Windows

optimism

Repository	https://github.com/celo-org/optimism
Version	`50977c3`
Type	Go, Solidity
Platform	Celo L2, Linux, macOS, Windows

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:

- **Token duality:** Token duality allows the native currency, CELO, to be used also as an ERC-20 token. Thus, a token transfer of CELO will update the native balance of the sender and the recipient. We performed a manual review of this system, focusing on the following:
    - We checked whether it is possible to bypass the access controls on the transfer precompile, which would allow an attacker to steal funds from other users.
    - We reviewed the `GoldToken` smart contract logic to ensure that the use of the transfer precompile does not lead to any unexpected edge cases or a violation of any ERC-20 invariants.
    - We reviewed the transfer precompile to ensure that errors are handled correctly and that it is implemented correctly.
- **Alternative fee currencies:** Alternative fee currencies allow users to pay for gas using whitelisted ERC-20 tokens. We performed a manual review of this logic, focusing on the following:
    - We checked whether it is possible for an attacker to bypass gas fees or pay less in gas fees by using an alternative fee currency.
    - We reviewed the state-transition function to ensure that the user pays for the EVM-level calls to the debit and credit gas fee functions.
    - We reviewed the state-transition logic to ensure that there are no direct comparisons between values that represent CELO and values that represent a fee currency. Through this review, we discovered that the logic incorrectly increments a CELO-denominated value with a fee currency–denominated value, which could cause user transactions to fail (TOB-CELO-L2-1).
    - We reviewed the mempool logic to ensure that transactions are correctly ordered when various fee currencies are handled. Through this review, we discovered that the miner tip is incorrectly calculated for mempool transactions, which could lead to incorrect ordering (TOB-CELO-L2-4). Additionally, we identified that transaction replacements could unexpectedly fail during transaction validation in the mempool (TOB-CELO-L2-5 and TOB-CELO-L2-6).

    - We reviewed the miner and worker logic to ensure that the multi-gas implementation correctly limits the gas usage of each fee currency and that the fee currency context is correctly updated per block.

    - We reviewed compatibility with the RPC layer to ensure that the introduction of fee currencies did not lead to unexpected edge cases or the returning of incorrect response data. Through this review, we discovered that the gas estimation RPC call may unexpectedly fail when using an alternative fee currency for gas (TOB-CELO-L2-7).

- **Smart contract logic:** The smart contract logic in the `celo-monorepo` project holds the various smart contracts that will be predeployed on the L2 blockchain. These contracts have storage layout updates and code modifications that need to be accounted for when the migration occurs. Additionally, a new epoch management system that is responsible for managing epochs, validator rewards, and validator elections was implemented. Note that the epoch management system will not be immediately deployed into production when the L2 blockchain is first initialized. As scoped, the diff between branches `release/core-contracts/11` and `release/core-contracts/12` was reviewed manually and dynamically. Additionally, the epoch manager and its associated contracts (e.g., unreleased treasury) were reviewed comprehensively:

    - We reviewed the epoch manager logic to ensure that epoch rewards are correctly distributed to validators and voters, epoch-related state is tracked and updated correctly, epoch rewards are not double-counted, and validator payments cannot be hijacked by an external party.

    - We reviewed the unreleased treasury to verify that there is no way for an attacker to steal funds from it.

    - We used a static analyzer and manually reviewed the diff of all smart contracts that will be upgraded to ensure that the updates are backward compatible and that the requisite modifiers are placed to differentiate behavior that should be available on L1 and/or L2. We discovered that the to-be-deprecated `Attestations` contract is missing the `onlyL1` checks for the relevant functions (TOB-CELO-L2-10).

    - We used a static analyzer and manually reviewed the diff of all smart contracts that will be upgraded in the next release to ensure that any state changes made to the implementation contract do not cause state collisions or overrides in the proxy's storage layout. We discovered that `FeeHandlerSeller` storage changes can break functionality of `UniswapFeeHandlerSeller` (TOB-CELO-L2-8).

- **Data migration:** The data migration logic is responsible for migrating all the blocks from the Celo L1 database into the new L2 database. During this migration, each block's header and body is transformed to be compatible with the existing `op-geth` API and its RPC layer. We performed a manual review of this system and investigated the following:
  - We reviewed the migration logic to identify off-by-one errors and to ensure that the block header and body transformations would comply with the `op-geth` API.
  - We reviewed the updates made to the genesis configuration for general correctness.
  - We reviewed the creation of the first L2 block and the rollup configuration for general correctness. Additionally, we reviewed the creation of the unreleased treasury to ensure that the correct CELO balance is added to the contract state.
  - We reviewed the error handling and data validation performed in the logic to ensure that all errors are sufficiently handled and that there are no opportunities to trigger a panic that would prevent node startup.
  - We reviewed the use of `rcopy` to ensure that the configuration of the tools and the flags used will properly migrate the non-ancient database.
- **Alt-DA:** The alt-DA support feature allows the batcher to upload the L2 transaction batches to a data availability layer such as the EigenDA layer to reduce the cost of transactions and increase the chain throughput. The rollup process then fetches the L2 input transactions from the configured data availability service and creates the new L2 state. We performed a manual review of this system, focusing on the following:
  - We checked whether the new configuration values are parsed and used correctly by the codebase.
  - We checked whether concurrent submission of the L2 transaction batches to the data availability service results in a crash or unexpected behavior.
  - We checked whether users can execute a denial-of-service attack on the system by challenging the input data commitment or by any other means.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Custom gas token:** The custom gas token logic was not reviewed during this audit. As discussed during scoping, this was deprioritized in favor of the other components in the system.

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

Areas of Focus

Our automated testing and verification work focused on ensuring that contract changes from the previous version of Celo match the specifications and the provided documentation.

Given Celo's particular directory and contract structure for different compiler versions, a helper script was created to compile and make all contracts available for later analysis. This script makes all the necessary changes to the configuration files to compile contracts from the `contracts` and `contracts-0.8` folders and stores the result on disk. Slither can later read these files, so it does not require the contracts to be recompiled every time the analysis is performed.

The following table summarizes the scripts' descriptions and names, and appendix D gives a more thorough description of how they work and how they can be used to help detect bugs earlier.

Description	Script Name
Join all contracts in the `contracts` and `contracts-0.8` folders into zip files	`generate_zip.py`
Find all functions that have the specified modifiers	`find_modifiers.py`
Check all of a contract's functions for the specified modifier	`find_modifiers_by_contract.py`

Generate a call trace given a contract and a function	`trace_calls.py`
Review storage changes given two versions of the contracts	`check_storage.py`

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	We identified two arithmetic issues when reviewing the fee currency logic (TOB-CELO-L2-1 and TOB-CELO-L2-4). The root cause for these issues is the performing of operations on values that represent different currencies. In general, updates to any critical path in `op-geth` should be accompanied with unit tests (with tracing, if possible) and integration tests. Both issues are hidden deep in the codebase's complexity and cannot be identified with the existing test suite of `op-geth`. No other arithmetic issues were identified in the other components within scope. It is important to note that the `celo-monorepo` project has a number of contracts that still use Solidity version 0.5. It would be beneficial to, in a future release, upgrade these contracts to a more stable version (at least 0.8) that has built-in arithmetic overflow protection.	**Moderate**
Auditing	Most critical state-changing functions in the smart contract logic emit events, but some do not (TOB-CELO-L2-2). Asserting on the emission of an event(s) in unit tests would aid in preventing these sorts of issues (e.g., using `expectEmit` in Foundry). The off-chain components have sufficient logging of events and metrics.	**Satisfactory**
Authentication / Access Controls	We did not identify any vectors that could enable access controls to be bypassed or unauthorized actions to be taken.	**Strong**
Complexity Management	The smart contract logic is generally well structured. Even though the call stacks and execution flows can be	**Moderate**

