Executive Summary

Project Dashboard

Engagement Goals

Coverage

Recommendations Summary

   Short Term
   Long Term

Findings Summary

  1. Spam attack through out-of-bound access on fractionMulExp and transfer precompiled contracts
  2. Lack of contract existence check on delegatecall will lead to unexpected behavior
  3. Lack of contract existence check in MultiSigWallet will lead to unexpected behavior
  4. Lack of contract existence check in Governance will lead to unexpected behavior
  5. Race condition in the ERC20 approve function may lead to token theft
  6. Quick buy and sell allows vote manipulation
  7. Linked list compromise through incorrect insertion
  8. Missing validation of message signatures
  9. Gas cost of precompiled transfer contract is unclear
 10. Unsafe Solidity type conversion
 11. Oracle exchange rates can be manipulated by calling removeExpiredReports
 12. Compromise of a single oracle allows limited control of the price
 13. Arithmetic rounding leads to non-constant product
 14. Lack of validation in update allows for SortedFractionMedianList compromise
 15. Exchange fallback function will lead to trapping ether
 16. Incorrect access control allows anyone to burn tokens’ reserve
 17. Missing validation in contract initializations
 18. Celo identity attestation vulnerable to SIM-swapping attacks
 19. Oracle’s median can be compromised with zero value
 20. Exchange susceptible to front-running
 21. On-chain mitigation does not prevent reserve from becoming under-collateralized
 22. Attestation validator selection takes place in a single transaction
 23. MultiSig contract is missing address validation
 24. Missing validation allows for Istanbul message forgery
 25. Missing validation allows for Istanbul message replay
 26. Future messages can crash a node through out-of-memory condition
 27. A malicious or unreachable proposer can trap the system
28. Integer overflow allows for arbitrary priorities in stored message
29. Liveness depends on local clock synchronization
30. Use of static constants for gas is error-prone
31. Missing error check can lead to incorrect randomness commitment
32. Unhandled errors can lead to invalid node state
33. Proposed blocks can be out of sequence
34. Integer overflow allows for early revocation of payments
35. Attestation validator can add their address to any identity

A. Vulnerability Classifications

B. Code Quality Recommendations
   Stability
   Governance
   consensus/istanbul/core/backlog.go

C. Slither delegatecall upgradeable proxy checks

D. Property testing of LinkedList

E. Detecting correct inheritance initialization with Slither
Executive Summary

From August 7 through September 6, 2019, Celo engaged Trail of Bits to review the security of the Celo blockchain. Trail of Bits conducted this assessment over the course of eight person-weeks, with two engineers working from the celo-blockchain (8360bec4) and celo-monorepo (4f257e39) GitHub repositories.

The Celo blockchain aims to provide a stable coin to its users. It is composed of a fork of geth, for which the main modification is the implementation of the Istanbul Byzantine Fault Tolerance (IBFT) consensus protocol. Several smart contracts provide on-chain validator selection and decentralized governance.

Trail of Bits used the first week to familiarize ourselves with the Celo codebase. We started our review of the Governance contracts during the second week and looked for common Solidity flaws in the other contracts. We finished our review of the Governance contracts during the third week and started our review of the Stability contracts. During the fourth week, Trail of Bits focused on the Stability contracts, as well as the beginning of our IBFT consensus protocol review. Finally, we spent the fifth week finishing our review of the IBFT consensus protocol.

During our assessment of the Celo smart contracts, we also developed custom Slither scripts to ensure the correct review of the upgradability mechanism (Appendix C) and the inheritance initialization (Appendix E). Additionally, we used Echidna to check properties on the linked list implementation (Appendix D). Appendix B contains code quality recommendations.

Trail of Bits identified 35 issues, ranging in severity from undetermined to high, including:

- The ability to crash remote nodes through an out-of-bound access
- The ability to crash remote nodes through an out-of-memory condition
- Multiple price manipulations, including:
  - Missing access control for expired report removal functionality
  - A malicious oracle having limited control over the price, and
  - A malicious oracle being able to compromise the sorted list due to a lack of input validation
- The ability to predict the outcome of random attestation validator selection
- A missing check of IBFT messages, which allows a malicious validator to send forged messages on behalf of another validator
- A proposer being able to trap the system by being silent
- The SMS-based identity mechanism used by Celo being vulnerable to SIM-swapping attacks
While the smart contracts codebase comprises several high-risk and complex components, Celo developed its smart contracts with a clear understanding of common Solidity flaws. The Celo team consciously avoided several issues frequently associated with contracts of this complexity. However, Celo based the stability algorithm on a new mechanism (the constant-product decentralized one-to-one mechanism), which lacks real-world evaluation. Moreover, the lack of high-level contract documentation highlighting how contracts are composed and interact with each other made the review more difficult.

The Go codebase represents a significant work in progress. Not all functionality was implemented, and some parts of the code were redundant, making its review more difficult. A lack of adequate data validation resulted in multiple findings.

Trail of Bits recommends that Celo fix the identified smart contracts issues and carefully evaluate the economic problems that can arise from the stability mechanism. We also recommend fixing all of the identified Go issues, adding `gosec` to Celo’s continuous integration pipeline, and carefully reviewing the data validation of the system.

Once the IBFT consensus implementation has been finalized, Trail of Bits recommends performing a follow-up assessment to review the updated codebase.
# Project Dashboard

## Application Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Celo-blockchain, celo-monorepo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>8360bec4, 4f257e39</td>
</tr>
<tr>
<td>Type</td>
<td>Go, Solidity</td>
</tr>
<tr>
<td>Platforms</td>
<td>Ethereum</td>
</tr>
</tbody>
</table>

## Engagement Summary

<table>
<thead>
<tr>
<th>Dates</th>
<th>August 7 - September 6, 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Whitebox</td>
</tr>
<tr>
<td>Consultants Engaged</td>
<td>2</td>
</tr>
<tr>
<td>Level of Effort</td>
<td>8 person-weeks</td>
</tr>
</tbody>
</table>

## Vulnerability Summary

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th><a href="#">20 High-Severity Issues</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total High-Severety Issues</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Medium-Severety Issues</td>
<td>4</td>
<td><a href="#">4 Medium-Severity Issues</a></td>
</tr>
<tr>
<td>Total Low-Severety Issues</td>
<td>3</td>
<td><a href="#">3 Low-Severity Issues</a></td>
</tr>
<tr>
<td>Total Informational-Severety Issues</td>
<td>6</td>
<td><a href="#">6 Informational-Severity Issues</a></td>
</tr>
<tr>
<td>Total Undetermined-Severety Issues</td>
<td>2</td>
<td><a href="#">2 Undetermined-Severity Issues</a></td>
</tr>
<tr>
<td>Total</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

## Category Breakdown

<table>
<thead>
<tr>
<th>Category</th>
<th>Issues</th>
<th><a href="#">Issues</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Control</td>
<td>2</td>
<td><a href="#">2</a></td>
</tr>
<tr>
<td>Authentication</td>
<td>1</td>
<td><a href="#">1</a></td>
</tr>
<tr>
<td>Configuration</td>
<td>1</td>
<td><a href="#">1</a></td>
</tr>
<tr>
<td>Cryptography</td>
<td>1</td>
<td><a href="#">1</a></td>
</tr>
<tr>
<td>Data Validation</td>
<td>23</td>
<td><a href="#">23</a></td>
</tr>
<tr>
<td>Denial of Service</td>
<td>2</td>
<td><a href="#">2</a></td>
</tr>
<tr>
<td>Patching</td>
<td>1</td>
<td><a href="#">1</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
<td>---</td>
</tr>
<tr>
<td>Timing</td>
<td>3</td>
<td>■ ■ ■</td>
</tr>
<tr>
<td>Undefined Behavior</td>
<td>1</td>
<td>■</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>35</strong></td>
<td></td>
</tr>
</tbody>
</table>
Engagement Goals

Celo and Trail of Bits scoped the engagement to provide a security assessment of the Celo Blockchain. The Celo team identified Governance, Stability and IBFT consensus as the highest priorities for review.

Specifically, we sought to answer the following questions:

- Is governance voting correctly implemented?
- Can the Reserve contract be drained?
- How does the system react in case of an oracle error or compromise?
- Does stability work as intended?
- Is the IBFT consensus protocol working as intended?

Coverage

Governance contracts. Trail of Bits reviewed the validators voting mechanism for correctness and searched for a way to vote with more weight than purchased, or prevent other users from voting. We reviewed the D'Hondt implementation, while considering potential out-of-gas issues. We also focused on the correct state transition of governance voting. Finally, we checked whether the notice period of the bonded deposit could be bypassed.

Stability contracts. Celo based its stability algorithm on the constant-product-market-maker model. We reviewed the model, taking into account its arithmetic imprecision, handling of oracle compromise, and the stability of the buckets. We also reviewed the sorted oracle list implementation to ensure the correct order of its elements.

Common contracts. We reviewed the implementation of the gold token for correctness. As several contracts rely on the delegatecall proxy pattern, we reviewed them for the most common flaws.

IBFT consensus. We focused on the message event protocol, including event types such as PREPREPARE and COMMIT. We looked for ways to compromise a node through crafted messages, prevent a quorum from occurring, and trigger invalid state transitions.

The Celo team was aware of the following issues prior to the beginning of the audit:

- The issues presented in Correctness Analysis of IBFT.
- The out-of-memory issue triggered by round change messages.
• The lack of consistency in the process to re-join the consensus after a crash. (As a result, we did not evaluate how nodes re-synchronize after leaving the network.)
• Some changes of a significant IBFT liveness pull request was not merged (https://github.com/celo-org/celo-blockchain/pull/366)
• Stability fee updates were not retroactive.
Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

- **Check the input length in fractionMulExp and transfer (core/vm/contracts.go).** Incorrectly sized input can potentially crash the node.

- **Check for contract existence prior to a low-level call with non-empty data or delegatecall with the EXTCODESIZE opcode in Proxy.sol, MultiSig.sol and Governance.sol.** The lack of an existence check can lead to unexpected behavior if no contract exists.

- **Document that suicide or selfdestruct can lead to unexpected behavior. Prevent future upgrades from introducing these functions.** A self-destructed contract might lead to an incorrect state of the proxy.

- **Implement increaseAllowance and decreaseAllowance from OpenZeppelin in GoldToken.sol, StableToken.sol.** These functions offer a mitigation to the ERC20 approval race condition.

- **Consider implementing a weighted stake (with the weight decreasing over time) to incentivize users to vote earlier, or requiring a minimal staking period.** Users have no incentive to vote early, which makes the voting system vulnerable to quick buy or sell orders.

- **Be sure that users are aware of the risk of front-running, and properly document the arbitrage opportunity.** This issue is inherent in the nature of on-chain exchange and is present in several similar platforms.

- **Prevent key from being equal to previousKey and nextKey in LinkedList.insert (LinkedList.sol).** Without these checks, it is possible to compromise the linked list.

- **Use OpenZeppelin’s ECDSA helper library to validate signatures and consistently validate that the returned value is not equal to an address of zero.** Direct calls to ecrecover do not contain the checks necessary for its correct usage.

- **Update the requiredGas function in /core/vm/contracts.go#L459-L461 to return a value of zero.** The current gas price is undefined and might lead to an incorrect assumption.

© 2019 Trail of Bits
Ensure that all unsafe type conversions (e.g., going from a larger integer to a smaller integer) are validated in BondedDeposits.sol. The codebase relies on numerous unsafe type conversions, which might introduce errors in future code updates.

Allow only privileged addresses (e.g., an owner address) to call removeExpiredReports (SortedOracles.sol). Calling the function affects the price, which creates an undocumented arbitrage potential.

Implement on-chain monitoring of the exchange and oracle contracts to report any suspicious activity. Each oracle has limited control over the price, which can allow them to manipulate it to their benefit. On-chain monitoring will help detect malicious behavior.

Add validation present in insert to update (SortedFractionMedianList.sol#L99-L101). Missing validation allows an oracle to compromise the list.

Remove payable from the Exchange’s fallback function. Otherwise, Ether sent to the contract will be trapped.

Remove the burn function or add the onlyOwner modifier in Reserve.sol. Consider removing the private mintToken function. The burn function allows any caller to burn the balance to an arbitrary token.

Apply all the missing validations listed in TOB-CELO-17. Missing validation can result in incorrectly deployed contracts.

Consider more secure forms of electronic identity, such as email addresses or domain names. SMS is an insecure protocol that is vulnerable to SIM-swapping attacks.

Document the risk of SIM-swapping attacks. Users must be aware of the risk they incur when using SMS for attestation.

Prevent the contract from accepting either a numerator and denominator of 0 in SortedOracles.sol. An oracle can compromise the list with these values.

