Cyberscope

Audit Report

Chrysus

April 2025

Network	ETH
Address	0xf1fA5809163c25049C4EE2883e4bc3d6fddd83a7
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **4**
Review **5**
Audit Updates 5
Source Files 5
Overview **6**
Governance Functionality 6
Voting Eligibility 6
Daily Minting 6
Vesting Schedule Management 7
Vesting Enforcement in Transfers 7
Contract Initialisation 7
Owner Functionalities 7
Roles 8
Team 8
onlyVoter 8
vestingContract 8
Findings Breakdown **9**
Diagnostics **10**
FGA - Flashloan Governance Attack 12
Description 12
Recommendation 14
ITVC - Inaccurate Token Vesting Calculation 15
Description 15
Recommendation 17
MVV - Multiple Voting Vulnerability 18
Description 18
Recommendation 19
ST - Stops Transactions 20
Description 20
Recommendation 21
CCR - Contract Centralization Risk 23
Description 23
Recommendation 23
DMO - Daily Mint Overflow 24
Description 24
Recommendation 25
DDP - Decimal Division Precision 26
Description 26

Recommendation 26
IBR - Incorrect Balance Restriction 27
Description 27
Recommendation 28
MVN - Misleading Variables Naming 29
Description 29
Recommendation 29
MVSV - Missing Vesting Schedule Validations 30
Description 30
Recommendation 31
MZVV - Missing Zero Vote Validation 32
Description 32
Recommendation 32
MU - Modifiers Usage 33
Description 33
Recommendation 33
PTAI - Potential Transfer Amount Inconsistency 34
Description 34
Recommendation 35
REF - Redundant Execution Flag 36
Description 36
Recommendation 36
RC - Repetitive Calculations 37
Description 37
Recommendation 37
UEC - Unoptimized External Calls 39
Description 39
Recommendation 39
UECE - Unrestricted External Call Execution 40
Description 40
Recommendation 41
UTPD - Unverified Third Party Dependencies 43
Description 43
Recommendation 43
L04 - Conformance to Solidity Naming Conventions 44
Description 44
Recommendation 44
L09 - Dead Code Elimination 45
Description 45
Recommendation 45
L13 - Divide before Multiply Operation 46
Description 46

Recommendation 46
L18 - Multiple Pragma Directives 47
Description 47
Recommendation 47
L19 - Stable Compiler Version 48
Description 48
Recommendation 48
Functions Analysis **49**
Inheritance Graph **50**
Flow Graph **51**
Summary **52**
Disclaimer **53**
About Cyberscope **54**

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	Governance
Compiler Version	v0.8.20+commit.a1b79de6
Optimization	200 runs
Explorer	https://etherscan.io/address/0xf1fa5809163c25049c4ee2883e4bc3d6fddd83a7
Address	0xf1fa5809163c25049c4ee2883e4bc3d6fddd83a7
Network	ETH
Symbol	GOV
Decimals	18
Total Supply	140,000,000

Audit Updates

Initial Audit	28 Apr 2025

Source Files

Filename	SHA256
Governance.sol	ab27f4cf146b2f3baa88dbcc639524fdfed58704989b36681332fe8b33b2b04c

Overview

The `Governance` smart contract is an ERC20 token-based system that manages decentralised governance and token vesting within a protocol. It allows token holders with active stakes to propose, vote on, and execute governance decisions. Additionally, it includes mechanisms for minting daily rewards under a capped policy and handles vesting schedules for investors or stakeholders. The contract tightly integrates with a `StabilityModule` to ensure that only active, engaged stakeholders can participate in governance. It also introduces voting logic tied to stake activity, enforces restrictions around token transfers based on vesting schedules, and provides a minting mechanism for reward and reserve allocation.

Governance Functionality

The contract introduces a voting system allowing stakeholders to propose and vote on protocol changes. A stakeholder must hold a significant amount of tokens staked in the `StabilityModule` and satisfy time-based eligibility requirements to propose a vote. Each `Vote` includes details such as the target function to be called, associated data, and the initiator. Users vote by selecting support, opposition, or abstention, and their voting power is determined by their token balance. A vote is executed only if 75% of the pool participates and over 51% supports it. Execution happens via a low-level call to the target function encoded in the vote, providing a flexible and modular upgrade pathway.

Voting Eligibility

Voting is permissioned via the `onlyVoter` modifier, ensuring that only users with recent staking activity or voting participation in the last 90 days can vote. Additionally, the stake must be at least 30 days old. These constraints prevent governance attacks from newly staked or inactive users and promote sustained engagement. The logic also updates the user's last governance interaction in the `StabilityModule` to keep their voter status current.

Daily Minting

The contract allows the team to mint new tokens daily via the `mintDaily` function, under the restriction of a daily cap (`DAILY_MINT_CAP`). If no mint has occurred yet, it

starts by minting for one day; otherwise, it calculates how many days have passed since the last mint. From the total daily minted amount, 90,000 tokens per day are minted to the `rewardDistributor`, and 10,000 are sent to the team's address. This ensures both liquidity and incentive alignment while preventing excessive inflation.

