BlockApex

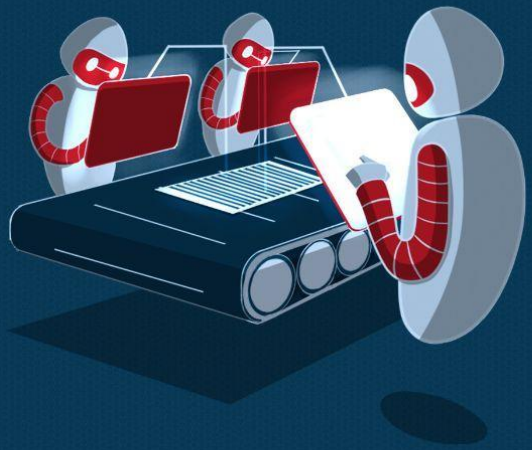
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Powered by XORD

PREFACE

Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings

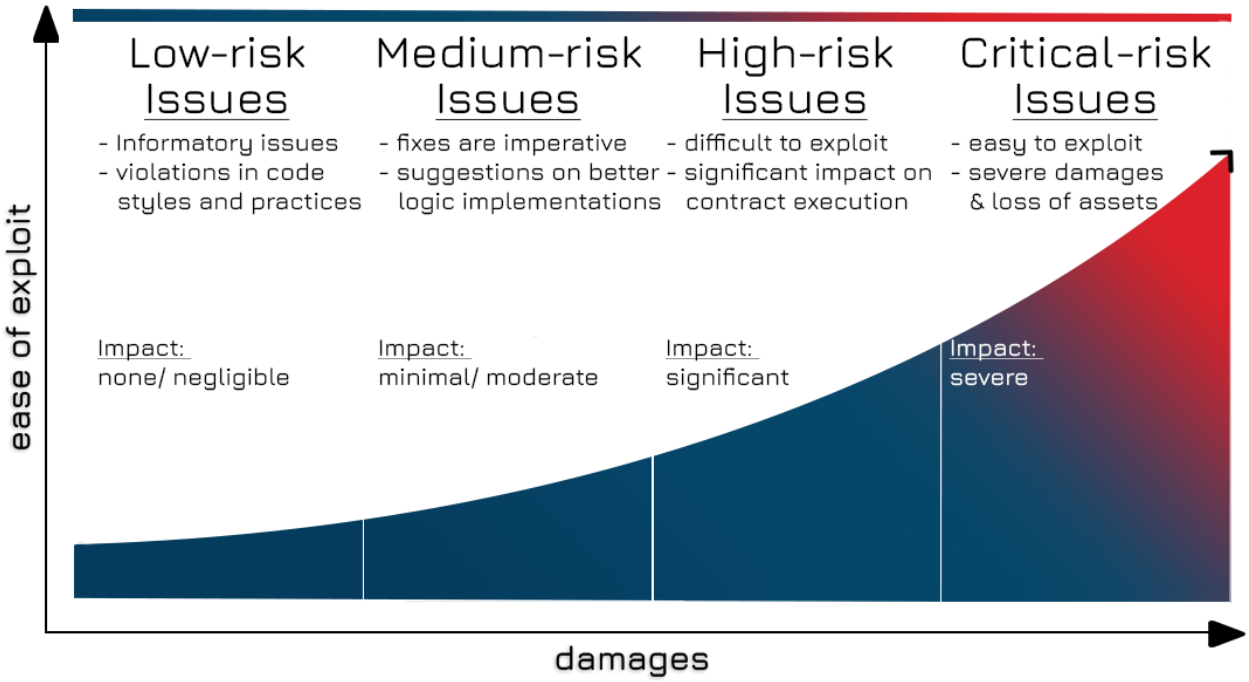


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Project Chrysus (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which started on 8th October, 2022

