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
Corrected Phase 5	20 Sep 2024

Source Files

StabilityModule.sol	4e8ec8ac5b025dc468d0b78a5ea761f20dc4c54e41c76502482861f908f948b1
RewardDistributor.sol	1c78f205360796b03b3d02d86707e77c0c958c8eb6950096c9b7ed73f5cb8ad4
Lending.sol	e0b0bc0ee70e319773cb1a3561d6bdd61f7a3b04502c22817c2b1ed3be65fe9a
Governance.sol	bb99556125394e2d4195c7a2bbeb25ca7dfb7b0155e40ccf9c2ffc0dde8df2b3
Chrysus.sol	30011581683fdeef5a69a80d830a73c313f16af40299aff6fbf41033fec3c87c

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. The sequencer functionality ensures that the oracles are up-to-date and operational before proceeding with any transaction, preventing stale price feeds or system downtime from affecting the collateralization process. This structure not only ensures liquidity generation but also provides a transparent and reliable 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:

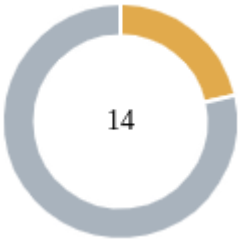
- function `initializeRewardPool`
- function `withdrawLeftovers`

Users

Users can interact with the following functions:

- function `recordParticipation`
- function `claimRewards`

Findings Breakdown



Critical	0
Medium	3
Minor / Informative	11

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	2	1	0	0
Minor / Informative	10	1	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	AOD	Array Order Disruption	Unresolved
Medium	CSD	Collateral Selection Discrimination	Acknowledged
Medium	DMC	Decimal Mismatch Calculation	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Acknowledged
Minor / Informative	FHI	Fee Handling Issue	Unresolved
Minor / Informative	IBV	Incorrect Balance Validation	Unresolved
Minor / Informative	ITC	Incorrect Type Casting	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MNOH	Missing Negative OraclePrice Handling	Unresolved
Minor / Informative	RFC	Redundant Fee Calculations	Unresolved
Minor / Informative	UET	Unnecessary External Transfer	Unresolved
Minor / Informative	L06	Missing Events Access Control	Unresolved
Minor / Informative	L13	Divide before Multiply Operation	Unresolved

AOD - Array Order Disruption

Criticality	Medium
Location	RewardDistributor.sol#L100,205
Status	Unresolved

Description

The contract is pushing values to the `userParticipationDays` array in a sequential manner, ensuring that the days are stored in increasing order. However, during the `claimRewards` function, the contract swaps the first element of the array with the last element before removing the last element. This operation disrupts the original sequential order, causing the latest entry to be at the start of the array instead of the end. As a result, the array no longer maintains a serial order, which can lead to inefficiencies in future operations that depend on the expected sequential structure. This misalignment can cause additional computational overhead and potentially unexpected behavior in other functions relying on the array's order.

```
    function claimRewards() external mustInit nonReentrant {
        uint256 totalRewards = getAccruedRewards(msg.sender);
        require(totalRewards > 0, "No rewards to claim");

        uint256 currentDay = getCurrentDay();

        for (uint i = 0; i < poolContracts.length; i++) {
            address poolContract = poolContracts[i];
            rewardPools[poolContract].userLastClaim[msg.sender] =
                currentDay -
                1;

            uint256[] storage participationDays =
rewardPools[poolContract]
                .userParticipationDays[msg.sender];

            // Clear past participation days
            while (
                participationDays.length > 0 &&
                participationDays[0] <= currentDay - 1
            ) {
                participationDays[0] = participationDays[
                    participationDays.length - 1
                ];
                participationDays.pop();
            }
        }

    function _updateDailyCheckpoint(
        address poolContract,
        address user,
        uint256 amount
    ) internal mustInit {
        uint256 currentDay = getCurrentDay();

       ...
            rewardPools[poolContract].userParticipationDays[user].push(
                currentDay
            )
```

Recommendation

It is recommended to remove outdated entries from the array in a manner that preserves the desired sequential order. This will ensure the array maintains its intended structure, optimizing performance and reducing the risk of logical errors in functions that rely on the array's chronology. Adopting a consistent strategy for adding and removing elements can

help maintain the integrity of the data and improve overall contract efficiency. For instance, consider implementing a more efficient mechanism for managing and clearing old entries, such as using a mapping with a pointer or an optimized data structure, to further enhance performance.

CSD - Collateral Selection Discrimination

Criticality	Medium
Location	Chrysus.sol#L119,344
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);

        ...

        // Calculate the amount of CHC to mint
        uint256 amountToMint = DSMath.div((actualAmountTransferred) *
priceCollateral, priceCHC);

        // Adjust the amount to mint based on the CHC/XAU ratio and the
minimum collateral requirement
        amountToMint = DSMath.div(amountToMint, 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.

