Code Assessment

of the Frankencoin v2024

Smart Contracts

November 28, 2024

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Frankencoin Team,

Thank you for trusting us to help Frankencoin with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Frankencoin v2024 according to Scope to support you in forming an opinion on their security risks.

Frankencoin implements extensions to the already deployed Frankencoin stablecoin system. The extensions include a MintingHub with variable interest rates, a PositionRoller that enables flashloans, and a Savings module. The contracts must be accepted as Minters by Frankencoin Governance to become usable.

The most critical subjects covered in our audit are functional correctness and accounting correctness.

Functional correctness has been improved, as the new liquidation mechanism could interfere with the existing one, see buyExpiredCollateral Can Disincentivize Challenging. Additionally, the minimum collateral requirement for positions was not enforced, see Minimum Collateral Can Be Partially Withdrawn. Accounting correctness was improvable, as bad debt was not accounted correctly, see forceSale Does Not Account for Bad Debt.

The general subjects covered are specification and trust model. Specification is improvable, as the only specification provided was in the form of code comments. Security regarding the trust model is high, as the system still relies on the same trust model as the original Frankencoin contracts, with no additional trusted roles.

In summary, we find that the codebase provides a high 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.

Critical -Severity Findings	0
High -Severity Findings	2
• Code Corrected	2
Medium -Severity Findings	4
• Code Corrected	4
Low -Severity Findings	4
• Code Corrected	1
• Code Partially Corrected	1
• Risk Accepted	1
• Acknowledged	1

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 v2024 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	22 Oct 2024	0d25d77110c3d50d6de3f7c4aec05e4abd64ddef	Initial Version
2	27 Nov 2024	fa1457b9c2b370fbcecc5442ae4e0c51b62e454f	Version 2

For the solidity smart contracts, the compiler version `^0.8.0` was chosen.

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

- Leadrate.sol
- Savings.sol
- PositionRoller.sol
- Frankencoin.sol
- MintingHub.sol
- Position.sol
- PositionFactory.sol
- utils/FPSWrapper.sol

2.1.1 Excluded from scope

Any file not listed explicitly above is excluded from the scope. Furthermore, external token contracts used as collateral in the system were not in the scope of this code assessment. Moreover, third party libraries are assumed to behave correctly according to their specification.

2.2 System Overview

This system overview describes the initially received version (⟮**Version 1**⟯) of the contracts as defined in the Assessment Overview.

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

Frankencoin v2024 is an updated version of the existing Frankencoin system, extending its functionality with the Leadrate, Savings, PositionRoller and FPSWrapper contracts. To integrate the new modules, the MintingHub and Position contracts have also been updated.

This system overview focuses on the parts that have changed in the new version. For an overview of the full functionality, refer to the first Frankencoin Audit report.

2.2.1 Leadrate

The Leadrate contract provides the functionality to set and update the system's interest rate. Upon construction, an initial interest rate is set. Any party with at least 2% voting power in the system may invoke the `proposeChange` function to propose a new interest rate. This sets `nextRatePPM` in the contract's state to the proposed rate. Since any user with sufficient voting power can make a proposal, the value of `nextRatePPM` may be overwritten at any time by a subsequent proposal. To cancel a proposal, it can be overwritten with a proposal that uses the current rate.

A proposed rate is applied to the system only if it is different from the current interest rate (`currentRatePPM`) and has remained unchanged for at least seven days (no overwrites). It is presumed that only proposals with prior off-chain consensus within governance will ultimately be applied to the system. Applying an interest rate of zero is allowed.

For interest calculations, the system uses interest `ticks` as a way to track accumulated interest over time. The `currentTicks` function in the Leadrate contract returns the cumulative system ticks, which are initialized to zero and increase monotonically over time based on the current interest rate.