Name
Project Chrysus
Auditor
BlockApex
Platform
Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
https://github.com/blackalbino01/ProjectGold
White paper/ Documentation
https://bit.ly/3D9oYqo
Document log
Initial Audit Completed: October 17th, 2022
Final Audit Completed: December 22nd, 2022

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulnerabilities**. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Project Chrysus aims to be a fully decentralized ecosystem revolving around Chrysus Coin. Chrysus Coin (Chrysus) is an ERC20 token, deployed on the Ethereum network which is pegged to the price of gold (XAU/USD) using Decentralized Finance (DeFi) best practices. The ecosystem around Chrysus will involve a SWAP solution, a lending solution and an eCommerce integration solution allowing for the use of Chrysus outside of the DeFi ecosystem.

One of the main goals of Chrysus is to not just closely follow the price of gold, but also to be a cash flow generating token. This is achieved through the Chrysus Governance Token (CGT) which will serve both as a decentralization tool for the system and as a reward tool for Chrysus token minters. Fees collected through the different components of the Project Chrysus ecosystem will be re-distributed to CGT token holders who actively participate in the stability mechanisms of the platform

System Architecture

Project Chrysus

Project Chrysus (PC) is a Decentralized Finance (DeFi) ecosystem, revolving around the Chrysus Coin (CHC: a synthetic asset tracking the price of gold) and the Chrysus Governance Token (CGT). Users can generate CHC by leveraging ETH and DAI as collateral, also known as Collateralized Debt Positions (CDPs)

Chrysus Coin

Chrysus (CHC) aims to become a de-facto stable coin pegged to the price of gold. The pegging mechanism will be fully decentralized and will be a hybrid between the solutions developed by *Maker* and *Synthetix* .
It borrows the best aspect from both systems where it has:

- Lower collateral requirement than Synthetix
- Improved reward structure for opening of positions compared to Maker
- Non-inflationary stability mechanisms for recapitalization of the system
- 1:1 profit with the actual tracked asset

Governance Token (CGT)

Chrysus Governance Token (CGT) is the governance token issued as a reward for participants in the ecosystem. CGT will be a perpetually inflationary token, which will be rewarded for:

- Minting Chrysus
- Providing liquidity to Chrysus
- Chrysus Lending and Borrowing

The CGT token's main purpose is to be used for governance via voting on the system's parameters. An additional use of the CGT token will be the stability system, where people who stake CGT will receive a percentage allocation of all fees collected throughout the Chrysus Ecosystem. This ensures both the stability of Chrysus versus its peg and well providing multiple cash flow streams.

For instance: A user deposits ETH & DAI to mint CHC, based on this they will be eligible for part of the daily CGT allocation. The user can use this Chrysus token in the *Lending Borrowing module*, enabling them to receive additional CGT. The user then stakes CGT in the *Stability Module*, enabling them to receive % of all fees collected on the platform.

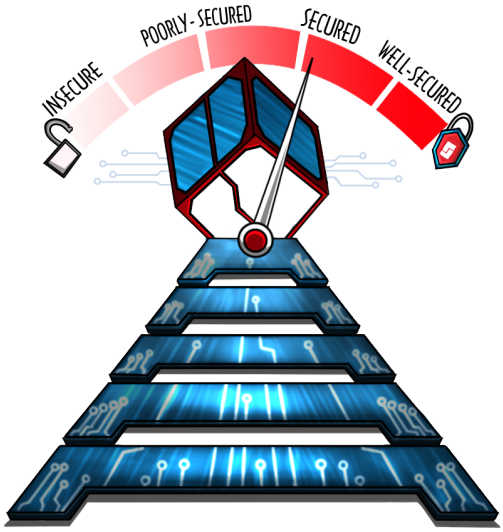
Methodology & Scope

The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process spanning over a time of one (1) week.

Starting with the recon phase, a basic understanding was developed and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software and security design patterns, code styles, best practices and identifying false positives that were detected by automated analysis tools like Slither.