Vesting Schedule Management

The contract supports time-locked token vesting via `VestingSchedule` structs, which are assigned per address and schedule ID. Vesting schedules include cliff durations, monthly vesting rates, and total allocations. The `setVestingSchedule` function can only be called by a dedicated `vestingContract`, ensuring that only authorised contracts can grant vesting rights. The `calculateAvailableTokens` function determines how many tokens are unlocked at any point in time, based on the cliff and vesting period.

Vesting Enforcement in Transfers

The `_update` function is overridden from the ERC20 base contract to enforce vesting constraints during token transfers. If the sender is not the `vestingContract`, the contract calculates how many of the sender's tokens are either vested or unencumbered. It then ensures that the amount being transferred does not exceed the sum of vested and free tokens. This prevents users from transferring locked tokens prematurely and ensures adherence to the vesting schedule.

Contract Initialisation

To avoid reentrancy or duplicate setup, the `init` function must be called exactly once by the `team` address. This function sets up the `rewardDistributor`, the `stabilityModule`, and the `vestingContract`. The `mustInit` modifier is used to gate critical functionality (e.g., minting and governance proposals) until the contract is fully configured, safeguarding against usage in an uninitialised state.

Owner Functionalities

The contract allows the deployer-defined `team` address to perform privileged actions such as initialising the contract and minting new tokens. The `onlyTeam` modifier restricts access to these critical functions. The team address is set during deployment and

is immutable. These administrative capabilities enable the team to bootstrap the protocol while preserving decentralised control via governance mechanisms thereafter.

Roles

Team

The team address has authority over the following functions:

- function init
- function mintDaily

onlyVoter

The onlyVoter addresses can interact with the following functions:

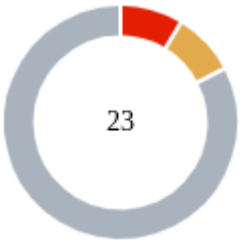
- function proposeVote
- function executeVote
- function vote

vestingContract

The vestingContract addresses can interact with the following functions:

- function setVestingSchedule

Findings Breakdown



Critical	2
Medium	2
Minor / Informative	19

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	2	0	0	0
Minor / Informative	19	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	FGA	Flashloan Governance Attack	Unresolved
Critical	ITVC	Inaccurate Token Vesting Calculation	Unresolved
Medium	MVV	Multiple Voting Vulnerability	Unresolved
Medium	ST	Stops Transactions	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	DMO	Daily Mint Overflow	Unresolved
Minor / Informative	DDP	Decimal Division Precision	Unresolved
Minor / Informative	IBR	Incorrect Balance Restriction	Unresolved
Minor / Informative	MVN	Misleading Variables Naming	Unresolved
Minor / Informative	MVSV	Missing Vesting Schedule Validations	Unresolved
Minor / Informative	MZVV	Missing Zero Vote Validation	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved

●	REF	Redundant Execution Flag	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	UEC	Unoptimized External Calls	Unresolved
●	UECE	Unrestricted External Call Execution	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

FGA - Flashloan Governance Attack

Criticality	Critical
Location	Governance.sol#L990,1030
Status	Unresolved

Description

The contract is vulnerable to manipulation through a flashloan-based governance attack due to its reliance on the voter's `balanceOf(msg.sender)` at the time of vote casting. This allows an attacker to temporarily inflate their token balance via a flashloan and use it to initiate and vote on proposals, bypassing the intended distribution and fairness of governance power. If a sufficient quorum is reached, the proposal can be executed, even though the attacker's actual stake in the system was only momentary.

This risk is magnified by the fact that successful proposals result in an external call to an arbitrary address and function, as described in the related finding `UECE - Unrestricted External Call Execution.` In such a case, an attacker can not only influence the outcome of governance but also execute malicious logic on-chain by targeting an address they control. Together, these issues represent a severe threat to protocol integrity and user funds.

```
  function vote(
        uint256 numberOfVote,
        bool isSupports,
        bool isAbstains
    ) external onlyVoter mustInit voteExists(numberOfVote) {
        Vote storage v = voteInfo[numberOfVote];
        require(v.tallyTime == 0, "Vote has already been tallied");
        require(!v.voted[msg.sender], "Sender has already voted");

        v.voted[msg.sender] = true;
        if (isAbstains) {
            v.amountAbstained += balanceOf(msg.sender);
        } else if (isSupports) {
            v.amountSupporting += balanceOf(msg.sender);
        } else {
            v.amountAgainst += balanceOf(msg.sender);
        }
        ...
        }

  function executeVote(
        uint256 numberOfVote
    ) external onlyVoter mustInit voteExists(numberOfVote) {
        //75 percent of pool needs to vote

        Vote storage v = voteInfo[numberOfVote];

        require(
            v.amountSupporting + v.amountAgainst + v.amountAbstained >
                (stabilityModule.getTotalPoolAmount() * 3) / 4,
            "75% of pool has not voted yet!"
        );
        require(v.tallyTime == 0, "Vote has already been tallied");
        uint256 _duration = 2 days;
        require(
            block.timestamp - v.startTime > _duration,
            "Time for voting has not elapsed"
        );

        if (
            v.amountSupporting >
            (stabilityModule.getTotalPoolAmount() * 51) / 100
        ) {
            v.result = true;
            address _destination = v.voteAddress;
            bool _succ;
            bytes memory _res;
            (_succ, _res) = _destination.call(
                abi.encodePacked(v.voteFunction, v.data)
            );
```

```
            //When testing _destination.call can require higher gas than
the standard. Be sure to increase the gas if it fails.
            require(_succ, "error running _destination.call");
            ...
            }
```