The Leadrate is used for two purposes: It is the rate savers earn in the Savings module, and it is the minimum interest rate that Positions must pay to mint Frankencoin. Note that interest will be paid on Frankencoins that are in the reserve, but those coins can never participate in the savings module. As a result, there should always be more interest earned by the system than paid out (unless the lead rate is increased).

2.2.2 Savings

The Savings contract inherits from the Leadrate contract and allows users to deposit Frankencoins and earn interest on their deposits. The contract maintains a mapping of savings `Accounts` in storage, keeping track of the `saved` Frankencoins and the `ticks` counter per user address.

The contract allows users to deposit Frankencoins into their own savings account or on behalf of other account addresses using the `save` function. The initial deposit to a user's savings account is locked for three days. During the lockup period, the deposited funds do not accrue interest and can not be withdrawn. Adding additional ZCHF to a savings account that already holds Frankencoins will apply a shorter lockup period, calculated as a weighted average based on the previously stored and newly added funds. Note that if the system's interest rate changes while an account is in a locked state, the lockup period will adjust inversely to the rate change.

Interest does not compound automatically; it must be explicitly realized by refreshing the savings account balance. The `refresh` function collects the accrued interest according to the account's ticks, adds it to the savings balance and updates the ticks to match the current system ticks.

Users can `withdraw` a specified amount of funds from their savings account at any time, provided the account is not in a locked state. If the user passes an `amount` that is greater than their full balance, the full balance will be withdrawn instead of `amount`. Before the withdrawal is transferred to the owner, the savings balance is updated to include any outstanding earned interest (`refresh`).

Note that the interest earned on the deposited Frankencoins is drawn from the system's `equity`. If the Frankencoin system holds insufficient equity, accrued interest will not be paid out to savers. This would be an extreme scenario, which could happen if the lead rate is quickly increased by a large amount and/or there is a lot of bad debt created through liquidations and/or there are a lot of FPS redemptions. In such a case, the equity outflows from interest payments to savers could exceed the inflows from interest payments made by lenders.

2.2.3 PositionRoller

The PositionRoller contract enables a position owner to close a nearly-expired position and "reopen" it without needing to come up with additional Frankencoin liquidity just to close the position.

This is done by allowing users to take a Frankencoin flash loan to roll over debt from an existing position, the `source`, to a different position, the `target`. The caller of the `roll` function must be the owner of the `source` position. In case the target position is not owned by the caller or the user wants a position with a shorter expiry, a clone of the `target` will be deployed.

To provide the flash loan, the `PositionRoller` must be registered as a minter in the Frankencoin contract. Called with pre-computed parameters that allow to fully roll over the debt from the source position, the `roll` function performs the following steps (assuming no cloning of `target` is required):

1. Take a flashloan by minting `repay` amount of ZCHF

2. Use the flashloan to repay the outstanding debt in the source position

3. Withdraw the corresponding amount of collateral from the source position, transferring it back to the owner

4. Transfer the specified amount of collateral to the target position

5. Mint the according amount of ZCHF from the target position to the caller

6. Repay the flash loan by burning `repay` amount of ZCHF from the caller

If the new mint is less than the repaid amount, (which will be the case due to fees, unless additional collateral is added) the additional ZCHF will be burnt from the caller's address. Should the caller not hold a sufficient amount of Frankencoin at the end of the roll function, repayment of the flash loan will fail, causing execution to revert.

The `rollFully` and `rollFullyWithExpiration` functions can be used in order to compute the parameters to fully roll a position, on-chain. However, as these on-chain computations may be costly due to the `binarySearch` involved, it is expected that users will call `roll` directly with pre-computed values. If the `target` position of `rollFully()` does not have enough limit to mint the required amount of tokens, the call will revert.

