Cyberscope

Audit Report

Chrysus

September 2024

Files Chrysus, Governance, Lending, RewardDistributor, StabilityModule

Audited by © cyberscope

Table of Contents

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

Audit Updates

Initial Audit	26 Feb 2024
Corrected Phase 2	26 Mar 2024
Corrected Phase 3	26 Mar 2024
Corrected Phase 4	06 Sep 2024

Source Files

Filename	SHA256
StabilityModule.sol	ae9bae6eb7c9465d93b24fe95848da7f25ebe98842b42a040fcd2949e90d44c1
RewardDistributor.sol	a3e9290a2e5d40a701c2f1ae7b66d64025a12df2b8578476d6028204dbc713ff
Lending.sol	90d629a3ccfb62c46331c95af759db78106e30b852c26de6cfcee11e3a13659b
Governance.sol	cf64a79ddf4ea98e58a38b0970871d42fdac3efc2c4cfa0c832ed51a1658c60f
Chrysus.sol	56ebdc47b0e8e03f057c2557ae9cd7ee828fdad78a0d5e8cae3b0e2227c7b08f
libraries/UQ112x112.sol	88e01a4a372e75e0f0d52e924e6e4f34449ae085c51d81c34722aca826d48930
libraries/Math.sol	107b7cba5446c0c8a822c7780b5efab414030fbc54d1ece6e5a6923d2b60a801

interfaces/IStabilityModule.sol	7db5defa43276eacb2a45aac8bd19068a0 04f2099b964a7497fc34e8b57dafaa
interfaces/IRewardDistributor.sol	09b5eee7d20fe8644c0ce0b9cedb232847 e6d8a97977088899d4e84d9a3d779b
interfaces/ILending.sol	afb210358215dfd20466115b45f6542c807 352e8d013fc86fe308654f4ebd920
interfaces/IGovernance.sol	45ea50d78e4c48569de96a486f6ac867f0f dff64a34d9e2328d68e27d94fc8ca
interfaces/IChrysus.sol	bc5f0bbd8b32056d379ca532952ff31458c 66f4831ceb70f359928105a711d3d
interfaces/AggregatorV3Interface.sol	827afc70fa4a24952f4b424062992f0d88a0 221e8913057f86aab97c048bd3c3

Overview

Chrysus contract

The Chrysus contract is a multi-functional decentralized financial instrument designed to provide liquidity through collateralized positions while maintaining robust risk management and governance mechanisms. It allows users to deposit approved collateral types and mint Chrysus (CHAU) tokens in return, based on a precise ratio defined by the value of the collateral, which is secured by oracles. The contract emphasizes flexibility, allowing for new collateral types to be introduced via governance decisions, enhancing the ecosystem's adaptability. Key functionalities include collateral management, liquidation processes for undercollateralized positions, and fee withdrawal mechanisms that redistribute accumulated fees to core components of the platform like the treasury, swap solution, and stability module.

Deposit Collateral Functionality

The `depositCollateral` function lets users lock in collateral (native token or other tokens) and mint CHC tokens. The contract calculates the amount of CHC to mint based on real-time oracle prices for CHC, XAU, and the collateral asset. A fee is applied during the deposit, ensuring that a portion of the collateral is allocated to the contract. This structure not only ensures liquidity generation but also provides a transparent system where users can securely mint CHC, backed by assets.

Withdraw Collateral Functionality

The `withdrawCollateral` function enables users to reclaim their collateral by burning an equivalent amount of CHC tokens. The contract calculates the collateral's value using the oracle's pricing data, ensuring a fair and accurate exchange. This functionality is essential for enabling users to manage their collateralized positions, with built-in security mechanisms to prevent fraudulent actions, such as double withdrawals or over-claims.

Liquidate Functionality

The `liquidate` function safeguards the system by allowing the liquidation of positions that fall below the required collateralization ratio. Users can liquidate undercollateralized

positions and earn a liquidation reward. The liquidation process is calculated based on oracle data, ensuring that all liquidation events are fair and follow the protocol's risk management standards. This feature ensures the health of the overall system by removing unstable positions and maintaining proper collateralization.

Fee Withdrawal Mechanism

The contract's fee withdrawal mechanism allows governance to redistribute accumulated fees across various areas, such as the treasury, swap solution, and stability module. Fees are earned from user interactions, and the withdrawal process ensures that key areas of the platform continue to receive necessary funding, promoting long-term stability.

