hexens × c·Labs

Jun.23

SECURITY REVIEW REPORT FOR CLABS

CONTENTS

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading web3 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Coinstats, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
15.06.2023

Audit Completion Date
14.07.2023

---

hexens × c·Labs

METHODOLOGY

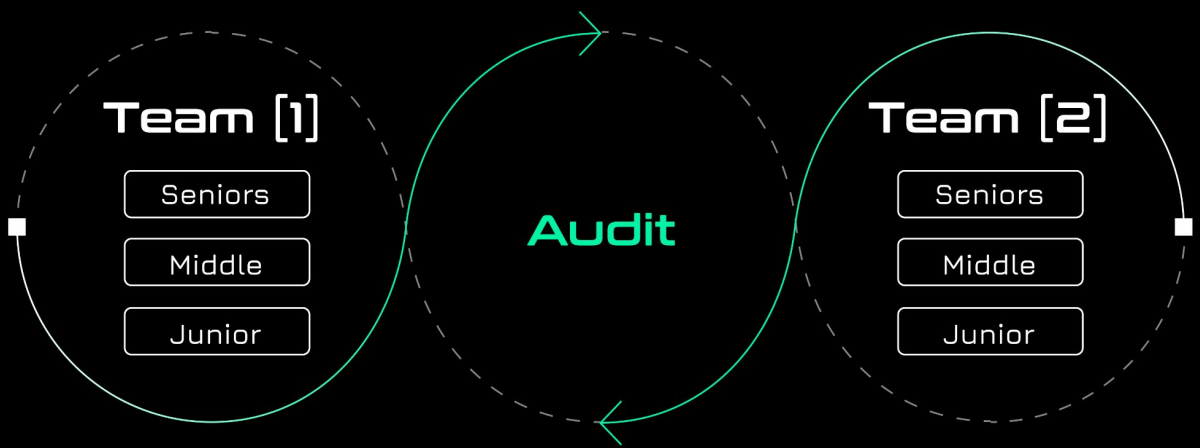
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of smart contracts. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered various pull requests for the Celo Network smart contracts as developed by cLabs. Most notably, the changes included significant changes to the Governance contracts to allow for vote delegation, as well as new fee handler contracts to handle swapping, burning and distributing of fee assets according to the Ultra Green Celo initiative.

Our security assessment was a full review of these pull requests, spanning a total of 4 weeks.

During our audit, we have identified 8 Critical severity vulnerabilities. Most of these vulnerabilities would result in an attacker being able to manipulate the Governance voting process by generating an infinite amount of voting power.

We have also identified 2 High severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality of the project have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/celo-org/celo-monorepo/issues/10375
https://alfajores.celoscan.io/token/0x2a14c01a5875315ca218145bca27a0a548361a5e#code

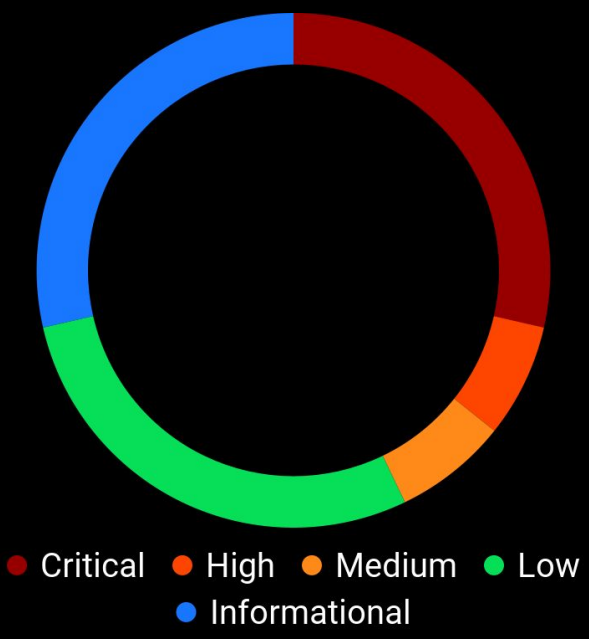
The issues described in this report were fixed in the following commit:
https://github.com/celo-org/celo-monorepo/commit/0056611df77c93acd1291db561f8b80920cb6b07
https://github.com/celo-org/celo-monorepo/pull/10437/commits/1582d398181e5ee24ad9e37f897f20b54660f7a8

SUMMARY

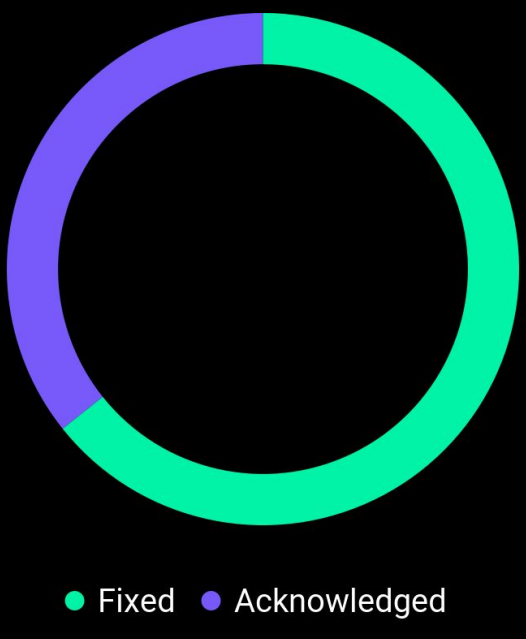
SEVERITY	NUMBER OF FINDINGS
CRITICAL	8
HIGH	2
MEDIUM	2
LOW	8
INFORMATIONAL	8

TOTAL: 28

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

CLAB-6. INCORRECT DELEGATED AMOUNT UPDATE

SEVERITY: Critical

PATH: PR #10325 LockedGold.sol:getDelegatorDelegateeExpectedAndRealAmount: L580-598

REMEDIATION: line 588 in LockedGold.sol should be replaced with:

DelegatedInfo storage currentDelegateeInfo = delegatorInfo[delegatorAccount]

STATUS: fixed

DESCRIPTION:

The function should return the expected and real amount of a delegation such that the values can be updated.

However, the function uses **delegatorInfo[delegator]** instead of **delegatorInfo[delegatorAccount]** which means that it retrieves the **DelegatedInfo** struct from the vote signer instead of the account. The vote signer's struct never gets set (as it shouldn't), but it is incorrectly used here and its values are by default 0.

Therefore, the percentage will always be 0 and as a result, the expected amount and real amount also become 0.

These incorrect amounts get saved back in **updateDelegatedAmount** in the **Delegated** struct of the actual **delegatorAccount** and **totalDelegatedCelo[delegateeAccount]** does not get lowered.

This bug can be exploited to get unlimited governance voting power. This is because the delegated amount in the delegator's struct gets set to 0, while the total delegated amount for the delegated stays the same.

So afterwards, the delegator can call the delegate function again and delegate the full amount to the delegates again. This can be repeated an arbitrary amount of times.

```
function getDelegatorDelegateeExpectedAndRealAmount(address delegator, address delegatee)
  public
  view
  returns (uint256 expected, uint256 real)
{
  address delegatorAccount = getAccounts().voteSignerToAccount(delegator);
  address delegateeAccount = getAccounts().voteSignerToAccount(delegatee);

  DelegatedInfo storage currentDelegateeInfo = delegatorInfo[delegator]
    .delegateesWithPercentagesAndAmount[delegateeAccount];

  uint256 amountToDelegate = FixidityLib
    .newFixed(getAccountTotalLockedGold(delegatorAccount))
    .multiply(FixidityLib.newFixedFraction(currentDelegateeInfo.percentage, 100))
    .fromFixed();

  expected = amountToDelegate;
  real = currentDelegateeInfo.currentAmount;
}
```

CLAB-8. UNDELEGATED VOTING POWER NOT REMOVED FROM PROPOSAL

SEVERITY: Critical

PATH: PR #10325 Governance.sol:_removeVotesWhenRevokingDelegatedVotes: L1558-1586

REMEDIATION: the function Governance.sol:_removeVotesWhenRevokingDelegatedVotes should remove the votes from both the user's vote record and the proposal's total vote tallies using proposal.update

STATUS: fixed

DESCRIPTION:

When a user revokes their delegation, the delegated votes should be removed from the delegatee. This is done from the LockedGold contract by calling **Governance.removeVotesWhenRevokingDelegatedVotes**.

However, this function only removes the votes from the user's voting record and not from the proposal's total vote tallies.