Recommendation

It is recommended to implement snapshot-based voting to mitigate flashloan attacks, where voting power is calculated from balances at a predefined block. Additionally, reinforcing governance with time delays or multisig review for external call proposals would reduce the risk of exploitation. These changes would significantly increase the protocol’s resilience to short-term manipulation and malicious governance behaviour.

ITVC - Inaccurate Token Vesting Calculation

Criticality	Critical
Location	Governance.sol#L895
Status	Unresolved

Description

The contract is designed to manage token vesting through the `calculateAvailableTokens` function, which determines the number of tokens available for an account based on a given vesting schedule. The calculation relies on the `VestingSchedule` struct, which includes `lastClaimTime` to track the timing of the last claim and `amountClaimed` to track the cumulative amount claimed so far.

However, the contract does not update either `lastClaimTime` or `amountClaimed` at any point in its logic, including when tokens are transferred via `_update`. As a result, `lastClaimTime` remains perpetually fixed at the `cliffEnd` timestamp set during vesting schedule creation, and `amountClaimed` stays at zero. This leads the `calculateAvailableTokens` function to always calculate vested tokens starting from the original `cliffEnd`, ignoring any previously transferred or claimed tokens.

Consequently, users may repeatedly transfer tokens beyond their intended vested amounts, leading to over-claiming. Even if the design was intended to allow token transfer tracking solely from the start of vesting, the contract should at least deduct previously transferred amounts or mark them as claimed. Otherwise, users can call `_update` multiple times and transfer tokens each time without reducing their future entitlement, effectively bypassing the vesting logic.

This flaw breaks the core integrity of the vesting mechanism and introduces a critical financial risk by allowing users to transfer more tokens than they should be entitled to under the vesting schedule.

```
    function calculateAvailableTokens(
        address account,
        uint8 _schedule
    ) public view returns (uint256) {
        ...
        uint256 monthsSinceLastClaim = (block.timestamp -
            schedule.lastClaimTime) / 30 days;
        uint256 newVestedAmount = monthsSinceLastClaim *
schedule.monthlyAmount;
        uint256 totalVested = schedule.amountClaimed + newVestedAmount;

        return
            totalVested > schedule.totalAmount
                ? schedule.totalAmount
                : totalVested;
    }

    function _update(
        address from,
        address to,
        uint256 value
    ) internal virtual override {
        if (from != address(0) && from != vestingContract) {
            uint256 totalVested = 0;
            uint256 vestedBalance = 0;

            uint8[] storage schedules = userSchedules[from];
            if (schedules.length > 0) {
                for (uint i = 0; i < schedules.length; i++) {
                    VestingSchedule storage schedule =
vestingSchedules[from][
                        schedules[i]
                    ];
                    if (block.timestamp >= schedule.cliffEnd) {
                        totalVested += calculateAvailableTokens(
                            from,
                            schedules[i]
                        );
                    }
                    vestedBalance += schedule.totalAmount;
                }

                uint256 freeBalance = balanceOf(from) > vestedBalance
                    ? balanceOf(from) - vestedBalance
                    : 0;
                require(
                    value <= freeBalance + totalVested,
                    "Amount exceeds available tokens"
                );
            }
```

```
        }
        super._update(from, to, value);
    }
```

Recommendation

It is recommended to implement a mechanism to correctly update the `lastClaimTime` and `amountClaimed` fields in the `VestingSchedule` struct each time a user claims tokens or when tokens are transferred as part of the vesting process. This should involve adding a function or modifying existing logic to record the timestamp of the claim and increment the `amountClaimed` by the claimed token amount. Additionally, ensure that any function allowing token claims or transfers checks and updates these fields to reflect the accurate vesting state. This will ensure that the `calculateAvailableTokens` function accurately calculates the available tokens based on the actual claim history, maintaining the intended vesting schedule and preventing over-claiming.