Roles

Governance

Governance holds authority over critical functions, ensuring that the contract remains adaptable and secure:

- function `addCollateralType`
- function `withdrawFees`
- function `updateLiquidatorReward`

Users

Users interact with the contract by utilizing core financial functions:

- function `liquidate`
- function `depositCollateral`
- function `withdrawCollateral`

Governance Contract

The Governance contract implements a robust and secure governance framework for a decentralized platform, enabling stakeholders to participate in crucial decision-making processes using governance tokens. It features a token-minting function to distribute rewards to various ecosystem components and includes advanced voting mechanisms to ensure a transparent and democratic decision-making process.

Minting and Reward Distribution

The `mintDaily` function mints new tokens on a daily basis, distributing 90,000 tokens to the `rewardDistributor` for incentivizing participants, while 10,000 tokens are allocated to a reserve controlled by the `team`. This distribution model supports the ongoing incentivization of liquidity providers, borrowers, lenders, and the overall ecosystem's stability.

ProposeVote Functionality

The `proposeVote` function allows eligible stakeholders to propose new votes on key changes or actions in the ecosystem. A stakeholder can propose a vote if their staked governance tokens exceed 10% of the total pool. The proposal includes details like the address being voted on, the function to be called, and any additional data needed. This ensures that only significant contributors can propose changes, aligning governance with those most invested in the platform’s success.

Vote and Execution Mechanism

The voting process is based on the `vote` and `executeVote` functions. Once a vote is proposed, stakeholders can cast their votes in favor, against, or abstain from the proposal. The weight of each vote is proportional to the number of tokens held by the voter. A vote can only be executed if at least 75% of the total token pool has participated. If the proposal receives over 51% support from the voting pool, the vote passes, and the specified function is executed. If not, the proposal fails.

Voting Eligibility

To maintain governance integrity, the contract includes strict voter eligibility requirements. Stakeholders can only vote or propose if they have actively staked their governance tokens and participated in governance within the past 90 days. Additionally, the stake must be at least 30 days old. This encourages long-term commitment and active participation in governance.

Roles

Team

The `team` address holds authority over critical functions, such as:

- function `init`
- function `mintDaily`

onlyVoter

Addresses with the `onlyVoter` role are eligible to:

- function `proposeVote`
- function `executeVote`
- function `vote`

Lending Contract

The Lending contract facilitates a decentralized lending system where users can lend, borrow, repay, and withdraw Chrysus tokens. This contract integrates key financial features, such as interest rate calculation and collateral handling, to maintain liquidity and ensure smooth user participation. Additionally, daily participation rewards are distributed to active users, incentivizing continued engagement.

Lend Functionality

The `lend` function allows users to supply Chrysus tokens to the lending pool. When a user lends, their `lendAmount` is updated, contributing to the platform's total supplied volume. This action also records the user's participation with the `rewardDistributor` for daily rewards. The lent amount is transferred from the user to the contract and tracked within their position.

Borrow Functionality

The `borrow` function allows users to take out loans by offering collateral. Borrowers are required to transfer an appropriate amount of collateral, either in tokens or native token, to secure their loan. The borrowed amount is transferred to the borrower, while the contract simultaneously tracks the loan and calculates the applicable interest. The interest rate is determined dynamically based on the utilization rate of the lending pool. This encourages fair borrowing while maintaining liquidity within the pool.

Repay Functionality

The `repay` function enables users to return their borrowed amounts along with accrued interest. Upon repayment, the borrower's outstanding debt is reduced, and any collateral tied to the loan is returned to the user. The repayment process ensures the total supplied volume increases, reinforcing the overall liquidity of the platform.

Withdraw Functionality

Users can retrieve their lent amounts using the `withdraw` function. This function checks that the requested withdrawal amount does not exceed the user's lend position and ensures

that the contract has enough balance to honor the request. Upon successful withdrawal, the user's `lendAmount` is reduced, and the requested amount is returned to the user.

Interest Rate Calculation

Interest rates are calculated dynamically based on the pool's `utilizationRate`, which is the ratio of borrowed to supplied funds. The interest rate is adjusted between a lower bound of 0.1% and an upper bound of 50%, balancing the supply and demand for loans. The contract also includes a `rebalanceInterestRate` function, allowing governance to recalibrate the upper bound of the interest rate when necessary.