As a result, the votes get erased from the record, allowing the delegatee to vote again, while the previous votes would still be counted in the proposal's vote tallies.

The delegator can then delegate to the delegatee again, and the delegatee can vote again. This can be repeated for infinite voting power.

This results in governance result manipulation.

```
function _removeVotesWhenRevokingDelegatedVotes(address account, uint256 newVotingPower)
  internal
{
  Voter storage voter = voters[account];

  for (uint256 index = 0; index < dequeued.length; index = index.add(1)) {
    Proposals.Proposal storage proposal = proposals[dequeued[index]];
    bool isVotingReferendum = (getProposalDequeuedStage(proposal) == Proposals.Stage.Referendum);

    if (!isVotingReferendum) {
      continue;
    }

    VoteRecord storage voteRecord = voter.referendumVotes[index];
    uint256 sumOfVotes = voteRecord.yesVotes.add(voteRecord.noVotes).add(voteRecord.abstainVotes);

    if (sumOfVotes > newVotingPower) {
      uint256 toRemove = sumOfVotes - newVotingPower;

      uint256 abstrainToRemove = getVotesPortion(toRemove, voteRecord.abstainVotes, sumOfVotes);
      uint256 yesToRemove = getVotesPortion(toRemove, voteRecord.yesVotes, sumOfVotes);
      uint256 noToRemove = getVotesPortion(toRemove, voteRecord.noVotes, sumOfVotes);

      voteRecord.abstainVotes = voteRecord.abstainVotes.sub(abstrainToRemove);
      voteRecord.yesVotes = voteRecord.yesVotes.sub(yesToRemove);
      voteRecord.noVotes = voteRecord.noVotes.sub(noToRemove);
    }
  }
}
```

CLAB-9. REVOKE DELEGATION VOTE AMOUNT UNDERFLOW

SEVERITY: Critical

PATH: PR #10325 LockedGold.sol:revokeDelegatedGovernanceVotes:L414-478

REMEDIATION: replace the unsafe subtraction operator with the safe equivalent in OpenZeppelin's SafeMath library

STATUS: fixed

DESCRIPTION:

The function **revokeDelegatedGovernanceVotes** is responsible for revoking some or all of a delegation. If the delegatee has used the delegated power as voting power, then this should also be removed from the Governance contract.

On line 453, the unused votes are calculated:

```
uint256 unusedReferendumVotes = delegateeTotalVotingPower - totalReferendumVotes;
```

However, because Solidity 0.5.13 is used and SafeMath is not used here, underflow can happen if the total record votes is greater than the total voting power. This situation is possible by rounding error due to the usage of percentages and fixed point math.

When **unusedReferendumVotes** underflows, the following conditional on line 454 is skipped and consequently the vote removal is skipped:

```
if (unusedReferendumVotes < amountToRevoke) {
  getGovernance().removeVotesWhenRevokingDelegatedVotes(
    delegateeAccount,
    delegateeTotalVotingPower - amountToRevoke
  );
}
```

This vulnerability can be exploited for infinite voting power using the following proof of concept:

1. A delegates 100% of 1 ETH to B.
2. B votes partially for proposal, e.g. (1 ETH - 1 Wei, 1 Wei, 0).
3. A revokes 1% of delegation, this backs out (0.01 ETH, 0, 0) because 1% of 1 Wei is 0.
4. B now has total voting power of 0.99 ETH and total votes of 0.99 ETH + 1 Wei.
5. A revokes 99% of delegation, **unusedReferendumVotes** overflows and this causes it to skip the backing out of the votes on lines 454-459 leaving the votes in.
6. A can now delegate full balance to C and repeat.

```
function revokeDelegatedGovernanceVotes(address delegatee, uint256 percentageToRevoke) public {
  require(
    percentageToRevoke > 0 && percentageToRevoke <= 100,
    "revoked percents can be only between 1%..100%"
  );

  address delegatorAddress = getAccounts().voteSignerToAccount(msg.sender);
  Delegated storage delegated = delegatorInfo[delegatorAddress];
  require(
    delegated.totalDelegatedCeloInPercents >= percentageToRevoke,
    "Not enough total delegated percents"
  );

  address delegateeAccount = getAccounts().voteSignerToAccount(delegatee);
  updateDelegatedAmount(msg.sender, delegatee);

  DelegatedInfo storage currentDelegateeInfo = delegated
    .delegateesWithPercentagesAndAmount[delegateeAccount];

  require(currentDelegateeInfo.percentage >= percentageToRevoke, "Not enough delegated percents");

  currentDelegateeInfo.percentage = currentDelegateeInfo.percentage.sub(percentageToRevoke);

  uint256 delegateeTotalVotingPower = getAccountTotalGovernanceVotingPower(delegateeAccount);
  uint256 totalReferendumVotes = getGovernance().getAmountOfGoldUsedForVoting(delegateeAccount);

  uint256 totalLockedGold = getAccountTotalLockedGold(delegatorAddress);

  // in a worst case we will remove whole amount of delegator (even when amount is not updated)
  uint256 amountToRevoke = currentDelegateeInfo.percentage == 0
    ? currentDelegateeInfo.currentAmount
    : Math.min(
      FixidityLib
        .newFixed(totalLockedGold)
        .multiply(FixidityLib.newFixedFraction(percentageToRevoke, 100))
        .fromFixed(),
      currentDelegateeInfo.currentAmount
    );
```

```
uint256 unusedReferendumVotes = delegateeTotalVotingPower - totalReferendumVotes;
 if (unusedReferendumVotes < amountToRevoke) {
  getGovernance().removeVotesWhenRevokingDelegatedVotes(
    delegateeAccount,
    delegateeTotalVotingPower - amountToRevoke
  );
 }

 delegated.totalDelegatedCeloInPercents = delegated.totalDelegatedCeloInPercents.sub(
  percentageToRevoke
 );

 currentDelegateeInfo.currentAmount = currentDelegateeInfo.currentAmount.sub(amountToRevoke);
 totalDelegatedCelo[delegateeAccount] -= amountToRevoke;

 if (currentDelegateeInfo.currentAmount == 0) {
  delegated.delegatees.remove(delegateeAccount);
 }

 emit CeloDelegatedRevoked(
  delegatorAddress,
  delegateeAccount,
  percentageToRevoke,
  amountToRevoke
 );
}
```

CLAB-21. DELEGATION INCORRECTLY REMOVED WHEN UNLOCKING

SEVERITY: Critical

PATH: PR #10325 LockedGold.sol:revokeFromDelegatedWhenUnlocking:L483-510

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

revokeFromDelegatedWhenUnlocking is called from **unlock** to remove any delegated amounts when unlocking gold.

The function should only remove any delegated amount from the delegatee and delegated voting power from the Governance contract that was used by the delegatee.

However, the function also incorrectly removes any delegation if the new delegated power is 0, while a delegated percentage still exists in the mapping.

As a result, the delegatee is skipped in any loops over a delegator's delegatees. For example in **_updateDelegatedAmount**, while the user can still call it manually as the percentage is still set.

This can be exploited for infinite voting power using the following proof of concept:

1. A locks some gold
2. A delegates 100% to B
3. A unlocks all gold, this removes the delegation link and voting power of B
4. A relocks the gold from the queue, this skips the update of delegated amount to B
5. B can call **updateDelegatedAmount(A, B)** and the full voting power is added back to B
6. A unlocks all gold again, this skips B and does not remove the voting power of B.
7. A withdraws the gold and transfer it to C.
8. Repeat.

```
function revokeFromDelegatedWhenUnlocking(address delegator, uint256 amountToRevoke) private {
  address[] memory delegatees = getDelegateesOfDelegator(delegator);

  Delegated storage delegated = delegatorInfo[delegator];

  FixidityLib.Fraction memory amountToRevokeOnePercent = FixidityLib.newFixedFraction(
    amountToRevoke,
    100
  );

  for (uint256 i = 0; i < delegatees.length; i = i.add(1)) {
    DelegatedInfo storage currentDelegateeInfo = delegated
      .delegateesWithPercentagesAndAmount[delegatees[i]];
    (uint256 expected, uint256 real) = _getDelegatorDelegateeExpectedAndRealAmount(
      delegator,
      delegatees[i]
    );
    uint256 delegateeAmountToRevoke = amountToRevokeOnePercent
      .multiply(FixidityLib.newFixed(currentDelegateeInfo.percentage))
      .fromFixed();
    delegateeAmountToRevoke = delegateeAmountToRevoke.sub(expected.sub(real));
    _decreaseDelegateeVotingPower(delegatees[i], delegateeAmountToRevoke, currentDelegateeInfo);
    if (currentDelegateeInfo.currentAmount == 0) {
      delegated.delegatees.remove(delegatees[i]);
    }
    emit CeloDelegatedRevoked(delegator, delegatees[i], 0, delegateeAmountToRevoke);
  }
}
```

The vulnerability can be fixed by simply removing lines 527-529, which is the conditional that removes the delegation from the array when unlocking:

```
if (currentDelegateeInfo.currentAmount == 0) {
  delegated.delegatees.remove(delegatees[i]);
}
```

This is because a delegation should never be removed when unlocking gold. Instead, only the delegated power should be decreased, which is already done in the function.