MVV - Multiple Voting Vulnerability

Criticality	Medium
Location	Governance.sol#L1033
Status	Unresolved

Description

The contract is designed to facilitate governance through a voting mechanism where users cast votes using the `vote` function, with their voting power determined by their token balance at the time of voting. The function records the msg.sender's vote based on their `balanceOf(msg.sender)` and marks them as having voted to prevent multiple votes from the same address. However, the contract does not restrict token transfers after a user has voted. As a result, a user could exploit this by casting a vote, then transferring their tokens to another address that also qualifies as a voter (per the `onlyVoter` modifier). The new address could then cast another vote using the same tokens, effectively allowing the same token balance to be counted multiple times across different addresses in the same voting round. This vulnerability undermines the integrity of the voting process, as it enables vote amplification and could skew governance outcomes in favor of malicious actors.

```
function vote(
        uint256 numberOfVote,
        bool isSupports,
        bool isAbstains
    ) external onlyVoter mustInit voteExists(numberOfVote) {
        Vote storage v = voteInfo[numberOfVote];
        require(v.tallyTime == 0, "Vote has already been tallied");
        require(!v.voted[msg.sender], "Sender has already voted");

        v.voted[msg.sender] = true;
        if (isAbstains) {
            v.amountAbstained += balanceOf(msg.sender);
        } else if (isSupports) {
            v.amountSupporting += balanceOf(msg.sender);
        } else {
            v.amountAgainst += balanceOf(msg.sender);
        }

        ...
    }
```

Recommendation

It is recommended to evaluate whether allowing token transfers after voting is a desired functionality for the governance system. If preventing double voting is critical, consider implementing a snapshot mechanism that records each user's token balance at the start of a voting period and bases voting power on this snapshot, rather than the current balance. Alternatively, consider locking tokens used for voting until the vote is tallied or restricting token transfers for addresses that have voted until the voting period concludes. These measures would prevent the same tokens from being used to vote multiple times, ensuring the fairness and accuracy of the governance process. Additionally, consider documenting the intended behavior in the contract's documentation to clarify whether this functionality is by design or requires mitigation.

ST - Stops Transactions

Criticality	Medium
Location	Governance.sol#L1088
Status	Unresolved

Description

The `vestingContract` has the authority to effectively halt or severely limit token transfers for specific addresses by assigning vesting schedules with large `totalAmount` values and delayed cliff or vesting periods. Since the `_update` function enforces transfer restrictions based on the sum of all vesting schedule amounts (`vestedBalance`) and only allows transfers up to the sum of `freeBalance` and currently vested tokens, a malicious or misconfigured `vestingContract` can set a schedule where no tokens are vested yet but the total vested amount covers the entire balance of the user. This causes `freeBalance` and `totalVested` to be zero, making all transfer attempts revert. As a result, this design gives the `vestingContract` unilateral control over whether a user can transfer tokens, essentially allowing it to freeze tokens for any address.

```
    function _update(
        address from,
        address to,
        uint256 value
    ) internal virtual override {
        if (from != address(0) && from != vestingContract) {
            uint256 totalVested = 0;
            uint256 vestedBalance = 0;

            uint8[] storage schedules = userSchedules[from];
            if (schedules.length > 0) {
                for (uint i = 0; i < schedules.length; i++) {
                    VestingSchedule storage schedule =
vestingSchedules[from][
                        schedules[i]
                    ];
                    if (block.timestamp >= schedule.cliffEnd) {
                        totalVested += calculateAvailableTokens(
                            from,
                            schedules[i]
                        );
                    }
                    vestedBalance += schedule.totalAmount;
                }

                uint256 freeBalance = balanceOf(from) > vestedBalance
                    ? balanceOf(from) - vestedBalance
                    : 0;
                require(
                    value <= freeBalance + totalVested,
                    "Amount exceeds available tokens"
                );
            }
        }
        super._update(from, to, value);
    }
```

Recommendation

It is recommended to review whether granting the `vestingContract` this level of control aligns with the intended governance and token utility model. If not, consider introducing checks to ensure that the total vesting amount cannot exceed the user's balance or enforce validations to prevent vesting schedules that would entirely lock a user's

tokens. Additionally, access to the `vestingContract` should be tightly restricted, audited, or governed by multisig to avoid abuse or accidental lock-ups.

The team should carefully manage the private keys of the owner's account that can call the `vestingContract`. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership of the `vestingContract`, which will eliminate the threats but it is non-reversible.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Governance.sol#L933,1057
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
   function mintDaily() external mustInit onlyTeam {
    ...
    }

function setVestingSchedule(
        address investor,
        uint256 amount,
        uint8 schedule,
        uint256 vestingDuration,
        uint256 cliffDuration
    ) external {
        require(msg.sender == vestingContract, "Only vesting contract");
        ...
        }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DMO - Daily Mint Overflow

Criticality	Minor / Informative
Location	Governance.sol#L933
Status	Unresolved

Description

The contract is designed to allow a fixed amount of tokens (100,000 units) to be minted per day through the `mintDaily` function, subject to a maximum cumulative limit defined by `DAILY_MINT_CAP` (e.g., 72 million). However, the logic enforces a strict cap on the total tokens minted through this function without considering the passage of time. This means that if `mintDaily` is not called daily and a gap of several days occurs, the accumulated mint amount for the missed days may exceed the cap. In such cases, the function reverts entirely, and no tokens are minted at all, including those still within the remaining allowable supply. For example, if 71 million tokens have already been minted and `mintDaily` is called after 11 days, the calculated mint amount becomes 1.1 million (100k * 11), resulting in a total of 72.1 million, — which exceeds the cap and causes the transaction to revert. This behaviour leads to a permanent inability to mint the remaining supply, effectively locking unused tokens and rendering the minting mechanism rigid and unforgiving.

```
    function mintDaily() external mustInit onlyTeam {
        uint256 dailyMintAmount = 1e23;
        uint256 numDays = lastMintTimestamp == 0
            ? 1
            : (block.timestamp - lastMintTimestamp) / 1 days;

        require(numDays != 0, "number of days cannot be 0");

        uint256 totalMintAmount = dailyMintAmount * numDays;
        require(
            totalDailyMinted + totalMintAmount <= DAILY_MINT_CAP,
            "Daily mint cap of 72M reached"
        );
        ...
        }
```