Roles

Team

The `team` address, managed by governance, has authority over critical functions, such as:

- function `rebalanceInterestRate`

Users

Users can interact with the following functions:

- function `lend`
- function `borrow`
- function `repay`
- function `withdraw`

StabilityModule Contract

The `StabilityModule` contract plays a key role in managing decentralized governance staking, incentivizing stakeholder participation, and ensuring active engagement with the platform's decision-making process. By allowing users to stake governance tokens, the contract encourages long-term commitment while distributing rewards for sustained participation.

Staking Functionality

The `stake` function allows stakeholders to deposit their governance tokens into the StabilityModule. This action records the start time of the stake, locks the tokens for a specified period, and updates the total pool amount. The staking mechanism tracks user interaction with the governance system and ensures that the staking process aligns user incentives with the platform's stability and governance goals.

- Stakeholders must lock in their tokens for a minimum period (e.g., 30 days), promoting a long-term commitment to the platform’s success.
- Each user’s stake is recorded with details such as the start time and the amount staked, ensuring accurate tracking for reward distribution.

Withdrawal and Rewards

Stakeholders can withdraw their staked tokens after the lock-up period using the `withdrawStake` function. Upon withdrawal, the system calculates the stakeholder's share of rewards, which includes both staking rewards (in governance tokens) and collateral rewards. The reward amount is based on the stakeholder's proportion of the total pool, which incentivizes participants to remain engaged and stake significant amounts.

- **Staking Rewards**: Distributed in governance tokens based on the user's share of the total staked pool.
- **Collateral Rewards**: Users receive collateral rewards from a variety of approved collateral types, determined by the contract’s balance. If a user holds a large proportion of the total stake, they receive a larger share of the collateral rewards.

Roles

Governance

The `governance` address, managed by the team, holds authority over specific governance-related functions:

- function `updateLastGovContractCall`

Users

Users can interact with several functions within the StabilityModule contract:

- function `stake`
- function `withdrawStake`

RewardDistributor Contract

The `RewardDistributor` contract is a decentralized reward distribution system that allows users to earn incentives for their participation across various reward pools. The contract ensures fairness in distributing rewards based on user exposure and manages any leftover incentives for efficient use. This contract is essential for platforms where user engagement is rewarded through token distribution.

Reward Pools Initialization

The contract starts by initializing reward pools with the `initializeRewardPool` function. This function allows the governance team to set up multiple reward pools, each with a specific daily incentive allocation. These pools are designed to distribute rewards based on the amount of engagement (referred to as exposure) from users on a daily basis.

- **Daily Incentives**: Each pool is assigned a daily reward allocation, incentivizing users based on their participation in that pool.
- **Pool Contracts**: A list of contracts representing different pools is initialized, enabling the system to track and distribute rewards accurately across multiple pools.

Participation and Exposure Tracking

The contract tracks user participation in reward pools through the `recordParticipation` function. Whenever a user participates in an activity within a pool, their exposure is recorded for that day. This exposure determines how much of the daily reward they will receive compared to other participants in the same pool.

- **User Exposure**: A user’s exposure is logged daily, representing their activity and stake in that pool. The more exposure a user has, the larger their share of the daily incentive.
- **Daily Checkpoints**: The contract maintains checkpoints for each pool, recording the total exposure for a day and each user's share. This helps in distributing rewards fairly based on actual participation.

Claiming Rewards

Users can claim their accrued rewards through the `claimRewards` function. The system calculates rewards based on user exposure over multiple days, ensuring that only the days when the user actively participated are counted.

- **Accrued Rewards**: Rewards are accumulated based on a user's participation days and the amount of exposure they had relative to the pool's total exposure. Users can claim rewards in governance tokens proportional to their contribution.
- **Efficient Claiming**: The contract automatically clears past participation records after a claim to optimize future reward distributions.

Leftovers Management

One of the unique aspects of this contract is its handling of unclaimed rewards, referred to as "leftovers." If there is no user exposure on a particular day, the rewards for that day are not distributed. Instead, they accumulate in the pool's leftover balance, which can later be withdrawn by the governance team.