DMC - Decimal Mismatch Calculation

Criticality	Medium
Location	Chrysus.sol#L300
Status	Unresolved

Description

The contract is using a `for` loop to sum up the `totalCollateralValue` by adding each `singleCollateralValue` based on the balance and price of different collateral types. However, each `singleCollateralValue` is calculated without standardizing the decimals of the various assets involved. Since different tokens can have varying decimal places, this approach can lead to inconsistencies and inaccuracies in the final `totalCollateralValue`, resulting in potential miscalculations in the contract's logic that depends on this total.

```
for (uint256 i = 0; i < approvedTokens.length; i++) {
  collateralType = approvedTokens[i];
  //read oracle price
  collateralPrice = _checkOraclePrice(
      approvedCollateral[collateralType].oracle
  );
  //multiply collateral amount
  singleCollateralValue =
      approvedCollateral[collateralType].balance *
      collateralPrice;
  totalcollateralValue += singleCollateralValue;
       ...
```

Recommendation

It is recommended to standardize all collateral values to a common decimal format before performing any summation. This can be done by normalizing each `singleCollateralValue` to a consistent number of decimal places, ensuring accurate and reliable calculations for the total collateral value and avoiding potential misinterpretations of asset values due to differing decimals.

CR - Code Repetition

Criticality	Minor / Informative
Location	Chrysus.sol#L285,430
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically, the `getCollateralizationRatio` and `calculateCollateralAmount` functions share similar code segments.

```
    function getCollateralizationRatio() public view returns (uint256) {
        //get CHC price using oracle
        uint256 priceCHC = _checkOraclePrice(oracleCHC);
        //multiply CHC price * CHC total supply
        uint256 valueCHC = priceCHC * totalSupply();
        if (valueCHC == 0) {
            return liquidationRatio;
        }
        ...

        return DSMath.div(totalcollateralValue * 1e12, valueCHC) / 1e6;
    }

function calculateCollateralAmount(
        uint256 amount,
        address collateral
    ) public view returns (uint256) {
        // Read CHC/USD oracle
        uint256 priceCHC = _checkOraclePrice(oracleCHC);

       ...

        return collateralToReturn;
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Chrysus.sol#L139
Status	Acknowledged

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.

Team Update

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

The addCollateralType function is intentionally controlled by governance, not by a single entity. This design ensures that adding new collateral types is subject to a decentralized decision-making process while the governance model involves token holders voting on proposals.

The ability to add new collateral types is crucial for the platform's long-term adaptability to market changes.

We have implemented safeguards, including community scrutiny of proposals, time-locks on execution, and checks to prevent duplicate collateral additions.

While this design introduces some level of centralization, we believe it strikes a necessary balance between platform flexibility and security.

FHI - Fee Handling Issue

Criticality	Minor / Informative
Location	Lending.sol#L111
Status	Unresolved

Description

The contract is calculating the `actualTransferredAmount` based on the `_collateralAmount` specified for the collateral type. Following this, the contract includes a check that reverts the transaction if the `actualTransferredAmount` is less than the original `_collateralAmount` declared. This implementation creates an issue when the collateral asset has any associated transfer fees. In such cases, the `actualTransferredAmount` received by the contract will be less than the `_collateralAmount` due to these fees, causing the transaction to revert. This restricts the contract to only support collateral tokens that do not implement transfer fees, limiting its functionality.

```
        if (collateral != address(0)) {
            uint256 userBalanceBefore =
IERC20(collateral).balanceOf(msg.sender);
            success = IERC20(collateral).transferFrom(
                msg.sender,
                address(this),
                _collateralAmount
            );
            require(success, "failed to transfer specified amount");
            uint256 userBalanceAfter =
IERC20(collateral).balanceOf(msg.sender);
            uint256 actualTransferredAmount = userBalanceBefore -
userBalanceAfter;
            require(
                actualTransferredAmount >= _collateralAmount,
                "Insufficient collateral transferred"
            );
```

Recommendation

It is recommended to handle cases where the collateral asset may deduct fees upon transfer. The contract could either adjust the expected `actualTransferredAmount` to account for potential fees or include an explicit allowance for acceptable discrepancies between the declared and received amounts. Alternatively, if the current behavior is intentional, the team should clearly acknowledge that the contract only supports collateral types without fees, meaning that the transferred amount must exactly match the declared `_collateralAmount`.

IBV - Incorrect Balance Validation

Criticality	Minor / Informative
Location	Lending.sol#L111
Status	Unresolved

Description

The contract is calculating the `userBalanceBefore` and `userBalanceAfter` based on the balance of the `msg.sender` to determine the `actualTransferredAmount` during the transfer process. This approach may not accurately reflect the actual amount transferred to the contract due to potential fluctuations in the user's balance caused by other concurrent transactions or allowances. Consequently, this could result in incorrect assumptions regarding the actual amount transferred to the contract, leading to potential inaccuracies in collateralization checks or other balance-related validations.