AUDIT REPORT

Executive Summary

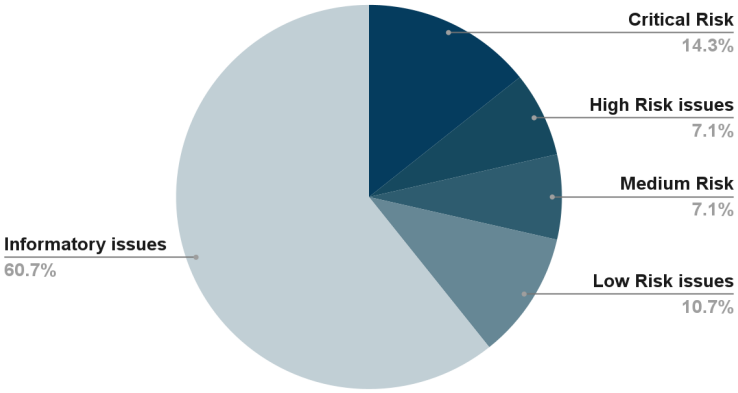


The analysis indicates that the contracts under scope of audit are **working properly.**
Our team performed a technique called “Filtered Audit”, where the contract was separately audited by three individuals. Followed by a rigorous process of manual testing and automated review using slither for static analysis and foundry for fuzzing invariants. All the flags raised were re-tested to identify the false positives.

Our team found:

# of issues	Severity of the risk
4	Critical Risk issue(s)
2	High Risk issue(s)
2	Medium Risk issue(s)
3	Low Risk issue(s)
17	Informatory issue(s)

Proportion of Vulnerabilities



Key Findings

#	Findings	Risk	Status
1.	Incorrect LP shares calculation	Critical	Fixed
2.	DoS attack stops minting completely	Critical	Fixed
3.	Incomplete functionality in the Chrysus contract	Critical	Fixed
4.	Potential reentrancy risk in Chrysus Token	Critical	Fixed
5.	Potential manipulation of Governance voteCount	High	Fixed
6.	Unchecked return values dependance	High	Fixed
7.	Rounding-off errors in collateralRatio	Medium	Fixed
8.	Unorthodox options to liquidate	Medium	Fixed
9.	Governance token contract does not emit relevant events	Low	Fixed
10.	Ensure contract is initialized	Low	Fixed
11.	Implement checks effects interactions pattern	Low	Fixed
12.	Improve functions layout	Informatory	Fixed
13.	Avoid using floating pragma solidity version	Informatory	Fixed
14.	Inexistent arithmetic overflow checks	Informatory	Fixed
15.	Low test coverage & documentation	Informatory	Acknowleged
16.	Use proper linting and prettier for codebase	Informatory	Fixed
17.	Use vetted codebase	Informatory	Fixed

18.	Remove unused state variables	Informatory	Fixed
19.	Solidity docs: Coding Styles	Informatory	Fixed
20.	Mark as immutable	Informatory	Fixed
21.	Misleading naming	Informatory	Fixed
22.	Wrapped calling of variables	Informatory	Fixed
23.	Constructor gas optimizations	Informatory	Fixed
24.	Improve constructor readability	Informatory	Fixed
25.	Gather all redundant checks inside modifiers	Informatory	Fixed
26.	Early return from function	Informatory	Fixed
27.	Introduce events in the contract Chrysus	Informatory	Fixed
28.	Redundant check against zero address for argument	Informatory	Fixed

Detailed Overview

Critical-risk issues

1. **Incorrect LP shares calculation**

   Source: contracts/Pair.sol

   Description:
   In the `Pair` contract, the `mint` function mints `liquidity` shares, using a formula as follows;

```
} else {
  liquidity =
    (amount0 * _totalSupply) / _reserve0 - //difference calculated here
    ((amount1 * _totalSupply) / _reserve1);
}
```

   The concerning issue is that the function uses the `difference` of two values to calculate the liquidity share.