- **Leftovers**: If no users participate in a pool on a specific day, the daily incentive for that day is stored as leftovers.
- **Withdrawal**: The governance team can withdraw accumulated leftovers, ensuring that unused rewards are not left idle.

Roles

Governance Team

The `team` address, representing the governance team, holds special permissions within the contract:

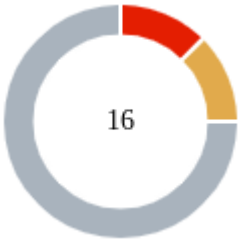
- function `initializeRewardPool`
- function `withdrawLeftovers`

Users

Users can interact with the following functions:

- function `recordParticipation`
- function `claimRewards`

Findings Breakdown



Critical	2
Medium	2
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	1	1	0	0
Minor / Informative	12	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	FDI	Fee Deduction Inaccuracy	Unresolved
●	ICW	Incorrect Collateral Withdrawal	Unresolved
●	CSD	Collateral Selection Discrimination	Acknowledged
●	IER	Incorrect Element Removal	Unresolved
●	CLU	Console Log Usage	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MZC	Missing Zero Check	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RDA	Redundant Decimal Adjustment	Unresolved
●	RVC	Redundant Value Check	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	UIV	Unnecessary Initialization Variable	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

FDI - Fee Deduction Inaccuracy

Criticality	Critical
Location	Chrysus.sol#L420
Status	Unresolved

Description

The contract is calculating a fee amount within the `depositCollateral` function but fails to deduct this fee from the `actualAmountTransferred` value before updating the `userDeposits[msg.sender][collateralType]` mapping. As a result, the mapping inaccurately reflects the total collateral deposited by the user, including the fee amount that should not be considered as part of the user's deposit. Additionally the fees can be withdrawn by the governance wallet using the `onlyGovernance` function, this inconsistency leads to the contract storing inaccurate data, which will cause discrepancies in the contract's state and misrepresent the actual value of user deposits.

```
function depositCollateral(
        address collateralType,
        uint256 amount
    ) public payable {
        ...

        // Calculate the 10% fee
        fees = DSMath.div(actualAmountTransferred, 10);

        approvedCollateral[collateralType].fees += fees;
        approvedCollateral[collateralType].balance +=
            actualAmountTransferred -
            fees;

        ...

        // Adjust the amount to mint based on the CHC/XAU ratio and the
minimum collateral requirement
        amountToMint = DSMath.div(
            amountToMint * 10000,
            ratio * approvedCollateral[collateralType].minCollateral
        );

        ...

        userDeposits[msg.sender][collateralType].minted += amountToMint;
        userDeposits[msg.sender][collateralType]
            .deposited += actualAmountTransferred;

        ..
        );
    }
```

Recommendation

It is recommended to update the `actualAmountTransferred` value immediately after calculating the fees to reflect the correct amount deposited by the user, excluding the fee. This adjustment ensures that the `userDeposits` mapping accurately tracks the net deposited amounts and maintains consistency in the contract's storage, preventing potential miscalculations and data inaccuracies.

ICW - Incorrect Collateral Withdrawal

Criticality	Critical
Location	StabilityModule.sol#L74
Status	Unresolved

Description

The contract contains the `withdrawStake` function, where, if the `collateralType` equals the zero address (indicating native token), the function sets the `collateralReward` to the entire balance of the contract. It then transfers this full balance directly to the user, without correctly calculating the portion of funds that the user is entitled to. This flaw allows any user who invokes the `withdrawStake` function to drain all the native funds from the contract in a single transaction, posing a significant risk of complete fund depletion.

```
    function withdrawStake() external mustInit nonReentrant {
        Stake storage s = governanceStakes[msg.sender];

        ...

        address[] memory Collaterals =
IChrysus(chrysus).getApprovedTokens();

        for (uint i = 0; i < Collaterals.length; i++) {
            address collateralType = Collaterals[i];
            uint256 collateralReward = collateralType == address(0)
                ? address(this).balance
                : DSMath.wmul(
                    stakingShare,
                    IChrysus(collateralType).balanceOf(address(this))
                );

            if (collateralReward > 0) {
                if (collateralType == address(0)) {
                    (bool success, ) = msg.sender.call{value:
collateralReward}(
                        ""
                    );
                    require(success, "Failed to transfer reward");
                ...
```

Recommendation

It is recommended that the withdrawal logic be modified to ensure that users only receive the portion of funds corresponding to their stake. The contract should calculate and send the exact amount the user is eligible to withdraw, rather than the entire contract balance, to prevent the unauthorized draining of funds.