	complex to analyze or follow, individual functions are easy to read and are well scoped. However, using two separate compilation versions and platforms adds complexity during building and testing. Additionally, it makes it difficult to integrate external tooling (e.g., Slither). It would be beneficial to migrate all the smart contracts and tests to use the same compilation platform. Since the system is an OP Stack fork, it inherits the complexity of the OP Stack. Features such as token duality have been implemented effectively without touching too many components of the stack. However, the fee currency logic is complicated and led to a large majority of the bugs during this audit. Having verbose comments wherever any change is made to vanilla `op-geth` or `optimism` will aid in maintaining the diff. This will aid in defining the boundaries between custom and upstream code.	
Configuration	No issues were identified in the data migration logic or the configuration of the rollup. The configuration was aligned with the provided specification. However, one issue identified in the configuration of the alt-DA service could lead to a node crash (TOB-CELO-L2-9).	**Satisfactory**
Data Handling	The fee currency logic resulted in a large majority of the data validation issues identified in the audit (e.g., TOB-CELO-L2-4 or TOB-CELO-L2-5). Similar to the arithmetic-related issues found around the fee currency logic, improving the testing of any custom code would prevent these types of issues from being introduced in the future. Additionally, the alt-DA service has one instance of insufficient data validation, which could cause a node to crash (TOB-CELO-L2-9). Validating all provided inputs to a function (e.g., making sure they are not `nil` and are within the expected values) will prevent issues like this in the future. The other components of the system have sufficient data validation to prevent any unexpected edge cases from being reached.	**Moderate**
Documentation	The provided documentation and specification are generally high quality. The specification clearly outlines	**Strong**

	the various capabilities of the system and the critical changes that were made as the blockchain migrates from an L1 to an L2. Additionally, the inline documentation generally aids in understanding where custom code was added and the rationale behind doing so.	
Low-Level Manipulation	We identified some low-level manipulation in the smart contract logic. However, as this logic was not changed in the provided diff, the low-level logic was not reviewed during the audit.	**Not Applicable**
Memory Safety and Error Handling	We identified a few direct pointer-to-pointer comparisons, one of which led to TOB-CELO-L2-6. Creating a standard practice to compare fee currencies would reduce the risk of comparing pointers directly. Error handling is generally sufficient throughout the codebase, except for a few instances in the data migration logic, highlighted in TOB-CELO-L2-3. Given the critical nature of the data migration logic, handling every error (including those in `defer` statements) will reduce the risk that the script will silently throw an error and cause the node to crash on startup.	**Moderate**
Testing and Verification	The smart contract logic is generally well tested using unit and integration tests. However, due to the structure of the `celo-monorepo` project, some tests can be difficult to implement. Adding custom scripts to the CI/CD could improve the test suite and help detect issues earlier. Consider creating upgradeability tests leveraging `slither-check-upgradeability` or similar tools. See appendix E for more information. The fee currency logic, however, is not sufficiently tested. To successfully implement the feature, a large number of `op-geth` components needed to be updated (e.g., state transition function, miner, mempool, and RPC). In vanilla `op-geth`, some of these components do not have any direct unit tests (e.g., the gas estimation logic). However, since these components were updated, it is essential that unit tests are implemented to test the custom functionality and verify its backward compatibility. The other off-chain components of the review (e.g., token duality and the alt-DA integration) are generally well	**Moderate**

	tested.	
Transaction Ordering	Transaction ordering risks were not considered during the current audit.	**Not Applicable**

Summary of Findings

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

ID	Title	Type	Severity
1	User transactions may unexpectedly fail	Data Validation	Low
2	Insufficient event generation	Auditing and Logging	Informational
3	Data migration process missing error handling and invariant checks	Error Reporting	Informational
4	Incorrect ordering of mempool transactions	Data Validation	Medium
5	Transaction replacements may fail when using CELO for gas	Data Validation	Low
6	Transaction replacements may fail when using a fee currency for gas	Data Validation	Low
7	Gas estimation may fail	Data Validation	Low
8	Storage change in FeeHandlerSeller affects UniswapFeeHandlerSeller	Data Validation	Medium
9	Lack of validation of op-node command line data availability flag	Data Validation	Informational
10	Missing onlyL1 modifier in Attestations contract's functions	Access Controls	Informational

Detailed Findings

1. User transactions may unexpectedly fail	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-1
Target: `op-geth/core/state_transition.go`	

Description

Due to incorrect token balance calculations during gas accounting, user transactions may unexpectedly fail.

As part of the alternative fee currency logic, a user may transfer CELO while paying for the transfer using an alternative fee currency. In `op-geth`, the gas accounting logic should validate that the user has enough CELO for the transfer and enough fee currency for gas (figure 1.1).

```go
func (st *StateTransition) buyGas() error {
    mgval := new(big.Int).SetUint64(st.msg.GasLimit)
    mgval.Mul(mgval, st.msg.GasPrice)
    var l1Cost *big.Int
    if st.evm.Context.L1CostFunc != nil && !st.msg.SkipAccountChecks {
        l1Cost = st.evm.Context.L1CostFunc(st.msg.RollupCostData, st.evm.Context.Time)
        if l1Cost != nil {
            mgval = mgval.Add(mgval, l1Cost)
        }
    }
    balanceCheck := new(big.Int).Set(mgval)
    if st.msg.GasFeeCap != nil {
        balanceCheck.SetUint64(st.msg.GasLimit)
        balanceCheck = balanceCheck.Mul(balanceCheck, st.msg.GasFeeCap)
    }
    balanceCheck.Add(balanceCheck, st.msg.Value)
    if l1Cost != nil {
        balanceCheck.Add(balanceCheck, l1Cost)
    }

    if st.evm.ChainConfig().IsCancun(st.evm.Context.BlockNumber, st.evm.Context.Time) {
        if blobGas := st.blobGasUsed(); blobGas > 0 {
            // Check that the user has enough funds to cover blobGasUsed * tx.BlobGasFeeCap
```

```
 327                          blobBalanceCheck := new(big.Int).SetUint64(blobGas)
 328                          blobBalanceCheck.Mul(blobBalanceCheck,
st.msg.BlobGasFeeCap)
 329                          balanceCheck.Add(balanceCheck, blobBalanceCheck)
 330                          // Pay for blobGasUsed * actual blob fee
 331                          blobFee := new(big.Int).SetUint64(blobGas)
 332                          blobFee.Mul(blobFee, st.evm.Context.BlobBaseFee)
 333                          mgval.Add(mgval, blobFee)
 334                  }
 335          }
 336          balanceCheckU256, overflow := uint256.FromBig(balanceCheck)
 337          if overflow {
 338                  return fmt.Errorf("%w: address %v required balance exceeds 256
bits", ErrInsufficientFunds, st.msg.From.Hex())
 339          }
 340
 341          if err := st.canPayFee(balanceCheckU256); err != nil {
 342                  return err
 343          }
 344          if err := st.gp.SubGas(st.msg.GasLimit); err != nil {
 345                  return err
 346          }
 347
 348          if st.evm.Config.Tracer != nil && st.evm.Config.Tracer.OnGasChange !=
nil {
 349                  st.evm.Config.Tracer.OnGasChange(0, st.msg.GasLimit,
tracing.GasChangeTxInitialBalance)
 350          }
 351          st.gasRemaining = st.msg.GasLimit
 352
 353          st.initialGas = st.msg.GasLimit
 354
 355          return st.subFees(mgval)
 356  }
```

Figure 1.1: `op-geth/core/state_transition.go#L304–L356`

However, notice in line 319, highlighted in figure 1.1, that the gas cost (denominated in the fee currency) is added to the message value (denominated in CELO). Thus, when the `canPayFee` function is called on line 341, the function may return an error, which would cause the transaction to fail. This is because the `canPayFee` function checks whether the user has a sufficient token balance to cover the transfer and the gas costs. However, as described above, the transfer and the gas costs are denominated by two different tokens.

Exploit Scenario

Alice wishes to transfer 10 CELO to Bob while paying for the transfer using aFEE, an alternative fee currency. This transaction fails to execute successfully.

Recommendations

Short term, update the gas accounting logic to validate native token balances separately from alternative fee currency balances. This can be done by updating the `canPayFee` function to accept two separate arguments (message value and gas fees) and evaluate whether the user has sufficient balances to cover both.

