

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	11
4	Terminology	12
5	Findings	13
6	Resolved Findings	17
7	Informational	34
8	Notes	37

1 Executive Summary

Dear Frankencoin Team,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Frankencoin according to [Scope](#) to support you in forming an opinion on their security risks.

The Frankencoin system is a set of smart contracts that issue the Frankencoin (ZCHF) on-chain, a stablecoin that is supposed to be pegged to the Swiss Franc. Each Frankencoin minted is backed either by collateral assets or other trusted Swiss Franc stablecoins. The governance of the system is based on veto rights of shareholders that control at least 2% of the total voting power.

The most critical subjects covered in our audit are asset solvency, functional correctness, and access control. Security regarding functional correctness and access control is high, while security regarding asset solvency is improvable, see [No Functionality to Recover From Bridge Failure](#).

The general subjects covered are code complexity, upgradeability, trustworthiness, documentation, and gas efficiency. Contracts in scope of this assessment are not upgradeable and have limited privileged roles. The code is well written. The documentation is improvable and the codebase could be more gas efficient, see [Findings](#).

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	1
•	1
-Severity Findings	3
•	2
•	1
-Severity Findings	11
•	8
•	2
•	1
-Severity Findings	16
•	8
•	3
•	1
•	2
•	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Frankencoin repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	07 Aug 2023	e7b7e379ba20c0e4ce0a3ce90db6ef46a0e56f89	Version 1
2	04 Sep 2023	851487f2221ed170b9e7be25eb833ec89375eb39	Version 2
3	18 Sep 2023	03a88fc298bd1ea886bd6f3b7c707946f6004fbc	Version 3-Dutch Auctions
4	27 Oct 2023	4605fe28ebdda9833dff200b0700e00393c16532	Version 4

For the solidity smart contracts, the compiler version 0.8.20 was chosen.

The following files in the folder `contracts` were in the scope of this review:

- Equity.sol
- ERC20.sol
- ERC20PermitLight.sol
- Frankencoin.sol
- MathUtil.sol
- MintingHub.sol
- Ownable.sol
- Position.sol
- PositionFactory.sol
- StablecoinBridge.sol

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope. Furthermore, external token contracts used as collateral or in the StablecoinBridge in the system were not in the scope of this code assessment. Finally, this assessment was focused on the correctness of the code implementation. The soundness of the financial model was not evaluated.

2.2 System Overview

This system overview describes the initially received version () of the contracts as defined in the [Assessment Overview](#).



At the end of this report section we have added subsections that document changes made in subsequent versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Frankencoin is an extendable stablecoin system, which aims to be self-governing and oracle-free.

2.2.1 Frankencoin

The Frankencoin (ZCHF) contract implements an ERC20 token, which can be minted by the minter role or by positions, which have been registered by minters. When minting, a `reserve` and `fee` amount can be specified, which will be sent to the Equity contract.

Initially, there will be only 2 minters: The MintingHub and the StablecoinBridge. Additional minters can be suggested by anyone that pays a 1000 ZCHF fee. If the suggested minter is not vetoed by governance within the `applicationPeriod`, they will gain the minter role. It is expected that any new minter will be vetoed, unless there is broad prior offchain consensus that the minter with the specified parameters should be added to Frankencoin. Minters should not be upgradeable, as they can never be removed once they have become active.

An unlimited allowance to spend any address's Frankencoin is given to minters and positions, as these are fully trusted roles which have the ability to mint an unlimited number of Frankencoin according to their code.

2.2.2 MintingHub

The MintingHub contract will be one of the initial minters of Frankencoin. It is used to open and challenge positions, which are used to mint/borrow Frankencoin against collateral.

Anyone can open a position, using any parameters they like, such as collateral asset, interest rate, liquidation price, `limit` and more. In particular, they can use a liquidation price that is much higher than the current price. This should lead to the position being challenged or vetoed. To do this, they must pay a 1000 ZCHF fee and deposit a minimum amount of collateral. Before this position is able to mint ZCHF, it must wait for the `initPeriod` of at least 3 days to pass. During this period, the position may be vetoed by governance, if it is found to have unacceptable parameters. A vetoed position cannot mint ZCHF.

At any point, during or after the `initPeriod`, a position can be challenged by anyone that is willing to put down the same type of collateral as is used in the position. Starting a challenge means "I think this position should be liquidated, the collateral is worth less than the liquidation price". A challenged position cannot mint ZCHF. During the challenge, anyone can make a bid on the collateral, saying how many ZCHF they are willing to pay to buy it. The challenge has an `end` time, which is increased to be at least 30 minutes in the future each time that a new bid is made. If a bidder is outbid by another bidder, they receive their full ZCHF bid amount back.

A challenge can end in two ways:

1. Challenge successful: The `end` time passes without the bid surpassing the position's liquidation price. This means the challenge was correct, the position's collateral was not able to fetch a price over the liquidation price, so it should be liquidated. Anyone can call the `end()` function, which transfers the collateral to the bidder and burns the ZCHF bid to close the position. The challenger receives 2% of the bid as a reward. If the bid amount was high enough that some of the position's buffer is remaining (it was still overcollateralized), the excess amount goes to the Equity as liquidation profit. If the position was undercollateralized, the Equity takes the loss. There can be multiple challenges in parallel, potentially with a combined size larger than the total collateral of the position. In this case, the challenges that end first will receive collateral until there is none left.
2. Challenge averted: Someone makes a bid that is greater or equal to the position's liquidation price. This means the challenge was incorrect. The position's collateral is able to fetch more than the liquidation price, so it should not be liquidated. In this case, the challenger is punished by being forced to sell his collateral at this price to the bidder. As the market price is higher, this leads to a

loss for the challenger. They also do not receive a challenge reward. When a challenge is averted, the position is restricted from minting for 1 day. This leaves enough time to start another challenge, in case the challenge was maliciously averted by the position owner. Averting a challenge maliciously by bidding above market price leads to a loss for the bidder and a profit for the challenger, as they are able to sell their collateral above market price. A challenge on a position that is expired cannot be averted.

A challenge can also be split into smaller pieces, in case a bidder does not want to bid on the full collateral amount in the challenge.

Instead of opening a new position and waiting for the `initPeriod` to pass, a user can also clone a position that is already active and is not being challenged. This will allow the user to immediately mint ZCHF at the same terms as the other position. The user may choose to set a shorter expiry, which will lead to a reduction in the fee (they pay the same interest rate, but for a shorter time). The `limit`, which is the maximum amount of ZCHF mintable, of the original position will be reduced by the amount of ZCHF minted in the cloned position. Positions that are clones can be re-cloned and split further, if they have unused `limit` (which happens after a position has been partially or fully repaid). Closed or expired positions cannot be cloned.

2.2.3 Position

Positions are used to mint/borrow ZCHF against collateral. They are created and registered as minters with the Frankencoin contract by the MintingHub. Each position has an owner. Ownership can also be transferred. The collateral is stored in the position contract.

While a position is not challenged, denied, expired, or on `cooldown`, the owner can mint new Frankencoin. The claimed liquidation value of the collateral must always be higher than the value of Frankencoin minted in the position. When minting, the new ZCHF are split into three parts:

- `usableMint`: This amount is sent to the position owner, who can use it for any purpose they desire.
- `reserve`: This amount is sent to the Equity contract, but the owner receives it back when repaying. It represents the overcollateralization of the position. If the position is liquidated through a challenge, any excess `reserve` will go to the Equity as profit.
- `fees`: This amount is sent to the Equity contract and is a profit for the governance shareholders. Together with the expiry, it represents an interest rate. For example, if there is a position with a reserve of 20%, a fee of 5%, and an expiry in two years, the effective interest rate on the `usableMint` will be $0.05 / 0.75 / 2 = 3.3\%$ per year. The fee is deducted upfront and is not returned in case the position is paid back before expiry. In order to fully close the position, the owner will need to acquire the fee amount of ZCHF from another source, such as buying it on a market.

An owner can repay their position partially or fully at any time. They can also increase the collateral deposited. While there is no challenge, they can also withdraw collateral, as long as the collateral value remains higher than the minted amount. They can also adjust the liquidation price of the collateral. The price can be reduced, given the collateral value is still enough. It can always be increased, but in this case, there will be a cooldown of 3 days before the position can mint ZCHF again. This gives enough time to start a challenge on the new price.

2.2.4 StablecoinBridge

The StablecoinBridge contract will be one of the initial minters of Frankencoin.

It is configured with the following immutable variables:

- `chf`: The source CHF stablecoin. This coin should be trusted to hold its value and not be upgradeable or contain functions like pause or freeze unless the admin with access to these functions "is known and enjoys an outstanding reputation with regards to respecting the token holders' rights".
- `zchf`: The Frankencoin contract.