Consider lowering the amount of gold tokens from which the tax fee is triggered. In its current form, the system is at risk of being under-collateralized.

Monitor the price to ensure the reserve is always over-collateralized from a given threshold. In its current form, the system is at risk of being under-collateralized.
- Remove the validator selection from request (Attestations.sol), and perform the selection in a separate transaction. The current validator selection process is deterministic, and users can know who the validator will be before sending a request.

- Add the `notNull(newOwner)` modifier to `replaceOwner` (MultiSig.sol). The lack of the modifier allows setting the zero address as an owner.

- Ensure that the message address is the message's sender, and the signer of the message is also the message's sender (core/handler.go). Missing validation of a message's sender address allows anyone to replay broadcasted messages, and allows a validator to send a message on behalf of other validators.

- Limit the number of requests per source stored in the backlog (consensus/istanbul/handler.go). The unbounded buffer can lead the node to crash, due to an out-of-memory error.

- Change the proposer if he never send a PREPREPARE message (IBFT state transition). If the proposer has not changed, he is able to trap the system indefinitely.

- Check for arithmetic overflows in `backlog.Priority` (consensus/istanbul/core/backlog.go). The overflow allows the user to set arbitrary priority.

- Document that users must keep an accurate local time when using Celo. An insecure source of local time can allow an attacker to isolate a target from the network.

- Create a config file to contain the gas limit, or allow unlimited gas from system calls. The statically hardcoded gas limits are error-prone and might cause the system to become trapped.

- Check the error value returned by `computeCommitment` in random.go. The missing error check might lead to unexpected behavior.

- Consider allowing only buy and sell orders that do not lead to loss of precision. StableToken uses the constant-product-market-maker model. The model relies on the product of the StableToken and GoldToken buckets being constant. Due to a rounding imprecision, the product does not stay constant.

- Perform consistent error handling. If a failed operation would result in an invalid node state, divert program control flow and return early. The lack of consistent error handling is error-prone and makes code review more difficult.
Perform consistent validation of all incoming messages. Do not assume client-side validation will prevent an attacker from crafting malicious messages.

Ensure that an overflow will not occur between timestamp and expirySeconds in Escrow. An overflow can allow an attacker to revoke its payments early.

Add a minimal number (>1) of required validators for attestation. If one validator is malicious or compromised, they can add their SMS number to any identity.

Use Slither printers to review that each contract has only required functions. This will prevent unintended functionality from being exposed.
Long Term

- **Avoid low-level Solidity calls.** Low-level calls are error-prone and do not have the same in-built protections that high-level calls.

- **Carefully review the Solidity documentation**, especially the entire Warnings section. Solidity contains several pitfalls that must be known when writing smart contracts.

- **Carefully review the pitfalls of using the delegatecall proxy pattern.** Delegatecall-based upgrades require a deep understanding of EVM and are highly error-prone.

- **Ensure that users are aware of the ERC20 increaseAllowance and decreaseAllowance functions, and encourage them to make use of them when appropriate.** These functions prevent the ERC20 approval race condition.

- **Add tests to the codebase that validate the proper handling of invalid signatures.** Signatures are an important part of the codebase and must be properly tested.

- **Write a specification of each new precompiled contract, and add tests to the codebase to check that they are followed.** The precompiled contracts must have a specification to ensure they are tested correctly.

- **Eliminate unnecessary type conversions from the codebase.** The codebase relies on numerous unsafe type conversions, which can introduce errors in future code updates.

- **Consider requiring that validators automatically call removeExpiredReports as the first transaction of every block.** Calling this function affects the price, which creates an undocumented arbitrage opportunity.

- **Document all expected arbitrage opportunities.** Users will benefit from a discussion of known arbitrage opportunities offered by the system.

- **Assume that an attacker may be able to compromise some of the oracles when designing the protocol.** Price computation should be robust in case of partial compromise.

- **Use Echidna and Manticore to:**
  - Ensure that a multisig’s owner can never be the zero address. Zero as owner will lead to unexpected behavior.
  - Check the sorted lists implementation. Both tools will help find any list compromises.
Use Manticore to ensure the correctness of the market model computation. Price computation requires extra care. Symbolic execution will help identify any arithmetic issues.

Use crytic.io or Slither to detect the most common Solidity flaws. Trail of Bits identified several of the presented findings using Slither.

Introduce additional software tests to check that initialization validation is adequately enforced across all contracts. The lack of validation can result in incorrectly deployed contracts.

Investigate solutions to reduce dependency on the Governance holders for system collateralization. Currently, the system relies on the Governance holders to send new collateral if the system is close to losing its collateralization. This is a manual solution and requires users to trust the Governance holders.

Carefully evaluate the evolution of the gas bound of the contracts. The contracts rely on gas-expensive computation operations that might lead to out-of-gas issues in the future.

Validate the messages after their parsing. Messages are controlled by external users and must be thoroughly validated. For example, the round field must always be less than 4.

Add on-chain monitoring to check the attestation requests, and report any suspicious activities.

Thoroughly validate each field that is decoded from a user-controlled source in the IBFT consensus implementation. Multiple issues were the result of missing data validation.

Use gofuzz to check the robustness of the nodes and the precompiled contracts. Fuzzing will help locate potential node crashes.

Long term, use Echidna and Manticore to ensure that invalid prices cannot be added to the list. Moreover, consider oracles as untrusted users, and validate and monitor their inputs.

Review IBFT state transitions to ensure that no state can lead to an infinite loop. IBFT is a new protocol and requires extra care in its state transitions.

Investigate moving away from consensus protocols that require a global clock. The local clock can be easily compromised.
- **Add additional documentation and testing for node error handling and crash recovery strategies.** Error handling is a frequent source of error in Go and should be carefully designed and implemented.

- **Use SafeMath for all arithmetic operations.** SafeMath will prevent all potential integer overflow.

- **Add tests to the codebase that validate proper error handling in random.go’s GenerateNewRandomnessAndCommitment function.** Ignoring errors can result in unexpected behavior.

- **Add tests to the codebase that check for proper data validation preprepare.go’s handlePreprepare function.** Consistent validation is important in ensuring that nodes conform to the protocol that Celo intends to implement.
# Findings Summary

<table>
<thead>
<tr>
<th>#</th>
<th>Title</th>
<th>Type</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Spam attack through out-of-bound access on fractionMulExp and transfer precompiled contracts</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>2</td>
<td>Lack of contract existence check on delegatecall will lead to unexpected behavior</td>
<td>Patching</td>
<td>High</td>
</tr>
<tr>
<td>3</td>
<td>Lack of contract existence check in MultiSigWallet will lead to unexpected behavior</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>4</td>
<td>Lack of contract existence check in Governance will lead to unexpected behavior</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>5</td>
<td>Race condition in the ERC20 approve function may lead to token theft</td>
<td>Timing</td>
<td>High</td>
</tr>
<tr>
<td>6</td>
<td>Quick buy and sell allows vote manipulation</td>
<td>Timing</td>
<td>High</td>
</tr>
<tr>
<td>7</td>
<td>Linked list compromise through incorrect insertion</td>
<td>Data Validation</td>
<td>Low</td>
</tr>
<tr>
<td>8</td>
<td>Missing validation of message signatures</td>
<td>Cryptography</td>
<td>Informational</td>
</tr>
<tr>
<td>9</td>
<td>Gas cost of precompiled transfer contract is unclear</td>
<td>Undefined Behavior</td>
<td>Informational</td>
</tr>
<tr>
<td>10</td>
<td>Unsafe Solidity type conversion</td>
<td>Data Validation</td>
<td>Informational</td>
</tr>
<tr>
<td>11</td>
<td>Oracle exchange rates can be manipulated by calling removeExpiredReports</td>
<td>Access Control</td>
<td>High</td>
</tr>
<tr>
<td>12</td>
<td>Compromise of a single oracle allows limited control of the price</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>Category</td>
<td>Severity</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td>13</td>
<td>Arithmetic rounding leads to non-constant product</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>14</td>
<td>Lack of validation in update allows for SortedFractionMedianList compromise</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>15</td>
<td>Exchange fallback function will lead to trapping ether</td>
<td>Data Validation</td>
<td>Medium</td>
</tr>
<tr>
<td>16</td>
<td>Incorrect access control allows anyone to burn tokens’ Reserve</td>
<td>Access Control</td>
<td>Informational</td>
</tr>
<tr>
<td>17</td>
<td>Missing validation in contract initializations</td>
<td>Data Validation</td>
<td>Low</td>
</tr>
<tr>
<td>18</td>
<td>Celo identity attestation vulnerable to SIM-swapping attacks</td>
<td>Authentication</td>
<td>High</td>
</tr>
<tr>
<td>19</td>
<td>Oracle’s median can be compromised with zero value</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>20</td>
<td>Exchange susceptible to front-running</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>21</td>
<td>On-chain mitigation does not prevent reserve from becoming under-collateralized</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>22</td>
<td>Attestation validator selection takes place in a single transaction</td>
<td>Timing</td>
<td>High</td>
</tr>
<tr>
<td>23</td>
<td>MultiSig contract is missing address validation</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>24</td>
<td>Missing validation allows for Istanbul message forgery</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>25</td>
<td>Missing validation allows for Istanbul message replay</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>26</td>
<td>Future messages can crash a node through out-of-memory condition</td>
<td>Data Validation</td>
<td>High</td>
</tr>
<tr>
<td>27</td>
<td>A malicious or unreachable proposer can trap the system</td>
<td>Denial of Service</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Issue Description</td>
<td>Category</td>
<td>Severity</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>28</td>
<td><strong>Integer overflow allows for arbitrary priorities in stored message</strong></td>
<td>Data Validation</td>
<td>Informational</td>
</tr>
<tr>
<td>29</td>
<td><strong>Liveness depends on local clock synchronization</strong></td>
<td>Denial of Service</td>
<td>Low</td>
</tr>
<tr>
<td>30</td>
<td><strong>Use of static constants for gas is error-prone</strong></td>
<td>Configuration</td>
<td>Informational</td>
</tr>
<tr>
<td>31</td>
<td><strong>Missing error check can lead to incorrect randomness commitment</strong></td>
<td>Data Validation</td>
<td>Medium</td>
</tr>
<tr>
<td>32</td>
<td><strong>Unhandled errors can lead to invalid node state</strong></td>
<td>Data Validation</td>
<td>Undetermined</td>
</tr>
<tr>
<td>33</td>
<td><strong>Proposed blocks can be out of sequence</strong></td>
<td>Data Validation</td>
<td>Undetermined</td>
</tr>
<tr>
<td>34</td>
<td><strong>Integer overflow allows for early revocation of payments</strong></td>
<td>Data validation</td>
<td>Medium</td>
</tr>
<tr>
<td>35</td>
<td><strong>Attestation validator can add their address to any identity</strong></td>
<td>Data validation</td>
<td>Medium</td>
</tr>
</tbody>
</table>
1. Spam attack through out-of-bound access on fractionMulExp and transfer precompiled contracts

Severity: High
Type: Data Validation
Finding ID: TOB-CELO-01
Target: core/vm/contracts.go

Description
Out-of-bounds access in the fractionMulExp precompiled contract allows an attacker to consume nodes’ resources without paying fees.

fractionMulExp accesses the input array without checking its length (ex: input[0:32]):

```go
func (c *fractionMulExp) Run(input []byte, caller common.Address, evm *EVM, gas uint64) ([]byte, uint64, error) {
    gas, err := debitRequiredGas(c, input, gas)
    if err != nil {
        return nil, gas, err
    }

    parseErrorStr := "Error parsing input: unable to parse %s value from %s"

    aNumerator, parsed := math.ParseBig256(hexutil.Encode(input[0:32]))
}

aNumerator, parsed := math.ParseBig256(hexutil.Encode(input[32:64]))
```

Figure 1: contracts.go#L497-L507

If the contract is called without enough data in the transaction, the function will trigger a panic. As a result, the node will not accept the transaction, and no fee will be removed from the user.

If the transaction executes costly operations prior to the call to the precompiled contract, these operations will not have to be paid. Figure 2 shows an example.

```solidity
contract Exploit{
    function attack() public{
        // Execute costly operations
        // ....

        // Call to fractionMulExp, triggering the panic and canceling the costly operations
        dst = address(0xff-3);
        dst.call("");
    }
}
```

Figure 2: Exploit.sol
An attacker can take advantage of the situation to spam the network for free.

A similar issue is present in the `transfer` precompiled contract, but has a lower impact as the contract is only callable by the gold token contract.

**Exploit Scenario**
Bob is a node in the Celo network. Eve deploys a contract triggering the panic and sends millions of transactions to Bob. As a result, Bob is unable to process the transactions and becomes unreachable.