Long term, improve the unit testing of the state-transition function to ensure that gas accounting is performed correctly in all possible scenarios.

2. Insufficient event generation

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-CELO-L2-2
Target: `celo-monorepo/packages/protocol/contracts-0.8/common/EpochManager.sol`, `celo-monorepo/packages/protocol/contracts-0.8/common/FeeCurrencyDirectory.sol`	

Description

Some critical operations do not emit events. As a result, it will be difficult to review the contracts' behavior for correctness once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

The following operations should trigger events:

- `EpochManager.setToProcessGroups`
- `EpochManager.processGroup`
- `FeeCurrencyDirectory.setCurrencyConfig`
- `FeeCurrencyDirectory.removeCurrencies`

Exploit Scenario

An attacker discovers a vulnerability in the `EpochManager` contract and modifies its execution. Because these actions generate no events, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

Recommendations

Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

3. Data migration process missing error handling and invariant checks	
Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-CELOL2-3
Target: `optimism/op-chain-ops/cmd/celo-migrate/`	

Description

We identified the following instances in the data migration process in which errors are not handled or an invariant is not checked:

- **All errors are not captured and returned in the following areas of the code.** Although errors are unlikely to occur in these areas, it is beneficial to catch any unexpected edge cases, including errors that may occur in `defer` statements, so that the data migration logic works as expected. Note that in the instance shown in figure 3.2, only the `os.ErrNotExist` error type is handled, whereas the `os.Stat` command can return more than one type of error.

```
22    output, _ := cmdHelp.CombinedOutput()
```

Figure 3.1: `optimism/op-chain-ops/cmd/celo-migrate/non-ancients.go#L22`

```
41    if _, err := os.Stat(chaindataPath); errors.Is(err, os.ErrNotExist) {
42          return nil, err
43    }
```

Figure 3.2: `optimism/op-chain-ops/cmd/celo-migrate/db.go#L41–L43`

```
112    _ = batch.Put(headerKey(number, hash), newHeader)
```

Figure 3.3: `optimism/op-chain-ops/cmd/celo-migrate/non-ancients.go#L112`

```
220    db, err := openDB(dbPath, true)
221    if err != nil {
222          return nil, fmt.Errorf("failed to open database: %w", err)
223    }
224    defer db.Close()
```

Figure 3.4: `optimism/op-chain-ops/cmd/celo-migrate/ancients.go#L220–L224`

- **There is no check to ensure that the total number of ancients in the new database is equal to that of the old database after the ancient migration is complete.** The premigration process will move all the ancients from the old database to the new one. The code should verify that the ancient migration occurred successfully before continuing the remaining portions of the migration process. This can be done by adding the following snippet to the end of the `migrateAncientsDb` function (figure 3.5).

```
if numAncientsNewAfter != numAncientsOld {
        return 0, 0, fmt.Errorf("failed to migrate all ancients from old to new db")
}
```

Figure 3.5: An invariant that the number of ancients in the old and new database must be equal after the migration

- **There is no check for the existence of a stray block in the ancient database before its removal.** Though it is unlikely that a stray block identified in the non-ancient database would be absent from the ancient database, it would be beneficial to validate that such a block exists in the ancient database before it is removed. This will ensure that the removal of the stray is justified. This validation can be performed in the `removeBlocks` function (figure 3.6).

```
  82    func removeBlocks(ldb ethdb.Database, numberHashes []*rawdb.NumberHash)
error {
  83          defer timer("removeBlocks")()
  84
  85          if len(numberHashes) == 0 {
  86                return nil
  87          }
  88
  89          batch := ldb.NewBatch()
  90
  91          for _, numberHash := range numberHashes {
  92                log.Debug("Removing block", "block", numberHash.Number)
  93                rawdb.DeleteBlockWithoutNumber(batch, numberHash.Hash,
numberHash.Number)
  94                rawdb.DeleteCanonicalHash(batch, numberHash.Number)
  95          }
  96          if err := batch.Write(); err != nil {
  97                log.Error("Failed to write batch", "error", err)
  98          }
  99
 100          return nil
 101    }
```

Figure 3.6: `optimism/op-chain-ops/cmd/celo-migrate/db.go#L82–L101`

Recommendations

Short term, address the instances described above in which errors are not handled or invariants are not checked. Doing so will improve the security posture of the data migration process.

4. Incorrect ordering of mempool transactions

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-4
Target: `op-geth/miner/ordering.go`	

Description

Due to an arithmetic miscalculation, user transactions may be incorrectly ordered.

As part of the alternative fee currency feature, `op-geth` needs to be able to order transactions, according to price and nonce, where each transaction may be paying for gas using different currencies. To do so, the system must do two things. First, it must identify the miner tip denominated in each transaction's requested fee currency. Second, it must convert all these tips back to their CELO equivalents so that the transactions can be ordered correctly.

```
// newTxWithMinerFee creates a wrapped transaction, calculating the effective
// miner gasTipCap if a base fee is provided.
// Returns error in case of a negative effective miner gasTipCap.
func newTxWithMinerFee(tx *txpool.LazyTransaction, from common.Address, baseFee *uint256.Int, rates common.ExchangeRates) (*txWithMinerFee, error) {
    tip := new(uint256.Int).Set(tx.GasTipCap)
    if baseFee != nil {
        baseFeeConverted := baseFee
        if tx.FeeCurrency != nil {
            baseFeeBig, err := exchange.ConvertCeloToCurrency(rates, tx.FeeCurrency, baseFee.ToBig())
            if err != nil {
                return nil, err
            }
            baseFeeConverted = uint256.MustFromBig(baseFeeBig)
        }

        if tx.GasFeeCap.Cmp(baseFeeConverted) < 0 {
            return nil, types.ErrGasFeeCapTooLow
        }
        tip = new(uint256.Int).Sub(tx.GasFeeCap, baseFee)
        if tip.Gt(tx.GasTipCap) {
            tip = tx.GasTipCap
        }
    }

    // Convert tip back into celo if the transaction is in a different currency
```

```
 62          if tx.FeeCurrency != nil {
 63                  tipBig, err := exchange.ConvertCurrencyToCelo(rates, 
tx.FeeCurrency, tip.ToBig())
 64                  if err != nil {
 65                          return nil, err
 66                  }
 67                  tip = uint256.MustFromBig(tipBig)
 68          }
 69
 70          return &txWithMinerFee{
 71                  tx:   tx,
 72                  from: from,
 73                  fees: tip,
 74          }, nil
 75  }
```

Figure 4.1: `op-geth/miner/ordering.go#L37–L75`

As shown in lines 44–50 in figure 4.1, the base fee, which is denominated in CELO, is converted into its fee currency equivalent (`baseFeeConverted`). The tip is then calculated, in line 55, by calculating the difference between the gas fee cap (denominated in the fee currency) from the *converted* base fee.

However, notice that the value used to find the difference is `baseFee`, not `baseFeeConverted`. Thus, the operation subtracts a fee currency amount from a CELO amount. This could cause the mempool to be ordered incorrectly since the tip amounts could be greater or less than expected, depending on the conversion rates.

Exploit Scenario

Alice submits a transaction using an alternative fee currency. She sets a high miner tip to ensure that her transaction will be in the next block. However, due to the incorrect calculation, her transaction does not get included.

Recommendations

Short term, update line 55 in figure 4.1 as follows:

```
 55                  tip = new(uint256.Int).Sub(tx.GasFeeCap, baseFeeConverted)
```

Figure 4.2: The tip is now calculated using the same denominated currency.

Long term, add tests of transaction ordering with alternative fee currencies. Currently, there are no tests for this specific feature.

5. Transaction replacements may fail when using CELO for gas

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-5
Target: `op-geth/core/txpool/legacypool/legacypool.go`	

Description
If a user attempts to resubmit a transaction while paying for gas using the native currency, CELO, their request may fail.