- `horizon`: The expiry time of the bridge, after which it cannot be used to mint more Frankencoin, but Frankencoin can still be burned to retrieve the `chf` stablecoin.
- `limit`: The maximum amount of Frankencoin that may be minted through this bridge at any time. In the worst-case scenario where the `chf` stablecoin is malicious, losses to the Frankencoin system should not exceed `limit`.

Users can deposit another stablecoin, `chf`, to the bridge and mint Frankencoin at a 1:1 ratio. This is intended to help keep the peg of Frankencoin. Frankencoin can be minted as long as there are less than `limit` coins outstanding from this bridge. Frankencoin can be burned to redeem `chf` as long as the `StablecoinBridge` has a balance of `chf`. Aside from this mechanism, a correct choice of interest rates for minting Frankencoin should be used to incentivize bringing the Frankencoin valuation to be equal to that of the Swiss Franc, according to the documentation. However, this does not guarantee a 1:1 peg at all times. Also note that it is not possible to choose a negative interest rate.

Once the `horizon` has passed, no more coins can be minted and it is expected that a new bridge with a new `horizon` will be proposed as a Frankencoin minter. The `limit` should be chosen such that a total failure of the `chf` coin does not lead to a failure of Frankencoin. For this case, see also [No Functionality to Recover From Bridge Failure](#).

2.2.5 Equity

The Equity contract implements an ERC20 token. It contains the governance functionality of the Frankencoin system.

Anyone can mint Frankencoin pool shares (FPS) by depositing ZCHF to the equity using the `invest` function. The price is calculated such that the Market Cap of FPS is always exactly 3 times the equity capital. The following formula is used to compute new shares that are issued:

$$\epsilon \epsilon(\epsilon K) = \epsilon \left(\epsilon \frac{K + \epsilon K}{K} \cdot 1 \right)$$

where θ is the number of existing shares, K is the equity capital, and $\Delta\theta$ is the number of new shares corresponding to the investment of ΔK .

Equity capital is the amount of ZCHF that belongs to the system, not including the Position reserves, which are also held on the Equity contract. For example, if the equity contains 1000 ZCHF and the supply of FPS is 1000, the price is 3. This leads to a Market Cap of 3000. If the system made profits and the equity was now 2000, the price would increase to 6. FPS can be minted at this price, but they can also be burned at this price. In return, the burner will receive ZCHF from the equity. Each mint or burn is subject to a 0.3% fee, which goes to the equity.

FPS holders accrue voting power over time. The voting power is calculated as `balance * averageTimeHeld`. So an address with 2 FPS that has held for one week will have the same voting power as another user with 1 FPS that has held for two weeks. If an FPS is transferred, the time held will be reset. The address receiving the FPS will not immediately receive more voting power, but they will accrue it more quickly in the future.

Votes are used to exercise veto power. Anyone with at least 2% of the total voting power can veto newly proposed minters or positions. Votes can also be delegated. If an address has received a delegation from addresses that have a summed voting power of 2% (including themselves), they can also veto. Someone who has delegated their votes can still exercise their veto power themselves. Delegations are transitive, so if A delegates to B and B delegates to C, C can use the combined vote power of A, B and C.

In order to avoid a malicious party vetoing everything (even legitimate proposals), there is the `kamikaze` function. Using this, an FPS holder can choose to destroy the votes of another address, while destroying the same amount of their own votes. Once the target has less than 2% (including delegation), they can no longer veto. The FPS balance is unaffected, only held time is lost. The consequence of this is that in the extreme case, the veto system falls back to a majority rule. If a group owns 51% of all votes, they can `kamikaze()` everyone else, bringing the voting power of all other holders below 2%. Now nobody else can veto, so the group can propose any minter or position they like, without recourse. After the

`initPeriod` ends, they could mint an unlimited number of ZCHF. However, this would cause severe losses to the equity, of which the group likely owns a significant share to have accrued so much voting power.

Finally, there is a function `restructureCapTable`. This is intended to be used in the worst-case scenario where the system has taken such severe losses that the equity is negative. Anyone with at least 2% of votes can call this function and burn the FPS balances of other addresses. This is intended to be used when someone wants to restore the system, taking the loss for the system by paying in ZCHF to the Equity (including covering socialized position reserve losses) until the equity capital is back to a positive amount. In this case, they should be able to own all FPS tokens, as the other tokens were worthless until the system was saved by this user.

2.3 Roles and Trust Model

The `minter` role is privileged in the system. Any `minter` in the Frankencoin contract is assumed to be fully trusted, with the power to mint unlimited Frankencoin. Initially, only the `MintingHub` and `StablecoinBridge` contracts should have the `minter` role. Frankencoin does not implement functionality to remove existing minters, hence we assume only non-upgradeable contracts that are carefully evaluated by the governance get the `minter` role.

We assume FPS holders that own 51% of the total voting power in the Equity contract always behave in the best interests of the system. FPS holders are trusted to continuously monitor and veto proposals for adding untrusted minters in Frankencoin or opening positions with unfair parameters in the `MintingHub`. In case a minority of FPS holders block the system by vetoing all proposals, the majority of shareholders should act and reduce their voting power.

It is assumed that markets are sufficiently efficient, and a sufficient number of challengers and bidders continuously monitor opened positions and efficiently liquidate unhealthy positions. Additionally, the size of positions should be limited such that liquidation of the full collateral amount is possible without affecting the market price so significantly that there is a loss for the system. It is also assumed that gas costs are low enough that they do not prevent positions from being liquidated without causing losses for the system.

The external stablecoins supported in `StablecoinBridge` are considered to be non-malicious, behave according to the specifications and maintain their peg to the Swiss Franc. These stablecoins should be ERC20-compliant with no special features (e.g., rebasing, fees on transfer, etc.) and revert on failed transfers.

Collateral assets are assumed to be compliant with the ERC20 standard, implement the `decimals` function, use less than 24 decimals and be non-malicious. Only ERC20-compliant tokens without special behavior (e.g., rebasing, fees on transfer) are supported. The collateral should be a liquid asset and easily available, as the overall health of the system depends on the ability to efficiently liquidate undercollateralized positions. The collateral token should not implement a whitelisting mechanism. The collateral token should revert on failed transfers. We also assume that governance limits the exposure of the system to collateral tokens that are upgradeable, by vetoing proposals for tokens that are untrustworthy or pose risks to the system. Collateral tokens should not have transfer hooks (e.g. ERC-777).

2.4 Changes in Version 2

- The Equity contract now also has an unlimited allowance to spend any address's ZCHF, just like minters do.
- The function `onTokenTransfer` has been removed from the Equity contract, hence the only way to mint new shares is by calling the function `invest`.

2.5 Changes in Version 3

introduces a change to the auction mechanism. Instead of using an ascending auction with an end-time that can be increased, a dutch auction mechanism is now used. The dutch auction is split into 2 phases. In the first phase, the price stays constant at the `liquidationPrice` of the position. If nobody bids on the collateral at that price by the end of the first phase, the second phase starts. In the second phase, the price starts at `liquidationPrice` and linearly decreases over time, reaching a price of zero at the `endTime` of phase 2. The auction ends immediately once the full amount of challenged collateral has been bid on. Bidders receive the collateral they bid on immediately, without needing to wait until the auction ends. In the current implementation, phase 1 and phase 2 have the same length, which can be freely chosen when creating a new position. Governance should veto any position created with an auction length that is too long or too short.

Positions that are clones of clones can now extend the expiration time up to the original position's expiration time if the previous clone had shortened the expiration. Clones must now always pay the interest of at least 4 weeks when minting, no matter how long the position is valid. Previously, a position with an expiry in 1 second would only pay interest for 1 second.

Additionally, the governance veto threshold was adjusted from 3% to 2%.

2.6 Changes in Version 4

In , a position is closed if it has less than `minCollateral` when a challenge is successful or averted. Previously this only happened on successful challenges. Additionally, a fee was introduced that is paid when a repaid position is liquidated. Previously, a challenged position could be repaid by the borrower at no cost while a challenge is ongoing, which would allow the borrower to avoid the liquidation loss by repaying.