2.2.4 FPSWrapper

The FPSWrapper contract allows users to wrap their Frankencoin Pool Share (FPS) tokens to maintain accrued voting power during transfers, for example when bridging across chains. The `wrap` and `depositFor` functions enable users to deposit FPS tokens and mint an equivalent amount of wrapped tokens (WFPS). The `unwrap` and `withdrawTo` functions burn the wrapped tokens and return the underlying FPS tokens to the user, preserving the time-held status. Additionally, the `unwrapAndSell` function bypasses the standard 90-day holding requirement and allows to redeem the underlying FPS tokens immediately, given that the wrapper's average holding time is greater than 90 days. The `halveHoldingDuration` function mitigates the risk of the wrapper contract accumulating too many votes for governance to function by halving the held tokens' holding duration. It can also be used to disallow bypassing the redemption period.

2.2.5 Updates in MintingHub contract

To support the new modules, the MintingHub now stores the addresses of the PositionRoller and Leadrate contracts in storage.

The updated MintingHub implements a `buyExpiredCollateral` function. This function enables users to purchase a specified amount of collateral from an expired position at the applicable `expiredPurchasePrice`. It provides a new mechanism for liquidating expired positions, which previously required initiating a challenge and providing the full collateral amount as challenge collateral. Additionally, it allows the position owner to sell the collateral at a price higher than the liquidation price, if its true market value exceeds the liquidation price. The `expiredPurchasePrice` starts at 10 times liquidation price and decreases linearly over time.

For the challenge mechanism, both bidding phases are now enforced to be the same length. Previously, the first phase could be shorter than the second phase if the position was close to expiry. As a result, it

was impossible for a challenge on an expired position to be averted. Now, challenges on expired positions can be averted normally.

The `reservePPM` variable when opening a new position is now enforced to be at least `2%`.

2.2.6 Updates in Position contract

The total amount of Frankencoins minted by a parent position and all its clones is now tracked in the `totalMinted` storage variable. It is only updated in the parent position. Any call to `mint()` or `repay()` in a clone triggers `notifyMint` and `notifyRepay` respectively, updating the `totalMinted` variable in the parent position accordingly.

The Position contract introduces a new `ownerOrRoller` modifier for access control in the `mint()` and `withdrawCollateral()` functions. Previously, these functions were restricted to the owner. The updated access modifier is needed to enable the `PositionRoller` to roll over debt from one position to another.

The annual interest model has been updated. Previously, the annual interest rate was a fixed parameter set during position construction, remaining constant throughout the position's lifetime. In the updated system, the annual interest rate consists of the system's current interest rate (through the Leadrate module) plus a risk premium. The `riskPremiumPPM` can be chosen by the owner and is passed to the constructor. Riskier collateral types may require a higher premium to be accepted by governance. As the current interest rate may change and adjust over time, positions can now be opened for longer as the interest paid for minting new ZCHF is adjustable. Interest is still paid upfront for the entire duration, using the leadrate at the time of minting. If the leadrate is changed, it only applies to Frankencoins minted from then on. Previously minted debt is unaffected.

Previously, minting Frankencoins required an upfront interest payment covering a minimum period of four weeks, even if the position expired sooner. Now, there is no minimum interest payment requirement; upfront interest is calculated and paid on a per-second basis, matching the actual time remaining until expiry.

A new `forceSale()` function has been added to the Position contract. It can only be called by the MintingHub on expired positions. It allows to fully or partially buy the collateral from an expired position at a price that ranges from 10 times the liquidation price to 0. It is called via `MintingHub.buyExpiredCollateral`.

2.2.7 Roles and Trust Model

For the updated version of the Frankencoin system, the same trust assumptions as for the original system are made. For the detailed trust model please refer to the "Roles and Trust Model" section of our previous code assessment of the Frankencoin system.

Governance should ensure that the new contracts are deployed and configured correctly before assigning the minter role to them.

The interest rate of the system is assumed to be set correctly by governance. Proposals for changes in the interest rate must be observed by governance and rate changes that are too large should be vetoed to ensure sanity of the system.

The same assumptions on collateral tokens as in the original version apply: They are assumed to be compliant with the ERC20 standard, implement the `decimals` function, use less than 24 decimals and be non-malicious. Only 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).