Users may resubmit a transaction with the same nonce if, for example, they want to increase the gas price for their transaction to improve the chance that it will be executed sooner. The `op-geth` mempool is responsible for validating this “replacement transaction” and ensuring that the user has enough funds to cover the new total cost (if the gas price increases, so does the total cost). Note that this validation must also account for changes to the underlying fee currency as well. As shown in figure 5.1, the `ValidateTransactionWithState` function is responsible for handling these fee currency changes and ensuring that the user has enough balance to pay for gas as well as any CELO they wish to transfer.

```
func ValidateTransactionWithState(tx *types.Transaction, signer types.Signer, opts *ValidationOptionsWithState) error {
[...]
	var (
		feeCurrencyBalance          = opts.ExistingBalance(from, tx.FeeCurrency())
		nativeBalance               = opts.ExistingBalance(from, &common.ZeroAddress)
		feeCurrencyCost, nativeCost = tx.Cost()
	)
	if feeCurrencyBalance == nil {
		return fmt.Errorf("feeCurrencyBalance is nil for FeeCurrency %x", tx.FeeCurrency())
	}
	if opts.L1CostFn != nil {
		if l1Cost := opts.L1CostFn(tx.RollupCostData()); l1Cost != nil { // add rollup cost
			nativeCost = nativeCost.Add(nativeCost, l1Cost)
		}
	}
	if feeCurrencyBalance.Cmp(feeCurrencyCost) < 0 {
		return fmt.Errorf("%w: balance %v, tx cost %v, overshot %v, fee currency: %v", core.ErrInsufficientFunds, feeCurrencyBalance, feeCurrencyCost,
new(big.Int).Sub(feeCurrencyCost, feeCurrencyBalance), tx.FeeCurrency().Hex())
 292            }
 293            if nativeBalance.Cmp(nativeCost) < 0 {
 294                    return fmt.Errorf("%w: balance %v, tx cost %v, overshot %v",
core.ErrInsufficientFunds, nativeBalance, nativeCost, new(big.Int).Sub(nativeCost,
nativeBalance))
 295            }
 296            // Ensure the transactor has enough funds to cover for replacements or
nonce
 297            // expansions without overdrafts
 298            feeCurrencySpent, nativeSpent := opts.ExistingExpenditure(from)
 299            if feeCurrencyPrev, nativePrev := opts.ExistingCost(from, tx.Nonce());
feeCurrencyPrev != nil {
 300                    // Costs from all transactions refer to the same currency,
 301                    // which is ensured by ExistingCost and ExistingExpenditure.
 302                    feeCurrencyBump := new(big.Int).Sub(feeCurrencyCost,
feeCurrencyPrev)
 303                    feeCurrencyNeed := new(big.Int).Add(feeCurrencySpent,
feeCurrencyBump)
 304                    nativeBump := new(big.Int).Sub(nativeCost, nativePrev)
 305                    nativeNeed := new(big.Int).Add(nativeSpent, nativeBump)
 306                    if feeCurrencyBalance.Cmp(feeCurrencyNeed) < 0 {
 307                            return fmt.Errorf("%w: balance %v, queued cost %v, tx
bumped %v, overshot %v, feeCurrency %v", core.ErrInsufficientFunds,
feeCurrencyBalance, feeCurrencySpent, feeCurrencyBump,
new(big.Int).Sub(feeCurrencyNeed, feeCurrencyBalance), tx.FeeCurrency())
 308                    }
 309                    if nativeBalance.Cmp(nativeNeed) < 0 {
 310                            return fmt.Errorf("%w: balance %v, queued cost %v, tx
bumped %v, overshot %v", core.ErrInsufficientFunds, nativeBalance, nativeSpent,
nativeBump, new(big.Int).Sub(nativeNeed, nativeBalance))
 311                    }
 312            } else {
 [...]
 327            }
 328            return nil
 329    }
```

Figure 5.1: `op-geth/core/txpool/validation.go#L258–L329`

(Note that in the following explanation "fee currency" could mean either a whitelisted ERC-20 token or the native CELO currency.) The function must do three things to validate a replacement transaction. First, it must validate that the user has enough funds to cover the replacement transaction (lines 277–295 in figure 5.1). Next, it must calculate the total cost of all that user's transactions in the mempool that use the same fee currency as the replacement transaction (line 298). Third, it must validate the additional balance the user must have to execute the replacement transaction in addition to all the other transactions they have in the mempool that use the same fee currency (lines 299–311).

The problem arises in the second step. The `ExistingExpenditure` function (figure 5.2) should return the total cost of the user's transactions in the mempool denominated in the fee currency and in CELO, separately.

```
ExistingExpenditure: func(addr common.Address) (*big.Int, *big.Int) {
        if list := pool.pending[addr]; list != nil {
                return list.TotalCostFor(tx.FeeCurrency()).ToBig(), list.TotalCostFor(nil).ToBig()
        }
        return new(big.Int), new(big.Int)
},
```

Figure 5.2: `op-geth/core/txpool/legacypool/legacypool.go#L690–L695`

However, as explained before, the fee currency could be the native CELO token itself (`tx.FeeCurrency()` would return `nil`). If CELO is used to pay for gas, the function should return zero for the cost of the fee currency and a nonzero amount for CELO. However, the function instead returns the same value for the fee currency amount and the amount of CELO. In the `ValidateTransactionWithState` function (figure 5.1), the `feeCurrencySpent` and the `nativeSpent` values are the same even though the `feeCurrencySpent` should be zero (line 298).

In the third step, the `feeCurrencyNeed` value will be incorrect. Instead of being zero, it will be the value of all the CELO required by the user's transactions in the mempool (`nativeSpent`). If the user does not own any alternative fee currencies, the check on line 306 in figure 5.1 will fail.

Exploit Scenario

Alice currently owns 1,000 CELO and no aFEE, an alternative fee currency. Additionally, Alice currently has a transaction in the mempool whose gas is paid for using CELO. The total cost of this transaction is 500 CELO. She wishes to execute the transaction faster, so she resubmits the same transaction with a higher gas price. The new total cost of the transaction is 505 CELO. The node, however, rejects her transaction since the node expects her to own 505 CELO and 500 aFEE.

Recommendations

Short term, update the `ExistingExpenditure` function to handle the case in which the `tx.FeeCurrency()` function returns `nil`. If the fee currency of the transaction is `nil`, the total fee currency cost should be zero.

Long term, improve the testing of the transaction replacement logic. As shown by this issue, by TOB-CELO-L2-6, and by some non-security-related issues described in appendix C, it would be beneficial to directly test this logic.

6. Transaction replacements may fail when using a fee currency for gas

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-6
Target: `op-geth/core/txpool/legacypool/legacypool.go`	

Description

If a user attempts to resubmit a transaction and use a fee currency for the gas, the request may fail.

Refer to TOB-CELO-L2-5 for a general explanation of transaction replacements and the various functions that are responsible for them.

The difference between this issue and TOB-CELO-L2-5 is that this bug is in the `ExistingCost` function (figure 6.1). As highlighted on line 706 in figure 6.1, the function attempts to compare two pointers. This comparison will always be evaluated to false if both fee currency pointers are not `nil`. Both will be `nil` only if CELO is being used to pay for gas both in the original transaction and in the replacement. Thus, when using a fee currency, regardless of whether the replacement transaction uses the same fee currency or a different one, `feeCurrencyCost` will evaluate to zero (line 708).

```
 696    ExistingCost: func(addr common.Address, nonce uint64) (*big.Int, *big.Int) {
 697            if list := pool.pending[addr]; list != nil {
 698                    feeCurrency := tx.FeeCurrency()
 699                    if tx := list.txs.Get(nonce); tx != nil {
 700                            feeCurrencyCost, nativeCost := tx.Cost()
 701                            if pool.l1CostFn != nil {
 702                                    if l1Cost := pool.l1CostFn(tx.RollupCostData());
l1Cost != nil { // add rollup cost
 703                                            nativeCost = nativeCost.Add(nativeCost,
l1Cost)
 704                                    }
 705                            }
 706                            if tx.FeeCurrency() != feeCurrency {
 707                                    // We are only interested in costs in the same
currency
 708                                    feeCurrencyCost = new(big.Int)
 709                            }
 710                            return feeCurrencyCost, nativeCost
 711                    }
 712            }
 713            return nil, nil
 714    },
```