CLAB-23. DELEGATION INCORRECTLY REMOVED WHEN REVOKING A ZERO AMOUNT DELEGATION

SEVERITY: Critical

PATH: PR #10325
LockedGold.sol:revokeDelegatedGovernanceVotes:L425-476

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

This issue is similar to CLAB-21, but the root cause is different.

The function **revokeDelegatedGovernanceVotes** is responsible for partially or fully removing any delegations. The function also incorrectly removes a delegation if the delegated amount is 0, while the percentage is still set in the mapping.

On lines 466-468 the delegation is removed from the set, but this is done using a conditional on the actual delegated amount instead of the percentage. If the delegator's balance is 0, then the delegated amount will also be 0 even though the percentage can still be 100.

If the delegator has no locked gold and delegates a 100% to a delegatee. They can then revoke 1% and the delegation will be removed from the set, but the struct still keeps 99%.

The delegator can then lock some gold and call **updateDelegatedAmount** (which is still valid, because it only checks the percentage) and it will give 99% of the amount as voting power to the delegatee.

Unlocking and withdrawing would not remove this because the delegatee is not in the set.

This can then be repeated for infinite voting power.

```
function revokeDelegatedGovernanceVotes(address delegatee, uint256 percentageToRevoke) public {
  require(
    percentageToRevoke != 0 && percentageToRevoke <= 100,
    "revoked percents can be only between 1%..100%"
  );

  address delegatorAccount = getAccounts().voteSignerToAccount(msg.sender);
  Delegated storage delegated = delegatorInfo[delegatorAccount];
  require(
    delegated.totalDelegatedCeloInPercents >= percentageToRevoke,
    "Not enough total delegated percents"
  );

  address delegateeAccount = getAccounts().voteSignerToAccount(delegatee);
  _updateDelegatedAmount(delegatorAccount, delegateeAccount);

  DelegatedInfo storage currentDelegateeInfo = delegated
    .delegateesWithPercentagesAndAmount[delegateeAccount];

  require(currentDelegateeInfo.percentage >= percentageToRevoke, "Not enough delegated percents");

  currentDelegateeInfo.percentage = currentDelegateeInfo.percentage.sub(percentageToRevoke);

  uint256 totalLockedGold = getAccountTotalLockedGold(delegatorAccount);

  uint256 amountToRevoke = currentDelegateeInfo.percentage == 0
    ? currentDelegateeInfo.currentAmount
    : Math.min(
      FixidityLib
        .newFixed(totalLockedGold)
        .multiply(FixidityLib.newFixedFraction(percentageToRevoke, 100))
        .fromFixed(),
      currentDelegateeInfo.currentAmount
    );
```

```
  _decreaseDelegateeVotingPower(delegateeAccount, amountToRevoke, currentDelegateeInfo);

  delegated.totalDelegatedCeloInPercents = delegated.totalDelegatedCeloInPercents.sub(
    percentageToRevoke
  );

  if (currentDelegateeInfo.currentAmount == 0) {
    delegated.delegatees.remove(delegateeAccount);
  }

  emit CeloDelegatedRevoked(
    delegatorAccount,
    delegateeAccount,
    percentageToRevoke,
    amountToRevoke
  );
}
```

The vulnerability can be fixed by changing the conditional on line 492 to check the delegated percentage instead of the amount:

```
if (currentDelegateeInfo.percentage == 0) {
  delegated.delegatees.remove(delegateeAccount);
}
```

This will be in line with the use case of delegating a zero amount of gold, where the percentage is higher than zero and represents the existence of a delegation, and the amount can be zero or anything else.

CLAB-25. INACTIVE TOKENS CAN STILL BE PROCESSED IN FEEHANDLER

SEVERITY: Critical

PATH: PR #10227
FeeHandler.sol

REMEDIATION: add checks whether a token is marked as active in the handle and distribute functions

STATUS: acknowledged

DESCRIPTION:

The FeeHandler contract manages the selling, distributing and burning of fees. It requires tokens to be added and activated before they can be processed.

After a token is deactivated, it gets removed from the **activeTokens** set such that it won't be iterated on in **handleAll**.

However, the token can still be sold, distributed and burned by manually calling the **handle** or **distribute** function on the token because it still keeps a valid **tokenState**.

The functions are missing checks whether the given token is active and therefore inactive tokens could still be processed.

```
function handle(address tokenAddress) external {
  return _handle(tokenAddress);
}

function _handle(address tokenAddress) private {
  // Celo doesn't have to be exchanged for anything
  if (tokenAddress != registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID)) {
    _sell(tokenAddress);
  }
  _burnCelo();
  _distribute(tokenAddress);
  _distribute(registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID));
}

/**
  @dev Distributes the available tokens for the specified token address to the fee beneficiary.
  @param tokenAddress The address of the token for which to distribute the available tokens.
*/
function distribute(address tokenAddress) external {
  return _distribute(tokenAddress);
}

function _distribute(address tokenAddress) private onlyWhenNotFrozen nonReentrant {
  require(feeBeneficiary != address(0), "Can't distribute to the zero address");
  IERC20 token = IERC20(tokenAddress);
  uint256 tokenBalance = token.balanceOf(address(this));

  TokenState storage tokenState = tokenStates[tokenAddress];

  // safty check to avoid a revert due balance
  uint256 balanceToDistribute = Math.min(tokenBalance, tokenState.toDistribute);

  if (balanceToDistribute == 0) {
    // don't distribute with zero balance
    return;
  }

  token.transfer(feeBeneficiary, balanceToDistribute);
  tokenState.toDistribute = tokenState.toDistribute.sub(balanceToDistribute);
}
```

CLAB-26. WRONG CALCULATION FOR TOKEN DISTRIBUTION FOR SELL

SEVERITY: Critical

PATH: PR #10227
FeeHandler.sol:_sell:L295-335

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In the **_sell** function of the contract **FeeHandler.sol**, the calculation of **tokenState.toDistribute** should be corrected to ensure proper functionality. Currently, on line 306, it is calculated such that the value is doubled on every call:

```
tokenState.toDistribute += tokenState.toDistribute.add(
  balanceToProcess.fromFixed().sub(balanceToBurn)
);
```

This introduces a vulnerability that inflates the distribution value, which can lead to much funds being distributed or DoS on the core protocol.

A malicious actor can exploit this issue by repeatedly calling the **sell** function, when available, causing the distribution amount to double each time. Eventually, this will lead to incorrect calculations and cause the subtraction operations (**sub(tokenState.toDistribute)**) to revert, such as in line 301. As a result, it will become impossible to call the **sell** function.

```
function _sell(address tokenAddress) private onlyWhenNotFrozen nonReentrant {
  IERC20 token = IERC20(tokenAddress);

  TokenState storage tokenState = tokenStates[tokenAddress];
  require(tokenState.handler != address(0), "Handler has to be set to sell token");
  FixidityLib.Fraction memory balanceToProcess = FixidityLib.newFixed(
    token.balanceOf(address(this)).sub(tokenState.toDistribute)
  );

  uint256 balanceToBurn = (burnFraction.multiply(balanceToProcess)).fromFixed();

  tokenState.toDistribute += tokenState.toDistribute.add(
    balanceToProcess.fromFixed().sub(balanceToBurn)
  );

  // small numbers cause rounding errors and zero case should be skipped
  if (balanceToBurn < MIN_BURN) {
    return;
  }

  if (dailySellLimitHit(tokenAddress, balanceToBurn)) {
    // in case the limit is hit, burn the max possible
    balanceToBurn = tokenState.currentDaySellLimit;
    emit DailyLimitHit(tokenAddress, balanceToBurn);
  }

  token.transfer(tokenState.handler, balanceToBurn);
  IFeeHandlerSeller handler = IFeeHandlerSeller(tokenState.handler);

  handler.sell(
    tokenAddress,
    registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID),
    balanceToBurn,
    FixidityLib.unwrap(tokenState.maxSlippage)
  );

  tokenState.pastBurn = tokenState.pastBurn.add(balanceToBurn);
  updateLimits(tokenAddress, balanceToBurn);

  emit SoldAndBurnedToken(tokenAddress, balanceToBurn);
}
```