   A simple edge case test confirms the vulnerability, that is, if a user deposits 100 and 100, no LP tokens are minted, completely forbidding the user to claim back the deposited amount of tokens entirely via the `burn` function which fails with with the UniswapV2 error message of `insufficient burn liquidity`.

   In the meanwhile, Uniswap V2 uses the same formula as above but selecting the `minimum` of the above two factors, the user share is calculated accordingly to `mint` an appropriate amount of LP tokens and burning these LP tokens results in claim of appropriate deposited tokens.

   This is a critical flaw in the `Chrysus Pair contract` tokenomics concerning the minting and burning functionalities and should be handled with immense care during restructuring and thorough testing.

   Proof of Concept:
   Consider a scenario, a user deposits 100 and 80 amounts of tokens 0 and 1 respectively, they will receive 20LP tokens representing the user's share of `liquidity`.

If the user then tries to `burn` these 20LP tokens, the burn formula (being indifferent from the UniswapV2's standard `burn`) miscalculates the reserves to give the user an irrational amount of the original deposited tokens as shown in the below test case.

Hence, in all cases of subtracting the reserves, users won't be able to withdraw their actual share of reserves resulting in loss of the original deposited tokens.

The following test verifies the values;

```
 it("check liquidity share calculation formula", async function () {

   await dai.connect(daiHolder).approve(pair.address, BigInt(10000E20))
   await dai.connect(daiHolder).transferFrom(DAI_HOLDER, pair.address,
BigInt(100E18))
   await chrysus.transfer(pair.address, BigInt(100E18))
   await pair.connect(team).mint(team.address)

   console.log("\n User1 has minted \n");

   await dai.connect(daiHolder).approve(pair.address, BigInt(10000E20))
   await dai.connect(daiHolder).transferFrom(DAI_HOLDER, pair.address,
BigInt(100E18))
   await chrysus.transfer(pair.address, BigInt(80E18))
   await pair.mint(accounts[0].address)

   console.log("\n User2 has minted \n");

   await pair.transfer(pair.address, BigInt(await
pair.balanceOf(accounts[0].address)));
   await pair.burn(accounts[0].address);

   console.log("\n User2 has burnt \n");

   console.log("dai received: ", await dai.balanceOf(accounts[0].address));
   console.log("chu received: ", await chrysus.balanceOf(accounts[0].address));
 });
```

Results from the tests:

```
  Chrysus tests
amount0 minted:          10000000000000000000
amount1 minted:          10000000000000000000

 User1 has minted

amount0 minted:          10000000000000000000
amount1 minted:          80000000000000000000

 User2 has minted

liquidity to burn:        2000000000000000000


 User2 has burnt

dai received:  BigNumber { value: "3333333333333333333" }
chu received:  BigNumber { value: "9850000000000000000000" }
    ✔ check liquidity share calculation formula (225ms)
```

```
/** User2 amounts:
 * Deposited DAI: 10000000000000000000
 * Received DAI:  98500000000000000000000
 *
 * Deposited CHC: 80000000000000000000
 * Received CHC:  33333333333333333333
 */
```

Remedy:
It is recommended to restructure the formula following the UniswapV2 logic, as above, to compute over the `minimum` value of the two reserves instead of the `difference`.
Alternatively, the burning functionality can also be redefined by following the minting formula in an `off-by-one manner`, i.e. the current minting logic could be mirrored in the burning function as well when the user tries to `burn` their deposited tokens.

Status: Fixed

2. **DoS attack stops minting completely**

   Source: contracts/Governance.sol

   Description:
   In the `Governance` contract, the function `mintDaily` mints governance tokens (CGT) on a daily basis to multiple contracts. However, if it is called right before 24 hours have passed, the function still executes with a zero minting amount but also resets the variable `lastMintTimestamp`.