The specification has also been updated to clarify that tokens with transfer hooks, such as ERC-777 tokens, are not supported as collateral.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the [Resolved Findings](#) section. The findings are split into these different categories:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
--------------------	---

-Severity Findings	1
--------------------	---

- [No Functionality to Recover From Bridge Failure](#)

-Severity Findings	1
--------------------	---

- [Restructuring Equity Shares Does Not Enforce Payment](#)

-Severity Findings	5
--------------------	---

- [Challenge Can Leave Dust Amount](#)
- [Rounding Error in `_calculatePrice`](#)
- [Cloned Positions Can Have Arbitrary Expiration](#)
- [Mismatch of Natspec With Implementation in ERC20](#)
- [Missing Sanity Checks](#)

5.1 No Functionality to Recover From Bridge Failure

CS-ZCHF-002

The documentation states:

```
"In order to protect the Frankencoin from a crash of the connected stablecoins,
the bridge contract is limited in time and volume."
```

However, there is no functionality to recover from such a situation.

Suppose there are 10 million Frankencoin minted in total. 3 million of them are equity. 1 million were minted through the stablecoin bridge.

Now the other stablecoin loses peg and becomes worthless. So there is no value in burning any Frankencoin using the bridge. There is more than enough equity in the Equity contract to cover the loss, but there is no mechanism to update the accounting such that the Equity can take the loss. Governance cannot vote to burn 1 million ZCHF that are in equity. There would be 1 million "unbacked" Frankencoins in circulation, with no way of changing the accounting so that they are backed again.



This should cause Frankencoin to depeg, falling below a price of 1 CHF, even though the loss in the bridge was relatively small and could be covered by equity.

Risk accepted:

Frankencoin understands and accepts the risk.

Frankencoin responded:

The purpose of the StablecoinBridge is to help with bootstrapping the system. In the long run, we would provide swap facilities on decentralized exchanges along the following lines: <https://github.com/Frankencoin-ZCHF/FrankenCoin/issues/10>

5.2 Restructuring Equity Shares Does Not Enforce Payment

CS-ZCHF-009

In case the system suffers losses and its equity goes below `MINIMUM_EQUITY` (1000 ZCHF), any user with at least 3% of the voting power should be able to receive 100% of the equity shares if they pay for the losses. For that, the function `restructureCapTable` allows the caller to wipe equity shares for any address. As noted in [restructureCapTable May Take Multiple Blocks](#), depending on the number of shareholders, `restructureCapTable` can take more than one block to complete.

The function does not enforce that the surviving FPS shareholder actually pays for the losses. Therefore, it is possible that a user with 3% of the voting power calls `restructureCapTable` quickly to remove other shareholders but do not bootstrap the system as expected by paying for the losses.

Risk accepted:

Frankencoin is aware about this behavior of the restructuring functionality but has decided to keep the code unchanged given that pool shares do not have any value when restructuring conditions are met.

5.3 Challenge Can Leave Dust Amount

CS-ZCHF-033

When a challenge is started, it is checked that the challenged amount is larger than `minCollateral`.

However, the amount remaining in the position is not considered. The challenge could be for an amount that leaves a dust amount of collateral in the position after the challenge is successful. If the dust amount is not worth the gas of holding an auction, it won't be liquidated.

Having unliquidated dust collateral can lead to small losses for the system.

Risk accepted:

Frankencoin is aware of this issue but has decided to keep the respective code unchanged.



5.4 Rounding Error in `_calculatePrice`

CS-ZCHF-040

The function `_calculatePrice` in `MintingHub` performs a division before multiplication which introduces a rounding error:

```
return (liqPrice / phase2) * timeLeft;
```

The rounding error is in favor of the bidder against the position's owner or the system (equity pays slightly more if the position causes a loss).

Risk accepted:

Frankencoin is aware of this issue but has decided to keep the function unchanged.

5.5 Cloned Positions Can Have Arbitrary Expiration

CS-ZCHF-006

`MintingHub` implements two functions for cloning a position:

```
- function clonePosition(address position, uint256 _initialCollateral,  
    uint256 _initialMint) public returns (address);  
  
- function clonePosition(address position, uint256 _initialCollateral,  
    uint256 _initialMint, uint256 expiration) public returns (address);
```

The second function takes `expiration` as a user input and does not perform any check if it is in the future. The function `reduceLimitForClone` enforces that `expiration` of the clone is between `start` and `expiration` of the original position. However, one can still create a clone that has `expiration` in the past if current timestamp has passed `start` of the original timestamp.

The position will be able to mint tokens in the `initializeClone` function even though it is expired, circumventing the `alive` check, hence minting ZCHF from an expired position.

Acknowledged:

Frankencoin is aware of the possibility to clone positions that are expired at creation time but considers it to be an expected behavior.

In `clonePosition`, a clone of a position can have an expiry that is at most the expiry of the original position that was created. In particular, if cloning reduces the expiry but then that cloned position is cloned again, the expiry of the second clone can be equal to the original position.



5.6 Mismatch of Natspec With Implementation in ERC20

CS-ZCHF-012

The natspec of the function `approve` states:

```
- `spender` cannot be the zero address.
```

This restriction is not implemented in the function `approve` and it is possible to call the function with `spender` being the zero address.

Similar statements are present in natspec comments of functions `_approve`, and `_burn` but the respective checks are not implemented.

Specification partially changed:

The stated restrictions have been removed from the natspec of `_burn()` and `_approve()`.

The natspec for `approve()` still incorrectly states that `spender` cannot be the zero address. However, this should not be relevant in practice, as the zero address should never have a non-zero balance.

5.7 Missing Sanity Checks

CS-ZCHF-015

The following functions set or update important state variables but do not implement any sanity check on the inputs:

1. `kamikaze()` could check that `target` address is not `msg.sender`.
 2. `Frankencoin.constructor()` could enforce a minimum value for `MIN_APPLICATION_PERIOD` to prevent deployments with wrong parameters.
 3. `MintingHub.constructor()` does not check for `address(0)` on the input parameters.
 4. `Position`'s parameters `reserveContribution` and `yearlyInterestPPM` (deciding the interest rate of the loan) should not exceed 100%, to avoid underflows in the function `getUsableMint`.
 5. `Position.constructor()` should enforce that `challengePeriod` is at least 30 minutes to ensure that a new challenge does not end before ongoing challenges in which bids postpone end time by 30 minute.
 6. `StablecoinBridge.constructor()` does not check for zero values in the inputs.
-

Acknowledged:

Frankencoin is aware of the missing sanity checks but and has decided to add checks only for point 4.



6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Findings](#) section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	1
<ul style="list-style-type: none">• Successfully Challenged Positions May Not Close	
-Severity Findings	2
<ul style="list-style-type: none">• Bidding After Challenge Ends Adds No Time• Position Limits Can Be Used for Free	
-Severity Findings	10
<ul style="list-style-type: none">• Challenges of Expired Loans Can Be Averted• Double Entry Point Protection Can Be Insufficient• Possible to DoS Minting Functionalities• Reentrant Collateral Could Mint During Liquidation• redeemFrom Does Not Reduce Allowance• Bids Could Be Reverted by Frontrunning• Bridge's Risk Exposure Limit Can Be Circumvented• Incorrect Comparison in Function <code>_mulDiv</code>• Minimum Collateral Can Be Partially Withdrawn• Parallel Challenges Are Expensive to Bid on	
-Severity Findings	11
<ul style="list-style-type: none">• Inconsistent Value Formats Emitted in Events• Challenge State Treated Inconsistently• Incorrect Documentation for Denied Positions• Incorrect Natspec Regarding Allowances in Frankencoin• Low Precision on Cubic Root Approximation• Missing Event for New Positions• Missing Implementation of Described Functions• Pool Shares Limit Not Enforced• Wrong Liquidation Price Emitted in Event• _cubicRoot Returns 0 for Large Inputs• calculateSharesInternal Can Return Large Numbers	
Informational Findings	3
<ul style="list-style-type: none">• Unused Import• Misleading Function Name <code>isPosition</code>	

6.1 Successfully Challenged Positions May Not Close

CS-ZCHF-034

When a position is below its liquidation price, one or multiple successful challenges should sell all its collateral, and the position should be closed.

However, positions do not get closed if there is still an open challenge.

```
if (balance < minimumCollateral && challengedAmount == 0) {  
    _close();  
}
```

Consider the following situation:

1. Attacker Alice creates a new position with normal parameters and deposits `minCollateral + 1` of collateral.
2. After the `initPeriod` has passed (can no longer be vetoed), Alice increases the `liquidationPrice` to a very high price. This restricts minting for 3 days.
3. Alice immediately challenges herself, with a size of `minCollateral`. It is not worth the gas for anyone else to challenge the remaining 1 wei of unchallenged collateral.
4. Alice waits until phase 2 of the first challenge starts, then starts another challenge with size of 1.
5. Someone bids the true price of the collateral in the first auction and buys all the collateral except 1 wei. `notifyChallengeSucceeded()` is called on the position, but the `challengedAmount` is still more than 0, so the position is not closed. The minting restriction is reset to 3 days.
6. Alice bids on her second challenge before it enters phase 2. This averts the challenge and reduces `challengedAmount` to 0. There is still 1 wei of collateral in the position, but it is not worth the gas for anyone to challenge it. The position is not closed, as the challenge was not successful.
7. Alice waits until the 3 day minting restriction is over.
8. Alice deposits collateral and immediately mints the `limit` amount of the position at the very high (above market) price that she had set in step 2. She passes the `noChallenge` and `noCooldown` checks. The minted tokens are undercollateralized, leading to losses for the system.