Figure 6.1: `op-geth/core/txpool/legacypool/legacypool.go#L696–L714`

In the `ValidateTransactionWithState` function (figure 6.2), which handles fee currency changes, `feeCurrencyPrev` will always be zero. If the fee currency did not change between the two submissions, `feeCurrencyPrev` should be greater than zero. Thus, `feeCurrencyBump` will become larger than expected (subtracting zero from a nonzero value) which will cause `feeCurrencyNeed` to be larger than expected. Thus, the user will require a larger fee currency balance than required to resubmit the transaction. If the user does not have enough funds, the request to replace the transaction will fail.

```go
func ValidateTransactionWithState(tx *types.Transaction, signer types.Signer, opts *ValidationOptionsWithState) error {
[...]
	feeCurrencySpent, nativeSpent := opts.ExistingExpenditure(from)
	if feeCurrencyPrev, nativePrev := opts.ExistingCost(from, tx.Nonce()); feeCurrencyPrev != nil {
		// Costs from all transactions refer to the same currency,
		// which is ensured by ExistingCost and ExistingExpenditure.
		feeCurrencyBump := new(big.Int).Sub(feeCurrencyCost, feeCurrencyPrev)
		feeCurrencyNeed := new(big.Int).Add(feeCurrencySpent, feeCurrencyBump)
		nativeBump := new(big.Int).Sub(nativeCost, nativePrev)
		nativeNeed := new(big.Int).Add(nativeSpent, nativeBump)
		if feeCurrencyBalance.Cmp(feeCurrencyNeed) < 0 {
			return fmt.Errorf("%w: balance %v, queued cost %v, tx bumped %v, overshot %v, feeCurrency %v", core.ErrInsufficientFunds, feeCurrencyBalance, feeCurrencySpent, feeCurrencyBump, new(big.Int).Sub(feeCurrencyNeed, feeCurrencyBalance), tx.FeeCurrency())
		}
		if nativeBalance.Cmp(nativeNeed) < 0 {
			return fmt.Errorf("%w: balance %v, queued cost %v, tx bumped %v, overshot %v", core.ErrInsufficientFunds, nativeBalance, nativeSpent, nativeBump, new(big.Int).Sub(nativeNeed, nativeBalance))
		}
	}
[...]
}
```

Figure 6.2: `op-geth/core/txpool/validation.go#L258–L329`

Exploit Scenario

Alice currently owns 10 aFEE, an alternative fee currency. Additionally, Alice currently has a transaction in the mempool whose gas is paid for using aFEE. The total cost of this transaction is 9 aFEE. She wishes to execute the transaction faster, so she resubmits the same transaction with a higher gas price. The new total cost of the transaction is 9.5 aFEE. The node, however, rejects her transaction since the node expects her to own 18.5 aFEE instead of 9.5 aFEE.

Recommendations

Short term, update the `ExistingCost` function so that it does not compare the fee currency pointers. Instead, use a helper function such as `common.AreSameAddress` or the `Cmp` function of the `Address` type.

Long term, create a `FeeCurrency` type that implements a variety of helper functions (e.g., `Cmp` or `IsEqual`) to operate on and compare fee currencies. As shown in this issue and in two related but non-security-critical instances described in appendix C, there are many instances where fee currency pointers are compared. Standardizing the process will prevent these sorts of issues. An alternative option that may be less overengineered is to use type aliasing, as is done here, across all uses of fee currencies and use the existing functions of the `Address` type.

7. Gas estimation may fail

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-7
Target: `op-geth/internal/ethapi/api.go`, `op-geth/eth/gasestimator/gasestimator.go`	

Description

Gas estimation may fail if an alternative fee currency is used to pay for gas while attempting a value transfer.

The gas estimation logic attempts to identify the minimum gas limit for a given transaction to successfully execute. As part of the logic, the gas estimator, as a safety check, ensures that the user has sufficient funds to cover the call value. This check prevents users from erroneously sending transactions that will definitely fail. On lines 87–92 in figure 7.1, the `Estimate` function checks to make sure that the user has enough funds.

```
  50    // Estimate returns the lowest possible gas limit that allows the
transaction to
  51    // run successfully with the provided context options. It returns an error
if the
  52    // transaction would always revert, or if there are unexpected failures.
  53    func Estimate(ctx context.Context, call *core.Message, opts *Options, gasCap
uint64, exchangeRates common.ExchangeRates, balance *big.Int) (uint64, []byte,
error) {
[...]
  73          // Recap the highest gas limit with account's available balance.
  74          if feeCap.BitLen() != 0 {
  75                  available := balance
  76                  if call.FeeCurrency != nil {
[...]
  86                  } else {
  87                          if call.Value != nil {
  88                                  if call.Value.Cmp(available) >= 0 {
  89                                          return 0, nil,
core.ErrInsufficientFundsForTransfer
  90                                  }
  91                                  available.Sub(available, call.Value)
  92                          }
  93                  }
[...]
 210    }
```

Figure 7.1: `op-geth/eth/gasestimator/gasestimator.go#L50-L210`

The `available` value, which is equal to `balance`, is an input argument (line 53 in figure 7.1). This value, as shown on line 1333 in figure 7.2, may be one of two things. If the user is paying gas with CELO, it is the user's native balance. If it is an alternative fee currency, then it is the user's token balance for that currency.

```
1332    // Celo specific: get balance
1333    balance, err := b.GetFeeBalance(ctx, blockNrOrHash, call.From,
args.FeeCurrency)
1334    if err != nil {
1335        return 0, err
1336    }
1337
1338    // Run the gas estimation and wrap any revertals into a custom return
1339    estimate, revert, err := gasestimator.Estimate(ctx, call, opts, gasCap,
exchangeRates, balance)
1340    if err != nil {
1341        if len(revert) > 0 {
1342            return 0, newRevertError(revert)
1343        }
1344        return 0, err
1345    }
```

Figure 7.2: `op-geth/internal/ethapi/api.go#L1332–L1345`

However, the `Estimate` function does not account for both possibilities; it accounts only for the possibility that the balance provided is that of CELO. Thus, if the user uses an alternative fee currency, the balance check is incorrect, as it is comparing a CELO amount to an ERC-20 token balance. If the user's token balance is less than or equal to the call value, the gas estimation will fail.

Exploit Scenario

Alice attempts to transfer 1,000 CELO to Bob and wishes to pay for the gas using aFEE, an alternative fee currency. Alice owns less than 1,000 CELO equivalent to aFEE. Her wallet prevents her from sending the transaction since the gas estimation for the transaction notifies Alice that the transaction is guaranteed to fail, even though it should not.

Recommendations

Short term, update the `Estimate` function to accept both the CELO balance and the balance of the alternative fee currency being used. Next, separate the function's calculations related to value transfers from calculations related to gas. Which balance should be used for gas calculations is a question of whether a fee currency was used.

Long term, improve the testing of the gas estimation logic to test the two cases related to this issue (gas paid in native currency and in an alternative fee currency). In general, any changes made to vanilla `op-geth` or `geth` must be tested to ensure that the original behavior is preserved and the new behavior works as expected.

8. Storage change in FeeHandlerSeller affects UniswapFeeHandlerSeller

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CELO-L2-8
Target: `celo-monorepo/packages/protocol/contracts/common/UniswapFeeHandlerSeller.sol`	

Description
The `UniswapFeeHandlerSeller` contract inherits from `FeeHandlerSeller`, and it is an upgradeable contract in Celo. The `oracleAddresses` mapping has been added to the `FeeHandlerSeller` contract, changing the contract's storage. Since `UniswapFeeHandlerSeller` also defines storage variables, the layout for the old version is different from the layout for the new version.