Change lines 306-308 to:

```
tokenState.toDistribute = tokenState.toDistribute.add(
  balanceToProcess.fromFixed().sub(balanceToBurn)
);
```

CLAB-29. INCORRECT CALCULATION OF BURN FRACTION AND DISTRIBUTION AMOUNT

SEVERITY: Critical

PATH: PR #10227
FeeHandler.sol

REMEDIATION: convert all non-CELO tokens to CELO in _sell and only divide the total CELO amount into burn and distribution in the _burnCelo function

STATUS: fixed

DESCRIPTION:

In the FeeHandler, the sell function handles selling any other token into CELO. This function already takes a burn fraction into account (e.g. 80%) and thus adds 20% to **tokenState.toDistribute.**

Afterwards, in the **_burnCelo** function, all CELO is burned and this function also takes the burn fraction into account.

The previous 80% that was converted to CELO would get divided again and consequently only 64% would actually get burned, while 36% gets distributed.

Because both functions take the burn fraction into account, the distribution calculation is done twice and too much is distributed and too little is burned each time.

```
function handle(address tokenAddress) external {
  return _handle(tokenAddress);
}

function _handle(address tokenAddress) private {
  // Celo doesn't have to be exchanged for anything
  if (tokenAddress != registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID)) {
    _sell(tokenAddress);
  }
  _burnCelo();
  _distribute(tokenAddress);
  _distribute(registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID));
}

function _sell(address tokenAddress) private onlyWhenNotFrozen nonReentrant {
  IERC20 token = IERC20(tokenAddress);

  TokenState storage tokenState = tokenStates[tokenAddress];
  require(tokenState.handler != address(0), "Handler has to be set to sell token");
  FixidityLib.Fraction memory balanceToProcess = FixidityLib.newFixed(
    token.balanceOf(address(this)).sub(tokenState.toDistribute)
  );

  uint256 balanceToBurn = (burnFraction.multiply(balanceToProcess).fromFixed());

  tokenState.toDistribute += tokenState.toDistribute.add(
    balanceToProcess.fromFixed().sub(balanceToBurn)
  );

  // small numbers cause rounding errors and zero case should be skipped
  if (balanceToBurn < MIN_BURN) {
    return;
  }

  if (dailySellLimitHit(tokenAddress, balanceToBurn)) {
    // in case the limit is hit, burn the max possible
    balanceToBurn = tokenState.currentDaySellLimit;
    emit DailyLimitHit(tokenAddress, balanceToBurn);
  }
```

```
  token.transfer(tokenState.handler, balanceToBurn);
  IFeeHandlerSeller handler = IFeeHandlerSeller(tokenState.handler);

  handler.sell(
    tokenAddress,
    registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID),
    balanceToBurn,
    FixidityLib.unwrap(tokenState.maxSlippage)
  );

  tokenState.pastBurn = tokenState.pastBurn.add(balanceToBurn);
  updateLimits(tokenAddress, balanceToBurn);

  emit SoldAndBurnedToken(tokenAddress, balanceToBurn);
}

function _distribute(address tokenAddress) private onlyWhenNotFrozen nonReentrant {
  require(feeBeneficiary != address(0), "Can't distribute to the zero address");
  IERC20 token = IERC20(tokenAddress);
  uint256 tokenBalance = token.balanceOf(address(this));

  TokenState storage tokenState = tokenStates[tokenAddress];

  // safty check to avoid a revert due balance
  uint256 balanceToDistribute = Math.min(tokenBalance, tokenState.toDistribute);

  if (balanceToDistribute == 0) {
    // don't distribute with zero balance
    return;
  }

  token.transfer(feeBeneficiary, balanceToDistribute);
  tokenState.toDistribute = tokenState.toDistribute.sub(balanceToDistribute);
}
```

CLAB-19. PROPOSAL QUEUE UPVOTES NOT BACKED OUT

SEVERITY: High

PATH: PR #10325 Governance.sol:_removeVotesWhenRevokingDelegatedVotes: L1547-1610

REMEDIATION: the function _removeVotesWhenRevokingDelegatedVotes should also check the user's upvote record and accordingly update the queue by removing any votes for the proposal that the user upvoted

STATUS: fixed

DESCRIPTION:

Revoking delegation will call **_removeVotesWhenRevokingDelegatedVotes** with the new amount of voting power for any delegatee that had their delegated amount decreased.

However, this function does not back out the **weight** of the user's upvote. This means that by voting, redelegating to another account and voting again, the user can infinitely increase the **weight** of a proposal in the queue.

Using this vulnerability, a malicious user could always push their proposal to the head of the queue and permanently block normal proposals from going through.

```
function _removeVotesWhenRevokingDelegatedVotes(address account, uint256 newVotingPower)
  internal
{
  Voter storage voter = voters[account];

  for (uint256 index = 0; index < dequeued.length; index = index.add(1)) {
    Proposals.Proposal storage proposal = proposals[dequeued[index]];
    bool isVotingReferendum = (getProposalDequeuedStage(proposal) == Proposals.Stage.Referendum);

    if (!isVotingReferendum) {
      continue;
    }

    VoteRecord storage voteRecord = voter.referendumVotes[index];
    uint256 sumOfVotes = voteRecord.yesVotes.add(voteRecord.noVotes).add(voteRecord.abstainVotes);

    if (sumOfVotes > newVotingPower) {
      uint256 toRemove = sumOfVotes.sub(newVotingPower);

      uint256 abstainToRemove = getVotesPortion(toRemove, voteRecord.abstainVotes, sumOfVotes);
      uint256 yesToRemove = getVotesPortion(toRemove, voteRecord.yesVotes, sumOfVotes);
      uint256 noToRemove = getVotesPortion(toRemove, voteRecord.noVotes, sumOfVotes);

      uint256 totalRemoved = abstainToRemove.add(yesToRemove).add(noToRemove);

      uint256 yesVotes = voteRecord.yesVotes.sub(yesToRemove);
      uint256 noVotes = voteRecord.noVotes.sub(noToRemove);
      uint256 abstainVotes = voteRecord.abstainVotes.sub(abstainToRemove);

      if (totalRemoved < toRemove) {
        // in case of rounding error
        uint256 roundingToRemove = toRemove.sub(totalRemoved);

        uint256 toRemoveRounding = Math.min(roundingToRemove, yesVotes);
        yesVotes = yesVotes.sub(toRemoveRounding);
        roundingToRemove = roundingToRemove.sub(toRemoveRounding);

        if (roundingToRemove > 0) {
          toRemoveRounding = Math.min(roundingToRemove, noVotes);
          noVotes = noVotes.sub(toRemoveRounding);
          roundingToRemove = roundingToRemove.sub(toRemoveRounding);
        }
```

```
      if (roundingToRemove > 0) {
        toRemoveRounding = Math.min(roundingToRemove, abstainVotes);
        abstainVotes = abstainVotes.sub(toRemoveRounding);
      }
    }

    proposal.updateVote(
      voteRecord.yesVotes,
      voteRecord.noVotes,
      voteRecord.abstainVotes,
      yesVotes,
      noVotes,
      abstainVotes
    );

    voteRecord.abstainVotes = abstainVotes;
    voteRecord.yesVotes = yesVotes;
    voteRecord.noVotes = noVotes;
  }
 }
}
```

CLAB-30. PROPOSAL DEQUEUEING IGNORES VOTING QUEUE ORDER

SEVERITY: High

PATH: PR #10268 Governance.sol:dequeueProposalIfReady:L1258-1294

REMEDIATION: remove or reimplement this function as it currently completely breaks the queue of the Governance contract

STATUS: fixed

DESCRIPTION:

PR #10268 and PR #10324 introduce a new function **dequeueProposalIfReady(uint proposalId)** for the **Governance.sol** contract. The function is only called in **Governance.sol:upvote** on line 536 with the given **proposalId** and it can be called externally as it's marked **public**.