CSD - Collateral Selection Discrimination

Criticality	Medium
Location	contracts/Chrysus.sol#L115,345
Status	Acknowledged

Description

The contract calculates the amount of tokens to mint based on the `minCollateral` value associated with each `_collateralType`. This approach inadvertently incentivizes users to select the collateral type with the lowest `minCollateral` value for depositing, as it results in a higher amount of tokens being minted for the same deposited value. Given the current setup, where different collateral types have significantly varied `minCollateral` values (e.g., ETH having a lower `minCollateral` value compared to DAI), users will deposit the collateral type perceived to offer the better minting ratio, potentially skewing the distribution of collateral types within the platform. This bias towards selecting certain collateral types based solely on their `minCollateral` values could lead to an imbalance in the platform's collateral.

```
        _addCollateralType(_ab.daiAddress, 120, _ab.oracleDAI);
        _addCollateralType(address(0), 267, _ab.oracleETH);

        ...

        uint256 amountToMint = DSMath.div((actualAmountTransferred -
fees) * uint256(priceCollateral), uint256(priceCHC));

        // Adjust the amount to mint based on the CHC/XAU ratio and the
minimum collateral requirement
        amountToMint = DSMath.div(amountToMint * 10000, ratio *
approvedCollateral[collateralType].minCollateral);

        );
```

Recommendation

It is recommended to revise the token minting calculation to normalize the influence of the `minCollateral` value across different collateral types. One approach could involve

adjusting the minting ratio to account for the relative market values or volatility of the collateral types, ensuring that the amount of tokens minted reflects not just the `minCollateral` value but also the inherent risk and liquidity characteristics of the collateral. These changes should mitigate the bias towards selecting certain collateral types and promote a more balanced and diversified collateral portfolio.

Team Update

The team has acknowledged that this is not a security issue and states:

The varying minCollateral values across different collateral types are an intentional design choice, fundamental to the Chrysus coin (CHC) system's collateralization mechanism. The minCollateral values are carefully calibrated to facilitate dynamic adjustments in the collateralization ratios based on the deviation of CHC from its peg to the price of gold. Lower minCollateral values (e.g., ETH) allow for easier CHC creation when traded above the peg, while higher values (e.g., DAI) make CHC creation harder when traded below the peg. This design enables the system to regulate CHC supply and demand, ultimately stabilizing its price around the gold peg, as outlined in our tokenomics paper. While it may incentivize certain collateral selections, this is a necessary trade-off to achieve the project's primary goal of maintaining a decentralized peg.

IER - Incorrect Element Removal

Criticality	Medium
Location	RewardDistributor.sol#L99
Status	Unresolved

Description

The contract is attempting to remove outdated participation days for users by checking the first element (index 0) of the `userParticipationDays` array within a `while` loop. However, it incorrectly uses the `.pop()` method to perform the removal. The `.pop()` method removes the last element of the array, not the first. As a result, the contract does not effectively clear past participation days as intended, potentially leaving old and irrelevant data in the array and causing inaccurate reward calculations or unintended behavior during execution.

```
    function claimRewards() external mustInit nonReentrant {
        ...
        for (uint i = 0; i < poolContracts.length; i++) {
            ...

            // Clear past participation days
            while (

rewardPools[poolContract].userParticipationDays[msg.sender].length > 0
&&

rewardPools[poolContract].userParticipationDays[msg.sender][0] <=
getCurrentDay() - 1
            ) {

rewardPools[poolContract].userParticipationDays[msg.sender].pop();
            }
        }

        ...
    }
```

Recommendation

It is recommended to implement a method that correctly removes the first element of the array when clearing past participation days. This can be achieved by shifting the first with the last element and the use the `.pop` method or utilizing a different data structure that allows for efficient removal from the beginning. Ensuring that the array manipulation aligns with the contract's intended logic will prevent any inconsistencies or errors in the contract's operation.