2.2.8 Changes in Version 2

In ⸨Version 2⸩, it is no longer possible to challenge expired positions, as the `alive` modifier has been added to `notfiyChallengeStarted()`. Moreover, calling `buyExpiredCollateral()` is only allowed for positions that are not in a challenged state.

Losses for the system are now accounted for via `coverLoss()` in the following two scenarios: Firstly, when earned interest in the Savings module is collected. Secondly, in case the proceeds of buying the full amount of expired collateral via `forceSale()` do not suffice to account for the position debt, which results in bad debt that must be covered by the equity or the reserves.

The `PositionRoller.roll()` function now validates the addresses of the source and target positions by checking if they had previously been registered in the Frankencoin `positions[]` mapping.

The Position contract now stores a `closed` flag to indicate whether a position is closed, meaning that it has at some point held less collateral than the minimum collateral amount. The newly added `backed` modifier checks whether a position is closed and is used on the `assertCloneable()`, `_adjustPrice()` and `_mint()` functions of a Position, disallowing these functions if the position has ever had less than `minimumCollateral`. Moreover, a new check in `_checkCollateral()` ensures that it is not possible to reduce the collateral of a position below `minimumCollateral` by withdrawing, unless the position has zero debt.

More general changes in ⸨Version 2⸩ include :

- Public getter functions in the Savings module to compute the `accruedInterest()`.
- A helper function `Savings.adjust()` to set the balance of a savings account to the `targetAmount`.
- Completion and correction of code documentation.

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	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	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:

- **Design**: Architectural shortcomings and design inefficiencies
- **Correctness**: Mismatches between specification and implementation

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

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	0

Low-Severity Findings	3

- Frontrunning Denial of Service **Code Partially Corrected**
- burnWithoutReserve Event Can Overestimate Profit **Acknowledged**
- buyExpiredCollateral Can Leave Dust **Risk Accepted**

5.1 Frontrunning Denial of Service

Design **Low** **Version 1** **Code Partially Corrected**

CS-ZCHF2-007

The following functions could be permissionlessly frontrun to cause reverts:

- `Savings.withdraw()` could be frontrun with `Savings.save()`, donating a very small amount to the user, causing a withdrawal cooldown of 1 second. This will cause the withdrawal to revert.

- `Position.repay()` of the full minted amount, which is also used in `PositionRoller.roll()`, could be frontrun with `Position.repay()`, paying back 1 wei of ZCHF for the user. This will cause a revert, as the user will try to repay more than the minted amount.

- `Mintinghub.buyExpiredCollateral()` of the full collateral amount could be frontrun with `buyExpiredCollateral()` of 1 wei. As it will not be possible to transfer the full collateral when some was already bought, the transaction will revert.

- `Position.withdrawCollateral()` of the full collateral amount could be frontrun with transferring 1 wei of collateral to the position. This will cause a revert, as the withdraw would leave the position with less than `minCollateral` (this assumes that the check for `minCollateral` which is missing in **Version 1** is readded).

Generally, users can avoid frontrunning by using a private mempool like Flashbots Protect. An attacker could still try to speculatively frontrun a transaction, but this is unlikely to be profitable.

The reverts would only cause a lasting problem if the user utilizes a timelocked smart contract to execute their transactions. If this contract does not support retrying failed executions, the user may be repeatedly frontrun without being able to react to it.

Code partially corrected:

`buyExpiredCollateral` now takes an `upToAmount` argument, instead of `amount`. If there is less than the requested amount of collateral available, the function will buy as much as possible. This prevents the frontrunning attack described above.

`withdrawCollateral()` now allows withdrawals that leave less than the `minCollateral` in the position, as long as there is no outstanding debt.

The Frankencoin team has chosen not to change the `withdraw` and `repay` functions.