However, the function only checks if the dequeue time has been reached (e.g. last dequeue time + dequeue frequency) and if the given proposal has not yet expired. It does not take the queue order that is established through upvoting with voting power into account.

In contrary to **dequeueProposalsIfReady**, which pops the head proposals of the queue, it also only supports dequeueing a single proposal each time. It will set **lastDequeue** time and consequently block any other proposals from getting dequeued.

This vulnerability can be used to instantly dequeue a proposal without the need for upvotes, as well as block any normal proposals from getting dequeued by repeatedly proposing and dequeueing malicious or bogus proposals.

```
function dequeueProposalIfReady(uint256 proposalId) public returns (bool isProposalDequeued) {
  isProposalDequeued = false;
  // solhint-disable-next-line not-rely-on-time
  if (block.timestamp >= lastDequeue.add(dequeueFrequency)) {
    Proposals.Proposal storage proposal = proposals[proposalId];

    if (_isQueuedProposalExpired(proposal)) {
      emit ProposalExpired(proposalId);
      return isProposalDequeued;
    }

    // Updating refunds back to proposer
    refundedDeposits[proposal.proposer] = refundedDeposits[proposal.proposer].add(
      proposal.deposit
    );
    queue.remove(proposalId);
    // solhint-disable-next-line not-rely-on-time
    proposal.timestamp = block.timestamp;
    if (emptyIndices.length > 0) {
      uint256 indexOfLastEmptyIndex = emptyIndices.length.sub(1);
      dequeued[emptyIndices[indexOfLastEmptyIndex]] = proposalId;
      delete emptyIndices[indexOfLastEmptyIndex];
      emptyIndices.pop();
    } else {
      dequeued.push(proposalId);
    }

    // solhint-disable-next-line not-rely-on-time
    emit ProposalDequeued(proposalId, block.timestamp);
    isProposalDequeued = true;

    // solhint-disable-next-line not-rely-on-time
    lastDequeue = block.timestamp;
  }

  return isProposalDequeued;
}
```

CLAB-16. INCORRECT TOTAL DELEGATION WHEN GOLD IS ZERO

SEVERITY: Medium

PATH: PR #10325 LockedGold.sol:delegateGovernanceVotes:L331-396

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **delegateGovernanceVotes** allows a user to delegate a given percentage of their locked gold to another user. If the delegator has 0 locked gold, they can still prepare their delegations.

It handles this case specially with a conditional on lines 370-383.

However, it incorrectly sets the total delegation percentage by summing the new percentage and not subtracting the old percentage.

Therefore, if someone with 0 locked gold sets a percentage of 50% and later changes this 100%, the **totalDelegatedCeloFraction** gets set to 150%.

This leads to the user not able to delegate anymore as the check for a maximum of 100% will always fail.

```
function delegateGovernanceVotes(address delegatee, uint256 delegateFraction) external {
  FixidityLib.Fraction memory percentageToDelegate = FixidityLib.wrap(delegateFraction);
  require(
    FixidityLib.lte(percentageToDelegate, FixidityLib.fixed1()),
    "Delegate fraction must be less than or equal to 1"
  );
  address delegatorAccount = getAccounts().voteSignerToAccount(msg.sender);
  address delegateeAccount = getAccounts().voteSignerToAccount(delegatee);

  IValidators validators = getValidators();
  require(!validators.isValidator(delegatorAccount), "Validators cannot delegate votes.");
  require(
    !validators.isValidatorGroup(delegatorAccount),
    "Validator groups cannot delegate votes."
  );

  Delegated storage delegated = delegatorInfo[delegatorAccount];
  delegated.delegatees.add(delegateeAccount);
  require(delegated.delegatees.length() <= maxDelegateesCount, "Too many delegatees");

  DelegatedInfo storage currentDelegateeInfo = delegated
    .delegateesWithPercentagesAndAmount[delegateeAccount];

  require(
    FixidityLib.gte(percentageToDelegate, currentDelegateeInfo.percentage),
    "Cannot decrease delegated amount - use revokeDelegatedGovernanceVotes."
  );

  FixidityLib.Fraction memory requestedToDelegate = delegated
    .totalDelegatedCeloFraction
    .subtract(currentDelegateeInfo.percentage)
    .add(percentageToDelegate);

  require(
    FixidityLib.lte(requestedToDelegate, FixidityLib.fixed1()),
    "Cannot delegate more than 100%"
  );

  uint256 totalLockedGold = getAccountTotalLockedGold(delegatorAccount);
  if (totalLockedGold == 0) {
    currentDelegateeInfo.percentage = percentageToDelegate;
    delegated.totalDelegatedCeloFraction = delegated.totalDelegatedCeloFraction.add(
      percentageToDelegate
    );
```

```
    emit CeloDelegated(
      delegatorAccount,
      delegateeAccount,
      FixidityLib.unwrap(percentageToDelegate),
      currentDelegateeInfo.currentAmount
    );
    return;
  }

  uint256 totalReferendumVotes = getGovernance().getAmountOfGoldUsedForVoting(delegatorAccount);

  if (totalReferendumVotes > 0) {
    FixidityLib.Fraction memory referendumVotesInPercents = FixidityLib.newFixedFraction(
      totalReferendumVotes,
      totalLockedGold
    );
    require(
      FixidityLib.lte(referendumVotesInPercents.add(requestedToDelegate), FixidityLib.fixed1()),
      "Cannot delegate votes that are voting in referendum"
    );
  }

  // amount that will really be delegated - whatever is already
  // delegated to this particular delagatee is already subracted from this
  uint256 amountToDelegate = FixidityLib
    .newFixed(totalLockedGold)
    .multiply(percentageToDelegate)
    .subtract(FixidityLib.newFixed(currentDelegateeInfo.currentAmount))
    .fromFixed();

  delegated.totalDelegatedCeloFraction = delegated
    .totalDelegatedCeloFraction
    .subtract(currentDelegateeInfo.percentage)
    .add(percentageToDelegate);
  currentDelegateeInfo.percentage = percentageToDelegate;

  currentDelegateeInfo.currentAmount = currentDelegateeInfo.currentAmount.add(amountToDelegate);
  totalDelegatedCelo[delegateeAccount] = totalDelegatedCelo[delegateeAccount].add(
    amountToDelegate
  );

  emit CeloDelegated(
    delegatorAccount,
    delegateeAccount,
    FixidityLib.unwrap(percentageToDelegate),
    currentDelegateeInfo.currentAmount
  );
}
```

In LockedGold.sol, lines 372-374 should be replaced with:

```
delegated.totalDelegatedCeloFraction = delegated.totalDelegatedCeloFraction
  .subtract(currentDelegateeInfo.percentage)
  .add(percentageToDelegate);
```

CLAB-28. INCORRECT MINIMAL LIMIT OF BURNING TOKENS

SEVERITY: Medium

PATH: PR #10227
FeeHandler.sol:_sell:L295-335

REMEDIATION: in the TokenState struct, we suggest adding a variable called 'minBurn' that can be set individually for each token. This will allow for a more flexible approach, ensuring that tokens with different decimal places can have their own appropriate minimal limits

STATUS: acknowledged

DESCRIPTION:

In the **_sell** function, there is a check that verifies if **balanceToBurn** is less than **MIN_BURN**, which is set to 200.

However, **MIN_BURN** is a static and constant argument that is applied the same to all tokens, regardless of their decimal places.

This results in an incorrect minimal limit, where tokens such as USDT with 6 decimal places and BUSD with 18 decimal places have the same minimal limit.

```
function _sell(address tokenAddress) private onlyWhenNotFrozen nonReentrant {
    IERC20 token = IERC20(tokenAddress);

    TokenState storage tokenState = tokenStates[tokenAddress];
    require(tokenState.handler != address(0), "Handler has to be set to sell token");
    FixidityLib.Fraction memory balanceToProcess = FixidityLib.newFixed(
      token.balanceOf(address(this)).sub(tokenState.toDistribute)
    );

    uint256 balanceToBurn = (burnFraction.multiply(balanceToProcess).fromFixed());

    tokenState.toDistribute += tokenState.toDistribute.add(
      balanceToProcess.fromFixed().sub(balanceToBurn)
    );
```

CLAB-1. DUPLICATE ARRAY LENGTH CHECK IN FEEHANDLER

SEVERITY: Low

PATH: PR #10227
FeeHandler.sol:initialize:L92-122

REMEDIATION: remove the duplicate check

STATUS: fixed

DESCRIPTION:

The **initialize** function has multiple checks for the lengths of array parameters.