This attack can be repeated multiple times. The tokens minted in the first attack will be challenged successfully, which will reset the `minted` amount of the position to zero. Alice can again start a second challenge as soon as the other challenge is about to enter phase 2, which avoids the position from being closed. After averting the second challenge and waiting 3 days, she can once again deposit collateral and mint `limit` amount of tokens. This can be repeated infinitely.

Anyone (for example an FPS share holder with an interest in avoiding system losses) can delay the attack by challenging the position even when it has no collateral. This has a capital cost (`minCollateral` required for challenge) and a gas cost. The attacker can counter this by starting another challenge afterwards, just before the first challenge enters phase 2. This also has the same capital and gas cost for the attacker as for the "defender". The attacker has a direct incentive to pay this cost (the value of the minted tokens if the attack is successful), while the defender is being altruistic and just avoids a loss for the system without any direct reward for themselves. Each time the defender starts a challenge, they delay the attack by at least 3 days. However, they can never stop it entirely if the attacker always answers with a challenge of their own.

Ultimately, the issue is that there can be open positions with small amounts of collateral that are not worth challenging. This can lead to attacks that require altruistic users to pay costs forever to delay them, or that can lead to unlimited losses if not answered.

Code corrected:

The issue has been fixed by adding a check that closes a position if it has less than `minCollateral` when a challenge is averted (not just when a challenge is successful).

6.2 Bidding After Challenge Ends Adds No Time

CS-ZCHF-001

When Frankencoin's `bid()` is called, the challenge `endTime` will be extended such that it is at least 30 minutes in the future.

```
uint256 earliestEnd = block.timestamp + 30 minutes;
if (earliestEnd >= endTime && block.timestamp < endTime) {
    challenge.end = earliestEnd;
}
```

However, if the `endTime` has already passed, the time will not be extended.

It is intentionally possible to bid after the end of a challenge by frontrunning the execution of `end()`. However, as the time will not be extended in this case, a bidder can guarantee that they win the auction by bidding after `endTime` and then immediately calling `end()`. This will make it so that no other bidders can overbid him. If all the bidders follow this strategy, the one with the first transaction in the block after `endTime` will win the auction. This changes the auction mechanism to a gas bidding war, instead of the intended auction design, which could increase the losses of the system significantly.

Code corrected:

The function `bid` has been revised to always push the auction deadline with at least 30 minutes when a new bid is recorded. This change mitigates the frontrunning issue described above as it is not possible anymore to call functions `bid` and `end` in the same block. The code implementation is revised as follows:

```
uint256 earliestEnd = block.timestamp + 30 minutes;
if (earliestEnd >= endTime) {
    challenge.end = earliestEnd;
}
```

Note that bids that are submitted after the `endTime` will not be considered if they are executed after the function `end` was called to close the auction.

6.3 Position Limits Can Be Used for Free

CS-ZCHF-003

Opening a new position through the MintingHub requires users to pay a fee of 1000 ZCHF and wait for at least 3 days for the initialization period to pass. Users cannot clone positions during the initialization period and position owners cannot mint Frankencoin during this time.

Once the initialization period is complete, users can clone the original position which reduces its limit. When minting, they pay interest fees to the Equity. The system allows users to freely choose the expiration of the cloned position and the yearly interest rate is applied such that interest is only paid for the loan duration. Cloning a position that is expired or with a short loan duration (a few seconds), results in zero or very small fees for the interest. However, the minting limit of the original position gets consumed anyway. As new positions should wait for the initialization period (+3 days) to complete before they become usable, one could consume the limit of all positions by cloning positions for a very short amount of time (paying nearly no fees) and cause a denial-of-service, while also wasting the 1000 ZCHF fee that was paid for opening the position.

The issue is even easier to be exploited in the current codebase due to [Cloned Positions Can Have Arbitrary Expiration](#). An attacker can set the expiration of cloned position into the past or `block.timestamp` to have zero interest fees. This way, the attacker can reduce the limit of a position to zero by cloning, repaying, and withdrawing the collateral in the same transaction. Flashloans can also be used for the required collateral to consume large positions.

Code partially corrected:

The functionality for creating and cloning positions has been re-organized in `Position` such that the amount of ZCHF for which the owner has already deposited collateral is "reserved" and cannot be used by clones. This logic is enforced in function `reduceLimitForClone` which relies on the new function `limitForClones`:

```
function limitForClones() public view returns (uint256) {
    uint256 backedLimit = (_collateralBalance() * price) / ONE_DEC18;
    if (backedLimit >= limit) {
        return 0;
    } else {
        // due to invariants, this is always below (limit - minted)
        return limit - backedLimit;
    }
}
```

This approach only mitigates the problem for position owners, and only for those owners which deposit the full amount of collateral they will want to use upfront.

For other users, the issue is still present. An attacker can consume from the `limit` of positions without paying any fees, until only the owner-reserved amount is left. Hence, nobody else will be able to clone and the position cloning functionality can be rendered useless. According to the documentation it is supposed to be the main way for ordinary users to mint ZCHF:

[Cloning] is the standard way to obtain Frankencoins against a collateral. Unlike creating an entirely new position, which takes a lot of time, borrowing by cloning an established position can be done immediately.

Code corrected:

The Position contract in `Position` introduces a minimum interest fee that is applied to all new positions independently of their duration. Therefore, creating a new position that is already expired or has a short

duration (a few seconds), as described above, now incurs a cost that corresponds to the interest rate for a duration of 4 weeks.

6.4 Challenges of Expired Loans Can Be Averted

CS-ZCHF-035

Each position has an expiration which is used to compute the interest fee when minting new frankencoins. The system assumes that users will pay their loan on time as expired positions should be challenged and then liquidated with the auctioning mechanism. However, in users can postpone repayments indefinitely as long as the position's liquidation price remains below the market price. Any challenge initiated for such positions would be averted, hence challengers make a loss.

In earlier versions, it was not possible to avert challenges on expired positions.

Code corrected:

The liquidation process has been adjusted in such that the duration of the phase 1 (when a challenge can be averted) is capped by the position's expiration. Therefore, launching a challenge for an expired position immediately triggers phase 2 (dutch auction) of the liquidation. This change is implemented in the following function:

```
function challengeData(uint256 challengeStart) external view returns (uint256 liqPrice, uint64 phase1, uint64 phase2) {
    uint256 timeToExpiration = challengeStart >= expiration ? 0 : expiration - challengeStart;
    return (price, uint64(_min(timeToExpiration, challengePeriod)), challengePeriod);
}
```

Note that position owners should always withdraw their collateral from expired positions, otherwise they will be liquidated, even if the position is over-collateralized.

6.5 Double Entry Point Protection Can Be Insufficient

CS-ZCHF-036

In Position, the `withdraw` function looks as follows:

```
function withdraw(address token, address target, uint256 amount) external onlyOwner {
    if (token == address(collateral)) {
        withdrawCollateral(target, amount);
    } else {
        uint256 balance = _collateralBalance();
        IERC20(token).transfer(target, amount);
        require(balance == _collateralBalance()); // guard against double-entry-point tokens
    }
}
```

The comment states that this should protect from double-entry-point tokens. That is, a token that supports multiple addresses through which transfers can happen.

However, it could be possible that the double-entry-point token has a transfer hook that calls the target address. In this case, there could be a reentrant call to `adjust()`, which deposits an equal amount of tokens as are being withdrawn. The second `require` statement would pass, even though there was a double-entry-point token used to incorrectly withdraw collateral.

If there is a pre-transfer hook, the attacker could deposit and mint tokens using reentrancy, then withdraw collateral using the second entry-point (that is not considered to be the collateral address) after the reentrant call returns, circumventing the `_checkCollateral` check that usually happens upon withdrawing collateral. This could leave the position undercollateralized, leading to losses for the system.