5.2 burnWithoutReserve Event Can Overestimate Profit

Correctness **Low** **Version 1** **Acknowledged**

CS-ZCHF2-009

In `burnWithoutReserve()`, the Frankencoin contract burns the full minted amount of tokens from the caller and reduces the MinterReserve by `reservePPM` of the minted amount. This full amount is then emitted in the Profit event. However, in the special case where part of the MinterReserve has been used to cover bad debt (after the equity is fully emptied), the position's will only be partially left. As a result, the emitted Profit will be too high.

Only the remaining part of the position's minter reserve should be emitted.

Acknowledged:

The Frankencoin team acknowledged the issue but has not changed the code, as the Frankencoin contract is already deployed.

However, they have added the following comment to the code:

```
//The Profit event can overstate profits in case there is no equity capital left.
```

5.3 buyExpiredCollateral Can Leave Dust

Design **Low** **Version 1** **Risk Accepted**

CS-ZCHF2-010

The `buyExpiredCollateral` function transfers a user-provided amount of collateral from the expired position. There is no `minimumcollateral` check, so the position could be left with a small amount of collateral that is not worth claiming by calling `buyExpiredCollateral` again.

Risk Accepted:

Frankencoin accepts the risk with the following response:

```
This is accepted. The assumption is that someone will altruistically buy
the remaining collateral in order to trigger correct accounting.
```

If nobody buys the full amount of collateral, the bad debt created by that position will stay unaccounted. See buyExpiredCollateral Can Dodge Liquidation Penalty.

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.

`Critical`-Severity Findings	0

`High`-Severity Findings	2

- buyExpiredCollateral Can Disincentivize Challenging `Code Corrected`
- forceSale Does Not Account for Bad Debt `Code Corrected`

`Medium`-Severity Findings	4

- Minimum Collateral Can Be Partially Withdrawn `Code Corrected`
- Savings Can Spend MinterReserve `Code Corrected`
- Untrusted Call Can Break Roller Approval `Code Corrected`
- buyExpiredCollateral Can Dodge Liquidation Penalty `Code Corrected`

`Low`-Severity Findings	1

- Incorrect Calculation in LockedFunds Error `Code Corrected`

Informational Findings	5

- Event Indexing `Code Corrected`
- Outdated Comment `Specification Changed`
- Roller Cannot Extend Expiry for the Owner `Code Corrected`
- Savings Must Be Taken Into Account for Profit Calculation `Code Corrected`
- Users Should No Longer Challenge Expired Positions `Code Corrected`

6.1 buyExpiredCollateral Can Disincentivize Challenging

`Design` `High` `Version 1` `Code Corrected`

CS-ZCHF2-001

The `buyExpiredCollateral` function can remove collateral from a challenged position. As a result, the challenger will not receive a reward for challenging the position. This discourages challenging positions that are close to expiry.

This could be a problem in the following scenario:

1. A position has a challengePeriod of more than 3 days.

2. At 3.01 days before expiry, the position owner adjusts the price to a very high value. This puts the position on cooldown for 3 days.

3. Someone must challenge the position, otherwise the owner will be able to mint unbacked ZCHF (the cooldown ends before expiry and the price is too high).

4. If there is a challenge, the owner can use `buyExpiredCollateral` to repay their position and remove the collateral. The challenger will not receive a reward for challenging the position.

5. The challenger would make a loss, as they must pay gas and lock up collateral to challenge the position.

In this scenario, challenging the position is critical for system health, but the challenger is disincentivized from doing so. It only affects positions with a challengePeriod of more than 1.5 days (as otherwise the challenge price will reach zero before the cooldown is over).

Code corrected:

The issue has been resolved by adding two restrictions:

1. The `buyExpiredCollateral` function can now only be called if there is no ongoing challenge.

2. A position can no longer be challenged after it has expired.

This ensures that the challenger will always receive a reward for successfully challenging a position, as the collateral cannot be removed during a challenge.