**Recommendation**
Check the input length in `fractionMulExp` and `transfer`.

Consider using a tool such as [gofuzz](https://github.com/trailofbits/gofuzz) on the Celo-specific, precompiled contracts to ensure their correct behavior.
2. Lack of contract existence check on delegatecall will lead to unexpected behavior

Severity: High
Type: Patching
Finding ID: TOB-CELO-02
Target: Proxy.sol

Description
Proxy uses the delegatecall proxy pattern. If the implementation is incorrectly set or self-destructed, the proxy can exhibit unexpected behavior.

A delegatecall to a self-destructed contract will return success, as part of the EVM specification. The solidity documentation warns:

> The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

Proxy uses delegatecall without checking for code account existence:

```solidity
function _setAndInitializeImplementation(
    address implementation,
    bytes calldata callbackData
) external payable onlyOwner
{
    _setImplementation(implementation);
    bool success;
    bytes memory returnValue;
    (success, returnValue) = implementation.delegatecall(callbackData);
    require(success, "initialization callback failed");
}
```

Figure 1: Proxy.sol#L83-L96

```assembly
assembly {
    let implementationAddress := sload(implementationPosition)
    let newCallDataPosition := mload(0x40)
    mstore(0x40, add(newCallDataPosition, calldatasize))

    calldatacopy(newCallDataPosition, 0, calldatasize)

    let delegatecallSuccess := delegatecall(
        gas,
        implementationAddress,
        newCallDataPosition,
        calldatasize,
```
As a result, the proxy will not throw an error if its implementation is incorrectly set or self-destructed. It will instead return success, while no code was executed.

**Exploit Scenario**
Bob upgrades GoldToken to an incorrect new implementation. As a result, all the calls to `transfer` and `transferFrom` return success, while they do not change the state and do not perform token transfers. Eve uses the situation to scam Celo's users.

**Recommendation**
Check for contract existence prior to a `delegatecall` with the `EXTCODESIZE` opcode. Document that `suicide` or `selfdestruct` can lead to unexpected behavior. Prevent future upgrades from introducing these functions.

Carefully review the [Solidity documentation](https://docs.soliditylang.org/en/v0.8.11/), especially the entire Warnings section. In addition, carefully review the pitfalls of using `delegatecall` proxy pattern.

**References**
- [Contract upgrade anti-patterns](https://github.com/trailofbits/contract-upgrade-anti-patterns)
3. Lack of contract existence check in MultiSigWallet will lead to unexpected behavior

Severity: High  Difficulty: High  Type: Data Validation  Finding ID: TOB-CELO-03

Target: MultiSig.sol

**Description**
Failure to check for a contract's existence may lead to incorrect assumptions about the success of a call in the MultiSig contract.

**MultiSig** is meant to allow for both the execution of code and the transfer of Ether. In both cases, the call is done through an assembly call:

```solidity
function external_call(
    address destination,
    uint value,
    uint dataLength,
    bytes memory data
) private
returns (bool)
{
    bool result;
    /* solhint-disable max-line-length */
    assembly {
        let x := mload(0x40)  // "Allocate" memory for output (0x40 is where "free memory"
pointer is stored by convention)
        let d := add(data, 32) // First 32 bytes are the padded length of data, so exclude that
result := call(
    sub(gas, 34710),  // 34710 is the value that solidity is currently emitting
    // It includes callGas (700) + callVeryLow (3, to pay for SUB) +
callValueTransferGas (9000) +
    // callNewAccountGas (25000, in case the destination address does
not exist and needs creating)
    destination,
    value,
    d,
    dataLength,    // Size of the input (in bytes) - this is what fixes the padding
problem
    x,
    0             // Output is ignored, therefore the output size is zero
)
    /* solhint-enable max-line-length */
    return result;
}
```

*Figure 1: MultiSig.sol#L254-L275*

The [Solidity documentation](https://docs.solidity-lang.org/en/v0.6.11/) warns:
The low-level call, delegatecall, and callcode will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

As a result, if the destination does not contain code, MultiSig will return success. Calls with non-empty data are likely made with the intent of executing code, while in this case no code can be executed.

**Exploit Scenario**
Bob proposes a transaction that is meant to transfer tokens. The destination is incorrectly set. Bob’s transaction is accepted and executed. As a result, Bob incorrectly assumes that the tokens transfer succeeded.

**Recommendation**
Check the contract’s existence prior to every low-level call with non-empty data, using the EXTCODESIZE opcode.

Avoid low-level calls. If unavoidable, carefully review the [Solidity documentation](https://docs.soliditylang.org/en/latest/), especially the entire Warnings section.
4. Lack of contract existence check in Governance will lead to unexpected behavior

Severity: High
Type: Data Validation
Finding ID: TOB-CELO-04
Target: Governance.sol

**Description**

Failure to check for a contract's existence may lead to incorrect assumptions about the success of a call in the Governance contract.

**Governance.externalCall** uses assembly to call an external destination:

```solidity
/* solhint-disable max-line-length */
let x := mload(0x40) // "Allocate" memory for output (0x40 is where "free memory"
pointer is stored by convention)
let d := add(data, 32) // First 32 bytes are the padded length of data, so exclude that
result := call(
    sub(gas, 34710), // 34710 is the value that solidity is currently emitting
    // It includes callGas (700) + callVeryLow (3, to pay for SUB) +
callValueTransferGas (9000) +
    // callNewAccountGas (25000, in case the destination address does
not exist and needs creating)
    destination,
    value,
    d,
    dataLength, // Size of the input (in bytes) - this is what fixes the padding
    problem
    x,
    0 // Output is ignored, therefore the output size is zero
)
```

*Figure 1: Governance.externalCall (Governance.sol#L1062-L1075)*

The Solidity documentation warns:

*The low-level call, delegatecall, and callcode will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

As a result, if the destination does not contain code, Governance.externalCall will return success. Calls with non-empty data are likely to be made with the intention of executing code, while in this case no code can be executed.

**Exploit Scenario**

Bob proposes a transaction that is meant to transfer tokens. The destination is incorrectly set. Bob's transaction is accepted and executed. As a result, Bob incorrectly assumes that the tokens transfer succeeded.
**Recommendation**
Check the contract’s existence prior to every low-level call with non-empty data, using the `EXTCODESIZE` opcode.

Avoid low-level calls. If unavoidable, carefully review the [Solidity documentation](#), especially the entire Warnings section.
5. Race condition in the ERC20 approve function may lead to token theft

Severity: High
Type: Timing
Target: GoldToken.sol, StableToken.sol
Finding ID: TOB-CELO-05

Description
A known race condition in the ERC20 standard affecting the approve function could lead to the theft of tokens.

The ERC20 standard describes how to create generic token contracts. Among others, an ERC20 contract defines these two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions give permission to a third party to spend tokens. Once the function `approve(spender, value)` has been called by a user, `spender` can spend up to the value of the user's tokens by calling `transferFrom(user, to, value)`.

This schema is vulnerable to a race condition when the user calls `approve` a second time on a `spender` that has already been allowed. If the `spender` sees the transaction containing the call before it has been mined, the `spender` can call `transferFrom` to transfer the previous value and still receive the authorization to transfer the new value.

Exploit Scenario
1. Alice calls `approve(Bob, 1000)`. This allows Bob to spend 1,000 tokens.
2. Alice changes her mind and calls `approve(Bob, 500)`. Once mined, this will decrease to 500 the number of tokens that Bob can spend.
3. Bob sees the second transaction and calls `transferFrom(Alice, X, 1000)` before `approve(Bob, 500)` has been mined.
4. If Bob's transaction is mined before Alice's, he can transfer 1,000 tokens from the initial call. But once Alice's transaction is mined, Bob can call `transferFrom(Alice, X, 500)`. Bob will have transferred 1,500 tokens, even though this was not Alice's intention.

Recommendation
One common workaround is to implement `increaseAllowance` and `decreaseAllowance` from OpenZeppelin. Allowance can also revert if it sets the allowance from a non-empty value to another non-empty value.

Ensure users are aware of this extra functionality, and encourage them to make use of it when appropriate.
6. Quick buy and sell allows vote manipulation

Severity: High  Difficulty: High
Type: Timing  Finding ID: TOB-CELO-06
Target: BoundedDeposit.sol, Validators.sol, Governance.sol

Description
Celo relies on a voting system, which allows anyone to vote with any weight at the last minute. As a result, anyone with a large fund can manipulate the vote.

The voting mechanism of Celo relies on staking. There is no incentive for users to stake tokens well before the voting ends. Users can buy a large amount of tokens just before voting ends and sell them right after it. As a result, anyone with a large fund can decide the outcome of the vote without being a market participant.

As all the votes are public, users voting earlier will be penalized, because their votes will be known by the other participants. An attacker can know exactly how much currency will be necessary to change the outcome of the voting, just before it ends.

Exploit Scenario
The system is deployed and requires one validator. Bob and Eve both register to become the new validator. Bob votes for $5,000 for himself. Eve votes for Bob with $10,000 as a validator. All the users expect Bob to win, and no one adds more votes. At the last minute, Eve changes her vote to become the new validator. As a result, Eve wins.

Recommendation
Blockchain-based online voting is a known challenge. No perfect solution has been found so far.

Consider implementing a weighted stake (with the weight decreasing over time) to incentivize users to vote earlier. While it will not prevent users with unlimited resources to manipulate the vote at the last minute, it will make the attack more expensive.

An alternative to mitigate the issue is to require users to have a minimal staking period.
7. Linked list compromise through incorrect insertion

Severity: Low
Type: Data Validation
Target: LinkedList.sol
Finding ID: TOB-CELO-07

Description
LinkedList is a library implementing a linked list data structure. A lack of validation in the insertion function allows an attacker to corrupt the list.

insert(list, key, previousKey, nextKey) inserts key between previousKey and nextKey in list:

```solidity
function insert(
    List storage list,
    bytes32 key,
    bytes32 previousKey,
    bytes32 nextKey
) public
{
    require(key != bytes32(0), "Key must be defined");
    require(!contains(list, key), "Can't insert an existing element");

    Element storage element = list.elements[key];
    element.exists = true;

    if (list.numElements == 0) {
        list.tail = key;
        list.head = key;
    } else {
        require(
            previousKey != bytes32(0) || nextKey != bytes32(0),
            "Either previousKey or nextKey must be defined"
        );

        element.previousKey = previousKey;
        element.nextKey = nextKey;

        if (previousKey != bytes32(0)) {
            require(
                contains(list, previousKey),
                "If previousKey is defined, it must exist in the list"
            );
            Element storage previousElement = list.elements[previousKey];
            require(
                previousElement.nextKey == nextKey,
                "previousKey must be adjacent to nextKey"
            );
            previousElement.nextKey = key;
        }
    }
}
```
```solidity
} else {
  list.tail = key;
}

if (nextKey != bytes32(0)) {
  require(contains(list, nextKey), "If nextKey is defined, it must exist in the list");
  Element storage nextElement = list.elements[nextKey];
  require(nextElement.previousKey == previousKey, "previousKey must be adjacent to nextKey");
  nextElement.previousKey = key;
} else {
  list.head = key;
}
}

list.numElements = list.numElements.add(1);
```

Figure 1: LinkedList.insert (LinkedList.sol#L33-L85)

Missing validation allows an attacker to insert a new key equal to previousKey, nextKey, or both. As a result, the added element will point to itself, creating a loop in the linked list.

If the new key is equal to previousKey, and nextKey is set to 0, the head of the list will point to the new element. As a result, iterating over the list elements will always return the same element.

Figure 2 shows an example of how this can be exploited:

```solidity
contract Attack {
  using LinkedList for LinkedList.List;

  LinkedList.List private list;

  function exploit() public{
    list.insert("AA", 0, 0);
    list.insert("BB", "AA", 0);
    list.insert("CC", "CC", 0);
  }

  function getKeys() public view returns(bytes32[] memory){
    return list.getKeys();
  }

  function contains() public view returns(bool){
    return list.contains("CC");
  }

  function size() public view returns(uint){
    return list.numElements;
  }
}
```

Figure 2: Attack.sol
After calling exploit, getKeys() will return ‘CC’ three times.

This issue does not directly affect the Celo codebase, as LinkedList.insert is either not directly callable, or the issue is mitigated by the caller.

Appendix D contains an Echidna test that triggers the bug.

**Exploit Scenario**
The Celo team updates the contracts and uses LinkedList instead of SortedLinkedList to store the validators’ votes. Eve adds her candidate to the list’s head and points the next candidate to itself. As a result, only Eve’s candidate appears in the list and Eve’s candidate wins.

**Recommendation**
Prevent key to being equal to previousKey and nextKey in LinkedList.insert.

Use Echidna to test the correct state transitions of the linked list implementations.
8. Missing validation of message signatures

Severity: Informational
Type: Cryptography
Finding ID: TOB-CELO-08
Target: Multiple contracts

Description
The `ecrecover` usages do not follow security best practices. As a result, an incorrect validation schema might be introduced in the future.