However, it has a duplicate check for **tokens.length == newMaxSlippages.length** on lines 103-106 and 107-110. The second check has a different error message, but it effectively checks the same.

```
function initialize(
    address _registryAddress,
    address newFeeBeneficiary,
    uint256 newBurnFraction,
    address[] calldata tokens,
    address[] calldata handlers,
    uint256[] calldata newLimits,
    uint256[] calldata newMaxSlippages
  ) external initializer {
    require(tokens.length == handlers.length, "handlers length should match tokens length");
    require(tokens.length == newLimits.length, "limits length should match tokens length");
    require(
      tokens.length == newMaxSlippages.length,
      "maxSlippage length should match tokens length"
    );
    require(
      tokens.length == newMaxSlippages.length,
      "newMininumReports length should match tokens length"
    );
```

CLAB-2. FEEHANDLERSELLER SELL FUNCTION CAN BE CALLED BY ANYONE

SEVERITY: Low

PATH: PR #10227 UniswapFeeHandlerSeller.sol:sell:L147-212, MentoFeeHandlerSeller.sol:sell:L44-85

REMEDIATION: the UniswapFeeHandlerSeller and MentoFeeHandlerSeller are primarily used by the FeeHandler contract. It could be beneficial to only allow this contract (and an owner for stuck funds) to call the sell function or to change line 209 (Uniswap) and line 82 (Mento) to always send the funds to the FeeHandler instead of msg.sender

STATUS: acknowledged

DESCRIPTION:

The **sell** method of the **UniswapFeeHandlerSeller** and **MentoFeeHandlerSeller** is a permissionless function to sell any tokens that are in the contract. It does not validate that the caller is the **FeeHandler** contract.

On line 209 (Uniswap) and line 82 (Mento), the function sends the swapped funds back to **msg.sender**.

As such, if some funds get stuck in the contract or if a user or contract mistakenly transfer funds to the contract without immediately calling **sell** (e.g. by front-running an EOA), then someone could call **sell** and steal the funds.

```
function sell(
  address sellTokenAddress,
  address buyTokenAddress,
  uint256 amount,
  uint256 maxSlippage // as fraction,
) external {
  require(
    buyTokenAddress == registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID),
    "Buy token can only be gold token"
  );

  require(
    routerAddresses[sellTokenAddress].values.length > 0,
    "routerAddresses should be non empty"
  );

  // An improvement to this function would be to allow the user to pass a path as argument
  // and if it generates a better outcome that the ones enabled that gets used
  // and the user gets a reward

  IERC20 celoToken = getGoldToken();

  IUniswapV2RouterMin bestRouter;
  uint256 bestRouterQuote = 0;

  address[] memory path = new address[][2];

  for (uint256 i = 0; i < routerAddresses[sellTokenAddress].values.length; i++) {
    address poolAddress = routerAddresses[sellTokenAddress].get(i);
    IUniswapV2RouterMin router = IUniswapV2RouterMin(poolAddress);

    path[0] = sellTokenAddress;
    path[1] = address(celoToken);

    // Using the second return value becuase it's the last argument,
    // the previous values show how many tokens are exchanged in each path
    // so the first value would be equivalent to balanceToBurn
    uint256 wouldGet = router.getAmountsOut(amount, path)[1];

    emit ReceivedQuote(sellTokenAddress, poolAddress, wouldGet);
    if (wouldGet > bestRouterQuote) {
      bestRouterQuote = wouldGet;
      bestRouter = router;
    }
  }
```

```
require(bestRouterQuote != 0, "Can't exchange with zero quote");

  uint256 minAmount = 0;
  if (maxSlippage != 0) {
    minAmount = calculateAllMinAmount(sellTokenAddress, maxSlippage, amount, bestRouter);
  }

  IERC20(sellTokenAddress).approve(address(bestRouter), amount);
  bestRouter.swapExactTokensForTokens(
    amount,
    minAmount,
    path,
    address(this),
    block.timestamp + MAX_TIMESTAMP_BLOCK_EXCHANGE
  );

  celoToken.transfer(msg.sender, celoToken.balanceOf(address(this)));
  emit RouterUsed(address(bestRouter));
  emit TokenSold(sellTokenAddress, buyTokenAddress, amount);
}

function sell(
  address sellTokenAddress,
  address buyTokenAddress,
  uint256 amount,
  uint256 maxSlippage // as fraction,
) external {
  require(
    buyTokenAddress == registry.getAddressForOrDie(GOLD_TOKEN_REGISTRY_ID),
    "Buy token can only be gold token"
  );

  IStableTokenMento stableToken = IStableTokenMento(sellTokenAddress);
  require(amount <= stableToken.balanceOf(address(this)), "Balance of token to burn not enough");

  address exchangeAddress = registry.getAddressForOrDie(stableToken.getExchangeRegistryId());

  IExchange exchange = IExchange(exchangeAddress);

  uint256 minAmount = 0;
  if (maxSlippage != 0) {
    // max slippage is set

    ISortedOracles sortedOracles = getSortedOracles();

    require(
      sortedOracles.numRates(sellTokenAddress) >= minimumReports[sellTokenAddress],
      "Number of reports for token not enough"
    );

    (uint256 rateNumerator, uint256 rateDenominator) = sortedOracles.medianRate(sellTokenAddress);
    minAmount = calculateMinAmount(rateNumerator, rateDenominator, amount, maxSlippage);
  }
```

```
  // TODO an upgrade would be to compare using routers as well
  stableToken.approve(exchangeAddress, amount);
  exchange.sell(amount, minAmount, false);

  IERC20 goldToken = getGoldToken();
  goldToken.transfer(msg.sender, goldToken.balanceOf(address(this)));

  emit TokenSold(sellTokenAddress, buyTokenAddress, amount);
}
```

CLAB-3. ZERO COMPARISON OPTIMISATION FOR UINT TYPES

SEVERITY: Low

PATH: Governance.sol, LockedGold.sol

REMEDIATION: replace the > 0 checks with != to optimise for gas usage

STATUS: fixed

DESCRIPTION:

We have identified various functions that check if a **uint** type is greater than zero using the **> 0** operator.

In Governance.sol:

1. setConcurrentProposals
2. setMinDeposit
3. setQueueExpiry
4. setDequeueFrequency
5. setReferendumStageDuration
6. setExecutionStageDuration
7. upvote
8. vote
9. withdraw
10. revokeVotes

In LockedGold.sol:

1. delegateGovernanceVotes
2. revokeDelegatedGovernanceVotes
3. updateDelegatedAmount

Instead, using the `!=` operator to check for `0` costs less gas, allowing for gas savings, because a `uint` cannot be negative.

```
  function setConcurrentProposals(uint256 _concurrentProposals) public onlyOwner {
    require(_concurrentProposals > 0, "Number of proposals must be larger than zero");
    require(_concurrentProposals != concurrentProposals, "Number of proposals unchanged");
    concurrentProposals = _concurrentProposals;
    emit ConcurrentProposalsSet(_concurrentProposals);
  }

[...]

    function vote(uint256 proposalId, uint256 index, Proposals.VoteValue value)
    external
    nonReentrant
    returns (bool)
  {
    dequeueProposalsIfReady();
    (Proposals.Proposal storage proposal, Proposals.Stage stage) = requireDequeuedAndDeleteExpired(
      proposalId,
      index
    );
    if (!proposal.exists()) {
      return false;
    }

    require(stage == Proposals.Stage.Referendum, "Incorrect proposal state");
    require(value != Proposals.VoteValue.None, "Vote value unset");

    address account = getAccounts().voteSignerToAccount(msg.sender);
    uint256 weight = getLockedGold().getAccountTotalGovernanceVotingPower(account);
    require(weight > 0, "Voter weight zero");

    _vote(
      proposal,
      proposalId,
      index,
      account,
      value == Proposals.VoteValue.Yes ? weight : 0,
      value == Proposals.VoteValue.No ? weight : 0,
      value == Proposals.VoteValue.Abstain ? weight : 0
    );
    return true;
  }

[...]
```

CLAB-11. GOVERNANCE SLASHING WILL SEND REWARD TO ADDRESS ZERO

SEVERITY: Low

PATH: LockedGold.sol:slash:L846-885

REMEDIATION: move the _incrementNonvotingAccountBalance() function inside the if statement for reporter != address(0) such that the reward is not given to address(0)

STATUS: acknowledged

DESCRIPTION:

In the **slash** function, there is a address parameter called **reporter** which used to provide a reward to. The function includes a check to determine if the **reporter** address is the zero address (**address(0)**). However, after the check, outside of the if statement, the function calls **_incrementNonvotingAccountBalance(reporter, reward)** to give the reward. This would result in the reward being given to the zero address if the reporter address is zero, which is unintended.