Recommendation

It is recommended to modify the `mintDaily` logic to allow partial minting up to the remaining available supply if the calculated mint amount exceeds the `DAILY_MINT_CAP`. This would ensure that the remaining mintable tokens are not permanently locked due to a delay in calling the function and would improve the resilience of the minting mechanism to missed days or irregular execution.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	Governance.sol#L1067
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 monthlyAmount = (amount * 30 days) / vestingDuration;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IBR - Incorrect Balance Restriction

Criticality	Minor / Informative
Location	Governance.sol#L1088
Status	Unresolved

Description

The contract is designed to restrict token transfers to an amount less than or equal to the sum of a user's free balance and total vested tokens. However, the logic for calculating the free balance introduces an issue. The free balance is computed as the user's token balance minus their vested balance, but it is capped at zero if the balance is less than the vested balance. For example, if a user has a balance of 30 tokens, a vested balance of 100 tokens, and a total vested amount of 10 tokens, the free balance is calculated as zero (since 30 - 100 is negative). As a result, the user is restricted to transferring only the total vested amount (10 tokens), despite having a balance of 30 tokens. This discrepancy misrepresents the user's available tokens and unduly limits their transfer capability, deviating from the intended functionality of allowing transfers based on the actual token balance.

```
function _update(
        address from,
        address to,
        uint256 value
    ) internal virtual override {
        if (from != address(0) && from != vestingContract) {
            ...
                    if (block.timestamp >= schedule.cliffEnd) {
                        totalVested += calculateAvailableTokens(
                            from,
                            schedules[i]
                        );
                    }
                    vestedBalance += schedule.totalAmount;
                }

                uint256 freeBalance = balanceOf(from) > vestedBalance
                    ? balanceOf(from) - vestedBalance
                    : 0;
                require(
                    value <= freeBalance + totalVested,
                    "Amount exceeds available tokens"
                );
            }
        }
        super._update(from, to, value);
    }
```

Recommendation

It is recommended to reconsider the intended functionality of the transfer restriction. Specifically, examine whether the free balance calculation should directly use the user's actual token balance (balanceOf(from)) instead of capping it at zero when the balance is less than the vested balance. This would align the transfer restriction with the user's true token holdings, ensuring that users can transfer up to their actual balance plus any vested tokens, as intended.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	Governance.sol#L808,943
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically the variable name `DAILY_MINT_CAP` is misleading, as it actually enforces a cumulative minting cap rather than a true daily limit, and could be more accurately named `MINT_CAP` .

```
uint256 public constant DAILY_MINT_CAP = 72e24;
...
 require(
        totalDailyMinted + totalMintAmount <= DAILY_MINT_CAP,
        "Daily mint cap of 72M reached"
    );
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MVSV - Missing Vesting Schedule Validations

Criticality	Minor / Informative
Location	Governance.sol#L1057
Status	Unresolved

Description

The contract is missing input validation checks within the `setVestingSchedule` function, which may result in erroneous vesting entries or unintended overwrites. Specifically, the function does not validate that the `amount` and `vestingDuration` parameters are greater than zero. If `vestingDuration` is set to zero, the function will revert due to a division by zero error during monthly amount calculation. Moreover, allowing an `amount` of zero would result in a meaningless vesting entry, which could cause confusion or unintended behaviour in downstream logic.

In addition, the function does not check whether a vesting schedule with the same schedule ID already exists for the given investor. As a result, calling `setVestingSchedule` multiple times with the same schedule ID will silently overwrite the existing schedule, leading to loss of data or abuse scenarios where a previous vesting configuration is replaced without restriction.