Multiple Celo contracts use `ecrecover` to validate supplied cryptographic signatures. In all identified instances, a call to this function is not checked for failure, which is indicated by a return value of zero. There are also no checks to prevent signature malleability attacks. As stated by Solidity documentation:

> If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for transaction signatures (see EIP-2), but the `ecrecover` function remained unchanged.

> This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin have a ECDSA helper library that you can use as a wrapper for `ecrecover` without this issue.

Figure 1 lists the impacted functions. Note that even in cases where a user-supplied address is checked against a signer’s address (e.g., in `getSignerOfAddress` and `delegateRole`), a user-supplied address of zero would pass this check.

- `getSignerOfAddress` in `common/Signatures.sol#L12-L26`
- `delegateRole` in `governance/BondedDeposits.sol#L218-L246`
- `withdraw` in `identity/Escrow.sol#L132-L161`
- `validateAttestationCode` in `identity/Attestations.sol#L529-L554`

![Figure 1: List of functions with inadequate validation of message signatures.](image)

Exploit Scenario
The Celo team updates the codebase and allows 0 to be a valid signer address. As a result, anyone can pass the signature check by providing incorrect parameter to `ecrecover`.

Recommendation
Short term, whenever validating signatures, use OpenZeppelin’s ECDSA helper library, and consistently validate that the returned value is not equal to an address of zero.
Long term, add tests to the codebase that validate the proper handling of invalid signatures.
9. Gas cost of precompiled transfer contract is unclear

Severity: Informational  Difficulty: High
Type: Undefined Behavior  Finding ID: TOB-CELO-09
Target: core/vm/contracts.go

Description
The precompiled Transfer contract has an unclear gas cost. The undefined behavior might lead to cheaper-than-expected execution or incorrect client implementations.

Although the contract defines an amount of required gas, it is never explicitly debited from the transaction. Figure 2 demonstrates this with a missing call to debitRequiredGas, which is the function responsible for debiting the amount of gas returned in Figure 1.

```
func (c *transfer) RequiredGas(input []byte) uint64 {
    return params.TxGas
}
```

*Figure 1: /core/vm/contracts.go#L459-L461*

```
func (c *transfer) Run(input []byte, caller common.Address, evm *EVM, gas uint64) ([]byte, uint64, error) {
    celoGoldAddress, err := GetRegisteredAddressWithEvm(params.GoldTokenRegistryId, evm)
    if err != nil {
        return nil, gas, err
    }
    if caller != *celoGoldAddress {
        return nil, gas, fmt.Errorf("Unable to call transfer from unpermissioned address")
    }
    from := common.BytesToAddress(input[0:32])
    to := common.BytesToAddress(input[32:64])
    var parsed bool
    value, parsed := math.ParseBig256(hexutil.Encode(input[64:96]))
    if !parsed {
        return nil, gas, fmt.Errorf("Error parsing transfer: unable to parse value from " + hexutil.Encode(input[64:96]))
    }
    // Fail if we're trying to transfer more than the available balance
    if !evm.Context.CanTransfer(evm.StateDB, from, value) {
        return nil, gas, ErrInsufficientBalance
    }
    gas, err := evm.TobinTransfer(evm.StateDB, from, to, gas, value)
    return input, gas, err
}
```

*Figure 2: /core/vm/contracts.go#L463-L488*
The associated gas computation is unclear and missing from documentation, which may lead to errors in the future.

However, the codebase is not at immediate risk, as gas ends up being debited later in a call to TobinTransfer.

**Exploit Scenario**
Bob implements a fork of another Ethereum client to work with the Celo codebase. Bob’s precompiled transfer contract takes into account the gas amount in Figure 1, while Celo’s does not, resulting in a consensus split.

**Recommendation**
Short term, update the requiredGas function in /core/vm/contracts.go#L459-L461 to return a value of zero.

Long term, write a specification of each new precompiled contract, and add tests to the codebase to check that they are followed.
10. Unsafe Solidity type conversion

Severity: Informational
Type: Data Validation
Finding ID: TOB-CELO-10
Target: BondedDeposits.sol

Description

The BondedDeposit contract performs multiple type conversions without first checking the type's maximum value.

The code identified in the figures below take a value of one type and convert it to another, without validating that the conversion is safe (i.e., doesn't truncate the integer). Because none of the instances identified are exploitable, this is only an informational-severity issue.

```solidity
function updateBondedDeposit(
    Account storage account,
    uint256 value,
    uint256 noticePeriod
) private {
    Deposit storage bonded = account.deposits.bonded[noticePeriod];
    require(value != bonded.value);
    uint256 weight;
    if (bonded.value == 0) {
        bonded.index = uint128(account.deposits.noticePeriods.length);
        bonded.value = uint128(value);
        account.deposits.noticePeriods.push(noticePeriod);
        weight = getDepositWeight(value, noticePeriod);
        account.weight = account.weight.add(weight);
        totalWeight = totalWeight.add(weight);
    } else if (value == 0) {
        weight = getDepositWeight(bonded.value, noticePeriod);
        account.weight = account.weight.sub(weight);
        totalWeight = totalWeight.sub(weight);
        deleteDeposit(bonded, account.deposits, DepositType.Bonded);
    } else {
        uint256 originalWeight = getDepositWeight(bonded.value, noticePeriod);
        weight = getDepositWeight(value, noticePeriod);

        uint256 difference;
        if (weight >= originalWeight) {
            difference = weight.sub(originalWeight);
            account.weight = account.weight.add(difference);
            totalWeight = totalWeight.add(difference);
        } else {
            difference = originalWeight.sub(weight);
            account.weight = account.weight.sub(difference);
            totalWeight = totalWeight.sub(difference);
        }
    }
    bonded.value = uint128(value);
}
```

Figure 1: BondedDeposits.sol#L579-L634
function updateNotifiedDeposit(
    Account storage account,
    uint256 value,
    uint256 availabilityTime
)
private
{
    Deposit storage notified = account.deposits.notified[availabilityTime];
    require(value != notified.value);
    if (notified.value == 0) {
        notified.index = uint128(account.deposits.availabilityTimes.length);
        notified.value = uint128(value);
        account.deposits.availabilityTimes.push(availabilityTime);
        account.weight = account.weight.add(notified.value);
        totalWeight = totalWeight.add(notified.value);
    } else if (value == 0) {
        account.weight = account.weight.sub(notified.value);
        totalWeight = totalWeight.sub(notified.value);
        deleteDeposit(notified, account.deposits, DepositType.Notified);
    } else {
        uint256 difference;
        if (value >= notified.value) {
            difference = value.sub(notified.value);
            account.weight = account.weight.add(difference);
            totalWeight = totalWeight.add(difference);
        } else {
            difference = uint256(notified.value).sub(value);
            account.weight = account.weight.sub(difference);
            totalWeight = totalWeight.sub(difference);
        }
        notified.value = uint128(value);
    }
}

function createAccount()
external
returns (bool)
{
    require(isNotAccount(msg.sender) && isNotDelegate(msg.sender));
    Account storage account = accounts[msg.sender];
    account.exists = true;
    account.rewardsLastRedeemed = uint96(block.number);
    return true;
}

**Exploit Scenario**

Celo implements a code change that does not account for the identified unsafe type conversions, resulting in an exploitable issue within one of the Celo contracts.
**Recommendation**

Short term, ensure that all unsafe type conversions (e.g., going from a larger integer to a smaller integer) are validated, by first checking that the original number does not exceed the maximum value of the desired type.

Long term, consider eliminating unnecessary type conversions from the codebase.
11. Oracle exchange rates can be manipulated by calling `removeExpiredReports`

Severity: High
Type: Access Control
Finding ID: TOB-CELO-11
Target: SortedOracles.sol

**Description**

A lack of access control affecting report removal functionality allows for the manipulation of exchange rates.

The function `removeExpiredReports` deletes expired oracle reports, which can change an oracle's median rate and update its exchange rate. Because this function is public, users are free to call it whenever the exchange rate will be updated in their favor.

```solidity
function removeExpiredReports(address token, uint256 n) external {
    require(
        token != address(0) &&
        timestamps[token].tail != address(0) &&
        n < timestamps[token].numElements
    );
    for (uint256 i = 0; i < n; i++) {
        address oldest = timestamps[token].tail;
        uint128 timestamp = timestamps[token].elements[oldest].numerator;
        // solhint-disable-next-line not-rely-on-time
        if (uint128(now).sub(timestamp) >= uint128(reportExpirySeconds)) {
            removeReport(token, oldest);
        } else {
            break;
        }
    }
}
```

*Figure 1: SortedOracles.sol#L120-L136*

```solidity
function removeReport(address token, address oracle) private {
    SortedFractionMedianList.Element memory originalMedian =
    getMedianElement(rates[token]);
    rates[token].remove(oracle);
    timestamps[token].remove(oracle);
    emit OracleReportRemoved(token, oracle);
    emitIfMedianUpdated(token, originalMedian);
}
```

*Figure 2: SortedOracles.sol#L280-L286*

**Exploit Scenario**

An oracle's exchange rate is 650. Eve knows that calling `removeExpiredReports(token, 2)` would remove the last two expired reports, causing the median price to change and the oracle's exchange rate to increase to 680. Eve creates a single Ethereum transaction that
places a buy order on the exchange, calls `removeExpiredReports(token, 2)`, and then places a sell order, resulting in an immediate profit.

**Recommendation**

Short term, only allow privileged addresses (e.g., an owner address) to call `removeExpiredReports`.

Long term, consider requiring that validators automatically call `removeExpiredReports` as the first transaction of every block.

Alternatively, if this is considered to be an expected arbitrage opportunity, it must be properly documented to ensure all users are aware of it.
12. Compromise of a single oracle allows limited control of the price

Severity: High  Difficulty: High
Type: Data Validation  Finding ID: TOB-CELO-12
Target: SortedOracles.sol, SortedFractionMedianList.sol

Description
By compromising only one oracle, an attacker can control the median rate within a certain range.

SortedOracle computes the median of all the price oracles. If the number of oracles is odd, the median is the center value of the ordered list of the rates. If an attacker compromises one oracle, they can control the median within a range.

Exploit Scenario
3 oracles are available:

- $O_0$ with a price of 603
- $O_1$ with a price of 598
- $O_2$ which is compromised by Eve

Eve is able to set the median rate to any value in the range $[598, 603]$. Eve can adjust the rate when buying and selling to make a profit.

Recommendation
There is no simple fix for this issue. Consider on-chain monitoring of the exchange and oracle contracts to report any suspicious activity.

Long term, assume that an attacker may be able to compromise some of the oracles. The price computation should be robust in case of partial compromise.
13. Arithmetic rounding leads to non-constant product

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-CELO-13
Target: Exchange.sol

**Description**

StableToken uses the constant-product-market-maker model. The model relies on the product of the StableToken and GoldToken buckets being constant. Due to a rounding imprecision, the product does not stay constant.

The central equation of the stability model is the constant of the bucket product:

![Figure 1: Stability documentation](image)

For a purchase of gold, the equation can be reformulated as:

\[
goldBucket \times stableBucket = (goldBucket - goldBought) \times (stableBucket + StableSold)
\]

The function `getBuyTokenAmount` returns the amount of tokens to receive for a given amount of tokens sold:
function getBuyTokenAmount(
    uint256 sellAmount,
    bool sellGold
) external view returns (uint256)
{
    uint256 sellTokenBucket;
    uint256 buyTokenBucket;
    (buyTokenBucket, sellTokenBucket) = getBuyAndSellBuckets(sellGold);

    uint256 x = spread.denominator.sub(spread.numerator).mul(sellAmount);
    uint256 numerator = x.mul(buyTokenBucket);
    uint256 denominator = sellTokenBucket.mul(spread.denominator).add(x);

    return numerator.div(denominator);
}  

Figure 2: Exchange.sol#L159-L176

Rounding imprecision is introduced by the division: numerator.div(denominator). As a result, Figure 1’s equation does not hold.

Each buy and sell can introduce a slight change in the bucket’s product result, leading the conversion between StableToken and GoldToken to be incorrect.

**Exploit Scenario**
Bob wants to sell 10 gold tokens. The current gold bucket is 100, and the stable bucket is 100. The product of the two buckets is 1,000. Bob receives 9 stable tokens. The new gold bucket is 110, and the new stable bucket is 1010. The result of their product is 1,010. As a result, the market did not keep a constant product.