6.2 forceSale Does Not Account for Bad Debt

Correctness High Version 1 Code Corrected

CS-ZCHF2-002

The `forceSale` function can be used to liquidate a position after it has expired. The function checks whether the proceeds from the sale are enough to cover the full debt of the position.

```
if (proceeds + availableReserve >= minted) {
        //[...]
        } else {
            zchf.transferFrom(buyer, address(this), proceeds);
            uint256 repaid = zchf.burnWithReserve(proceeds, reserveContribution);
            _notifyRepaid(repaid);
        }
```

The case where we do not have enough to repay the full minted amount can be reached in two ways:

1. The bidder only bought part of the collateral, leaving debt and collateral in the position.

2. The bidder bought the full collateral, but the collateral is worth less than the debt. This leaves the position bad debt, not backed by any collateral.

Case 2 is not handled correctly. If there is bad debt, the system should account for this by calling the `Frankencoin.coverloss()` function, which will burn ZCHF from the Equity contract to cover the loss.

Not accounting for bad debt could cause the ZCHF to lose value (depeg), as there are now unbacked tokens in circulation. Additionally, the FPS price will be incorrect, as there are more ZCHF remaining in the Equity contract than there should be.

In case this happens, the accounting could be corrected by doing the following:

1. Send 1 wei of collateral to the bad debt position (a position with zero collateral cannot be challenged).

2. Challenge the position.

3. Wait for the first challenge period to end.

4. Bid on the position, paying a tiny amount (as there is only 1 wei collateral).

5. The `_finishChallenge` function will correctly account for the loss.

If another user calls `bid()` before the challenge period ends, the challenge will be averted, and the bad debt will remain unaccounted for. So the remediation only works if there are no uncooperative users.

In summary, bad debt must be accounted correctly when liquidating an expired position through `forceSale()`. Otherwise the ZCHF may depeg.

Code corrected:

The issue has been resolved. Bad debt incurred through insufficient proceeds from a `forceSale()` are now handled correctly by accounting for it via `coverLoss()`.

Note that the accounting is only updated if there is no remaining collateral in the position. As long as there is 1 wei of collateral remaining, the bad debt is not yet accounted.

6.3 Minimum Collateral Can Be Partially Withdrawn

`Design` `Medium` `Version 1` `Code Corrected`

CS-ZCHF2-003

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 `withdrawCollateral` function of 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.

The `minimumCollateral` check was present in the previously audited version of Position, but is no longer present in `Version 1` of this review.

Code corrected:

A check has been added to `_checkCollateral()`, which ensures that the collateral cannot be reduced below the minimum collateral amount, unless there is zero debt minted against the position.

6.4 Savings Can Spend MinterReserve

`Correctness` `Medium` `Version 1` `Code Corrected`

CS-ZCHF2-004

The `refresh` function in Savings contains the following check:

```
if (earnedInterest > 0 && zchf.balanceOf(address(equity)) >= earnedInterest) {...}
```

This is meant to ensure that the Savings contract cannot spend more than the ZCHF's equity.

However, checking `balanceOf()` does not enforce this correctly, as the Equity contract stores tokens for two separate purposes: the equity that belongs to the FPS holders, and the MinterReserve that belongs to Position owners. Due to the incorrect check, the MinterReserve could be spent on paying interest to savers, if the FPS equity has gone to zero.

The equity without including the MinterReserve can be queried using `Frankencoin.equity()`.

Code corrected:

Frankencoin resolved the issue. A `calculateInterest()` function has been introduced that checks the computed earned interest against `Frankencoin.equity()`. In case the `equity()` is smaller than the earned interest, only the amount in equity is paid to the saver.

6.5 Untrusted Call Can Break Roller Approval

`Correctness` `Medium` `Version 1` `Code Corrected`

CS-ZCHF2-005

In the PositionRoller, the `roll` function does not validate the `source` and `target` contracts supplied by the user. As a result, the `target` contract may be a malicious contract.