```
  function setVestingSchedule(
        address investor,
        uint256 amount,
        uint8 schedule,
        uint256 vestingDuration,
        uint256 cliffDuration
    ) external {
        require(msg.sender == vestingContract, "Only vesting contract");
        require(investor != address(0), "Invalid investor address");

        uint256 monthlyAmount = (amount * 30 days) / vestingDuration;

        vestingSchedules[investor][schedule] = VestingSchedule({
            totalAmount: amount,
            cliffEnd: block.timestamp + cliffDuration,
            vestingEnd: block.timestamp + cliffDuration +
vestingDuration,
            lastClaimTime: block.timestamp + cliffDuration,
            monthlyAmount: monthlyAmount,
            amountClaimed: 0,
            schedule: schedule
        });

        userSchedules[investor].push(schedule);
    }
```

Recommendation

It is recommended to add validation checks to ensure that both `amount` and `vestingDuration` are strictly greater than zero to prevent invalid or meaningless vesting schedules and avoid runtime exceptions. Furthermore, a check should be implemented to verify that a vesting schedule with the given ID does not already exist for the specified investor, thereby preventing unintended overwrites and preserving the integrity of the vesting system.

MZVV - Missing Zero Vote Validation

Criticality	Minor / Informative
Location	Governance.sol#L883
Status	Unresolved

Description

The contract is missing a validation in the `voteExists` modifier, which checks only whether the `numberOfVote` is less than or equal to the `voteCount`. However, it does not check whether the vote identifier is non-zero, even though a vote ID of zero does not represent a valid or meaningful vote in most governance implementations. Since vote IDs increment starting from one, passing a zero as the `numberOfVote` parameter bypasses logical expectations and may cause unexpected behaviour or unintended access to the default `Vote` struct stored at index zero in the `voteInfo` mapping. This could lead to execution or validation of an uninitialised or malformed vote and undermines the integrity of the governance process.

```
    modifier voteExists(uint256 numberOfVote) {
        require(numberOfVote <= voteCount, "Vote does not exist");
        _;
    }
```

Recommendation

It is recommended to strengthen the `voteExists` modifier by explicitly checking that the `numberOfVote` parameter is greater than zero. This ensures that only valid and properly initialised votes are considered during governance operations and prevents potential manipulation or logical errors arising from using a zero-based vote ID.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	Governance.sol#L921
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
if (rewardDistributorAddress == address(0)) revert ZeroAddress();
if (stabilityModuleAddress == address(0)) revert ZeroAddress();
if (vestingAddress == address(0)) revert ZeroAddress();
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Governance.sol#L1372
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

Specifically, the contract should ensure that it the tokens equal to the `amount` specified in the `setVestingSchedule` function, as this value is committed to the vesting schedule and must be available for future claims.

```
    function setVestingSchedule(
        address investor,
        uint256 amount,
        uint8 schedule,
        uint256 vestingDuration,
        uint256 cliffDuration
    ) external {
        require(msg.sender == vestingContract, "Only vesting contract");
        require(investor != address(0), "Invalid investor address");

        uint256 monthlyAmount = (amount * 30 days) / vestingDuration;

        vestingSchedules[investor][schedule] = VestingSchedule({
            totalAmount: amount,
            ...
            }
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

`Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

REF - Redundant Execution Flag

Criticality	Minor / Informative
Location	Governance.sol#L990
Status	Unresolved

Description

The contract is using a boolean flag `executed` to track whether a vote has already been executed, despite already enforcing this logic through the condition `require(v.tallyTime == 0, "Vote has already been tallied")`. Since the `tallyTime` variable alone is sufficient to determine whether the vote has been executed—by checking whether it is zero or non-zero—the additional `executed` state variable introduces unnecessary storage usage. This redundancy increases gas costs for state writes and adds complexity to the contract without providing any additional security or clarity.

```
function executeVote(
        uint256 numberOfVote
    ) external onlyVoter mustInit voteExists(numberOfVote) {
        ...
        require(v.tallyTime == 0, "Vote has already been tallied");
        ...
        v.executed = true;
        v.tallyTime = block.timestamp;
        ...
    }
```

Recommendation

It is recommended to remove the `executed` variable and rely solely on the `tallyTime` check to determine whether a vote has been executed. This will reduce storage writes, lower gas consumption, and simplify the contract's state management.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Governance.sol#L855
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
require(
    (stabilityModule.getGovernanceStake(msg.sender).startTime >
        block.timestamp - 90 days) ||
        (stabilityModule
            .getGovernanceStake(msg.sender)
            .lastGovContractCall > block.timestamp - 90 days) ||
        (lastVoteTimestamp[msg.sender] > block.timestamp - 90 days),
    "stake is inactive, hasn't been used in 3 months"
);
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and

gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

UEC - Unoptimized External Calls

Criticality	Minor / Informative
Location	Governance.sol#L930,965,997
Status	Unresolved

Description

The contract is making multiple external calls to the `stabilityModule` contract within a single transaction execution. Specifically, repeated invocations of functions such as `updateLastGovContractCall`, `getGovernanceStake`, and `getTotalPoolAmount` can result in unnecessary gas consumption. This pattern not only increases the execution cost for users interacting with the contract but may also reduce the efficiency and scalability of the system when operations are performed frequently or under high transaction volume. In these scenarios involving conditional logic that depend on these external values, the calls could significantly impact gas efficiency and user experience.

```
stabilityModule.updateLastGovContractCall(msg.sender);
...
stabilityModule.getGovernanceStake(msg.sender).amount >
    stabilityModule.getTotalPoolAmount() / 10,
...
```

Recommendation

It is recommended to optimise gas usage by caching the results of external calls to the `stabilityModule` contract locally at the beginning of the function execution. By retrieving each required value once and reusing the cached result throughout the function, the contract can avoid redundant calls and improve gas efficiency without altering its logical behaviour.