```
if (collateral != address(0)) {
            uint256 userBalanceBefore = IERC20(collateral).balanceOf(
                msg.sender
            );
            success = IERC20(collateral).transferFrom(
                msg.sender,
                address(this),
                _collateralAmount
            );
            require(success, "failed to transfer specified amount");
            uint256 userBalanceAfter =
IERC20(collateral).balanceOf(msg.sender);
            uint256 actualTransferredAmount = userBalanceBefore -
                userBalanceAfter;
            require(
                actualTransferredAmount >= _collateralAmount,
                "Insufficient collateral transferred"
            );
```

Recommendation

It is recommended to check the balance of the contract itself before and after the transfer operation, instead of relying on the balance of the `msg.sender`. This approach provides a more accurate assessment of the actual amount received by the contract, ensuring correct validations and preventing any discrepancies caused by concurrent user transactions or changes in allowances.

ITC - Incorrect Type Casting

Criticality	Minor / Informative
Location	StabilityModule.sol#L99
Status	Unresolved

Description

The contract is casting the `collateralType` address to the `IGovernance` interface to perform a token transfer using the `transfer()` function. Although this casting works in this particular context due to the fact that the contract at `collateralType` implements the ERC20 `transfer()` function, this approach is fundamentally incorrect. The `IGovernance` interface is intended for other functionalities, and using it for ERC20 transfers can lead to confusion and potential errors, especially if the interface changes or if `collateralType` refers to a contract that doesn't fully implement the expected ERC20 interface.

```
bool success = IGovernance(collateralType).transfer(
            msg.sender,
            collateralReward
        );
```

Recommendation

It is recommended to use the proper ERC20 interface for token transfers to ensure clarity and correctness in contract interactions. This can be achieved by casting `collateralType` to the standard ERC20 interface (`IERC20`) when performing token transfers. Doing so will make the code more readable, maintainable, and less prone to errors, while also aligning with the intended usage of each interface. Additionally, consider implementing checks to ensure that the `collateralType` is indeed an ERC20 token before proceeding with the transfer operation.

MC - Missing Check

Criticality	Minor / Informative
Location	Chrysus.sol#L379,504
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the contract is missing a check to verify that the `ratio` value is greater than zero, and is also missing a check to verify that the `deposited` value of the `userDeposits` is greater than the `collateralToReturn`.

```
// Read the CHC/USD oracle price
uint256 priceCHC = _checkOraclePrice(oracleCHC);
// Read the XAU/USD oracle price
uint256 priceXAU = _checkOraclePrice(oracleXAU);
// Calculate the CHC/XAU ratio
uint256 ratio = DSMath.div(priceCHC, priceXAU);
```

```
uint256 collateralToReturn = calculateCollateralAmount(
    amount,
    collateralType
);
userDeposits[msg.sender][collateralType].minted -= amount;
userDeposits[msg.sender][collateralType]
    .deposited -= collateralToReturn;
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MNOH - Missing Negative OraclePrice Handling

Criticality	Minor / Informative
Location	Chrysus.sol#L535
Status	Unresolved

Description

The contract is converting the `oraclePrice` obtained from the Chainlink Data Feeds, which is of type `int256`, into an unsigned integer (`uint256`) without verifying whether the price is negative. Since some prices in financial markets can be negative (e.g., during extreme market conditions), directly casting a negative `oraclePrice` to `uint256` can result in a large, incorrect value, potentially leading to erroneous calculations, mispricing, and unintended behaviors within the contract.

```
(, int256 oraclePrice, , uint256 updatedAt, ) = asset.latestRoundData();
...
price = uint256(oraclePrice);
```

Recommendation

It is recommended to add a require check to verify that the `oraclePrice` is greater than zero before performing any further calculations or conversions. This ensures that only valid, non-negative prices are used in the contract, preventing potential errors and safeguarding the integrity of price-dependent logic.

RFC - Redundant Fee Calculations

Criticality	Minor / Informative
Location	Chrysus.sol#L220
Status	Unresolved

Description

The contract is calculating the `feeShares` values (such as `treasuryFeesShare`, `swapSolutionFeesShare`, and `stabilityModuleFeesShare`) regardless of whether the `_fees` value is greater than zero. As a result, if the `_fees` value is zero, these calculations are unnecessary and redundant, leading to wasted computational resources and increased gas usage without any impact on the contract's state or logic. This inefficiency can accumulate over multiple transactions, resulting in higher costs and potential performance degradation, especially in a high-frequency usage scenario.

```
uint256 _fees = approvedCollateral[collateralType].fees;
approvedCollateral[collateralType].fees = 0;
uint256 treasuryFeesShare = DSMath.wdiv(
    DSMath.wmul(_fees, 3000),
    10000
);
uint256 swapSolutionFeesShare = DSMath.wdiv(
    DSMath.wmul(_fees, 2000),
    10000
);
uint256 stabilityModuleFeesShare = _fees -
    (treasuryFeesShare + swapSolutionFeesShare);
if (_fees > 0) {
```