The `check_storage.py` script output, shown in figure 8.1, shows that the storage of both the `MentoFeeHandlerSeller` and `UniswapFeeHandlerSeller` contracts has changed. However, since `MentoFeeHandlerSeller` does not define storage slots, the change does not affect it. In the `UniswapFeeHandlerSeller` contract, the new storage layout makes all accesses to the `oracleAddresses` variable actually access the old `routerAddresses` variable.

```
Contract MentoFeeHandlerSeller storage is different.
    Number of storage variables is different: 4 -> 5
        New variable added: oracleAddresses
Contract UniswapFeeHandlerSeller storage is different.
    Number of storage variables is different: 5 -> 6
        New variable added: routerAddresses
    Variable was renamed: routerAddresses -> oracleAddresses
```

Figure 8.1: Partial output of `check_storage.py`, showing the storage changes

Exploit Scenario
After the contracts are upgraded, the router addresses mapping is empty, breaking the sell functionality in `UniswapFeeHandlerSeller` contract. Additionally, all oracles in `FeeHandlerSeller` have incorrect values.

Recommendations
Short term, revert the storage changes that were added by the oracle setter functionality to `FeeHandlerSeller`. Consider using a different storage slot for the `oracleAddresses` variable.

Long term, add storage checks to the test suite or CI/CD pipeline to ensure that contract upgrades do not break functionality. Consider running `slither-check-upgradeability` to check for potential conflicts and using EIP-1967-like storage slots for upgradeable contracts.

9. Lack of validation of op-node command line data availability flag

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CELO-L2-9
Target: `optimism/op-batcher/batcher/service.go`	

Description

The op-node accepts an `OP_BATCHER_DATA_AVAILABILITY_TYPE` flag to configure the type of L1 transaction to send to push L2 data to L1. This flag can have one of the following three values:

1. `CallDataType`
2. `BlobType`
3. `AutoType`

The default value is `CallDataType`, if not provided as a command line flag. If alt-DA is enabled, the code to push the L2 transaction data to the alt-DA layer checks that the `txdata` fetched from the channel is not blob data:

```
func (l *BatchSubmitter) publishToAltDAAndL1(txdata txData, queue
*txmgr.Queue[txRef], receiptsCh chan txmgr.TxReceipt[txRef], daGroup
*errgroup.Group) {
	// sanity checks
	if nf := len(txdata.frames); nf != 1 {
		l.Log.Crit("Unexpected number of frames in calldata tx", "num_frames",
nf)
	}
	if txdata.asBlob {
		l.Log.Crit("Unexpected blob txdata with AltDA enabled")
	}
	...
```

Figure 9.1: A snippet of the `publishToAltDAAndL1` function of the batcher driver (`op-batcher/batcher/driver.go#L594–L601`)

However, the `initChannelConfig` function does not include validation checks to ensure that `OP_BATCHER_DATA_AVAILABILITY_TYPE` is set to `CallDataType` when `OP_BATCHER_ALTDA_ENABLED` is set to true (i.e., when alt-DA is enabled). The `BlobType` value for the `OP_BATCHER_DATA_AVAILABILITY_TYPE` flag sets the `ChannelConfig.UseBlobs` to true; if this happens when alt-DA is enabled, this will result in an early crash of the batcher. The `AutoType` value for the

`OP_BATCHER_DATA_AVAILABILITY_TYPE` flag may set the `ChanneConfig.UseBlobs` to true, depending on the L1 market conditions. In this case, the batcher may work for some time and crash at any later point. The error message will be `Unexpected blob txdata with AltDA enabled`, which may not help developers debug the issue.

Exploit Scenario
An operator runs the batcher with the command line flag value of `blobs` for the `OP_BATCHER_DATA_AVAILABILITY_TYPE` flag, and with alt-DA enabled and all other alt-DA flags set to a correct value. The batcher process crashes soon after, as it cannot process the blob type transaction data returned by the channel.

Recommendations
Short term, add a validation check in the `initChannelConfig` function to ensure that the value of the `OP_BATCHER_DATA_AVAILABILITY_TYPE` flag is set to `CallDataType` when alt-DA is enabled.

10. Missing onlyL1 modifier in Attestations contract's functions

Severity: **Informational**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-CELO-L2-10
Target: `contracts/identity/Attestations.sol`	

Description

According to the Celo documentation, the `Attestations` contract will be deprecated, but the withdraw functionality will still be enabled. Deprecated contracts are meant to stay on chain, but their functions should revert with the modifiers defined in the `isL2Check` contract.

None of the functions in the `Attestations` contract have the `onlyL1` modifier, meaning that the functions can be still called after the L2 migration. The output of the `find_modifiers_by_contract.py` script is shown in figure 10.1.

```
% python3 find_modifiers_by_contract.py Attestations onlyL1
Functions in contract Attestations (or inherited) with modifier onlyL1:
    constructor ()
    setRegistry (onlyOwner)
    owner () (doesn't modify state)
    isOwner () (doesn't modify state)
    renounceOwnership (onlyOwner)
    transferOwnership (onlyOwner)
    constructor ()
    initialize (initializer)
    revoke ()
    withdraw ()
    getUnselectedRequest () (doesn't modify state)
    getAttestationIssuers () (doesn't modify state)
    getAttestationStats () (doesn't modify state)
    batchGetAttestationStats () (doesn't modify state)
    getAttestationState () (doesn't modify state)
    getCompletableAttestations () (doesn't modify state)
    getAttestationRequestFee () (doesn't modify state)
    getMaxAttestations () (doesn't modify state)
    lookupAccountsForIdentifier () (doesn't modify state)
    requireNAttestationsRequested () (doesn't modify state)
    getVersionNumber () (doesn't modify state)
    setAttestationRequestFee (onlyOwner)
    setAttestationExpiryBlocks (onlyOwner)
    setSelectIssuersWaitBlocks (onlyOwner)
    setMaxAttestations (onlyOwner)
    validateAttestationCode () (doesn't modify state)
```

Figure 10.1: Output of `find_modifiers_by_contract.py` showing the lack of `onlyL1` modifiers

Recommendations

Short term, add the missing modifiers to the `Attestations` contract.

Long term, generate Slither rules that check for system integrity and regularly run them to ensure these kinds of issues are detected early. These checks can also be added to the CI/CD pipeline.

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
Configuration	The configuration of system components in accordance with best practices
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
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. Non-Security-Related Recommendations

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

- **Fix the following spelling errors.**

```
enum EpochProcessStatus
  NotStarted,
  Started,
  IndivudualGroupsProcessing
}
```

Figure C.1: *celo-monorepo/packages/protocol/contracts-0.8/common/EpochManager.sol#L40–L44*

```
require(newFraction.lt(FixidityLib.fixed1()), "New cargon fraction can't be greather than 1");
```

Figure C.2: *celo-monorepo/packages/protocol/contracts/common/FeeHandler.sol#L502*

- **Remove the reheap operation.** The call to reheap in figure C.3 (line 410) is superfluous because the `dropInvalidsAfterRemovalAndReheap` function (line 406) will perform the necessary reheap. Note that the code in between these two calls does not affect the internal state of the heaps.

```
406   invalids := l.dropInvalidsAfterRemovalAndReheap(removed)
407   // Reset total cost
408   l.subTotalCost(removed)
409   l.subTotalCost(invalids)
410   l.txs.reheap()
```

Figure C.3: *op-geth/core/txpool/legacypool/list.go#L406–L410*

- **Replace the direct comparisons of pointers shown in figures C.4 and C.5 with a helper function such as `common.AreSameAddress` or the `Cmp` function of the `Address` type.**

```
// Short circuit if the fee currency is the same.
if feeCurrency1 == feeCurrency2 {
      return val1.Cmp(val2), nil
}
```

Figure C.4: *op-geth/common/rates.go#L74–L77*

```
    if tx.FeeCurrency() != old.FeeCurrency() {
            thresholdFeeCapInCurrency = exchange.ConvertCurrency(rates,
thresholdFeeCap, old.FeeCurrency(), tx.FeeCurrency())
            thresholdTipInCurrency = exchange.ConvertCurrency(rates, thresholdTip,
old.FeeCurrency(), tx.FeeCurrency())
    }
```

Figure C.5: `op-geth/core/txpool/legacypool/list.go#L333–L336`

D. Slither Scripts

As explained in the Automated Testing section, Slither is a powerful tool for finding issues, code smells, misconfigurations, and other common mistakes in smart contracts. Given its public Python API, developers can easily extend Slither's behavior and customize it to their projects' needs.