UECE - Unrestricted External Call Execution

Criticality	Minor / Informative
Location	Governance.sol#L990
Status	Unresolved

Description

The contract includes functionality that allows the execution of arbitrary external calls through the `executeVote` function if a proposal passes governance. Once the vote meets quorum and support thresholds, the function performs a low-level call to an externally specified `voteAddress` with arbitrary function data, as defined by the proposal initiator. There are no restrictions or validations on what this address is or what kind of function is being executed.

This introduces a crucial security risk, as a malicious actor who gains sufficient voting power could craft a proposal that calls into a contract they control, potentially triggering destructive operations such as fund drains, token transfers, or altering governance parameters. Since the call is encoded with user-supplied data and executed without any further access control or input sanitisation, this mechanism provides a powerful vector for executing arbitrary logic on-chain.

```
  function executeVote(
        uint256 numberOfVote
    ) external onlyVoter mustInit voteExists(numberOfVote) {
        //75 percent of pool needs to vote

        Vote storage v = voteInfo[numberOfVote];

        require(
            v.amountSupporting + v.amountAgainst + v.amountAbstained >
                (stabilityModule.getTotalPoolAmount() * 3) / 4,
            "75% of pool has not voted yet!"
        );
        require(v.tallyTime == 0, "Vote has already been tallied");
        uint256 _duration = 2 days;
        require(
            block.timestamp - v.startTime > _duration,
            "Time for voting has not elapsed"
        );

        if (
            v.amountSupporting >
            (stabilityModule.getTotalPoolAmount() * 51) / 100
        ) {
            v.result = true;
            address _destination = v.voteAddress;
            bool _succ;
            bytes memory _res;
            (_succ, _res) = _destination.call(
                abi.encodePacked(v.voteFunction, v.data)
            );
            //When testing _destination.call can require higher gas than
the standard. Be sure to increase the gas if it fails.
            require(_succ, "error running _destination.call");
            ...
            }
```

Recommendation

It is recommended to place strict constraints on which contracts and functions can be targeted by governance proposals. This can be achieved by implementing an allowlist of permitted destination addresses and function selectors, or by introducing a vetting phase through a multisig or timelock. Furthermore, governance votes that lead to external

execution should undergo extensive off-chain review or incorporate formal verification tools to ensure safety.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	Governance.sol#L925,930,997,1051
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
rewardDistributor = rewardDistributorAddress;
stabilityModule = IStabilityModule(stabilityModuleAddress);
vestingContract = vestingAddress;
...
stabilityModule.updateLastGovContractCall(msg.sender);
...
require(
    v.amountSupporting + v.amountAgainst + v.amountAbstained >
        (stabilityModule.getTotalPoolAmount() * 3) / 4,
    "75% of pool has not voted yet!"
);
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Governance.sol#L897
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
_schedule
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	Governance.sol#L138,543
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _contextSuffixLength() internal view virtual returns (uint256)
{
        return 0;
    }

function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Governance.sol#L904,906
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 monthsSinceLastClaim = (block.timestamp -
            schedule.lastClaimTime) / 30 days;