This case can happen by default if someone is slashed by Governance in **GovernanceSlasher.sol:slash**, which provides **address(0)** as reporter.

```
function slash(
  address account,
  uint256 penalty,
  address reporter,
  uint256 reward,
  address[] calldata lessers,
  address[] calldata greaters,
  uint256[] calldata indices
) external onlySlasher {
  uint256 maxSlash = Math.min(penalty, getAccountTotalLockedGold(account));
  require(maxSlash >= reward, "reward cannot exceed penalty.");
  // `reporter` receives the reward in locked CELO, so it must be given to an account
  // There is no reward for slashing via the GovernanceSlasher, and `reporter`
  // is set to 0x0.
  if (reporter != address(0)) {
    reporter = getAccounts().signerToAccount(reporter);
  }
  // Local scoping is required to avoid Solc "stack too deep" error from too many locals.
  {
    uint256 nonvotingBalance = balances[account].nonvoting;
    uint256 difference = 0;
    // If not enough nonvoting, revoke the difference
    if (nonvotingBalance < maxSlash) {
      difference = maxSlash.sub(nonvotingBalance);
      require(
        getElection().forceDecrementVotes(account, difference, lessers, greaters, indices) ==
          difference,
        "Cannot revoke enough voting gold."
      );
    }
    // forceDecrementVotes does not increment nonvoting account balance, so we can't double count
    _decrementNonvotingAccountBalance(account, maxSlash.sub(difference));
    _incrementNonvotingAccountBalance(reporter, reward);
  }
  address communityFund = registry.getAddressForOrDie(GOVERNANCE_REGISTRY_ID);
  address payable communityFundPayable = address(uint160(communityFund));
  require(maxSlash.sub(reward) <= address(this).balance, "Inconsistent balance");
  communityFundPayable.sendValue(maxSlash.sub(reward));
  emit AccountSlashed(account, maxSlash, reporter, reward);
}
```

CLAB-12. AMBIGUOUS LOGIC FOR MAX SLIPPAGE IN FEEHANDLERSELLER

SEVERITY: Low

PATH: PR #10227 UniswapFeeHandlerSeller.sol:sell:L139-204, MentoFeeHandlerSeller.sol:sell:L41-82

REMEDIATION: the conditional logic for max slippage in the sell function of both Uniswap and Mento should be in accordance with the logic in calculateMinAmount

STATUS: fixed

DESCRIPTION:

In both the Uniswap and Mento seller contracts, the max slippage is used to calculate the minimum amount for the swap in the router. This maximum slippage comes from a setting in FeeHandler.

In the **sell** functions, there is a conditional that if the maximum slippage is set to 0, the minimum amount will also be 0.

However, this is in contradiction with how the maximum slippage is used in the calculation in **FeeHandlerSeller.sol:calculateMinAmount**. Here the maximum slippage is a fraction of the total amount that is to be subtracted from the total amount to get the minimum amount.

Therefore, a maximum slippage of 100% would equal a minimum amount of 0, while 0% would equal the total amount. The conditional expresses the exact opposite.

```
function calculateMinAmount(
  uint256 midPriceNumerator,
  uint256 midPriceDenominator,
  uint256 amount,
  uint256 maxSlippage // as fraction
) public pure returns (uint256) {
  FixidityLib.Fraction memory maxSlippageFraction = FixidityLib.wrap(maxSlippage);

  FixidityLib.Fraction memory price = FixidityLib.newFixedFraction(
    midPriceNumerator,
    midPriceDenominator
  );
  FixidityLib.Fraction memory amountFraction = FixidityLib.newFixed(amount);
  FixidityLib.Fraction memory totalAmount = price.multiply(amountFraction);

  return
    totalAmount
      .subtract(price.multiply(maxSlippageFraction).multiply(amountFraction))
      .fromFixed();
}
```

CLAB-20. UNNECESSARY USAGE OF SAFEMATH

SEVERITY: Low

PATH: see description

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

SafeMath is used in an almost all contracts (currently 31) for incrementing the iterator in for-loops.

The iterator cannot realistically overflow, as execution would be out-of-gas first. Therefore, the usage of SafeMath for incrementing the iterator becomes redundant and a waste of gas.

```
for (uint256 i = 0; i < [..]; i.add(1)) {
  [..];
}
```

The iterator should be incremented with ++i instead:

```
for (uint256 i = 0; i < [..]; ++i {
  [..];
}
```

This should be replaced in all for-loops that increment i with 1 each iteration, in each contract of the project.

CLAB-24. EARLY EXIT GAS OPTIMISATION FOR CONDITIONALS IN VOTE REVOKING

SEVERITY: Low

PATH: Governance.sol:revokeVotes:L802-875

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function loops through the dequeue with indices and checks whether the vote record's proposal ID is equal to the proposal in the dequeue position on lines 814-819:

```
if (
  (voteRecord.yesVotes > 0 ||
    voteRecord.noVotes > 0 ||
    voteRecord.abstainVotes > 0 ||
    voteRecord.deprecated_weight > 0) &&
  voteRecord.proposalId == dequeued[dequeueIndex]
) { [..] }
```

The EVM compiler checks those conditions from left to right and the most important check is if the current **voteRecord.proposalId** is equal to **dequeued[dequeueIndex]**. Otherwise it will check the **voteRecord** votes each time redundantly and it produces many unnecessary opcodes during execution.

```
function revokeVotes() external nonReentrant returns (bool) {
    address account = getAccounts().voteSignerToAccount(msg.sender);
    Voter storage voter = voters[account];
    for (
      uint256 dequeueIndex = 0;
      dequeueIndex < dequeued.length;
      dequeueIndex = dequeueIndex.add(1)
    ) {
      VoteRecord storage voteRecord = voter.referendumVotes[dequeueIndex];

      // Skip proposals where there was no vote cast by the user AND
      // ensure vote record proposal matches identifier of dequeued index proposal.
      if (
        (voteRecord.yesVotes > 0 ||
          voteRecord.noVotes > 0 ||
          voteRecord.abstainVotes > 0 ||
          voteRecord.deprecated_weight > 0) &&
        voteRecord.proposalId == dequeued[dequeueIndex]
      ) {
```

It would be more gas efficient to first check the most unlikely condition by switching the checks and allowing for an early exit:

```
if (voteRecord.proposalId == dequeued[dequeueIndex] &&
  (voteRecord.yesVotes > 0 ||
    voteRecord.noVotes > 0 ||
    voteRecord.abstainVotes > 0 ||
    voteRecord.deprecated_weight > 0)
) { [..] }
```

CLAB-34. LOCKEDGOLD SLASH OPTIMISATION

SEVERITY: Low

PATH: LockedGold.sol:slash:L846-885

REMEDIATION: remove the require check and just call forceDecrementVotes directly

STATUS: acknowledged

DESCRIPTION:

This function is used to slash the locked gold of a user. It also reduces votes if this locked gold was used as voting power.

On line 870-874, there is a **require** check:

```
require(getElection().forceDecrementVotes(account, difference, lessers, greaters, indices)
== difference);
```

However, the function **forceDecrementVotes** always returns **difference** from the parameters, otherwise it will revert.