Consider the following example:

1. A position has a collateral that has a pre-transfer hook to the receiver and has two entry points. 1000 collateral tokens are deposited. Each token has a value of 1 ZCHF. 1000 ZCHF are already minted.
2. The position owner calls `withdraw()`, using the address of the token's second entry point (which is not the collateral address). The `_collateralBalance` of 1000 is saved and the transfer call happens. Now, the pre-transfer hook is executed and gives execution control to the attacker.
3. The attacker calls `adjust(2000,2000,1)`. 1000 additional collateral are deposited, then `mint()` is called to mint an additional 1000 ZCHF. There are 2000 collateral tokens, so the collateral check in `mint()` passes.
4. The `adjust()` call returns, then the pre-transfer hook returns.
5. The transfer of the second entry-point of the collateral happens.
6. Now, the saved `_collateralBalance` of 1000 is compared to the current `_collateralBalance`, which is also 1000, as that additional amount was deposited. The check passes.
7. Now, there are 2000 ZCHF minted against a position with 1000 ZCHF worth of collateral.

The undercollateralized minting can only happen if there is a collateral that has a double-entry-point AND a pre-transfer hook that calls the recipient (ERC-777 tokens are not an example, they only call the recipient after a transfer, not before). This combination of traits does not seem to appear in commonly used tokens.

Specification changed:

Frankencoin has decided to revise the criteria of acceptable tokens that can be used as collateral. More specifically, tokens that implement transfer hooks (such as ERC-777) should not be used as collateral and governance should deny any position with such collateral tokens.

6.6 Possible to DoS Minting Functionalities

CS-ZCHF-037

In , the system allows any user to launch a challenge for any position and immediately avert it in the same transaction. Besides gas costs, there are no other costs for such behavior. However, the side effect of the `challenge/avert` operations is that the victim position is set in cooldown for 1 day:

```
function notifyChallengeAverted(uint256 size) external onlyHub { ...
    _restrictMinting(1 days);
}
```

An attacker can exploit this behavior to disrupt the system by putting specific positions on cooldown, therefore blocking cloning and minting functionalities in victim positions.

This was not possible in earlier versions.



Code corrected:

The function `_avertChallenge` now checks that the challenge has not been launched in the same block (based on timestamps):

```
function _avertChallenge(Challenge memory _challenge, uint32 number, uint256 liqPrice, uint256 size) internal {
    require(block.timestamp != _challenge.start);
    ...
}
```

A challenger can still challenge a position with a correct liquidation price (below market price) to put it on cooldown for one day. However, the challenger's collateral can then be sold to other bidders in future blocks, as the challenge cannot be immediately cancelled. The challenger will still be able to cancel their challenge in a later block if their cancellation transaction is included in the chain before a bid from another bidder. However, this will likely require the challenger to incur a high gas cost.

6.7 Reentrant Collateral Could Mint During Liquidation

CS-ZCHF-038

When a challenge is successful, `notifyChallengeSucceeded()` is called on `Position`.

This function does the following 3 things in order:

1. reduce `challengedAmount`
2. send collateral to bidder
3. restrict minting for 3 days

Step 2. can cause a reentrancy during an inconsistent state (minting not yet restricted) if the collateral is reentrant (e.g. ERC-777).

Consider the following attack:

1. Attacker Alice creates a new position with normal parameters and deposits an ERC-777 collateral.
2. After the `initPeriod` has passed (can no longer be vetoed), Alice increases the `liquidationPrice` to a very high price. This restricts minting for 3 days.
3. A challenge is started. Assume the duration of phase 1 + phase 2 is more than 3 days.
4. When the auction price gets close to the value of the collateral, Alice calls `bid()`.
5. `notifyChallengeSucceeded()` is called, which reduces the `challengedAmount` to 0.
6. The `Position`'s collateral is sent to Alice, which allows her to reenter. She deposits additional collateral to the position and mints `limit` tokens at the high price she set in step 2. The `noChallenge` modifier passes, as the `challengedAmount` is 0. The `noCooldown` modifier passes as 3 days have passed since the price was adjusted and `notifyChallengeSucceeded()` has not yet restricted minting. The tokens are minted at an incorrectly high price, which leads to losses for the system.

If the auction duration is less than 3 days but more than 1 day, Alice could start the challenge herself and avert it for free before it reaches phase 2. This will set the cooldown to 1 day. While Alice has an open challenge for the full collateral amount, nobody else is incentivized to start another challenge.

Specification changed:



Frankencoin has decided to revise the criteria of acceptable tokens that can be used as collateral. More specifically, tokens that implement transfer hooks (such as ERC-777) should not be used as collateral and governance should deny any position with such collateral tokens.

Futhermore, the function `notifyChallengeSucceeded` has been updated to set the cooldown to 3 days first, and later withdraw the collateral:

```
function notifyChallengeSucceeded(address _bidder, uint256 _size)
    external onlyHub returns (address, uint256, uint256, uint32) {
    ...
    _restrictMinting(3 days);

    _withdrawCollateral(_bidder, _size); // transfer collateral to the bidder and emit update
    ...
}
```

6.8 redeemFrom Does Not Reduce Allowance

CS-ZCHF-026

In `redeemFrom`, the `redeemFrom` function is added to `Equity`. It allows redeeming FPS tokens on behalf of other users if the user has given an approval.

The approved amount is checked, but it is not reduced.

```
require(_allowance(owner, msg.sender) >= shares);
```

As a result, if an approval of 1 share is given, it can be reused to redeem an unlimited number of shares.

Code corrected:

The allowance is now reduced when calling `redeemFrom()`.

6.9 Bids Could Be Reverted by Frontrunning

CS-ZCHF-004

The `MintingHub`'s `bid` function contains the following check:

```
if (expectedSize != challenge.size) revert UnexpectedSize();
```

This exists so that a bidder does not accidentally bid a wrong amount on a challenge that was split using `splitChallenge()`. However, this can also be used as a DoS vector.

Consider the following situation:

There is a position of 1 million ZCHF with `200 * minCollateral` of collateral that should be liquidated. A challenger creates a challenge for the full amount and immediately bids 1 ZCHF for the collateral. There are no bids until 30 minutes before the end of the auction, as bidders want to limit the risk of the collateral losing value after their bid is placed. Starting from 31 minutes before the end, the challenger calls `splitChallenge(challengeNumber, minCollateral + R)`, where `R` is a small random number, in each block with a relatively high amount of gas, such that they expect their transaction to happen in the block before any bids. Now, any bid will revert, as the position no longer has

the expected size. The `end` time is not increased, as there is no new bid. At the `end` time, the challenger can buy the collateral worth over 1 million ZCHF for 1 ZCHF. The system takes a massive loss.

As there are only about 150 Ethereum blocks per 30 minutes, the attacker has the ability to split their challenge once every block without hitting the `minCollateral` limitation.

In case the attacker fails to split early enough in one of the blocks and a bid comes through, they have only lost gas and incur no other cost. The bidders have little incentive to pay a lot of gas to avoid being frontrun, as they are not guaranteed to win the auction by making a single bid. They may be outbid by someone else, which would make their high gas bid worthless.

The challenger could also use more sophisticated frontrunning methods, where they only split their challenge if they really expect a bid in that block.

A bidder could work around the reverts by deploying a contract that reads the current size of the challenge on-chain and places a bid of the correct size. This cannot revert, as the size will always be up to date. It requires the bidder to have considered this situation already and have the infrastructure in place beforehand, as it is unlikely they will be able to set it up within 30 minutes.

Code corrected:

When splitting a challenge in `splitChallenge`, it is now required that the resulting bid on both resulting challenges is at least 2500 CHF, so no very small bids can be made. It is still possible to cause bids to revert by splitting a challenge, but it now requires a higher bid on the initial challenge to do so.

In `splitChallenge` the `splitChallenge` function has been removed with the change to dutch auctions.

6.10 Bridge's Risk Exposure Limit Can Be Circumvented

CS-ZCHF-005

The `StablecoinBridge` contract implements a mechanism to restrict the exposure towards an external stablecoin that might lose value or get compromised, e.g., due to a malicious implementation upgrade. The immutable variable `limit` sets an upper limit on the amount of Frankencoin that can be minted from the bridge. The limit is enforced in the internal function `mintInternal`:

```
require(chf.balanceOf(address(this)) <= limit, "limit");
```

The correctness of the protecting mechanism relies on the external contract returning a correct balance when the `balanceOf` function is called. However, if the `chf` contract becomes malicious it could return arbitrary values as `balanceOf`. For example, the contract could be upgraded to return 0 balance, or it could have an admin function that allows burning of a specific address's balance.

As a result, `limit` can be circumvented and the maximum exposure of the Frankencoin system to the external stablecoin is not bounded by `limit`. Instead, there could be an infinite number of Frankencoin minted through the bridge.

Code corrected:

Internal accounting for the number of tokens minted by the bridge has been added in `mintInternal`. This means `limit` is now correctly enforced without relying on any functions of the `chf` contract.