CLU - Console Log Usage

Criticality	Minor / Informative
Location	RewardDistributor.sol#L7,130
Status	Unresolved

Description

The contract is currently utilizing the `hardhat/console.log` import within its codebase, a practice typically reserved for development and testing phases to debug and verify contract logic. While `console.log` is an invaluable tool for developers to trace and debug contract execution in a test environment, its presence in a contract intended for deployment on the mainnet (production environment) raises concerns. The primary issue is that these logging statements are part of the Hardhat environment and do not function in a live network. Their inclusion in a production-ready contract suggests that the contract may not have been adequately cleaned up post-testing, potentially indicating oversight in the development process.

```
import "hardhat/console.sol";
...
console.log("Total leftovers: ", totalLeftovers/1e18);
```

Recommendation

It is recommended to remove the `hardhat/console.log` import from the contract before deployment to the mainnet or any production environment. This cleanup step is crucial since it reduces the contract's bytecode size, thereby lowering deployment and execution costs, and it eliminates unnecessary code that serves no purpose in a live environment.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Chrysus.sol#L151
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.

Specifically, the `addCollateralType` function centralizes authority with the governance, giving it the power to set the correct and appropriate collateral addresses that the contracts will utilize. This governance control includes the ability to approve new collateral types, set minimum collateral thresholds, and assign oracles for price feeds. While this ensures that the governance can maintain oversight and adapt the system to changing requirements, it also introduces a centralization risk. Any mismanagement or malicious actions by the governance could lead to the approval of unsuitable collateral, potentially compromising the platform's stability.

```
    function addCollateralType(
        address collateralType,
        uint256 minCollateral,
        address oracleAddress
    ) external onlyGovernance {
        if (approvedCollateral[collateralType].approved)
            revert CollateralTypeAlreadyApproved();
        _addCollateralType(collateralType, minCollateral,
oracleAddress);
    }
```

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. It is recommended to decentralize the reward distribution mechanism to mitigate the risks associated with centralized control and to ensure timely and consistent reward disbursement.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	RewardDistributor.sol#L60
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
startDay
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MZC - Missing Zero Check

Criticality	Minor / Informative
Location	StabilityModule.sol#L66,81
Status	Unresolved

Description

The contract is structured with a `withdrawStake` function that includes a `require` statement to ensure the staked amount is greater than zero before allowing a withdrawal. However, the corresponding `stake` function does not have a similar check to validate that the staking amount is greater than zero. As a result, it is possible to create a stake with a zero amount, which would be stored in the contract. However, this zero-value stake would then fail to be withdrawn later due to the `require` check in the `withdrawStake` function, potentially causing confusion or unintended behavior.

```
   function stake(uint256 amount) external {

        Stake storage s = governanceStakes[msg.sender];
        s.startTime = block.timestamp;
        s.amount += amount;
        s.lastGovContractCall = block.timestamp;
        ...

    }


    function withdrawStake() external mustInit nonReentrant{
        Stake storage s = governanceStakes[msg.sender];

        require(block.timestamp >= s.endTime, "Stake must be at least 30
days");
        require(s.amount > 0, "Insufficient stake balance");
        ...
        }
```

Recommendation

It is recommended to include a `require` check within the `stake` function to ensure that the staking amount is greater than zero. This additional validation will prevent zero-amount stakes from being created in the first place, ensuring consistent logic throughout the contract and avoiding potential issues with withdrawals.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	Chrysus.sol#L282,385
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
(, int256 priceCHC, , , ) = oracleCHC.latestRoundData();
...

( , int256 priceCHC, , , ) = oracleCHC.latestRoundData();

( , int256 priceXAU, , , ) = oracleXAU.latestRoundData();
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Lending.sol#L101
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

```
if (collateral != address(0)) {
    success = IERC20(collateral).transferFrom(
        msg.sender,
        address(this),
        _collateralAmount
    );
```

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
```

RDA - Redundant Decimal Adjustment

Criticality	Minor / Informative
Location	Lending.sol#L
Status	Unresolved

Description

The contract is performing calculations to adjust amounts by the difference between `MAX_DECIMALS` and the decimal precision of the `chrysus` token. However, since both the `MAX_DECIMALS` and the decimals of the `chrysus` token are set to 18, the multiplication by `10**(MAX_DECIMALS - IChrysus(chrysus).decimals())` does not provide any meaningful adjustment. The result of this calculation is always 1, making the adjustment redundant. This unnecessary computation adds complexity to the code without affecting the outcome of the calculations.