   Consider a scenario, an adversary calls this function after 23 hours have passed since last calling, the function (having no checks) will calculate the `number of days` passed since last mint, that is, `zero`, will successfully mint `zero` amounts for the current day and reset the `lastMintTimestamp` to `block.timestamp`. This design leads to DoS attacks as the minting can be permanently stopped repetitively calling this function in the nick of time as it approaches a 24 hour cycle.

   Proof of Concept:
   The following code snippet exhibits the failing functionality;

```
function mintDaily() external {
    uint256 numDays = (block.timestamp - lastMintTimestamp) / 86400;
   ...
}
```

   Remedy:
   Place a check to revert the execution if the value of variable `numDays` is not equal to `one`.

   Status: Fixed

3. Incomplete functionality in the Chrysus contract

Source: contracts/Chrysus.sol

Description:
In the `Chrysus` contract, the `liquidate` function enables self liquidation for users who initially deposit collateral to the contract and executes a **swap** functionality on the `UniswapV2Pair` while liquidating. This swap requires that the amounts to be swapped should be transferred before calling the `swap` function leading to a conflicting assumption that;

1. `msg.sender` in this scenario shall always be an EOA, since it allows users to `deposit` and `withdraw` collateral,
2. meanwhile, assuming that the same user should interact with `Chrysus` contract (as it looks up to the mapping of `user deposited collateral`) expecting to transfer the amounts of tokens before swap occurs through a callback feature OR externally via atomic transactions.

This leads to two possible scenarios:

- Either the `msg.sender` is considered a smart contract in context of the `liquidate` function as it will require external actions to pre-transfer the amounts of tokens to `UniswapV2Pair swap` call using the callback feature implementation,
- Or all interactions with the `Chrysus` contract are done through a periphery layer that implements all the validatory checks and transfers the amounts on EOA user's behalf during the `liquidate` call, leading to an invalid state of `msg.sender` being the periphery itself.

Thus if the swap callback is considered, the issue arises that only contracts implementing it will be allowed to liquidate themselves leading to the point that originally just these contracts should be allowed to make deposits and withdrawals.

Currently, only EOAs are assumed to make deposits and withdrawals, that is, users can directly interact with the token contract, this makes the liquidation mechanism confusing to call, rather test, and the functionality is inevitably rendered as incomplete.

Remedy:
It is highly recommended that the `Chrysus` contract liquidation mechanisms be restructured following the remedies in *issue # 08* of this report and go through effective unit testing confirming all the positive and negative test cases work correctly.

Developer Response:
"At the early stages of the system you need to self-liquidate because you cannot rely on 3rd parties until the system becomes used enough. And even then, there is no good reason not to self-liqdudate... Sure, if the recommendation is to include external liquidations, we can do that. I suggest:

- 1% of the liquidation amount
- Hide the liquidation amounts in order to avoid cherry picking the liquidations"

Conclusion: 1% of the liquidation amount will be given as a reward to the liquidator though the percentage is subjected to change based on the average position size.

Auditor Response: The Client's response elaborates on a new functionality that includes potential changes in the business layer. This iteration of audit is, henceforth considered as, completed from the Auditor.

Status: Fixed

4. Potential reentrancy risk in Chrysus Token

Source: contracts/Governance.sol

Description:
In the `Chrysus` contract, the function `withdrawCollateral` is prone to `reentrancy`; a particular attack vector with which a malicious contract can reenter a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, a second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. There are certain principles advised that mitigate such attack vectors like mutual exclusion (mutex sections), `checks-effects-interactions` (design pattern), `pull over push` (payment method) and `whitelisted calls` (trusted users only).

Proof of Concept:
The following code snippet exhibits the failing functionality;

```
function withdrawCollateral(address _collateralType, uint256 _amount)
  external { ... }
```

This function relies on `msg.sender.call()` to transfer the value to the caller. However, this design is vulnerable as `fallback` calls to `msg.sender` can be easily crafted to reenter the Chrysus contract. Apparently, the interaction with the external contract happens (line 375) when the `msg.sender` is sent `value` during the withdrawal. In this particular case, the external contract may hold certain hidden logic that can allow it to reenter the vulnerable contract.