This can cause a problem if a contract with the following behavior is used:

- `target.collateral()` is the Frankencoin token contract
- `target.hub()` returns a victim position when it is called the first time, but returns an attacker controlled contract that implements the `clone()` function when it is called the second time.

With this setup, an attacker could cause the PositionRoller to make a `Frankencoin.approve()` call, approving the victim contract to spend a small amount of ZCHF. This would make the victim Position unable to use the roller to roll to a new position, as the roller will no longer have infinite approval.

This is because of the special case in the Frankencoin's `_allowance` function:

```
function _allowance(address owner, address spender) internal view override returns (uint256) {
    uint256 explicit = super._allowance(owner, spender);
    if (explicit > 0) {
        return explicit; // don't waste gas checking minter
    } else if (isMinter(spender) || isMinter(getPositionParent(spender)) || spender == address(reserve)) {
        return INFINITY;
    } else {
        return 0;
    }
}
```

Usually, Position contracts have an infinite allowance to spend Frankencoin tokens, as they have the minter role. However, if an explicit allowance is set, the explicit allowance is used instead.

In summary, a malicious `target` contract could cause a Position to lose its ability to use the roller. The approval could be reset to zero in the same way, re-enabling the Position to use the roller.

Code corrected:

The `source` and `target` contracts are now validated. They must be Positions registered by a valid Minter in the Frankencoin contract.

This ensures that the PositionRoller can only call position contracts that were accepted by governance.

6.6 buyExpiredCollateral Can Dodge Liquidation Penalty

`Design` `Medium` `Version 1` `Code Corrected`

When a position is challenged successfully, the user forfeits their reserve contribution as a liquidation penalty. The reserve will be assigned to the equity as a profit.

However, the user can dodge the liquidation penalty by calling the `buyExpiredCollateral` function, if the position was challenged near expiry. `buyExpiredCollateral` will allow the user to repay their debt and receive their reserve contribution back.

Code corrected:

Frankencoin resolved the issue by adding the `noChallenge` modifier to `forceSale()`. Any call to `buyExpiredCollateral()` on a challenged position will thus revert.

6.7 Incorrect Calculation in LockedFunds Error

`Correctness` `Low` `Version 1` `Code Corrected`

The data passed to the `FundsLocked` error in the revert upon early withdrawal is incorrect. The error takes the `remainingSeconds` parameter but the data passed is not in the unit of seconds, as there are missing parentheses.

```
revert FundsLocked(uint40(account.ticks - currentTicks() / currentRatePPM));
```

Code corrected:

The code has been altered accordingly and the correct units are passed.

6.8 Event Indexing

`Informational` `Version 1` `Code Corrected`

- The `Saved`, `Withdrawn` and `InterestCollected` events do not index the address of the account.
- The `RateProposed` event does not index the address of the proposer.

Indexing the above mentioned fields may be useful for filtering events.

Code corrected:

- The `Saved`, `Withdrawn` and `InterestCollected` events now index the address field.
- The `RateProposed` event remains unchanged.

6.9 Outdated Comment

Informational | Version 1 | Specification Changed

The comment in the Savings contract states:

```
The saved ZCHF are subject to a lockup of up to 14 days and
only start to yield an interest after the lockup ended.
```

This comment is outdated. In Version 1, the lockup period is three days.

Specification changed:

The comment has been updated to reflect the correct lockup period of three days.

6.10 Roller Cannot Extend Expiry for the Owner

Informational | Version 1 | Code Corrected

The Roller contract creates a clone of the `target` contract if the caller is not the owner of the `target`, or if the `expiry` is reduced. However, there is one unlikely edge case that is not handled correctly.

If the caller is the owner of `target` and gives an expiry that is greater than the expiry of the `target`, the Roller contract will not create a clone. It may be possible that the user wants to create a clone that has a longer expiry than the `target` (but not longer than the `original`), if the `target` is a clone that reduced `expiry` when it was created.