```
uint256 totalBorrowed = volume.totalBorrowed * 10**(MAX_DECIMALS -
IChrysus(chrysus).decimals());
uint256 totalSupplied = volume.totalSupplied * 10**(MAX_DECIMALS -
IChrysus(chrysus).decimals());
```

Recommendation

It is recommended to set the local variables directly with the values of the necessary variables (`totalBorrowed` and `totalSupplied`) without performing the redundant calculation involving decimals. Removing these unnecessary adjustments will simplify the code, improve readability, and reduce gas costs without changing the contract's functionality.

RVC - Redundant Value Check

Criticality	Minor / Informative
Location	Lending.sol#L83,109
Status	Unresolved

Description

The contract is designed with a `borrowAmount` function that includes a `require` statement ensuring the `borrowAmount` is greater than zero. However, there is an additional `require` statement that checks whether `msg.value` is greater than zero. This second check is redundant because the code also contains a `require` that ensures `borrowAmount` equals `msg.value`. Since the initial check already validates that `borrowAmount` is greater than zero, it inherently guarantees that `msg.value` is also greater than zero when both are equal.

```
   function borrow(uint256 borrowAmount, address collateral) external
payable{
        require(borrowAmount > 0, "Collateral must be greater than 0");
        ...
        require(msg.value > 0, "Invalid Amount");
        require(borrowAmount == msg.value, "Amount Mismatch");
        ...
        }
```

Recommendation

It is recommended to remove the redundant requirement that checks if `msg.value` is greater than zero, as this is already indirectly validated by the condition that ensures `borrowAmount` equals `msg.value`. Simplifying the code in this way can enhance readability and maintainability.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Lending.sol#L170
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.

```
uint256 totalBorrowed = volume.totalBorrowed * 10**(MAX_DECIMALS -
IChrysus(chrysus).decimals());
uint256 totalSupplied = volume.totalSupplied * 10**(MAX_DECIMALS -
IChrysus(chrysus).decimals());
```

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

UIV - Unnecessary Initialization Variable

Criticality	Minor / Informative
Location	StabilityModule.sol#L51
Status	Unresolved

Description

The contract is currently utilizing an `initialized` variable to check whether the contract has already been initialized. However, this variable is redundant because the contract already verifies the initialization status by checking that the `chrysusAddress` is not set to the zero address. Since the presence of a valid `chrysusAddress` inherently indicates that the contract has been initialized, the use of the `initialized` variable adds unnecessary complexity and could be removed.

```
    function init(
        address chrysusAddress
    ) external onlyTeam {
        require(!initialized, "contract is initialized");
        require(chrysusAddress != address(0), "Invalid address");

        chrysus = chrysusAddress;
        initialized = true;
    }
```

Recommendation

It is recommended to eliminate the redundant `initialized` variable and rely solely on checking that the `chrysusAddress` is not equal to the zero address. This approach simplifies the code, reduces storage costs, and maintains the same level of security by ensuring the contract is properly initialized before allowing further actions.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	RewardDistributor.sol#L64,65
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.