6.11 Incorrect Comparison in Function `_mulDiv`

CS-ZCHF-032

The internal function `_mulDiv` checks if the computation `x * factor` fits in a `uint256` before performing the multiplication. If the result is larger than `type(uint256).max`, the function performs the division first to avoid the potential overflow of the multiplication.

The implementation incorrectly does the opposite, dividing first when the result of `x * factor` fits in `uint256`, and overflowing otherwise:

```
function _mulDiv(uint256 x, uint256 factor, uint256 divisor) internal pure returns(uint256) {
    if (...) {
        ...
    } else if (type(uint256).max / factor > x) {
        ...
        return x > factor ? x / divisor * factor : factor / divisor * x;
    } else {
        return x * factor / divisor;
    }
}
```

Code corrected:

The function has been revised to perform the division first only when the result of `x * factor` does not fit in a `uint256`.

6.12 Minimum Collateral Can Be Partially Withdrawn

CS-ZCHF-007

The documentation states:

```
[...]if the minimum collateral is 1 WETH, one cannot reduce the collateral to
0.9 WETH even if there is no outstanding Frankencoins.
```

However, this is not enforced in the Position contract.

This may lead to leftover collateral amounts that are too small to be effectively liquidated, due to gas costs. This can lead to losses for the system.

Code corrected:

The public function `withdrawCollateral` has been updated to revert if a dust amount of collateral remains in a position after a withdrawal:

```
function withdrawCollateral(address target, uint256 amount) public onlyOwner noChallenge {
    ...
    uint256 balance = _withdrawCollateral(target, amount);
    ...
}
```

```
if (balance < minimumCollateral && balance > 0) revert InsufficientCollateral();  
}
```

6.13 Parallel Challenges Are Expensive to Bid on

CS-ZCHF-008

There can be many parallel challenges for the same position, each with their own `end` time. In case the sum of challenged collateral is higher than the position collateral, only the first ended challenges will be able to receive collateral, until there is none left.

Consider the following situation:

There is a position that should be liquidated, where challenges will likely not be averted. This may be due to the market price being significantly below the liquidation price, or because the position was expired. Now, a challenger starts many challenges (e.g. 100) on the same position within the same block, meaning they have the same `end` time. The challenger will need at least $100 \times \text{minCollateral}$ to do this, but they know the challenge will not be averted, so they are not risking their collateral. Assume the total challenged amount is significantly more than the position collateral. Assume there are not many bids before the final 30 minutes of the auction (this seems likely as the bidders take the risk of the collateral value falling before the end of the auction). With 29 minutes left, a bidder bids a competitive price on one of the auctions. This extends the auction time by 30 minutes. However, the `end` time of the other 99 auctions is not increased. This means that if nothing else happens, the full collateral will be sold to the auctions with no or small bids, instead of the competitively priced one. To change this, someone will need to bid a small amount (or a competitive amount, but this has capital costs) on each of the other 99 positions before they end, paying gas fees. Whenever this happens, the current highest bidder of the non-competitive positions has the option of making a bid on the competitive auction to extend the `end` once again and force someone else to bid on all the 99 other positions once again.

Overall, creating multiple challenges with more collateral in total than the position collateral can massively increase the gas costs of auctions at a relatively small cost to the attacker. The cost consists of the gas for opening the challenges plus the capital cost of locking the challenge collateral, though for a common collateral like ETH this may not be a big hurdle.

If nobody is willing to pay the gas to extend the time of the uncompetitively priced auctions, the collateral could be sold massively under its market value, leading to losses for the system.

Code corrected:

The issue has been resolved by changing the auction type to a dutch auction in `Challenge`. There is no longer a moving end time.

6.14 Inconsistent Value Formats Emitted in Events

CS-ZCHF-039

The format of parameters in events `Loss` and `Profit` is not consistent. Specifically, the variables included in the event `Loss` represent raw frankencoin amounts, while the variables in `Profit` are in `E6` format (multiplied by $1e6$).

Code corrected:

The codebase has been updated to emit events with parameters in frankencoin amounts. Events now include only the reporting minter and the realized loss or profit Here is one example from function `burnWithoutReserve`:

```
emit Profit(msg.sender, reserveReduction / 1000_000);
```

6.15 Challenge State Treated Inconsistently

CS-ZCHF-027

Function `isChallengeOpen` returns `true` if `challenge.end` is in the future, otherwise it returns `false`. However, this is not in line with the `bid` function, which accepts new bids until a challenge has been either averted or settled by the function `end`.

Code corrected:

The function was removed in

6.16 Incorrect Documentation for Denied Positions

CS-ZCHF-010

The documentation states:

```
In case a new position is both challenged and vetoed, the challenge cannot be  
averted any more and the collateral is simply auctioned off to the highest bidder.
```

However, this is not correct. Challenges can be averted in this situation. They can only not be averted when the position is expired.

Specification changed:

The documentation has been updated to reflect the code.

6.17 Incorrect Natspec Regarding Allowances in Frankencoin

CS-ZCHF-011

The natspec comment for function `openPosition` states:



* For a successful call, you must set allowances for both ZCHF and the collateral token, allowing the minting hub to transfer the initial collateral amount to the newly created position and to withdraw the fees.

This sentence is partially incorrect as the user does not need to provide allowance to the minting hub for ZCHF (Frankencoin). MintingHub has the `minter` role in Frankencoin and has infinite approval transfer on behalf of any account.

The same incorrect information is also present in the documentation.

Specification changed:

The inline code comments have been revised. They no longer state that the user must provide allowance for ZCHF to the minting hub.

6.18 Low Precision on Cubic Root Approximation

CS-ZCHF-031

The function `_cubicRoot` implements the Halley approximation to compute the value $x^{1/3}$. The returned value has a maximum error of 0.01 shares. Given that this function is used to compute the number of pool shares an LP receives after investing, the rounding error might be significant if the pool shares have a high value (e.g., thousands of ZCHF).

Code corrected:

Function `_cubicRoot` has been updated to use a more efficient starting value for the approximation instead of 10^{18} that was used in the previous iteration:

```
// Good first guess for _v slightly above 1.0, which is often the case in the Frankencoin system
uint256 x = _v > ONE_DEC18 ? (_v - ONE_DEC18) / 3 + ONE_DEC18 : ONE_DEC18;
```

This enables the approximation algorithm to converge faster than before, which allows using a higher precision threshold, namely 10^{-12} .

6.19 Missing Event for New Positions

CS-ZCHF-013

The contract `PositionFactory` does not emit any event when a new position is opened or cloned. No event is emitted by `MintingHub` either. Deploying a new position is an important update for the system. An event helps challengers and liquidators to easily learn about existing positions.

In general, it is recommended to emit events for important state updates and index the relevant parameters in events to allow integrators and dApps to quickly search for these and simplify UIs.

Code corrected:

The event `PositionOpened` is now emitted by the minting hub whenever a new position is created or cloned.



6.20 Missing Implementation of Described Functions

CS-ZCHF-014

The comment in the contract ERC20 states that functions `decreaseAllowance` and `increaseAllowance` are implemented:

```
* Finally, the non-standard `decreaseAllowance` and `increaseAllowance`  
* functions have been added to mitigate the well-known issues around setting  
* allowances. See `IERC20.approve`.
```

However, none of the above functions have been implemented by the contract. The frontrunning attack against `approve()` is therefore still applicable as it always overwrites the current value without checking if the allowance has been consumed or not.

Specification changed:

The comment has been removed.

6.21 Pool Shares Limit Not Enforced

CS-ZCHF-030

Inline comments in the Equity contract suggest that a limit in the total pool shares is enforced in the code:

```
In fact, a limit of about 2**30 shares (that's 2**90 Bits when taking into account the  
decimals) is imposed when minting.
```

However, the contract does not implement such a restriction.

Code corrected:

The updated function `invest` now implements the following check that restricts the total number of pool shares minted:

```
require(totalSupply() <= type(uint96).max, "total supply exceeded");
```

6.22 Wrong Liquidation Price Emitted in Event

CS-ZCHF-016

Function `initializeClone` emits the variable `existing.price` in the event `PositionOpened` which matches the price of the original position. The liquidation price of the cloned position is stored in the state variable `pos.price` and may be smaller than `existing.price`.



Code corrected:

The event is now emitted by the minting hub using the correct price.