As part of this audit, we developed some simple scripts to ensure that the code matches the provided specifications and that the new changes will not introduce unexpected errors in production code.

As a first step, we needed to be able to access all contracts in the project. By default, Slither reads the project's build environment configuration files and uses internal logic to call the build tool and compile all contracts. In the specific case of Celo, Slither could not compile all contracts because they were divided into two different folders that required different compiler versions and configuration options.

Checking for differences between two versions of the same contract can be challenging. Tools like `diff` exist for code, but keeping track of storage changes is complex and error-prone. The same goes for ensuring that functions that require modifiers do have them.

This appendix explains how to run the provided scripts, what can be done with them, and how to interpret their results. To execute them, install the `slither-analyzer` and `colorama` packages.

As a last note, the script output to the screen usually uses colors to convey more information with the least amount of text. Some information might be lost due to the formatting in the example runs shown in this appendix. It is recommended that the team run the scripts and review their output.

generate_zip.py

This script reads all contracts using Slither and generates zip files with serialized Slither objects that can be later read without recompiling or modifying the build configuration files.

The output files can then be used as input for the other scripts. Figure 1 shows how to execute it, and the expected output is two new files in the current directory: `contracts.zip` and `contracts-08.zip`. These files contain the serialized Slither objects for the contracts in the `contracts` and `contracts-0.8` folders, respectively.

```
% python3 generate_zip.py
```

Figure D.1: The command to run `generate_zip.py`

find_modifiers.py

This script analyzes the previously generated zip files and lists all contracts' functions with any specified modifiers. It can be used, for example, to show what functions are deprecated in L2 by looking for the `onlyL1` modifier.

```
% python3 find_modifiers.py [modifier1] [modifier2] ...
```

Figure D.2: The command to run `find_modifiers.py`

Figure D.3 shows output from an example run, which lists all functions that have the `nonReentrant` and `onlyL2` modifiers:

```
% python3 find_modifiers.py nonReentrant onlyL2
Functions with nonReentrant modifier:
    CompileExchange.buy (onlyWhenNotFrozen, updateBucketsIfNecessary, nonReentrant)
    CompileExchange.sell (onlyWhenNotFrozen, updateBucketsIfNecessary, nonReentrant)
    [ ... snipped ... ]
    Validators.setSlashingMultiplierResetPeriod (nonReentrant, onlyOwner)
    Validators.setDowntimeGracePeriod (nonReentrant, onlyOwner, onlyL1)
Functions with onlyL2 modifier:
    PrecompilesOverride.validatorAddressFromCurrentSet (onlyL2)
    Election.validatorAddressFromCurrentSet (onlyL2)
    [ ... snipped ... ]
    Validators.registerValidatorNoBls (nonReentrant, onlyL2)
    Validators.mintStableToEpochManager (onlyL2, nonReentrant,
onlyRegisteredContract)
```

Figure D.3: Output from an example run of `find_modifiers.py` to look for functions with the `nonReentrant` and `onlyL2` modifiers

find_modifiers_by_contract.py

This script is similar to the previous one, but it analyzes all functions in the specified contract and shows information about them. In particular, it shows which ones have the specified modifier and which functions do not alter the storage of the contract (view or pure functions).

This script can be used to, for example, detect whether all deprecated contracts have the correct `onlyL1` modifier or ensure that privileged contracts do not have any functions that do not have access control modifiers. A limitation of this script is that it checks only one contract and one modifier at a time, but it can be easily extended if required.

```
% python3 find_modifiers_by_contract.py contract modifier
```

Figure D.4: The command to run `find_modifiers_by_contract.py`

Figure D.5 shows output from an example run that analyzes all functions from the to-be-deprecated `Random` contract to ensure they have the `onlyL1` modifier. The output is red for functions that do not have the modifier and green for the ones that do.

```
% python3 find_modifiers_by_contract.py Random onlyL1
Functions in contract Random (or inherited) with modifier onlyL1:
    fractionMulExp (onlyL1) (doesn't modify state)
    getEpochSize (onlyL1) (doesn't modify state)
    getEpochNumberOfBlock (onlyL1) (doesn't modify state)
    getEpochNumber (onlyL1) (doesn't modify state)
    validatorSignerAddressFromCurrentSet (onlyL1) (doesn't modify state)
    validatorSignerAddressFromSet (onlyL1) (doesn't modify state)
    numberValidatorsInCurrentSet (onlyL1) (doesn't modify state)
    numberValidatorsInSet (onlyL1) (doesn't modify state)
    checkProofOfPossession (onlyL1) (doesn't modify state)
    getBlockNumberFromHeader (onlyL1) (doesn't modify state)
    hashHeader (onlyL1) (doesn't modify state)
    getParentSealBitmap (onlyL1) (doesn't modify state)
    getVerifiedSealBitmapFromHeader (onlyL1) (doesn't modify state)
    minQuorumSize (onlyL1) (doesn't modify state)
    minQuorumSizeInCurrentSet (onlyL1) (doesn't modify state)
    owner () (doesn't modify state)
    isOwner () (doesn't modify state)
    renounceOwnership (onlyOwner)
    transferOwnership (onlyOwner)
    constructor ()
    initialize (initializer)
    revealAndCommit (onlyL1, onlyVm)
    random (onlyL1) (doesn't modify state)
    commitments (onlyL1) (doesn't modify state)
    randomnessBlockRetentionWindow (onlyL1) (doesn't modify state)
    getBlockRandomness (onlyL1) (doesn't modify state)
    getVersionNumber () (doesn't modify state)
    setRandomnessBlockRetentionWindow (onlyL1, onlyOwner)
    computeCommitment () (doesn't modify state)
```

Figure D.5: Output from an example run of `find_modifiers_by_contract.py` to look for functions in the Random contract with the `onlyL1` modifier

check_storage.py

This script looks for differences in storage slots between two versions of the Celo contracts.

When executed, it looks for four zip files in the current directory: two for the old version and two for the new. The filenames are `contracts.zip`, `contracts-old.zip`, `contracts-08.zip`, and `contracts-08-old.zip`. To generate them, the `generate_zip.py` script must be run in the new and old versions' directories.

Since the project has a large number of contracts, the output can quickly become too verbose, making it difficult to find critical information. By default, only the contracts with different storage structures are shown, but several output options can be configured in the script using Boolean flags.

```
% python3 check_storage.py
```

Figure D.6: The command to run `check_storage.py`

Figure D.7 shows output from an example run. Contracts whose storage structures were changed between the two versions are marked in red, and the storage changes are marked in yellow. This example run shows the detection of issue TOB-CELO-L2-8.

```
% python3 check_storage.py
Contract CompileExchange was not found in the old version
Contract FeeCurrencyWhitelist storage is different.
    Variable was renamed: whitelist -> deprecated_whitelist
Contract FeeHandler storage is different.
    Number of storage variables is different: 10 -> 13
        New variable added: totalFractionOfOtherBeneficiaries
        New variable added: otherBeneficiaries
        New variable added: otherBeneficiariesAddresses
    Variable was renamed: lastLimitDay -> deprecated_lastLimitDay
    Variable was renamed: burnFraction -> ignoreRenaming_inverseCarbonFraction
    Variable was renamed: feeBeneficiary -> ignoreRenaming_carbonFeeBeneficiary
Could not get storage for old version of contract GoldToken. Consider reviewing
manually.
Contract MentoFeeHandlerSeller storage is different.
    Number of storage variables is different: 4 -> 5
        New variable added: oracleAddresses
Contract ProxyFactory was not found in the old version
Contract UniswapFeeHandlerSeller storage is different.
    Number of storage variables is different: 5 -> 6
        New variable added: routerAddresses
    Variable was renamed: routerAddresses -> oracleAddresses
Contract CeloUnreleasedTreasuryProxy was not found in the old version
[ ... snipped ... ]
```

Figure D.7: Output from an example run of `check_storage.py` to look for contracts whose storage structures were changed

trace_calls.py

This script generates and shows the full recursive call stack, types of calls, event emissions, and `require` statements in the execution path of a function. It simplifies the analysis of complex functions or functions that have a significant number of external calls, helps ensure that the correct functions are called, and indicates the possible revert reasons that can be expected.