Remedy:
It is highly recommended to use `lock modifier` with this function to ensure a mutex section is created and that the function is entirely executed in its atomicity.

Status: Fixed

High-risk issues

5. **Potential manipulation of Governance voteCount**

Source: contracts/Governance.sol

Description:

In the `Governance` contract, the function `executeVote` makes an external call to the `destination` address with `function name` and bytes `data` as arguments. In all cases when the call to execute a vote at the `destination` address either passes or fails, the `Vote.result` is always saved as `true`. The return values from this external call are never checked or validated opening a window of false assumptions regarding execution of votes.

*An instance of critical severity is the defected usage of the `executed` state variable inside the struct `Vote`, which is assumed to be updated during this execution but is **never set** throughout the codebase.*

The design to execute votes in the governance contract hints at enforcing execution of arbitral contracts' functions, only requiring a successful `proposal`. However, the assumption that reflects this is that the `proposed vote` is always benevolent and contains trusted contract calls only. In most cases of executing votes in smart contracts, this is a severe design flaw and is therefore recommended as requiring restructure.

Proof of Concept:
The following code snippet exhibits the failing functionality;

```
if (...) {
    v.result = true;
    address _destination = v.voteAddress;
    bool _succ;
    bytes memory _res;
    (_succ, _res) = _destination.call(
        abi.encodePacked(v.voteFunction, v.data)
```

```
    ); //When testing ...
} else { ...
```

Consider a scenario where a vote is successfully proposed through `proposeVote`. Upon calling the `executeVote` function, if the external call to the `destination` address fails due to any inherent internal calls failing; the `vote.result` will be stated as true, but the variable `voteCount` will still be incremented. This shows that the `governance` contract aims to execute votes with failing results and does not handle the cases of reverting external calls.

Remedy:
Place checks on return values of `destination.call` for `success` to always be `true` and restructure the function to maintain the state of vote using the available variable `executed`.

Status: Fixed

6. Unchecked return values dependance

Source: contracts/Chrysus.sol

Description:
In the `Chrysus` contract, there are multiple instances of value transfer functions like `call`, `transfer` and `transferFrom` that need return values to be checked before continuing execution. This ensures that all misassumptions are cleared during external calls and that the return value, specifically the boolean status of the call, is always checked according to the expected result ahead of remaining function execution.

Proof of Concept:
The following code snippets exhibit the failing functionality, in the `withdrawCollateral` function.

Chrysus.sol - Line # 408:
```
(success,) = address(swapSolution).call{ value: DSMath.div(_fees, 5) }("");
```

Chrysus.sol - Line # 412:
```
(success, ) = address(stabilityModule).call{ value: DSMath.wdiv( DSMath.wmul(_fees, 5000), 10000)}("");
```

Chrysus.sol - Line # 440:
```
IERC20(collateralType).transferFrom(address(this), pair, amount);
```

Chrysus.sol - Line # 442:
```
success = IERC20(collateralType).transfer(address (stabilityModule), DSMath.div(_fees, 2));
```

Status: Fixed

Medium-risk issues

7. **Rounding-off errors in collateralRatio**

   Source: contracts/Chrysus.sol

   Description:
   In `Chrysus` contract, the view function `collateralRatio` returns a zero value for the `collateralizationRatio` variable when the `totalCollateral` may be less than the `price of CHC`, causing a proper fractional division. Since solidity by design does not handle floating points, the `collateralizationRatio` is returned after rounding-off to 0 in all cases where the division outputs a number between `0` and `1`. For instance, if `collateralizationRatio` is supposed to be `0.99` it is instead stored as `0`, negating all following calculations.

   Remedy:
   It is highly recommended that the `collateralizationRatio` be restructured such that a factorial `multiplier` of the collateral is always returned instead of `zero` hence, all following arithmetic operations will be handled with precision.