Recommendation

It is recommended to move the `feeShares` calculations inside the conditional block that checks whether `_fees` is greater than zero. This ensures that the calculations are only performed when they are relevant, avoiding unnecessary computations. Additionally, implementing this check will optimize the gas usage and improve the overall performance of the contract, reducing costs for users. It is also advisable to document this condition clearly

within the code to avoid future misunderstandings or misuse by other developers, ensuring the contract remains efficient and maintainable.

UET - Unnecessary External Transfer

Criticality	Minor / Informative
Location	Chrysus.sol#L497
Status	Unresolved

Description

The contract is using an external transfer function (`safeTransferFrom`) to move tokens between users within the same contract. This process involves additional checks and operations, leading to increased gas consumption. Since the ERC20 token is represented by the same contract (`address(this)`), performing an external transfer is redundant and inefficient for internal accounting purposes. This approach can significantly increase the operational costs of the contract due to unnecessary interactions with the ERC20 interface.

```
IERC20(address(this)).safeTransferFrom(
    liquidator,
    userToliquidate,
    amount - liquidatorReward
)
```

Recommendation

It is recommended to perform the transfer using an internal transfer mechanism, such as updating internal state variables or balances within the contract. This approach will reduce gas costs and simplify the process, while still accurately reflecting the transfer of value between users.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	StabilityModule.sol#L50
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
chrysus = chrysusAddress
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Chrysus.sol#L441,449
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 ratio = DSMath.div(priceCHC, priceXAU)
uint256 unadjustedAmountMinted = DSMath.mul(
            amount,
            ratio * approvedCollateral[collateral].minCollateral
        )
```

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.

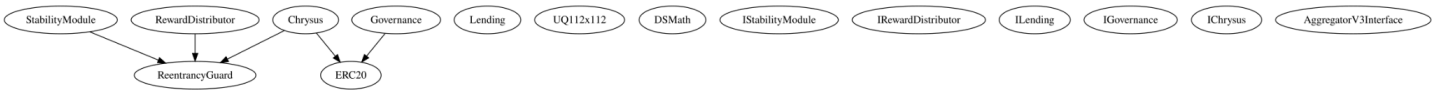
Functions Analysis

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		-

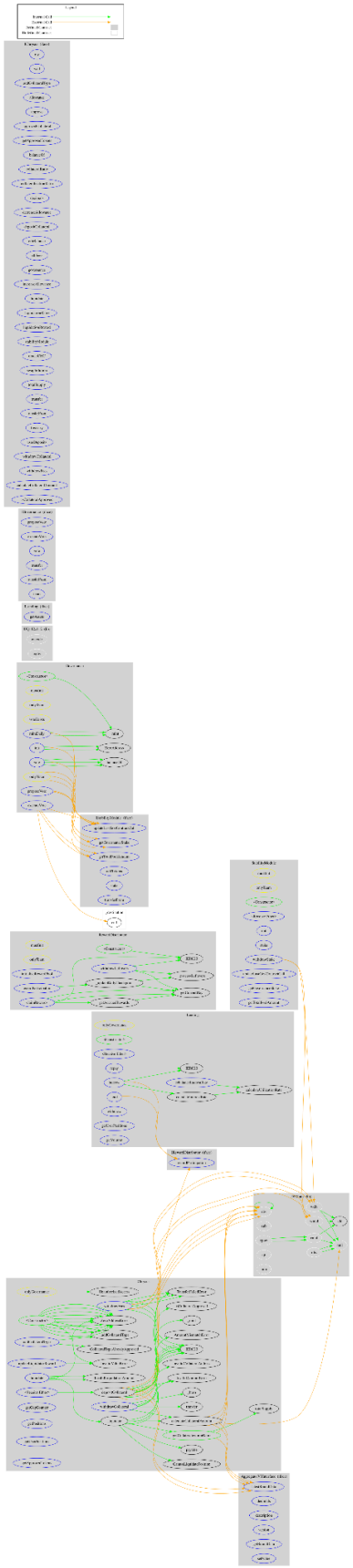
	_updateDailyCheckpoint	Internal	✓	mustInit
	_processLeftovers	Internal	✓	mustInit
Lending	Implementation			
		Public	✓	-
		External	Payable	-
	lend	External	✓	-
	borrow	External	Payable	-
	repay	External	✓	-
	withdraw	External	✓	-
	getUserPositions	External		-
	getVolume	External		-
	calculateUtilizationRate	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	✓	
	_checkOraclePrice	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