**Recommendation**
Consider allowing only buy and sell orders that do not lead to loss of precision.

Use Manticore to ensure the correctness of the market model computation.
14. Lack of validation in update allows for SortedFractionMedianList compromise

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-CELO-14
Target: SortedFractionMedianList.sol

Description
A lack of validation in `SortedFractionMedianList.update` allows a malicious oracle to compromise the token's rate list.

`SortedFractionMedianList` allows oracles to insert new elements or update existing ones. The following checks are present in `SortedFractionMedianList.insert`:

```solidity
require((lesserKey != address(0) || greaterKey != address(0)) || list.numElements == 0);
require(contains(list, lesserKey) || lesserKey == address(0));
require(contains(list, greaterKey) || greaterKey == address(0));
```

**Figure 1: SortedFractionMedianList.sol#L99-L101**

However, these checks are missing in `SortedFractionMedianList.update`:

```solidity
function update(
    List storage list,
    address key,
    uint128 numerator,
    uint128 denominator,
    address lesserKey,
    address greaterKey
) public {
    Element storage element = list.elements[key];
    // TODO: abstract repeated checks
    require(
        key != address(0) && key != lesserKey && key != greaterKey && contains(list, key),
        "key was null or equal to lesserKey or equal to greaterKey or already in DLL"
    );
    // TODO(asa): Optimize by not making any changes other than value if lesserKey and
greaterKey
    // don't change.
    // TODO(asa): Optimize by not updating lesserKey/greaterKey for key
    remove(list, key);
    (lesserKey, greaterKey) = getLesserAndGreater(
        list,
        numerator,
        denominator,
        lesserKey,
        greaterKey
    );
    _insert(list, element, key, numerator, denominator, lesserKey, greaterKey);
```
As a result, it is possible to compromise the list through the following scenarios:

- lesserKey and greaterKey both equal to 0
- A nonexistent lesserKey
- A nonexistent greaterKey

For example, submitting a price with a nonexistent greaterKey will lead to the creation of a new list, with only the newly added element. The tail of the list will point to this element.

**Exploit Scenario**

Eve is a malicious oracle. Eve compromises the rate's list and manipulates the median computation.

**Recommendation**

Add in update the same checks present in insert
(SortedFractionMedianList.sol#L99-L101)

Use [Echidna](https://echidna.tools) to test the robustness of the linked lists.
15. Exchange fallback function will lead to trapping ether

Severity: Medium
Type: Data Validation
Target: Exchange.sol
Finding ID: TOB-CELO-15

Description
The Exchange contract’s fallback function allows the contract to receive Ether. Ether sent to this contract will be trapped.

Note that the exchange’s fallback function is payable:

```solidity
function() external payable {} // Figure 1: Exchange.sol#L61
```

No function in Exchange allows the caller to withdraw or transfer their ether (or GoldToken). As a result, their ether is trapped.

Exploit Scenario
Bob sends 100 Ether through the fallback function by mistake. Bob’s ether is lost.

Recommendation
Remove payable from the fallback function.

Use cryptic.io or Slither to detect the most common Solidity flaws.
16. Incorrect access control allows anyone to burn tokens’ reserve

Severity: Informational  Difficulty: High
Type: Access Control  Finding ID: TOB-CELO-16
Target: Reserve.sol

Description
A lack of access control on the Reserve contract’s burn function allows anyone to burn tokens from the reserve.

Reserve.burn is callable by anyone, as shown in Figure 1:

```solidity
/**
 * @notice Burns all tokens held by the Reserve.
 * @param token The address of the token to burn.
 */
function burnToken(address token) external isStableToken(token) returns (bool) {
    IStableToken stableToken = IStableToken(token);
    require(stableToken.burn(stableToken.balanceOf(address(this))), "reserve token burn failed");
    return true;
}

Figure 1: Reserve.sol#L135-L143
```

The codebase is not at immediate risk, as Reserve is not allowed to burn StableToken tokens. After adding other stable tokens or updating the StableToken, the burn function can lead to a loss of funds.

Additionally the contract has the private mintToken function that is never called, but could allows anyone to mint tokens.

Exploit Scenario
The Celo team adds a new stable token. The token allows its users to burn their balances. Eve calls burn on the reserve’s tokens and empties the reserve’s balance.

Recommendation
Remove the burn function or add the onlyOwner modifier. Consider removing the private mintToken function.

Use Slither printers to review that each contract has only required functions.
17. Missing validation in contract initializations

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-CELO-17
Target: Multiple contracts

Description
Multiple contracts are missing adequate validation in their initialization functions. For example, the following contracts do not check that `registryAddress` isn’t equal to zero:

- GasPriceMinimum
- BondedDeposits
- Governance
- Attestations
- Escrow
- Exchange
- Reserve
- StableToken

The following validations are also missing:

- The Exchange contract does not validate that `spreadDenominator` and `reserveFractionDenominator` are not equal to zero.
- The Reserve contract does not validate that `tobinTaxStalenessThreshold` is greater than zero, even though the `setTobinTaxStalenessThreshold` function does.
- The StableToken contract does not validate that `name_` and `symbol_` are not blank, and that `decimals_` does not equal zero.

Exploit Scenario
The Celo team deploys a new version of the Attestations contract. Due to missing initialization validation and a bug in a deployment script, `registryAddress` is set to zero, and attestations no longer work.

Recommendation
Short term, apply all of the missing validations listed above.

Long term, introduce additional software tests to check that initialization validation is adequately enforced across all contracts.
18. Celo identity attestation vulnerable to SIM-swapping attacks

Severity: High
Difficulty: High
Type: Authentication
Finding ID: TOB-CELO-18
Target: SMS-based attestation

Description
The Celo blockchain relies on SMS for user identity attestations. SMS-based authentication schemes are known to be insecure, due to their susceptibility to “SIM-swapping” attacks. This makes Celo's mapping of mobile numbers to Ethereum addresses less trustworthy, and can lead to users unknowingly sending tokens to an attacker's address.

After a user submits an attestation request, a randomly selected validator sends a challenge code ("secret message") to the user's mobile number over SMS. If the user submits a correct challenge code in return, this is treated as a proof that the user's Ethereum address is associated with that mobile number. Other Celo users can then perform a lookup of the user's Ethereum address by providing the user's mobile number.

Mobile numbers offer a weak form of identity and should not be relied on for user authentication, because they can be transferred to other mobile accounts and providers. Attackers regularly exploit this vulnerability to take over mobile numbers and defeat SMS-based authentication schemes. Unauthorized mobile number transfers are commonly referred to as SIM-swapping or “SIM-porting” attacks.

Many mobile providers offer customers additional security features to protect against unauthorized number transfers. However, these features are optional and may not always withstand social engineering efforts and insider threats. Recent stories of successful SIM-swapping attacks illustrate the problem:

- The Most Expensive Lesson Of My Life: Details of SIM port hack
- 'Sim swap' gives fraudsters access-all-areas via your mobile phone
- Many Bengalureans lose cash to sim card swap fraud

Exploit Scenario
Bob wants to use Celo. Eve runs a SIM-swapping attack against Bob, allowing them to submit a valid attestation and change the Ethereum address associated with Bob's mobile number. As a result, Eve receives all the funds sent to Bob.

Recommendation
Electronic identity verification is a challenging problem that does not have one simple solution. Consider more secure forms of electronic identity, such as email addresses or domain names.
If Celo still intends to use SMS to identify Celo users, consider educating users on the dangers of SIM-swapping attacks and encouraging them to add security PINs to their mobile accounts.
19. Oracle's median can be compromised with zero value

Severity: High
Type: Data Validation
Difficulty: High
Finding ID: TOB-CELO-19
Target: SortedOracles.sol, FractionUtil.sol

Description
A lack of validation allows an oracle to report a price of zero (0/0) in any place of the sorted price's list. As a result, a malicious oracle can set the median to zero.

report lets oracles add new values to the sorted price's list:

```solidity
function report(
    address token,
    uint128 numerator,
    uint128 denominator,
    address lesserKey,
    address greaterKey
) external
onlyOracle(token)
{
    SortedFractionMedianList.Element memory originalMedian = getMedianElement(rates[token]);
    rates[token].insertOrUpdate(msg.sender, numerator, denominator, lesserKey, greaterKey);
}
```

Figure 1: SortedOracles.sol#L148-L159

Values are fractions. isLessThanOrEqualTo and isGreaterThanOrEqualTo ensure that the element is inserted at the correct position:

```solidity
function isLessThanOrEqualTo(
    Fraction memory x,
    Fraction memory y
)
internal
pure
returns (bool)
{
    return x.numerator.mul(y.denominator) <= y.numerator.mul(x.denominator);
}
```

Figure 2: FractionUtil.sol#L191-L200
return x.numerator.mul(y.denominator) >= y.numerator.mul(x.denominator);
}

Figure 3: FractionUtil.sol#L157-L166

There is no check to ensure that the fraction is not 0/0. If the value is 0/0, both isLessThanOrEqualTo and isGreaterThanOrEqualTo always return true.

As a result, an oracle can set 0/0 at any place in the sorted price's list and compromise the median.

A median of 0/0 will lead, among others, to:

- An empty stableBucket, which will lead the attacker to buy the entire gold tokens reserve supply for spread*stable stable tokens. If spread is 1, the reserve can be bought with 1 stable token
- The impossibility to execute computeTobinTax

**Exploit Scenario**
There are 50,000 gold tokens in the reserve, worth $5,000,000. Eve is a malicious oracle. Eve changes the median to zero, buys the 50,000 gold tokens with 1 stable token and sells everything on a third-party market.

**Recommendation**
Short term, prevent the contract from accepting either a numerator and denominator of 0.

Long term, use Echidna and Manticore to ensure that invalid prices cannot be added to the list. Moreover, consider oracles as untrusted users, and validate and monitor their inputs.
20. Exchange susceptible to front-running

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-CELO-20
Target: Exchange.sol

Description
The Exchange contract's exchange function allows one to buy and sell tokens. Attackers can make a profit by front-running price updates.

Ethereum transactions are not instantaneously validated. An attacker can observe price updates before they have been accepted by the network. As a result, an attacker can place a buy/sell order just before an update and profit from the price change.

The ability for an attacker's transaction to get accepted before the original depends on the network state and the gas price of each transaction. However, an attacker can maintain control over the outcome by offering a high gas price to increase their chance of success.

Exploit Scenario
Eve sees an upcoming price update that is going to increase the on-chain price of gold tokens. Eve buys 1,000 gold tokens before the transaction. The price is then updated. Eve benefits from the price's increase.

Recommendation
This issue is inherent in the nature of on-chain exchange and is present in several similar platforms. Be sure that users are aware of the risk, and properly document the arbitrage opportunity.

Reference
- Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges
21. On-chain mitigation does not prevent reserve from becoming under-collateralized

Severity: High  Difficulty: Undetermined  Type: Data Validation  Finding ID: TOB-CELO-21

Target: Reserve.sol

Description
A missing on-chain mitigation to ensure the reserve's collateralization can lead to the system becoming under-collateralized.

To keep stability, the Reserve contract must hold more gold tokens' worth of dollar than the current supply of stable tokens. If the reserve's balance drops too low, it is assumed that Governance holders will increase the balance. No on-chain mechanism ensures that the reserve holds enough tokens.

One of the on-chain mechanisms to prevent an under-collateralized state is the tax to be applied to every transaction:

```solidity
function computeTobinTax() private view returns (uint256) {
    address sortedOraclesAddress = registry.getAddressForOrDie(SORTED_ORACLES_REGISTRY_ID);
    ISortedOracles sortedOracles = ISortedOracles(sortedOraclesAddress);
    uint256 reserveGoldBalance = address(this).balance;
    uint256 stableTokensValueInGold = 0;

    for (uint256 i = 0; i < _tokens.length; i++) {
        uint256 stableAmount;
        uint256 goldAmount;
        (stableAmount, goldAmount) = sortedOracles.medianRate(_tokens[i]);
        uint256 stableTokenSupply = IERC20Token(_tokens[i]).totalSupply();
        uint256 aStableTokenValueInGold = stableTokenSupply.mul(goldAmount).div(stableAmount);
        stableTokensValueInGold = stableTokensValueInGold.add(aStableTokenValueInGold);
    }

    // The protocol calls for a 0.5% transfer tax on Celo Gold when the reserve ratio < 2.
    // The protocol aims to keep half of the reserve value in gold, thus the reserve ratio
    // is two when the value of gold in the reserve is equal to the total supply of stable tokens.
    if (reserveGoldBalance >= stableTokensValueInGold) {
        return 0;
    } else {
        return 5;
    }
}
```

Figure 1: Reserve.sol#L187-L210
However, this tax will be applied only once the reserve has less value in gold token than the total supply of stable coins. There is an assumption that the reserve will have access to other collaterals, but this assumption is not verified on-chain. As a result, tokens holders must trust the Governance holders to preserve the collateralization of the system.