```
address[] memory _poolContracts
uint256[] memory _dailyIncentives
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Chrysus.sol#L375,381,387
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 unadjustedAmountMinted = DSMath.div(
            DSMath.mul(amount, ratio *
approvedCollateral[collateral].minCollateral),
            10000
        )
uint256 collateralToReturn = DSMath.div(unadjustedAmountMinted
* uint256(priceCollateral), uint256(priceCHC))
```

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.

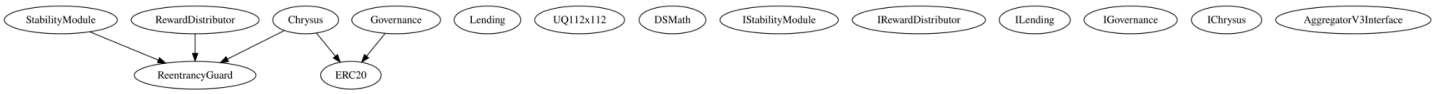
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
StabilityModule	Implementation	ReentrancyGuard		
		Public	✓	-
		External	Payable	-
	init	External	✓	onlyTeam
	stake	External	✓	-
	withdrawStake	External	✓	mustInit nonReentrant
	updateLastGovContractCall	External	✓	-
	getGovernanceStake	External		-
	getTotalPoolAmount	External		-
RewardDistributor	Implementation	ReentrancyGuard		
		Public	✓	-
	initializeRewardPool	External	✓	onlyTeam
	recordParticipation	External	✓	mustInit
	claimRewards	External	✓	mustInit nonReentrant
	withdrawLeftovers	External	✓	onlyTeam mustInit
	getAccruedRewards	Public		mustInit
	getCurrentDay	Public		-

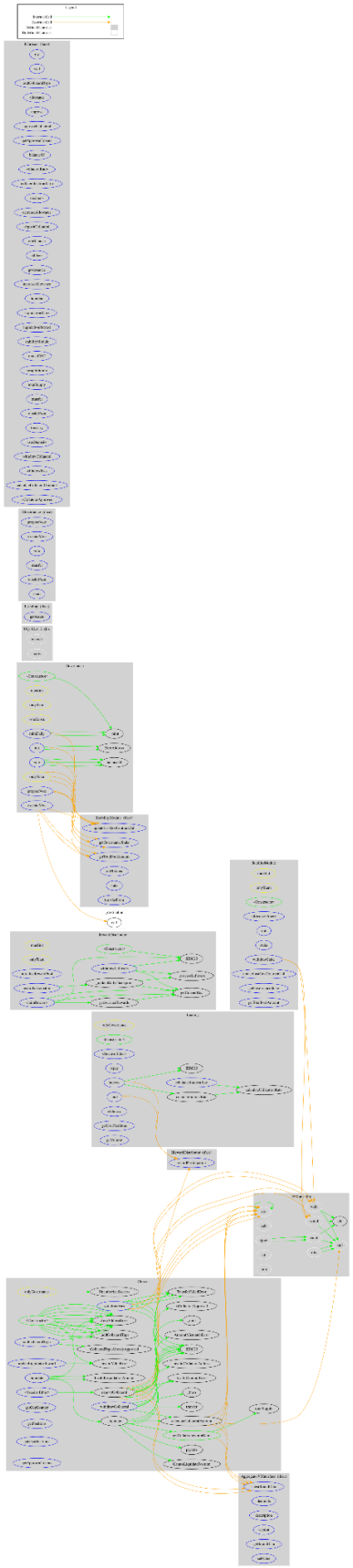
	_updateDailyCheckpoint	Internal	✓	mustInit
	_processLeftovers	Internal	✓	mustInit
Lending	Implementation			
		Public	✓	-
		External	Payable	-
	lend	External	✓	-
	borrow	External	Payable	-
	repay	External	✓	-
	withdraw	External	✓	-
	getUserPositions	External		-
	getVolume	External		-
	calculateUtilizationRate	Public		-
	calculateInterestRate	Public		-
	rebalanceInterestRate	External	✓	onlyGovernance
Governance	Implementation	ERC20		
		Public	✓	ERC20
	init	External	✓	onlyTeam
	mintDaily	External	✓	mustInit onlyTeam
	proposeVote	External	✓	onlyVoter mustInit
	executeVote	External	✓	onlyVoter mustInit voteExists

	vote	External	✓	onlyVoter mustInit voteExists
Chrysus	Implementation	ERC20, ReentrancyGuard		
		Public	✓	ERC20
		External	Payable	-
	addCollateralType	External	✓	onlyGovernance
	liquidate	External	✓	nonReentrant
	withdrawCollateral	External	✓	nonReentrant
	withdrawFees	External	✓	onlyGovernance
	updateLiquidatorReward	External	✓	onlyGovernance
	getCollateralizationRatio	Public		-
	getCdpCounter	External		-
	getPositions	External		-
	getUserPositions	External		-
	getApprovedTokens	External		-
	depositCollateral	Public	Payable	-
	calculateCollateralAmount	Public		-
	isCollateralApproved	Public		-
	_addCollateralType	Internal	✓	
	_liquidate	Internal	✓	

Inheritance Graph



Flow Graph



Summary

The Chrysus DApp implements a comprehensive decentralized financial system that includes governance staking, reward distribution, and lending and borrowing mechanisms to maintain stability and incentivize user participation. This audit investigates security vulnerabilities, business logic flaws, and potential optimizations across all contracts to enhance platform integrity and efficiency.

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