uint256 newVestedAmount = monthsSinceLastClaim * schedule.monthlyAmount;
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	Governance.sol#L7,89,117,147,312,623,662,791
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.20;
pragma solidity ^0.8.4;
...
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Governance.sol#L89,312
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
pragma solidity ^0.8.4;
...
```

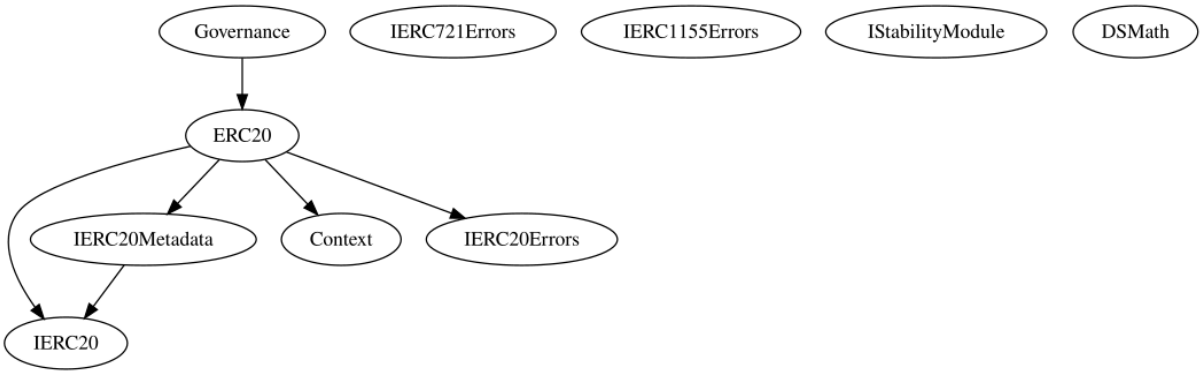
Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

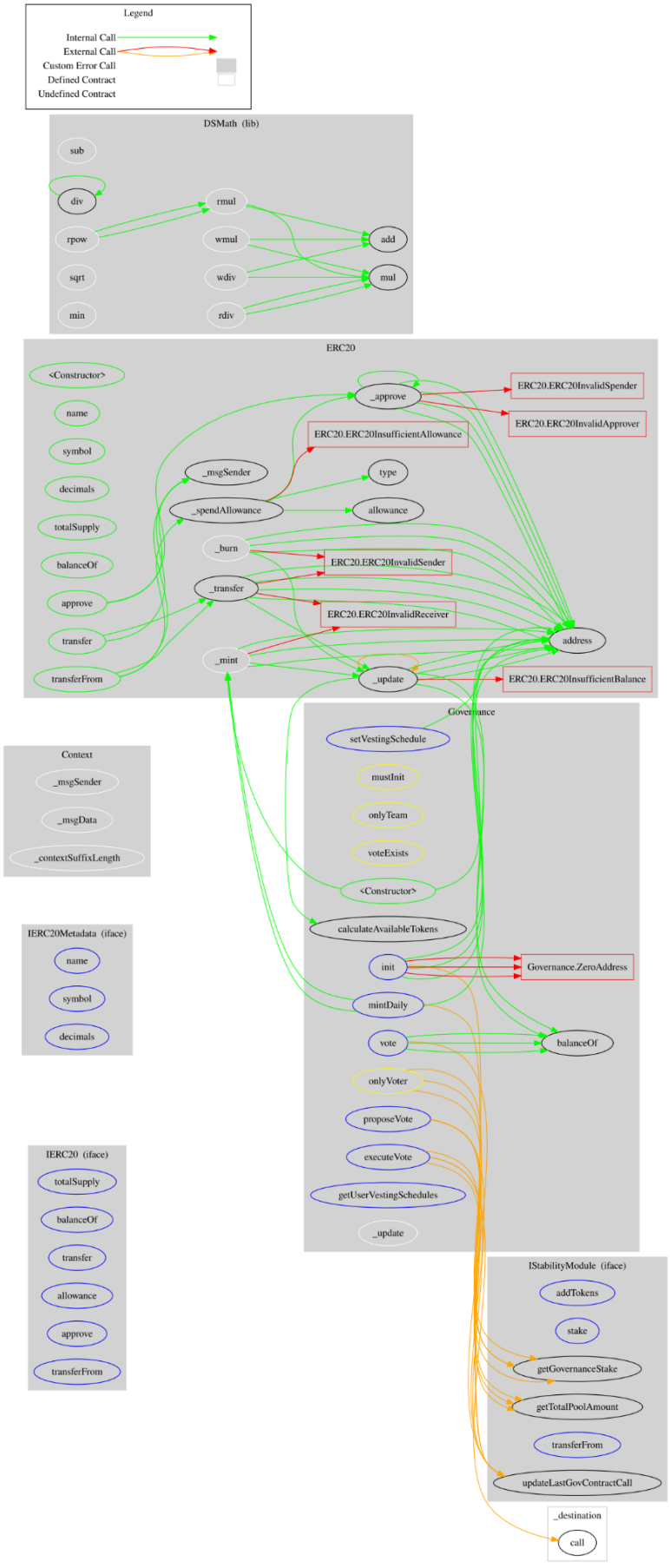
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Governance	Implementation	ERC20		
		Public	✓	ERC20
	calculateAvailableTokens	Public		-
	init	External	✓	onlyTeam
	mintDaily	External	✓	mustInit onlyTeam
	proposeVote	External	✓	onlyVoter mustInit
	executeVote	External	✓	onlyVoter mustInit voteExists
	vote	External	✓	onlyVoter mustInit voteExists
	setVestingSchedule	External	✓	-
	getUserVestingSchedules	External		-
	_update	Internal	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
DSMath (lib)
sub
div
rpow
sqrt
min
rmul
wmul
wdiv
rdiv
add
mul
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
approve
transfer
transferFrom
_approve
_msgSender
_spendAllowance
_burn
_transfer
_mint
_update
type
allowance
address
ERC20.ERC20InvalidSpender
ERC20.ERC20InvalidApprover
ERC20.ERC20InsufficientAllowance
ERC20.ERC20InvalidSender
ERC20.ERC20InvalidReceiver
ERC20.ERC20InsufficientBalance
Context
_msgSender
_msgData
_contextSuffixLength
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Governance
setVestingSchedule
mustInit
onlyTeam
voteExists
<Constructor>
calculateAvailableTokens
init
mintDaily
vote
onlyVoter
proposeVote
executeVote
getUserVestingSchedules
_update
Governance.ZeroAddress
balanceOf
IStabilityModule (iface)
addTokens
stake
getGovernanceStake
getTotalPoolAmount
transferFrom
updateLastGovContractCall
_destination
call

Summary

The Chrysus Governance contract implements a controlled daily minting and a decentralized voting system for the GOV token. This audit investigates security vulnerabilities, business logic issues, and potential improvements to ensure robust functionality and integrity.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io