**Exploit Scenario**
The reserve’s gold tokens are equal to $5,000,000. The total supply of stable coin is worth $4,500,000. The price of gold tokens drops 20%. The Governance holders do not act, and the system becomes under-collateralized. As a result, token holders panic and withdraw their funds. The withdrawals increase the under-collateralization and lead to an unrecoverable system.

**Recommendation**
Short term, consider lowering the amount of gold tokens from which the tax fee is triggered. Monitor the price to ensure the reserve is always over-collateralized with a given threshold.

Long term, investigate solutions to reduce dependency on the Governance holders.
22. Attestation validator selection takes place in a single transaction

Severity: High  Difficulty: High
Type: Timing  Finding ID: TOB-CELO-22
Target: Attestations.sol

Description
By exploiting the deterministic validator selection process that takes place during an attestation request, an attacker can request an attestation when they know that a validator under their control would be chosen.

According to Celo’s documentation:

As mentioned previously, when requesting new attestations, random validators are selected to perform phone number verification. This selection needs to be unpredictable to prevent Eve from creating an attestation for a phone number she doesn't control. Suppose, for example, that instead validators were selected in a round robin fashion. Eve could request an attestation when it was the turn of a validator she controls to perform verification. Instead of sending an SMS to the phone number (since she doesn't own it) she could just produce the correct verification code since she has access to the validator’s private key.

In an attempt to mitigate this issue, Celo uses a reveal-and-commit scheme that is meant to provide a source of entropy for the Attestations contract to use. When it is a validator’s turn to offer entropy, a commitment to a value is made such that it is to be revealed during their next turn, while the validator’s previous commitment is revealed and used during the current block.

However, this issue is not adequately mitigated by the proposed scheme. As highlighted in Figure 1 and Figure 2, an attestation request and random validator selection both happen within a single transaction. This makes it possible for an attacker to predict which validator would be selected if they were to call request.

```solidity
function request(
    bytes32 identifier,
    uint256 attestationsRequested,
    address attestationRequestFeeToken
) external
{
    require(
        attestationsRequestFees[attestationRequestFeeToken] > 0,
        "Invalid attestationsRequestFeeToken"
    );
    require(
        IERC20Token(attestationRequestFeeToken).transferFrom(}
msg.sender,
address(this),
attestationRequestFees[attestationRequestFeeToken].mul(attestationsRequested)
),
"Transfer of attestation request fees failed"
);
require(attestationsRequested > 0, "You have to request at least 1 attestation");
if (accounts[msg.sender].attestationRequestFeeToken != address(0x0)) {
  require(
    !isAttestationTimeValid(accounts[msg.sender].mostRecentAttestationRequest) ||
    accounts[msg.sender].attestationRequestFeeToken == attestationRequestFeeToken,
    "A different fee token was previously specified for this account"
  );
}
// solhint-disable-next-line not-rely-on-time
accounts[msg.sender].mostRecentAttestationRequest = uint96(now);
accounts[msg.sender].attestationRequestFeeToken = attestationRequestFeeToken;
IdentifierState storage state = identifiers[identifier];
addIncompleteAttestations(attestationsRequested, state.attestations[msg.sender]);

function addIncompleteAttestations(
  uint256 n,
  AttestationsMapping storage state
) internal {
  IRandom random = IRandom(registry.getAddressForOrDie(RANDOM_REGISTRY_ID));
  bytes32 seed = random.random();
  address[] memory validators = getValidators();
  uint256 currentIndex = 0;
  address validator;
  while (currentIndex < n) {
    seed = keccak256(abi.encodePacked(seed));
    validator = validators[uint256(seed) % validators.length];
  }

Exploit Scenario
An attacker deploys a contract that calculates which validator would be chosen if request were to be called, and waits until a validator under their control is selected. Several forged attestations are then generated and completed, compromising the identity of multiple users on the Celo network.
**Recommendation**
Due to the deterministic nature of blockchain transactions, there is no perfect on-chain solution to this problem.

Short term, remove the validator selection from request and perform the selection in a separate transaction. A more effective mitigation would be to:

- Ask for a number in request, \( n \geq 0 \), and store \( n \) and the original block.number of the transaction
- Create a `select_validator` function that uses `blockhash(original block.number + n)`. Ensure that `blockhash` does not return 0

Note that this solution is vulnerable to a malicious validator controlling the value of `blockhash`. Other schemas might be considered (e.g., requiring seeds from multiple validators).

Long term, use a monitoring solution to detect suspicious behavior such as a single user making many attestation requests.
23. MultiSig contract is missing address validation

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-CELO-23
Target: MultiSig.sol

Description
A missing modifier call can lead MultiSig to have the zero address as an owner.

Both addOwner and replaceOwner can be set to update the list of owners.

```solidity
function addOwner(address owner) public
onlyWallet
ownerDoesNotExist(owner)
notNull(owner)
validRequirement(owners.length + 1, required)
```

Figure 1: MultiSig.sol#L131-L136

```solidity
function replaceOwner(address owner, address newOwner) public
onlyWallet
ownerExists(owner)
ownerDoesNotExist(newOwner)
{
    for (uint i=0; i < owners.length - 1; i++)
        if (owners[i] == owner) {
            owners[i] = newOwner;
            break;
        }
    isOwner[owner] = false;
    isOwner[newOwner] = true;
    emit OwnerRemoval(owner);
    emit OwnerAddition(newOwner);
}
```

Figure 2: MultiSig.sol#L165-L180

replaceOwner lacks the notNull modifier. As a result, the zero address can become an multisig owner through an incorrect update.

Exploit Scenario
The Celo team changes the owner of the MultiSig contract. Due to a bug in a deployment script, the only owner address is set to zero and access to the contract cannot be recovered.

Recommendation
Short term, add the notNull(newOwner) modifier to replaceOwner. Long term, use Manticore and Echidna to ensure that a multisig's owner can never be the zero address.
24. Missing validation allows for Istanbul message forgery
Severity: High                  Difficulty: Low
Type: Data validation              Finding ID: TOB-CELO-24
Target: consensus/istanbul/core/core.go, consensus/istanbul/core/handler.go

Description
Missing validation allows an elected validator to send messages on behalf of any other validator.

When decoding a payload to a message, `c.validateFn` is called:

```
msg := new(message)
if err := msg.FromPayload(payload, c.validateFn); err != nil {
    logger.Error("Failed to decode message from payload", "err", err)
    return err
}
```

*Figure 1: core/handler.go#L134-L142*

c.validateFn is set to `c.checkValidatorSignature`:

```
c.validateFn = c.checkValidatorSignature
```

*Figure 2: core/core.go#L53*

`checkValidatorSignature` checks that the message has been signed by a validator, but does not check that the address of the message was the message's sender.

As a result, a validator can sign and send messages on behalf of other validators.

Exploit Scenario
Eve is a validator. Eve wants to prevent Bob's transaction from being accepted. Every time Bob's transaction is in a block to be committed, Eve sends fake RoundChange messages to all the validators. As a result, Eve prevents Bob's transaction from being processed.

Recommendation
Short term, ensure that the message address is the message's sender.

Long term, thoroughly validate any field decoded from a user-controlled source.
25. Missing validation allows for Istanbul message replay

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-CELO-25
Target: consensus/istanbul/core/core.go, consensus/istanbul/core/handler.go

Description
Missing validation allows anyone to replay broadcasted messages. As a result, an attacker can replay RoundChange messages to prevent a round from being processed.

When decoding a payload to a message, c.validateFn is called:

```go
msg := new(message)
if err := msg.FromPayload(payload, c.validateFn); err != nil {
    logger.Error("Failed to decode message from payload", "err", err)
    return err
}
```

*Figure 1: core/handler.go#L134-L142*

c.validateFn is set to c.checkValidatorSignature:

```go
c.validateFn = c.checkValidatorSignature
```

*Figure 2: core/core.go#L53*

checkValidatorSignature checks that the message has been signed by a validator, but does not check that the address of the message was the signer.

As a result, anyone, even nodes that are not validators, can replay broadcasted messages.

Exploit Scenario
A round change is made. Eve collects all the messages from the round change. Once the new round starts, Eve continuously sends the collected messages to all the validators, preventing the new round from processing.

Recommendation
Short term, ensure that the message is signed by the sender.

Long term, thoroughly validate any field decoded from a user-controlled source.
26. Future messages can crash a node through out-of-memory condition

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-CELO-26
Target: consensus/istanbul/core/handler.go, consensus/istanbul/core/backlog.go

Description
Validators store messages related to future blocks. There is no limit on the number of messages stored. As a result, an attacker can spam a node until it reaches an out-of-memory condition.

Messages returning the `errFutureMessage` error are stored for future processing:

```go
// Store the message if it's a future message
testBacklog := func(err error) error {
    if err == errFutureMessage {
        c.storeBacklog(msg, src)
    }

    return err
}

switch msg.Code {
case msgPreprepare:
    return testBacklog(c.handlePreprepare(msg, src))
case msgPrepare:
    return testBacklog(c.handlePrepare(msg, src))
case msgCommit:
    return testBacklog(c.handleCommit(msg, src))
case msgRoundChange:
    return testBacklog(c.handleRoundChange(msg, src))
}
```

Figure 1: handler.go#L157-L174

There is no limit on the number of messages stored:

```go
func (c *core) storeBacklog(msg *message, src istanbul.Validator) {
    logger := c.logger.New("from", src, "state", c.state)

    if src.Address() == c.Address() {
        logger.Warn("Backlog from self")
        return
    }

    logger.Trace("Store future message")

    c.backlogsMu.Lock()
```
As a result, an attacker can spam a node with messages classified as future messages, filling the entire memory of the node and making it crash.

**Exploit Scenario**
Bob is a validator. Eve wants to prevent Bob from participating. Eve spams Bob with future requests and makes Bob’s node crash.

**Recommendation**
Short term, limit the number of requests per source stored in the backlog.

Long term, use **gofuzz** to check the robustness of the nodes.
27. A malicious or unreachable proposer can trap the system

Severity: High
Type: Denial of Service
Finding ID: TOB-CELO-27

Description
A proposer not sending a PREPREPARE message will prevent the system from processing a new block.

Only the proposer can start a round. If the validators do not receive a PREPREPARE message, it triggers a timeout, and executes `handleTimeoutMsg`:

```go
func (c *core) handleTimeoutMsg() {
    // If we're not waiting for round change yet, we can try to catch up
    // the max round with F+1 round change message. We only need to catch up
    // if the max round is larger than current round.
    if !c.waitingForRoundChange {
        maxRound := c.roundChangeSet.MaxRound(c.valSet.F() + 1)
        if maxRound != nil && maxRound.Cmp(c.current.Round()) > 0 {
            c.sendRoundChange(maxRound)
            return
        }
    }

    lastProposal, _ := c.backend.LastProposal()
    if lastProposal != nil && lastProposal.Number().Cmp(c.current.Sequence()) >= 0 {
        c.logger.Trace("round change timeout, catch up latest sequence", "number", lastProposal.Number().Uint64())
        c.startNewRound(common.Big0)
    } else {
        c.sendNextRoundChange() // If we've received the minimum quorum size ROUND CHANGE messages, start a new round immediately.
    }
}
```

The validators will start the round change process. Once enough validators reach the RoundChange state, `startNewRound` is executed with `roundView.Round == 0`.