6.23 `_cubicRoot` Returns 0 for Large Inputs

CS-ZCHF-028

The function `_cubicRoot` computes incorrect results for inputs larger than `1e28` due to `_mulDiv()` rounding down as described in [Possible Rounding to 0 in Function `_mulDiv`](#). Therefore, for input values `_v` larger than `1e28`, the function always computes 0 in the following line:

```
uint256 xnew = _mulDiv(x, (powX3 + 2 * _v), (2 * powX3 + _v));
```

The consequences of this issue in the current codebase are limited given that `_cubicRoot()` is called with values slightly larger than `1` (`10**18`):

```
_cubicRoot(_divD18(capitalBefore + investmentExFees, capitalBefore))
```

In the codebase , a check is added on the input value `_v`, however it only impacts the initial guess of the approximation algorithm. For large inputs `_v`, the function `_cubicRoot` still returns incorrect results.

Code corrected:

The function `mulDiv` is removed from the codebase to avoid rounding down to 0, hence the function now fails for large inputs due to arithmetic overflows. This behavior is preferred by Frankencoin as the codebase calls `_cubicRoot()` only with limited values (slightly larger than `10**18`).

6.24 `calculateSharesInternal` Can Return Large Numbers

CS-ZCHF-017

In Equity, `calculateSharesInternal()` is mostly expected to be called with a `capitalBefore` of more than `1000E18`. However, there are edge cases where it could be called with small numbers, which lead to large return values.

Consider the following unlikely situation:

The Frankencoin equity has been almost wiped out, but not completely. The equity remaining is 1 wei of ZCHF. There are more than 1000 FPS shares still in circulation. Now, someone calls `invest()` with amount `1000E18` ZCHF. This will call `calculateSharesInternal(1, 1000E18)`.

The amount of shares to mint will be calculated using the cubic root of `997E18 / 1E(-18)`, which is a much larger number than expected.



Code corrected:

The function `calculateSharesInternal` has been renamed as `_calculateShares` and now it implements a short-circuit that returns 1000 shares whenever equity is below the minimum threshold or no pool shares are minted. This avoids small inputs:

```
function _calculateShares(uint256 capitalBefore, uint256 investment) internal view returns (uint256) {
    uint256 totalShares = totalSupply();
    ...
    uint256 newTotalShares = capitalBefore < MINIMUM_EQUITY || totalShares == 0 ? totalShares + 1000 * ONE_DEC18
    : ...;
    return newTotalShares - totalShares;
}
```

6.25 Incorrect Decimals Comments

CS-ZCHF-041

MintingHub's `launchChallenge()` and Position's `notifyChallengeSucceeded()` both have the following comment:

```
//@param _size      size of the collateral bid for (dec 18)
```

However, the expected input size should be given as a token amount (in token decimals), not with 18 decimals.

Code corrected:

The inline comments that described incorrectly the input values are removed in .

6.26 Misleading Function Name `isPosition`

CS-ZCHF-022

The function `isPosition` in Frankencoin contract returns the address of the minter if a position has been registered, and zero address otherwise. However, the name `isPosition()` hints that the function also checks if the position exists, which can be misleading. Additionally, the name could be read as implying that it returns a boolean (true if it is a position with minter role) instead of an address.

Code corrected:

The function has been renamed to `getPositionParent`.

6.27 Unused Import



The file `MintingHub.sol` imports the contract `Ownable`, which is not used.

Code changed:

The unused import has been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Compiler Version

CS-ZCHF-029

The compiler version used (0.8.20) has the following [known bugs](#).

This is just a note as we do not see any issue applicable to the current code.

The contracts should be deployed using a compiler version they have been thoroughly tested with. Using a very recent version may not be recommended, as it may not be considered battle-tested yet.

At the time of writing the most recent version is 0.8.21. For more information please refer to the [release notes](#).

7.2 Event Reentrancy

CS-ZCHF-018

Function `end` triggers a transfer in the collateral token before emitting the event. If the collateral token implements callbacks on transfer, one can reenter in the contract and therefore have events emitted out of order.

Acknowledged:

Frankencoin responded:

External observers are not expected to rely on the order of events within a transaction.

7.3 Gas Optimizations

CS-ZCHF-019

The codebase could be more efficient in terms of gas usage. Reducing the gas costs may improve user experience. Below is an incomplete list of potential gas inefficiencies:

1. The function `adjustRecipientVoteAnchor` does not clear the state recorded in the mapping `voteAnchor` for users that burn or transfer out all their shares.
2. The function `invest` could be marked as `external`.
3. The function `calculateShares` could be marked as `external`.
4. The function `getUsableMint` could be marked as `external`.

5. The field `end` in struct `Challenge` could use a smaller type such that it fits into a single storage slot with another field of type `address`.
 6. The modifier `onlyOwner` in function `adjust` is redundant when triggering calls to `withdrawCollateral()`, `mint()` and `adjustPrice()`.
 - :
 7. One external call to `*.collateral()` could be avoided in function `clonePosition`.
 8. The function `adjustPrice` performs two external calls to read the balance of the position via `_collateralBalance()` when price is lowered.
 - :
 9. The function `StablecoinBridge._mint` performs redundant SLOADs when accessing the state variable `minted`.
-

Code partially corrected:

2. The function `invest` has been marked as `external`.
3. The function `calculateShares` has been marked as `external`.
4. The function `getUsableMint` has been marked as `external`.
5. The field `end` has been removed, a new field `start` is added which is of type `uint64`.

7.4 Incomplete Natspec

CS-ZCHF-020

A large number of functions have incomplete Natspec, i.e., do not describe all input parameters and return values. Natspec descriptions help to more quickly understand the intention of functions, which improves code readability. Natspec of external functions also helps third-parties that integrate with the system, e.g., by providing information regarding the format of input values, or assumptions that are made about them.

Acknowledged:

Frankencoin added natspec comments for some functions.

7.5 Magic Numbers in Codebase

CS-ZCHF-021

Several magic values are used in the codebase that could be declared as constant. For instance, parameters of positions are stored in 6 decimals (parts-per-million), hence the number `1000_000` is used frequently. Similarly, voting thresholds are stored in basis points. Such values can be replaced with constant variables to improve code readability.

Acknowledged:



Frankencoin has decided to keep the code unchanged as they prefer to avoid adding new variables for constants.

7.6 Rounding Errors in Kamikaze Function

CS-ZCHF-023

The `kamikaze` function burns voting power from a `target` address and the caller of the function. The function is designed to burn the same voting power from both accounts as set in `votesToDestroy`. However, the internal function `reduceVotes` introduces errors due to rounding:

```
function reduceVotes(address target, uint256 amount) internal returns (uint256) {  
    ...  
    voteAnchor[target] = uint64(anchorTime() -  
        (votesBefore - amount) / balanceOf(target));  
    ...  
}
```

In the line above, the rounding error depends on `amount` (corresponding to the user input `votesToDestroy`) and the token balance of the affected address. Hence, the caller can choose `votesToDestroy` in such a way that its voting power is reduced slightly less compared to the other party.

In the worst case, the caller is able to destroy up to 1 second worth of votes more from a target than from themselves.

Acknowledged:

Frankencoin has acknowledged the issue but has decided to keep the function as-is due to the limited impact of the issue.

7.7 Transfer of Positions' Ownership

CS-ZCHF-024

Positions inherit the `Ownable` contract which records the address of current owner. `Ownable` implements a single-step transfer of ownership with a sanity check for zero address. Accidental ownership transfers are possible and would lock positions indefinitely, rendering functionalities to mint and withdraw collateral useless.

A mechanism such as `Ownable2Step` could be used to mitigate accidental transfers to an incorrect address.