   Status: Fixed

8. **Unorthodox options to liquidate**

   Source: contracts/Chrysus.sol

   Description:
   In the `Chrysus` contract, the current implementation of the `liquidate` function is bound for self liquidations only. Standalone self liquidation is an unorthodox practice, rare in industry standards of DeFi external liquidations. It is therefore suggested that the `liquidate` function be restructured such that:

   - New functionality is added alongside the main `liquidate` function to accommodate self liquidations,
   - `liquidate` shall accept arguments like `userToLiquidate, +`
   - `+collateralType` and `liquidationAmount` with appropriate checks for these added inputs.

   Status: Fixed

Low-risk issues

9. Governance token contract does not emit relevant events

Description:
It is necessary to emit relevant events on critical functions such as before proposing, executing votes or minting tokens.

Remedy:
`Emit` relevant `events` on all user facing functions in the `Governance` contract.

Status: Fixed

10. Ensure contract is initialized

Description:
In the `Governance` contract, a function named `mintDaily` mints tokens to multiple contracts on a daily basis, however the current implementation does not ensure the contract is `initialized` before it starts mining the governance voting powered tokens to different contracts in the protocol.

Remedy:
Place modifier on `mintDaily` function that checks whether the contract is `initialized` with an authorized role.

Status: Fixed

11. Implement checks effects interactions pattern

Description:
The following instances in `Chrysus` contract do not comply with the `checks-effects-interactions` pattern which is a recommended design pattern for smart contracts.

- Move this line before the `_mint` call `userDeposits[msg.sender][_collateralType].minted += amountToMint;`

- Move this line before the `_burn` call `userDeposits[msg.sender][_collateralType].minted -= amountToMint;`

Status: Fixed

Informatory Issues & Gas Optimizations

12. Improve functions layout

Description:
As stated in the Solidity style guide, the functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Status: Fixed

13. Avoid using floating pragma solidity version

Description:
Throughout the codebase, the current implementations of smart contracts have used floating pragma solidity versions. This practice leads to incompatible contracts during testing which is a matter of serious security because developers may have a different set of assumptions than auditors for testing of the contracts. It is therefore recommended to use a fixed solidity version across all smart contracts in the codebase.

Status: Fixed

14. Inexistent arithmetic overflow checks

Description:
Throughout the codebase there are certain contracts that use pre 0.8.x solidity versions and are liable to arithmetic overflow and underflow exceptions. It is

recommended that all the instances covering explicit mathematical operations be handled with safemath or using solidity versions greater than 0.8 otherwise.

Status: Pending

15. Low test coverage & documentation

Description:
Throughout the codebase, the testing is insufficient to support the positive and negative test cases from the protocol engineering team. The low level of testing points to a critical lacking in the project as many cases are

Status: Pending

16. Use proper linting and prettier for codebase

Description:
Throughout the codebase, cosmetic tools like prettier can be utilized to execute linting and code styling as suggested by the SOlidity docs so that code readability is improved.

Status: Fixed

17. Use vetted codebase

Description:
Throughout the codebase, libraries have been written in-house, though the functions and instances use similar conventions yet this strongly suggests that audited and vetted contract/ libraries and dependencies be used for all helper and auxiliary operations. It is therefore recommended that;

- The instances of `SafeERC20` and functions derived from it should be imported from the `OpenZeppelin's` suite of smart contracts.

- `Non-reentrant` functionality should be imported from openzeppelin instead of writing an `internal lock modifier`.

Status: Fixed

18. Remove unused state variables

Description:
In `Chrysus.sol`, the two state variables at the top of the `Chrysus` contract named `ethBalance` and `ethFees` are never used inside the contract and consume unnecessary storage. It is advised to prepare a structured version of the contract that is pre-reviewed for all issues like these before moving to audit.