Therefore, the check will always be **true**, making the **require** check redundant.

```
function slash(
    address account,
    uint256 penalty,
    address reporter,
    uint256 reward,
    address[] calldata lessers,
    address[] calldata greaters,
    uint256[] calldata indices
  ) external onlySlasher {
    uint256 maxSlash = Math.min(penalty, getAccountTotalLockedGold(account));
    require(maxSlash >= reward, "reward cannot exceed penalty.");
    // `reporter` receives the reward in locked CELO, so it must be given to an account
    // There is no reward for slashing via the GovernanceSlasher, and `reporter`
    // is set to 0x0.
    if (reporter != address(0)) {
      reporter = getAccounts().signerToAccount(reporter);
    }
    // Local scoping is required to avoid Solc "stack too deep" error from too many locals.
    {
      uint256 nonvotingBalance = balances[account].nonvoting;
      uint256 difference = 0;
      // If not enough nonvoting, revoke the difference
      if (nonvotingBalance < maxSlash) {
        difference = maxSlash.sub(nonvotingBalance);
        require(
          getElection().forceDecrementVotes(account, difference, lessers, greaters, indices) ==
            difference,
          "Cannot revoke enough voting gold."
        );
      }
      // forceDecrementVotes does not increment nonvoting account balance, so we can't double count
      _decrementNonvotingAccountBalance(account, maxSlash.sub(difference));
      _incrementNonvotingAccountBalance(reporter, reward);
    }
    address communityFund = registry.getAddressForOrDie(GOVERNANCE_REGISTRY_ID);
    address payable communityFundPayable = address(uint160(communityFund));
    require(maxSlash.sub(reward) <= address(this).balance, "Inconsistent balance");
    communityFundPayable.sendValue(maxSlash.sub(reward));
    emit AccountSlashed(account, maxSlash, reporter, reward);
  }
```

```
function forceDecrementVotes(
    address account,
    uint256 value,
    address[] calldata lessers,
    address[] calldata greaters,
    uint256[] calldata indices
  ) external nonReentrant onlyRegisteredContract(LOCKED_GOLD_REGISTRY_ID) returns (uint256) {
    require(value > 0, "Decrement value must be greater than 0.");
    DecrementVotesInfo memory info = DecrementVotesInfo(votes.groupsVotedFor[account], value);
    require(
      lessers.length <= info.groups.length &&
        lessers.length == greaters.length &&
        greaters.length == indices.length,
      "Input lengths must be correspond."
    );
    // Iterate in reverse order to hopefully optimize removing pending votes before active votes
    // And to attempt to preserve `account`'s earliest votes (assuming earliest = prefered)
    for (uint256 i = info.groups.length; i > 0; i = i.sub(1)) {
      info.remainingValue = info.remainingValue.sub(
        _decrementVotes(
          account,
          info.groups[i.sub(1)],
          info.remainingValue,
          lessers[i.sub(1)],
          greaters[i.sub(1)],
          indices[i.sub(1)]
        )
      );
      if (info.remainingValue == 0) {
        break;
      }
    }
    require(info.remainingValue == 0, "Failure to decrement all votes.");
    return value;
  }
}
```

CLAB-4. POSSIBILITY OF DUPLICATES IN WHITELIST

SEVERITY: Informational

PATH: FeeCurrencyWhitelist.sol:addToken:L70-73

REMEDIATION: the function addToken should check if the token has already been added

STATUS: acknowledged

DESCRIPTION:

This function adds a token to the **whitelist** array.

However, the function does not validate that the token has already been added to the whitelist. Therefore it becomes possible to add the same token multiple times.

```
function addToken(address tokenAddress) external onlyOwner {
  whitelist.push(tokenAddress);
  emit FeeCurrencyWhitelisted(tokenAddress);
}
```

CLAB-7. FUNCTIONS CAN BE MARKED AS EXTERNAL

SEVERITY: Informational

PATH: LockedGold.sol

REMEDIATION: change the visibility modifier from public to external

STATUS: fixed

DESCRIPTION:

Using external instead of public for the following functions will save gas, as they are not called from within a contract. This is because all the function arguments can be marked as calldata instead of memory.

In LockedGold.sol:

1. revokeDelegatedGovernanceVotes
2. getDelegatorDelegateeInfo

1.

```
function revokeDelegatedGovernanceVotes(address delegatee, uint256 percentageToRevoke) public {
  require(
    percentageToRevoke > 0 && percentageToRevoke <= 100,
    "revoked percents can be only between 1%..100%"
  );

[...]
```

2.

```
function getDelegatorDelegateeInfo(address delegator, address delegatee)
  public
  view
  returns (uint256 percentage, uint256 currentAmount)
{
  DelegatedInfo storage currentDelegateeInfo = delegatorInfo[delegator]
    .delegateesWithPercentagesAndAmount[delegatee];

  percentage = currentDelegateeInfo.percentage;
  currentAmount = currentDelegateeInfo.currentAmount;
}
```

CLAB-17. REDUNDANT LINE AND HANDLER CHECK IN FEEHANDLER

SEVERITY: Informational

PATH: PR #10227
FeeHandler.sol:_addToken:L236-244

REMEDIATION: remove the redundant line 238

STATUS: fixed

DESCRIPTION:

The contract **FeeHandler** implements the **IFeeHandlerSeller** interface, but in the function **_addToken** the line 238 becomes redundant because it doesn't call any function in the contract:

IFeeHandlerSeller(handlerAddress);

Additionally, it fails to check whether the contract actually implements the interface, as stated in the comment. Wrapping the address in the interface effectively does not nothing and does not check whether it actually implements the interface.

```
function _addToken(address tokenAddress, address handlerAddress) private {
    // Check that the contract implements the interface
    IFeeHandlerSeller(handlerAddress);

    TokenState storage tokenState = tokenStates[tokenAddress];
    tokenState.handler = handlerAddress;

    activeTokens.add(tokenAddress);
}
```

CLAB-18. UNUSED EVENTS

SEVERITY: Informational

PATH: PR #10227
FeeHandler.sol, FeeHandlerSeller.sol

REMEDIATION: remove these unused events from the contract or add them to the required places

STATUS: fixed

DESCRIPTION:

There are various events that are declared within the contracts but are never emitted in any function or operation:

1. FeeHandler.sol:RouterAddressSet:L77
2. FeeHandler.sol:RouterAddressRemoved:L78
3. FeeHandler.sol:RouterUsed:L79
4. FeeHandlerSeller.sol:TokenSold:L21

FeeHandler.sol

```
event RouterAddressSet(address token, address router);
event RouterAddressRemoved(address token, address router);
event RouterUsed(address router);
```

FeeHandlerSeller.sol

```
event TokenSold(address soldTokenAddress, address boughtTokenAddress, uint256 amount);
```

CLAB-22. REDUNDANT IMPORT OF SAFEMATH

SEVERITY: Informational

PATH: PR #10379
GasPriceMinimum.sol:L4

REMEDIATION: remove the import of OpenZeppelin's SafeMath library

STATUS: fixed

DESCRIPTION:

In the contract **GasPriceMinimum.sol** the pragma version is changed from **^0.5.13** to **>=0.8.7 <0.9.0**, so the import of OpenZeppelin's **SafeMath.sol** library is redundant and increases the contract's byte code size.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.7 <0.9.0;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";

[...]
```

CLAB-32. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: Steward.sol

REMEDIATION: the variables PAUSER_ROLE and MINTER_ROLE should be marked as private instead of public

STATUS: acknowledged

DESCRIPTION:

Setting constants to **private** in lines 1495 and 1496 will save deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table. If needed, the values can be read from the verified contract source code. the WithdrawalQueue, and the user would lose their ETH.

```
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
```

CLAB-33. REDUNDANT DECLARATION OF ABIENCODERV2

SEVERITY: Informational

PATH: Steward.sol

REMEDIATION: remove the line:

pragma experimental ABIEncoderV2;

STATUS: acknowledged

DESCRIPTION:

In solidity version >=0.8.0 ABI coder v2 is activated by default.

You can choose to use the old behavior using **pragma abicoder v1**;. The pragma **pragma experimental ABIEncoderV2;** is still valid, but it is deprecated and has no effect.

Solidity v0.8.0 Breaking Changes — Solidity 0.8.0 documentation

```
pragma solidity ^0.8.7;
pragma experimental ABIEncoderV2;
```

CLAB-35. MISSING ARRAY PARAMETERS LENGTH CHECK

SEVERITY: Informational

PATH: Steward.sol:mintMultiple:L1558-1574

REMEDIATION: check the input parameters' length so that the error is caught to improve user experience

STATUS: acknowledged

DESCRIPTION:

In the function **mintMultiple** there are 3 arrays as function parameters that require the same length. However, it is missing checks that lengths are equal.

```
function mintMultiple(
    address[] memory to,
    uint256[] memory tokenId,
    string[] memory uri
) public returns (bool) {
    if (!_publicMint) {
        require(
            hasRole(MINTER_ROLE, _msgSender()),
            "Steward: must have minter role to mint"
        );
    }
    for (uint256 i = 0; i < to.length; i++) {
        _safeMint(to[i], tokenId[i]);
        _setTokenURI(tokenId[i], uri[i]);
    }
    return true;
}
```