Acknowledged:

Frankencoin responded:

```
Worried users can create their own owner contract with arbitrary additional safety measures.
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Dutch Auction Duration

The price of the dutch auction decreases linearly throughout phase 2.

Note that only some prices will be available due to the Ethereum block time being 12 seconds. As a result, the duration of phase 2 should be chosen long enough such that relevant prices do not fall inbetween blocks.

For example, if the auction should have a price precision of 0.1%, the duration of phase 2 must be at least $1000 * 12$ seconds.

At the same time, the whole duration (phase 1 and phase 2) should not be too long, as a longer duration gives more time for the collateral to continue falling in value once the auction starts.

Positions with bad auction duration values should be denied by governance.

8.2 Interest Fees Are Computed on the Gross Minted Amount

Users pay an interest fee in Frankencoin (ZCHF) when minting from the positions created by the MintingHub. The interest is computed on the gross amount of minted ZCHF, which includes the reserves and the interest fee, not just the amount that the user receives in their wallet.

The relevant code is implemented in the function `mint` of the Frankencoin contract:

```
function mint(address _target, uint256 _amount, uint32 _reservePPM,
    uint32 _feesPPM) override external minterOnly {

    uint256 usableMint = (_amount * (1000_000 - _feesPPM - _reservePPM)) / 1000_000;
    _mint(_target, usableMint);
    _mint(address(reserve), _amount - usableMint); // rest goes to equity
    as reserves or as fees
    ...
}
```

In a scenario where the reserve contribution is set at 20%, the interest fee is 5%, and the expiration is in one year, the effective interest rate on the `usableMint` is $0.05 / 0.75 = 6.6\%$ per year, not 5%.

Users should be aware of the effective interest rate that they are paying, which may be unintuitive due to the way it is calculated.

8.3 Market Risk Taken by Challengers and Bidders

The auction process takes some time depending on the `challengePeriod` of the position and the number of bids received. During this time, the price of the collateral could change.

Market risk for challengers:

For instance, if at time t_1 the market price of the collateral is below the liquidation price of a position, one can start a challenge with the assumption that the challenge will succeed, i.e., the highest bid will be close to the market price, which is below the liquidation price. However, if the price goes above the liquidation price at time t_2 while the auction is ongoing, the challenger's collateral will be sold at the liquidation price, which will likely result in a small loss for them, as a bidder would only bid if they can buy the asset below market price.

As a result, it should be expected that challengers only start challenges when they think it is likely that it will not be averted. They will not challenge a position if market price is only a small amount below the liquidation price.

Market risk for bidders:

Similarly, bidders have exposure to the price change of the collateral when they place their bid. The exposure will always be at least 30 minutes. In case the price of the collateral drops during this time window, they may buy the collateral at a higher price than the current market price.

As a result, it should be expected that bids will not be at market price, but instead be at a discount which compensates for the risk taken (or the cost of hedging it elsewhere).

The `reservePPM` of positions should be set high enough to cover the loss occurred due to sub-market-price liquidations caused by the above factors.

In `of the codebase`, the liquidation process is based in dutch auctions, which removes the exposure of bidders to the price change. The auction is now resolved as soon as a bid is placed (the challenge is either averted or successful), and the bidder receives the collateral immediately.

8.4 Minimum Collateral Is Never Adjusted

The `minimumCollateral` for a position is immutable. On creation, it is enforced that the minimum is `>5000 ZCHF`. The price of the position can be adjusted up or down, but the `minimumCollateral` never changes. If the price is decreased, this could lead to positions with low value, which may not be worth the gas to liquidate using the auction mechanism.

Consider the following situation:

1. Position is created when collateral is worth a lot.
2. Price falls slowly but owner keeps adjusting liquidation price down. ZCHF limit is still large.
3. Someone clones the position many times using `minCollateralAmount` (which is significantly under 5000 ZCHF). Each position is not worth liquidating because of gas.

This could result in up to the position's minting `limit` amount of losses for the system.

Governance should keep this limitation in mind when evaluating the collateral asset, `expiration` and `limit` of new positions. An asset that may lose significant value before the `expiration` should likely not be allowed with a large `limit`.

8.5 Minters Should Not Change Their Code

Contracts with the minter role should not be able to change their code.

In particular, they should not be upgradeable and they should also not be able to `selfdestruct`. Using `create2`, it would be possible to propose a benign minter that is able to call `selfdestruct`. As soon as the veto period has ended, the deployer could `selfdestruct` the minter and deploy a different contract at the same address, which would now be able to mint unlimited Frankencoin.

Proposed minters and their related positions should be carefully evaluated during the veto process not to have any of the code-changing functionality such as the above or others.

8.6 Nested Iterations in Function `checkQualified`

The function `checkQualified` iterates through all addresses included in the array `helpers`. For each address in the array, the function `canVoteFor` is called recursively to verify that a valid delegation path exists between the FPS holder and `msg.sender`. The governance address exercising their veto right should carefully choose addresses in `helpers` to optimize their gas costs and make sure the transaction does not exceed gas limitations.

8.7 No Slippage Protection for Mint `onTokenTransfer`

There are two functions that can be used to mint tokens in Equity: `invest()` and `onTokenTransfer()`. The `invest()` function contains an `expectedShares` argument, which provides slippage protection. The `onTokenTransfer()` function does not have an equivalent.

Users should use the `invest()` function to mint shares when possible, in order to benefit from the slippage protection.

Code changed:

In `Equity`, the `onTokenTransfer` function was removed from `Equity`.

8.8 Non-registered Positions

The contract `PositionFactory` does not enforce any access control for its external functions `createNewPosition` and `clonePosition`, hence anyone can deploy arbitrary positions. The UI

should filter out such positions and users should validate that the positions they interact with are created by the correct MintingHub. Only contracts with the `minter` role in Frankencoin can register positions.

8.9 Position Owners Should Withdraw the Excess Collateral

Users should withdraw their excess collateral from positions, especially if they are expired or do not have any `minted` amount left. Otherwise, the position risks having its collateral liquidated if the market price falls below the liquidation price, even if the position itself might be over-collateralized (e.g., zero minted frankencoins).

8.10 Positions With Quickly Collapsing Collateral Must Be Challenged

In MintingHub, the Challenge reward is calculated as follows, where `offer` is the value that the collateral has been liquidated for:

```
uint256 reward = (offer * CHALLENGER_REWARD) / 1000_000;
```

This becomes problematic in a case where the expected liquidation value is close to zero.

Consider the following situation:

1. A token is accepted as collateral.
2. Positions are created with a price that is valid at the time.
3. Something catastrophic happens, and the value of the collateral declines very quickly towards 0 (e.g. hyperinflation or loss of backing for a wrapped asset).
4. Now, the expectation is that by the time `phase1` and `phase2` of the dutch auction finish, the collateral value will be 0. This means the challenger reward will also be 0.
5. If all challengers share this expectation, nobody is incentivized to start a challenge, as the gas cost of doing so will be larger than the reward.
6. The position goes unchallenged, even when the market price goes below the liquidation price. The price is unchanged, so the position is not on cooldown. A user can clone the position and deposit collateral token, which he bought for below liquidation price on the market. He can then profitably mint up to `limit` ZCHF, which are undercollateralized and will lead to a loss to the system.

This situation can be avoided if there is an attentive FPS holder, who is incentivized to avoid losses to the system. He should immediately challenge the position when it goes below liquidation price, even though they expect not to receive any challenge reward.

If the collateral value is 0, there is also no incentive for bidders to bid on a challenge once it reaches the ending price of 0. Someone also needs to call `bid()` altruistically (paying gas) to update the accounting and have the Equity take the loss.

In conclusion, FPS holders should monitor positions and challenge them even if they do not expect a challenge reward, in order to avoid losses to the system (which are absorbed by FPS Equity).

8.11 Possible Rounding to 0 in Function `_mulDiv`

Function `_mulDiv` in the library `MathUtil` stores the intermediate results in `uint256`, hence it performs the division before multiplication if the term `x * factor` does not fit in 256 bits to avoid overflowing. Note, the result is rounded to 0 for inputs where `divisor` is larger than both `x` and `factor`, e.g., `_mulDiv(2**250, 2**15, 2**251)`.

The function `_mulDiv` has been removed in [this commit](#) of the codebase to avoid rounding down as in the example described above.

8.12 Possible to Frontrun Veto Transactions

FPS holders can delegate their voting power to 3rd parties which can use the delegated votes to veto new proposals (e.g., opening new positions or adding minters into Frankencoin). A user vetoing a proposal submits a list of addresses that have delegated voting power to him (referred to as `helpers`).

The function `checkQualified` ensures that the total voting power (own voting power plus delegated votes) of the address exercising the veto right is above 3% and all delegations are still valid. The second condition enables a frontrunning possibility for a delegator in the list `helpers` to make the veto transaction revert by removing the delegation. Therefore, governance should exercise their veto rights as early as possible and include only trustworthy accounts as `helpers`.

8.13 Specifics of Modifier `minterOnly` in Frankencoin

The modifier `minterOnly` passes successfully if either `msg.sender` is a valid minter or it has been added in the mapping `positions` by a valid minter:

```
modifier minterOnly() {
    if (!isMinter(msg.sender) && !isMinter(positions[msg.sender])) revert NotMinter();
    _;
}
```

This behavior, among others, should be considered when evaluating proposals for adding new minters in the Frankencoin contract.

8.14 `restructureCapTable` May Take Multiple Blocks

If the system has incurred losses and equity is less than 1000 ZCHF, the function `restructureCapTable` allows existing shareholders to restructure the system. Any shareholder with more than 3% of the voting power can step in and bootstrap the system by paying for the losses (which

can be significantly higher than 1000 ZCHF) and become the only FPS shareholder. As the function `restructureCapTable` iterates through all FPS holders given as input and burns their shares, it is possible that the block gas limit prevents wiping all existing shares in a single block.