Status: Fixed

19. Solidity docs: Coding Styles

Description:
For the codebase, in general, it is highly recommended to follow the coding styles and best guides for layout in the solidity docs. For instance, variables in `Chrysus` contract like; `address public governance; address public treasury; address public auction;` can be moved to the top and arranged considering the tight variable design pattern.

Status: Fixed

20. Mark as immutable

Description:
In `Chrysus.sol`, there are some instances for state variables which can be marked as immutable such as; `ISwap public swapSolution; IStabilityModule public stabilityModule; IUniswapV2Pair public pair;`

Also ensure that variables have explicit visibility set as for the following; `AggregatorV3Interface oracleCHC; AggregatorV3Interface oracleXAU;`

Status: Fixed

21. Misleading naming

Description:
In `Chrysus.sol`, the variable `collateralRequirement` can be renamed for a more sensible and user readable convention such as `minCollateral`.
For another instance, the function `collateralRatio` the name `getCollateralizationRatio` can be used instead to increase readability.

Status: Fixed

22. Wrapped calling of variable

Description:
In `Chrysus.sol`, the constructor currently accepts the contract type argument for `ISwapRouter _swapRouter`. For ease of deployment, the argument type can be modified to accept the `address` for `_swapRouter`.

Status: Fixed

23. Constructor gas optimizations

Description:
In `Chrysus.sol`, the constructor currently checks against a list of provided addresses to not be `zero address` using `require` statement. It is suggested that all these checks be replaced with one accumulating `if` block or `revert` with an error message such as `ZeroAddress`.

Status: Fixed

24. Improve constructor readability

Description:
In `Chrysus.sol`, the constructor readability can be increased by applying the standard coding practices and naming conventions;

- Make the function `addCollateralType` as `public` and call it for both DAI and ETH during the constructor execution OR restructure using unchained `internal` function structure as follows;
  - Place all necessary checks in the external functions and then call the internal function to add the collateral type,
  - E.g. the internal function `_addCollateralType` can be called inside the external `addCollateralType` function.

Status: Fixed

25. Use modifiers for recurring checks

Description:
In the codebase for multiple instances some checks are used repetitively throughout different functions, for which it is suggested to use modifiers increasing the code readability and best style guides. For instance, the `feeToSetter` check in the swap contract and the check for governance role can be placed in modifiers.

Alternatively, for the sake of gas optimizations, it is suggested that all `require` statements be replaced with `if` statements accompanied by `revert` messages.

Status: Fixed

26. Early return from function

Description:
In `Chrysus.sol`, the function `collateralRatio` can be modified to have an early `return` implemented such that the expensive `for loop` is avoided in certain practically possible cases. The statement `if (valueCHC == 0)` could be moved to top, following the declaration of variable `uint256 valueCHC` and return right after, to avoid the `for loop`.

Status: Fixed

27. Introduce events in the contract Chrysus

Description:
In `Chrysus.sol`, to keep track of transactions and indexing through block explorers like etherscan it is suggested that all user facing functionalities have `events` implemented in them and emitted appropriately. Following instances are suggested:

- `CollateralDeposited(user, amount)`
- `CollateralWithdrawn(user,amount)`
- `AddedCollateralType(address collateralToken)`
- `Liquidated(liquidator, user, amountLiquidated)`
- `FeesWithdrawn(treasuryFees, swapSolutionFees, stabilityModuleFees)`

Status: Fixed

28. Redundant checking for zero address

Description:
In `Chrysus.sol`, there is a need to restructure code for elegant code blocks in the function named `depositCollateral` function. Currently the function checks the argument `_collateralType` for `zero address` twice during its execution. This is a costly pattern and introduces redundancy in the code blocks. For instance,

```
- if (_collateralType != address(0)) {
        approvedCollateral[_collateralType].fees += tokenFee;
    }
- if (_collateralType != address(0)) {
        bool success = IERC20(_collateralType).transferFrom(
          msg.sender,
          address(this),
          _amount
        );
        require(success);
    }
```

Status: Fixed

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.