The output of this script is meant to be interpreted by a human, and it cannot discover issues or bugs by itself. However, it can help developers understand how the system works and the dependencies between functions. Additionally, there is a limitation: the output does not show conditional execution paths, so the output might not match an actual execution flow.

```
% python3 trace_calls.py contract [function]
```

Figure D.8: The command to run `find_modifiers.py`

Figure D.9 shows partial output of an example run for the `sendValidatorPayment` function of the `EpochManager` contract. The script shows different colors for each call type, which are not shown in the figure.

```
% python3 trace_calls.py EpochManager sendValidatorPayment
EpochManager.sendValidatorPayment (Privileged call, modifiers:
onlySystemAlreadyInitialized)
  FixidityLib.newFixed (External call)
    FixidityLib.maxNewFixed (Internal call)
    require(x <= maxNewFixed(), "can't create fixidity number larger than
maxNewFixed()") (Solidity function)
  UsingRegistry.getValidators (Internal call)
    registry.getAddressForOrDie (External call)
      require(registry[identifierHash] != address(0), "identifier has no registry
entry") (Solidity function)
[ ... snipped ... ]
         SafeMath.sub (External call)
        require(b <= a, errorMessage) (Solidity function)
      SafeMath.add (External call)
        require(c >= a, "SafeMath: addition overflow") (Solidity function)
      Transfer (Event emission)
  require(stableToken.transfer(beneficiary, delegatedPayment), "transfer failed to
delegatee") (Solidity function)
  ValidatorEpochPaymentDistributed (Event emission)
```

Figure D.9: Output from an example run of `trace_calls.py` to show the execution path of the `EpochManager` contract's `sendValidatorPayment` function

E. Upgradability Checks with Slither

Trail of Bits developed the `slither-check-upgradability` tool to aid in the development of secure proxies; it performs safety checks relevant to both upgradeable and immutable `delegatecall` proxies. Consider using this tool during the development of the Celo smart contracts' codebase.

- Use `slither-check-upgradeability` to check for issues such as a corrupted storage layout between the previous and new implementations.

  ```
  slither-check-upgradeability . ContractV1 --new-contract-name ContractV2
  ```

  Figure E.1: An example of how to use `slither-check-upgradeability`

  For example, if a variable a is incorrectly added in the new implementation before other storage variables, `slither-check-upgradeability` will issue a warning like the one shown in figure E.2.

  ```
  ...
  INFO:Slither:
  Different variables between ContractV1 (contracts/versions/ContractV1.sol#9-54)
  and ContractV2 (contracts/versions/ContractV2.sol#11-133)
          ContractV1.__gap (contracts/versions/ContractV1.sol#15)
          ContractV2.a (contracts/versions/ContractV2.sol#20)
  Reference:
  https://github.com/crytic/slither/wiki/Upgradeability-Checks#incorrect-variables-w
  ith-the-v2
  ...
  ```

  Figure E.2: Example `slither-check-upgradeability` output

- Use `slither-read-storage` with the new implementation to manually check that the expected storage layout matches the previous implementation. Running the command shown in figure E.3 for the previous and new implementations will list the storage locations for variables in both versions. Make sure that variables in the previous implementation have the same slot number in the upgraded version.

  ```
  slither-read-storage . --contract ContractV1 --table
  ```

  Figure E.3: An example of how to use `slither-read-storage`

- If gaps are used in parent contracts, check the storage layout with `slither-read-storage` after adding new functionality and decrease the gap size accordingly. Make sure that storage is not overwritten or shifted in child contracts.

- Simulate the upgrade with a local mainnet fork, verify that all storage variables have the expected values, and run the tests on the new implementation.

F. 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.

From December 16 to December 18, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the cLabs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

The cLabs team provided two lists of pull requests (PRs), separating the blockchain fixes from the smart contract fixes. In total, seven PRs that fixed eight issues were provided. The team also provided explanations for the two unresolved issues (TOB-CELO-L2-8 and TOB-CELO-L2-10).

In summary, of the 10 issues described in this report, cLabs has resolved eight issues and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	User transactions may unexpectedly fail	Low	Resolved
2	Insufficient event generation	Informational	Resolved
3	Data migration process missing error handling and invariant checks	Informational	Resolved
4	Incorrect ordering of mempool transactions	Medium	Resolved
5	Transaction replacements may fail when using CELO for gas	Low	Resolved
6	Transaction replacements may fail when using a fee currency for gas	Low	Resolved
7	Gas estimation may fail	Low	Resolved
8	Storage change in FeeHandlerSeller affects UniswapFeeHandlerSeller	Medium	Unresolved

9	Lack of validation of op-node command line data availability flag	**Informational**	**Resolved**
10	Missing onlyL1 modifier in Attestations contract's functions	**Informational**	**Unresolved**

Detailed Fix Review Results

TOB-CELO-L2-1: User transactions may unexpectedly fail
Resolved in PR #266. The `canPayFee` function was modified to consider the amounts in CELO and the alternative fee currency separately. A new test script was added, and the existing send transaction test file was modified to comply with the changes.

TOB-CELO-L2-2: Insufficient event generation
Resolved in PR #11285. Four new events were added to the `EpochManager` and `FeeCurrencyDirectory` contracts to be emitted when the `setToProcessGroups`, `processGroups`, `setCurrencyConfig`, and `removeCurrencies` functions are executed. Test files for these contracts were modified to include new test cases that check for event emissions.

TOB-CELO-L2-3: Data migration process missing error handling and invariant checks
Resolved in PR #283. Several changes were implemented in the `celo-migrate` scripts to add the missing error checks. The invariant check to ensure that the number of ancients is the same after and before was also added.

TOB-CELO-L2-4: Incorrect ordering of mempool transactions
Resolved in PR #275. The tip is now calculated with the converted base fee, and the transactions are correctly ordered in the mempool. A new test case was added to check for transaction ordering.

TOB-CELO-L2-5: Transaction replacements may fail when using CELO for gas
Resolved as part of the TOB-CELO-L2-6 fix. The client provided the following context for this finding's fix status:

> *This was actually not a bug. It was working properly, but the code was a little misleading. So we refactored to something similar of what the audit fix was proposing.*

TOB-CELO-L2-6: Transaction replacements may fail when using a fee currency for gas
Resolved in PR #283. The pointer comparisons were replaced by `common.AreSameAddress`, and the `ValidateTransactionWithState` function now checks for a `nil` fee currency and sets the `feeCurrencyBalance` variable as expected. As mentioned in the previous issue's fix description, a refactor was also performed in the `ExistingExpenditure` function to return the correct values when the fee currency is `nil`.

A new test case was added to check that the replacement transaction is accepted if the account only owns CELO.

TOB-CELO-L2-7: Gas estimation may fail
Resolved in PR #282. The `Estimate` function was modified to account for gas denominated in CELO and fee currencies independently. Additional checks were added to ensure that the available CELO is enough to pay for the call value.

TOB-CELO-L2-8: Storage change in FeeHandlerSeller affects UniswapFeeHandlerSeller
Unresolved. The client provided the following context for this finding's fix status:

> *Determined to be a non-issue, because the contracts are being deprecated in favor of using `FeeCurrencyDirectory` contract.*

TOB-CELO-L2-9: Lack of validation of op-node command line data availability flag
Resolved in PR #274. A check was added to `initChannelConfig` to ensure that the flag's value is set to `CallDataType` when alt-DA is enabled.

TOB-CELO-L2-10: Missing onlyL1 modifier in Attestations contract's functions
Unresolved. The client provided the following context for this finding's fix status:

> *Determined to be non-issue, as the `Attestations` contract has already been deprecated on L1, in favor of `FederatedAttestations` contract.*

G. 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.