```go
else if num == c.valSet.MinQuorumSize() && (c.waitingForRoundChange || cv.Round.Cmp(roundView.Round) < 0) {

    // We've received the minimum quorum size ROUND CHANGE messages, start a new round immediately.
    c.startNewRound(roundView.Round)
}
```

© 2019 Trail of Bits
startNewRound will return, without effect, as:

- lastProposal.number == currentSequence - 1
- round == 0

```go
func (c *core) startNewRound(round *big.Int) {
    var logger log.Logger
    if c.current == nil {
        logger = c.logger.New("old_round", -1, "old_seq", 0)
    } else {
        logger = c.logger.New("old_round", c.current.Round(), "old_seq",
        c.current.Sequence())
    }

    roundChange := false
    // Try to get last proposal
    lastProposal, lastProposer := c.backend.LastProposal()
    if c.current == nil {
        logger.Trace("Start to the initial round")
    } else if lastProposal.Number().Cmp(c.current.Sequence()) >= 0 {
        c.sequenceMeter.Mark(new(big.Int).Add(diff, common.Big1).Int64())

        if !c.consensusTimestamp.IsZero() {
            c.consensusTimer.UpdateSince(c.consensusTimestamp)
            c.consensusTimestamp = time.Time{}
        }

        logger.Trace("Catch up latest proposal", "number",
        lastProposal.Number().Uint64(), "hash", lastProposal.Hash())
    } else if lastProposal.Number().Cmp(big.NewInt(c.current.Sequence().Int64()-1)) == 0 {
        if round.Cmp(common.Big0) == 0 {
            // same seq and round, don't need to start new round
            return
        }
    }
}
```

Figure 3: core.go#L208-L233

As a result, the proposer will not change. If the proposer never sends a PREPREPARE message, the system will be trapped.

**Exploit Scenario**

Eve is a malicious proposer. Eve never sends the PREPREPARE message and prevents the Celo blockchain from accepting new blocks.

**Recommendation**

Short term, change the proposer if they never send a PREPREPARE message and a timeout occurs.

Long term, review the IBFT state transaction to ensure that no other states can cause the system to reach an infinite loop.
28. Integer overflow allows for arbitrary priorities in stored message

Severity: Informational
Type: Data Validation
Target: backlog.go
Finding ID: TOB-CELO-28

Description
An integer overflow in backlog.toPriority allows an attacker to set arbitrary priorities for its stored messages.

toPriority computes the priorities of stored messages, according to their sequence, round, and type:

```go
func toPriority(msgCode uint64, view *istanbul.View) int64 {
    if msgCode == msgRoundChange {
        // For msgRoundChange, set the message priority based on its sequence
        return -int64(view.Sequence.Uint64() * 1000)
    }
    // FIXME: round will be reset as 0 while new sequence
    // 10 * Round limits the range of message code is from 0 to 9
    // 1000 * Sequence limits the range of round is from 0 to 99
    return -int64(view.Sequence.Uint64()*1000 + view.Round.Uint64()*10 + uint64(msgPriority[msgCode]))
}
```

Figure 1: backlog.go#L173-L182

The sequence and the round are user-controlled. Due to the lack of integer overflow protection or value range validation, the attacker can set arbitrary high priority.

Recommendation
Short term, check for overflows in backlog.Priority.

Long term, ensure that the messages are properly validated (e.g., round < 4).
29. Liveness depends on local clock synchronization

Severity: Low                      Difficulty: High
Type: Denial of Service          Finding ID: TOB-CELO-29
Target: IBFT protocol

**Description**
Nodes in Celo's consensus model treat messages differently depending on their local Unix time. Nodes relying on an insecure time mechanism can be isolated from the network.

Celo does not offer an in-protocol method of time synchronization. Users relying on an insecure time mechanism, such as NTP, can have their local time spoofed. This presents for an attacker with some network access a compelling vector to induce liveness failures.

**Exploit Scenario**
An attacker runs a rogue NTP server on the same network as some Celo nodes, advertising a time off by decades. These nodes set their local time accordingly and start ignoring messages sent by honest nodes, thinking they are out of date and inaccurate.

**Recommendation**
Short term, document this as a known issue so users are aware that they must keep accurate local time when using Celo.

Long-term, investigate moving away from consensus protocols that require a global clock.
30. Use of static constants for gas is error-prone

Severity: Informational
Type: Configuration
Target: consensus/istanbul/*, contract_comm/*
Finding ID: TOB-CELO-30

Description
The Celo codebase relies heavily on static constants for the gas given to a transaction. If the gas is insufficient, the call will always fail, and the user will not be able to adjust it.

For example, the gas provided to get the validators list is 10,000,000:

```
maxGasForGetValidators := uint64(10000000)
// TODO(asa) - Once the validator election smart contract is completed, then a more accurate gas value should be used.
_, err := contract_comm.MakeStaticCall(paramsValidatorsRegistryId, validatorsABI, "getValidators", []interface{}, &newValSetAddresses, maxGasForGetValidators, header, state)
```

Figure 1: validators.go#L120-L122

getValidators uses the D'Hondt algorithm to compute the list of validators. The number of validators is bounded by maxElectableValidators:

```
function getValidators() external view returns (address[] memory) {
    // Only members of these validator groups are eligible for election.
    uint256 numElectionGroups = maxElectableValidators;
    if (numElectionGroups > votes.list.numElements) {
        numElectionGroups = votes.list.numElements;
    }
    address[] memory electionGroups = votes.list.headN(numElectionGroups);
    // Holds the number of members elected for each of the eligible validator groups.
    uint256[] memory numMembersElected = new uint256[](electionGroups.length);
    uint256 totalNumMembersElected = 0;
    bool memberElectedInRound = true;
    // Assign a number of seats to each validator group.
    while (totalNumMembersElected < maxElectableValidators && memberElectedInRound) {
        memberElectedInRound = false;
        uint256 groupIndex = 0;
        FractionUtil.Fraction memory maxN = FractionUtil.Fraction(0, 1);
        for (uint256 i = 0; i < electionGroups.length; i = i.add(1)) {
            bool isWinningestGroupInRound = false;
            (maxN, isWinningestGroupInRound) = dHondt(maxN, electionGroups[i], numMembersElected[i]);
            if (isWinningestGroupInRound) {
                memberElectedInRound = true;
                groupIndex = i;
            }
        }
    }
```

Figure 2: Validators.sol#L629-L652
This function has a high gas cost, which can change over time. If its execution gas cost is greater than 10,000,000, the static call in validators.go will fail.

Other static gas constants exist in:

- consensus/istanbul/backend/engine.go#495
- consensus/istanbul/backend/engine.go#505
- consensus/istanbul/backend/engine.go#519
- contract_comm/currency.go#188
- contract_comm/currency.go#206
- contract_comm/currency.go#234
- contract_comm/gas_price_minimum.go#132
- contract_comm/gas_price_minimum.go#153
- contract_comm/gas_price_minimum.go#175
- contract_comm/random.go#123
- contract_comm/random.go#156
- contract_comm/random.go#170
- contract_comm/validators.go#103
- contract_comm/validators.go#122
- contract_comm/validators.go#135

Exploit Scenario
The number of validators to be elected becomes significantly high. As a result, getValidators require more than 10,000,000 gas, and the nodes are not able to fetch the validators list.

Recommendation
Short term, consider either:

- Creating a config file that will contain the gas limit
- Allowing unlimited gas cost for the calls from the system. This requires carefully reviewing the calls to prevent denial of service attacks

Long term, carefully evaluate the evolution of the gas bound of the contracts.
31. Missing error check can lead to incorrect randomness commitment

Severity: Medium
Type: Data Validation
Target: random.go
Finding ID: TOB-CELO-31

Description
A missing error check can lead to an incorrect random value being committed by a validator.

When a validator attempts to generate a new commitment, GenerateNewRandomnessAndCommitment calls the Random contract's computeCommitment function. If an error occurs during the call to computeCommitment, randomness is not updated, and err is overwritten in the next line when the VM's state database is updated. This makes it impossible for the program to ever check for and handle the original error.

This unchecked error can result in committing a random value of zero, which is later revealed and then used as a source of on-chain randomness.

```go
// GenerateNewRandomnessAndCommitment generates a new random number and a corresponding commitment.
// The random number is stored in the database, keyed by the corresponding commitment.
func GenerateNewRandomnessAndCommitment(header *types.Header, state vm.StateDB, db *ethdb.Database, seed []byte) (common.Hash, error) {
    commitment := common.Hash{
    // TODO(asa): Make an issue to not have to do this via StaticCall
    _, err := contract_comm.MakeStaticCall(params.RandomRegistryId, computeCommitmentFuncABI, "computeCommitment", []interface{}{randomness}, &commitment, gasAmount, header, state)
    err = (*db).Put(commitmentDbLocation(commitment), header.ParentHash.Bytes())
    if err != nil {
        log.Error("Failed to save last block parentHash to the database", "err", err)
    }
    return commitment, err
}
```

Figure 1: random.go#L150-L162

Exploit Scenario
When a validator attempts to generate a new commitment, the call to the Random contract's computeCommitment function fails. As a result, the validator commits a value of zero, thereby weakening the security of anything that relies on the Random contract as a source of on-chain randomness (e.g., attestation validator selection).
Recommendation
Short term, perform an error check immediately after calling the Random contract's computeCommitment function. If an error is returned, log the error and immediately return it to the caller so it can be properly handled.

Long term, add tests to the codebase that validate proper error handling in GenerateNewRandomnessAndCommitment.
32. Unhandled errors can lead to invalid node state

Severity: Undetermined  
Difficulty: High  
Type: Data Validation  
Finding ID: TOB-CELO-32  
Target: Multiple Go files

Description
The Celo Go codebase does not consistently perform error checking, which can lead to nodes entering invalid states.

While none of the identified instances in Figure 1 were found to be exploitable, missing error handling is dangerous and can lead to new issues being introduced in the future.

The instances listed in Figure 1 were found by running `gosec` on the in-scope Go source files.

- accounts/keystore/key.go#L204
- accounts/keystore/key.go#L201
- miner/worker.go#L511
- miner/worker.go#L653
- miner/worker.go#L980
- miner/worker.go#L985
- miner/worker.go#L1062
- accounts/keystore/key.go#L200
- core/types/block.go#L122
- core/types/block.go#L359
- core/types/transaction.go#L218
- light/lightchain.go#L174
- light/txpool.go#L215
- light/txpool.go#L478
- light/txpool.go#L551
- light/txpool.go#L554
- light/txpool.go#L564
- core/blockchain.go#L199
- core/blockchain.go#L744
- core/blockchain.go#L974
- core/blockchain.go#L993
- core/blockchain.go#L1495
- core/chain_indexer.go#L463
- core/chain_indexer.go#L492
- core/chain_indexer.go#L501
- core/genesis.go#L266
- core/genesis.go#L267
- core/headerchain.go#L182
- core/headerchain.go#L511
- core/tx_pool.go#L485
- core/tx_pool.go#L963
- core/tx_pool.go#L1184
- core/tx_pool.go#L1191
- p2p/discover/udp.go#L292
- p2p/discover/udp.go#L333
- p2p/discover/udp.go#L343
Exploit Scenario
A modification to the codebase makes one of the identified missing error checks exploitable, resulting in one or more nodes entering an invalid state.

Recommendation
Short term, perform consistent error handling. If a failed operation would result in an invalid node state, divert program control flow and return early.
Long term, add documentation and testing for node error handling and crash recovery strategies. In addition, consider adding `gosec` to Celo’s continuous integration pipeline.
33. Proposed blocks can be out of sequence

Severity: Undetermined
Type: Data Validation
Finding ID: TOB-CELO-33
Target: preprepare.go

Description
When broadcasting a PREPREPARE message, the sendPreprepare function first validates that the proposed block's number matches the message's sequence number. This prevents another node from broadcasting an out-of-order block. However, a similar check does not exist in handlePreprepare for the message processing from other nodes. This makes it possible for a malicious node to broadcast proposals for out-of-order blocks, which does not conform to the protocol that Celo intends to implement.

```go
func (c *core) sendPreprepare(request *istanbul.Request) {
    logger := c.logger.New("state", c.state)

    // If I'm the proposer and I have the same sequence with the proposal
    if c.current.Sequence().Cmp(request.Proposal.Number()) == 0 && c.isProposer() {
        curView := c.currentView()
        preprepare, err := Encode(&istanbul.Preprepare{
            View:     curView,
            Proposal: request.Proposal,
        })
        if err != nil {
            logger.Error("Failed to encode", "view", curView)
            return
        }
        c.broadcast(&message{
            Code: msgPreprepare,
            Msg:  preprepare,
        })
    }
}
```

*Figure 1: preprepare.go#L26-46, which contains the validation that is missing from handlePreprepare.*

Exploit Scenario
A malicious proposer proposes an out-of-order block, which—if accepted—can have an undetermined impact on the state of the network.

Recommendation
Short term, perform consistent validation of all incoming messages. Do not assume client-side validation will prevent an attacker from crafting malicious messages.
Long term, add tests to the codebase that check for proper data validation in handlePreprepare.
34. Integer overflow allows for early revocation of payments

Severity: Medium
Type: Data validation
Target: Escrow.sol
Finding ID: TOB-CELO-34

Description
An integer overflow affecting the `revoke` function allows users to revoke their payments at any moment.

`revoke` allows users to revoke payment that were not already withdrawn:

```solidity
function revoke(
    address paymentId
) external
nonReentrant
returns (bool)
{
    EscrowedPayment memory payment = escrowedPayments[paymentId];
    require(payment.sender == msg.sender, "Only sender of payment can attempt to revoke payment.");
    require(
        // solhint-disable-next-line not-rely-on-time
        now >= (payment.timestamp + payment.expirySeconds),
        "Transaction not redeemable for sender yet."
    );
}
```

Figure 1: Escrow.sol#L185-L198

If `payment.timestamp + payment.expirySeconds` overflows, the user can revoke the payment at any moment. As a result, the user can create a payment with an arbitrary long expiration period and then revoke the payment before the expected deadline.

Exploit Scenario
Eve creates a payment with an expiration period of $2^{256} - 1$. She justifies the long expiration to Bob by saying that it will never expire. Bob tries to withdraw the payment, but Eve front-runs the transaction and revokes it. As a result, Bob is unable to redeem the original transaction.

Recommendation
Short term, ensure that an overflow will not occur when `revoke` is later called by checking the payment's `timestamp` and `expirySeconds` in `Escrow.transfer`.

Long term, use SafeMath for all arithmetic operations.
35. Attestation validator can add their address to any identity

Severity: Medium  Difficulty: High
Type: Data Validation  Finding ID: TOB-CELO-35
Target: Attestations.sol

**Description**
Attestation validations require only one validator. As a result, a validator can validate their own attestation request and associate their address with any SMS number.

To validate an identity, a user must go through the attestation validation process. This process requires one of the validators to confirm the user's identity using an SMS-based challenge. The validation requires only one validator to confirm the identity. If a validator is compromised or malicious, they can validate their own attestation request and successfully associate their address with any SMS number.

**Exploit Scenario**
Eve is a malicious validator. Eve calls `request` with her own address and Bob's identifier. Eve ensures that she is selected as the validator and validates the request. Eve's address is added to Bob's identity.

**Recommendation**
Short term, consider adding a minimal number (>1) of required validators.

Long term, monitor on-chain attestation validations to detect any suspicious activity.
A. Vulnerability Classifications

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Controls</td>
<td>Related to authorization of users and assessment of rights</td>
</tr>
<tr>
<td>Auditing and Logging</td>
<td>Related to auditing of actions or logging of problems</td>
</tr>
<tr>
<td>Authentication</td>
<td>Related to the identification of users</td>
</tr>
<tr>
<td>Configuration</td>
<td>Related to security configurations of servers, devices, or software</td>
</tr>
<tr>
<td>Cryptography</td>
<td>Related to protecting the privacy or integrity of data</td>
</tr>
<tr>
<td>Data Exposure</td>
<td>Related to unintended exposure of sensitive information</td>
</tr>
<tr>
<td>Data Validation</td>
<td>Related to improper reliance on the structure or values of data</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Related to causing system failure</td>
</tr>
<tr>
<td>Error Reporting</td>
<td>Related to the reporting of error conditions in a secure fashion</td>
</tr>
<tr>
<td>Patching</td>
<td>Related to keeping software up to date</td>
</tr>
<tr>
<td>Session Management</td>
<td>Related to the identification of authenticated users</td>
</tr>
<tr>
<td>Timing</td>
<td>Related to race conditions, locking, or order of operations</td>
</tr>
<tr>
<td>Undefined Behavior</td>
<td>Related to undefined behavior triggered by the program</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informational</td>
<td>The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth</td>
</tr>
<tr>
<td>Undetermined</td>
<td>The extent of the risk was not determined during this engagement</td>
</tr>
<tr>
<td>Low</td>
<td>The risk is relatively small or is not a risk the customer has indicated is important</td>
</tr>
<tr>
<td>Medium</td>
<td>Individual user information is at risk, exploitation would be bad for</td>
</tr>
<tr>
<td>Difficulty</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Undetermined</td>
<td>The difficulty of exploit was not determined during this engagement</td>
</tr>
<tr>
<td>Low</td>
<td>Commonly exploited, and public tools exist or can be scripted that exploit this flaw</td>
</tr>
<tr>
<td>Medium</td>
<td>Attackers must write an exploit or need an in-depth knowledge of a complex system</td>
</tr>
<tr>
<td>High</td>
<td>The attacker must have privileged, insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue</td>
</tr>
</tbody>
</table>
B. Code Quality 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.

Stability

- **Check for the return value of mint (stability/Exchange.sol#L144) and burn (stability/Reserve.sol#L228).** The lack of return value check might lead to unpredicted behavior in the case of a code update.

Governance

- **Document that expired proposals not queued have their deposits lost.** The loss of deposit might not be known by the proposal’s owners.
- **Add the nonReentrant modifier to execute (Governance.sol#L597), vote (Governance.sol#L534) and propose (Governance.sol#L371).** Allowing proposals to execute these functions can lead to unintended behavior.

consensus/istanbul/core/backlog.go

- **Check for nil pointers after decoding messages in storeBacklog (backlog.go#L98, backlog.go#L105).** While messages are already checked for nil pointers prior to being added to the backlog, it is a better practice to enforce the checks locally.
C. Slither delegatecall upgradeable proxy checks

The Celo codebase allows upgrading most of the contracts through the use of the delegatecall proxy pattern. This pattern is error-prone. Incorrect setup or upgrade can break the contracts.

Trail of Bits used slither-check-upgradeability with a custom script to ensure that common upgradeable mistakes were not present. The tool looks for issues related to incorrect storage memory layout and checks that there is no function ID collision between the proxy and the contracts.

```python
import logging
from slither import Slither
from slither.tools.upgradeability.compare_variables_order import compare_variables_order_proxy
from slither.tools.upgradeability.compare_function_ids import compare_function_ids
from slither.tools.upgradeability.check_initialization import check_initialization

logging.basicConfig()
logging.getLogger("Slither-check-upgradeability").setLevel(logging.INFO)
logging.getLogger("Slither").setLevel(logging.INFO)

slither = Slither('.', truffle_ignore_compile=True)

proxy = slither.get_contract_from_name('Proxy')
proxy_targets = [c.name for c in slither.contracts if proxy in c.inheritance]
proxy_targets = [c[::len('Proxy')] for c in proxy_targets]

check_initialization(slither)

for target in proxy_targets:
    print('########################################################')
    print(f'Check {target}')

    compare_function_ids(slither, target, slither, proxy.name)

    compare_variables_order_proxy(slither, target, slither, proxy.name)
```

**Figure 1: Custom script to check contract upgradeability**
Exchange needs to be initialized by `initialize(address,address,uint256,uint256,uint256,uint256,uint256,uint256)`
GoldToken needs to be initialized by `initialize()`
GasCurrencyWhitelist needs to be initialized by `initialize()`
Validators needs to be initialized by `initialize(address,address,uint256,uint256,uint256,uint256,uint256,uint256)`
Governance needs to be initialized by `initialize(address,address,uint256,uint256,uint256,uint256,uint256,uint256)`
Registry needs to be initialized by `initialize()`
Reserve needs to be initialized by `initialize(address,uint256)`
StableToken needs to be initialized by `initialize(string,string,uint8,address,uint256,uint256,uint256)`

Figure 2: List of initialize functions that must be called per contract

Trail of Bits recommends that Celo:

- Ensure that Slither is up to date, as Trail of Bits continuously improves these upgradability checks
- Run `slither-check-upgradeability` before any deployment or upgrade of contracts
- Check the deployment scripts for correct initialize function calls
D. Property testing of LinkedList

Trail of Bits used Echidna to perform property testing of Celo’s LinkedList implementation. The following test shows how to trigger TOB-CELO-007.

```solidity
pragma solidity ^0.5.8;

contract LinkedList {
    struct Element {
        bytes32 previousKey;
        bytes32 nextKey;
        bool exists;
    }

    struct List {
        bytes32 head;
        bytes32 tail;
        uint256 numElements;
        mapping(bytes32 => Element) elements;
    }

    List list;

    /**
     * @notice Inserts an element into a doubly linked list.
     * @param key The key of the element to insert.
     * @param previousKey The key of the element that comes before the element to insert.
     * @param nextKey The key of the element that comes after the element to insert.
     */
    function insert(
        bytes32 key,
        bytes32 previousKey,
        bytes32 nextKey
    ) public {
        require(key != bytes32(0), "Key must be defined");
        require(!contains(key), "Can't insert an existing element");

        Element storage element = list.elements[key];
        element.exists = true;

        if (list.numElements == 0) {
            list.tail = key;
            list.head = key;
        } else {
            require(
                previousKey != bytes32(0) || nextKey != bytes32(0),
                "Either previousKey or nextKey must be defined"
            );

            element.previousKey = previousKey;
            element.nextKey = nextKey;

            if (previousKey != bytes32(0)) {
                require(
                    ...); // rest of the code
            }
        }
    }
}
```
contains(previousKey),
    "If previousKey is defined, it must exist in the list"
);
Element storage previousElement = list.elements[previousKey];
require(
    previousElement.nextKey == nextKey,
    "previousKey must be adjacent to nextKey"
);
previousElement.nextKey = key;
else {
    list.tail = key;
}
if (nextKey != bytes32(0)) {
    require(contains(nextKey), "If nextKey is defined, it must exist in the list");
    Element storage nextElement = list.elements[nextKey];
    require(nextElement.previousKey == previousKey, "previousKey must be adjacent to nextKey");
    nextElement.previousKey = key;
} else {
    list.head = key;
}
}
require(list.numElements + 1 >= list.numElements, "SafeMath: addition overflow");
list.numElements += 1;

/**
 * @notice Inserts an element at the tail of the doubly linked list.
 * @param key The key of the element to insert.
 */
function push(bytes32 key) public {
    insert(key, bytes32(0), list.tail);
}

/**
 * @notice Removes an element from the doubly linked list.
 * @param key The key of the element to remove.
 */
function remove(bytes32 key) public {
    Element storage element = list.elements[key];
    require(key != bytes32(0) && contains(key));
    if (element.previousKey != bytes32(0)) {
        Element storage previousElement = list.elements[element.previousKey];
        previousElement.nextKey = element.nextKey;
    } else {
        list.tail = element.nextKey;
    }
    if (element.nextKey != bytes32(0)) {
        Element storage nextElement = list.elements[element.nextKey];
        nextElement.previousKey = element.previousKey;
    } else {
        list.head = element.previousKey;
    }
    delete list.elements[key];
    require(list.numElements - 1 <= list.numElements, "SafeMath: subtraction underflow");
list.numElements -= 1;
}

/**
* @notice Updates an element in the list.
* @param key The element key.
* @param previousKey The key of the element that comes before the updated element.
* @param nextKey The key of the element that comes after the updated element.
*/
function update(
    bytes32 key,
    bytes32 previousKey,
    bytes32 nextKey
) public
{
    require(key != bytes32(0) && key != previousKey && key != nextKey && contains(key));
    remove(key);
    insert(key, previousKey, nextKey);
}

/**
* @notice Returns whether or not a particular key is present in the sorted list.
* @param key The element key.
* @return Whether or not the key is in the sorted list.
*/
function contains(bytes32 key) public view returns (bool) {
    return list.elements[key].exists;
}

/**
* @notice Returns the keys of the N elements at the head of the list.
* @param n The number of elements to return.
* @return The keys of the N elements at the head of the list.
*/
function headN(uint256 n) public view returns (bytes32[] memory) {
    require(n <= list.numElements);
    bytes32[] memory keys = new bytes32[](n);
    bytes32 key = list.head;
    for (uint256 i = 0; i < n; i++) {
        keys[i] = key;
        key = list.elements[key].previousKey;
    }
    return keys;
}

/**
* @notice Gets all element keys from the doubly linked list.
* @return All element keys from head to tail.
*/
function getKeys() public view returns (bytes32[] memory) {
    return headN(list.numElements);
}

/********************
* Echidna tests *
*******************/

function echidna_test_tail_previous_key() public returns (bool) {

function echidna_test_head_next_key() public returns (bool) {
    return list.elements[list.head].nextKey == 0;
}

function echidna_test_no_loops() public returns (bool) {
    return list.elements["test"].nextKey != "test";
}
E. Detecting correct inheritance initialization with Slither

The Celo codebase relies heavily on inheritance and the correct call of initialization functions. Some of the contracts need their derived contracts to call a specific function at initialization. For example, UserRegistry requires the contract to call setRegistry. Due to the size of the codebase, errors are possible.

Trail of Bits developed a Slither script to ensure the correct initialization throughout the codebase:

```python
from slither import Slither
slither = Slither('.', truffle_ignore_compile=True)

targets = {
    'UsingRegistry': 'setRegistry(address)',
    'Ownable': '_transferOwnership(address)'  
}

no_issue_found = True
for contract_name, function_to_call_signature in targets.items():
    contract_targeted = slither.get_contract_from_name(contract_name)
    for contract_derived in contract_targeted.derived_contracts:
        function_to_call = contract_derived.get_function_from_signature(function_to_call_signature)
        for f in contract_derived.functions:
            if not f.is_implemented:
                continue
            if f.name.startswith('initialize'):
                if not function_to_call in f.all_internal_calls():
                    print(f'\{f.canonical_name} does not call \{function_to_call}\')
                    no_issue_found = False

if no_issue_found:
    print('No issue found')
```

Figure 1: Slither script

This script will ensure that all contracts inheriting from UsingRegistry and Ownable define an initialize function that calls setRegistry and _transferOwnership. Other inheritance checks can be added to targets.