Code corrected:

The `roll` function has been updated to also create a clone if the caller is the owner of the `target` and the `expiry` is greater than the `target`'s expiry.

6.11 Savings Must Be Taken Into Account for Profit Calculation

Informational | Version 1 | Code Corrected

Whenever the Frankencoin Equity makes a profit (e.g. by collecting fees), the `Profit` event is emitted. When there is bad debt, the `Loss` event is emitted.

When the Savings contract takes ZCHF from the Equity contract in the `refresh()` function, it emits the `InterestCollected` event.

The current profit of the system can be calculated as `Profit - Loss - InterestCollected`.

Note that this does not take into account pending interest, which has not yet been collected. It will be fully realized when the `refresh` function is called on all users with pending interest.

Code corrected:

The `refresh()` function now uses the `coverLoss` function to transfer the ZCHF from the Equity contract to the Savings contract. The `coverLoss` function emits the `Loss` event. As a result, the system profit can be calculated as `Profit - Loss` in Version 2, as interest collected is counted as a loss.

Pending interest is still not taken into account.

6.12 Users Should No Longer Challenge Expired Positions

Informational **Version 1** **Code Corrected**

CS-ZCHF2-020

Expired positions can now be liquidated using the `buyExpiredCollateral()` function. However, it is also possible to challenge a position that is close to expiry or already expired. If the collateral is sold using `buyExpiredCollateral()` while there is an open challenge, the challenger will not receive a reward.

As long as the challenge started before the `expiry`, it will always be cheaper to bid on the challenge than use `buyExpiredCollateral()`. However, if the challenge is started after the `expiry`, bidding on the challenge can offer a worse price.

In conclusion, users should not challenge expired positions, as they may not receive a reward. This should be taken into account when writing bots that automatically challenge positions. Note that challenging expired positions was profitable in the previous version of the MintingHub.

Code corrected:

The issue has been resolved by adding two restrictions:

1. The `buyExpiredCollateral` function can now only be called if there is no ongoing challenge.

2. A position can no longer be challenged after it has expired.

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 Floating Pragma

Informational **Version 1** Acknowledged

The contracts use a floating pragma solidity ^0.8.0. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively (https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md).

Acknowledged:

Frankencoin acknowledges the issue but has decided not to change the compiler pragma.

7.2 Integer Division Before Multiplication

Informational **Version 1** Acknowledged

Depending on the time passed since position expiry, the `expiredPurchasePrice` is computed in the following ways:

```
liqprice + (((EXPIRED_PRICE_FACTOR - 1) * liqprice) / challengePeriod) * timeLeft;
```

```
(liqprice / challengePeriod) * timeLeft;
```

To minimize potential rounding errors, multiplication could be performed before division.

Acknowledged:

The Frankencoin team decided against changing the order with the following reasoning:

```
Rounding error is acceptable in comparison to the risk of overflows
```

7.3 Interest Rate Can Overflow

Informational **Version 1** Acknowledged

The annual interest rate of a position is calculated as follows:

```
function annualInterestPPM() public view returns (uint24) {
    return IHub(hub).rate().currentRatePPM() + riskPremiumPPM;
}
```

The `currentRatePPM` could be up to `uint24.max` and the `riskPremiumPPM` could be up to 1000000. This means that the sum of the two could overflow a `uint24`.

The `currentRatePPM` is set by governance in the Leadrate contract. Governance should likely never set the rate to the maximum, so this is not a problem. However, enforcing a lower maximum could help prevent mistakes.

Acknowledged:

Frankencoin acknowledges the issue but has decided not to make a code change.

7.4 Owner Can Bid at Any Expired Collateral Price

`Informational` `Version 1` `Acknowledged`

CS-ZCHF2-016

The `buyExpiredCollateral` function starts with a collateral price of `10 * liquidationPrice` and reduces the price over time. Usually, users need to wait until the price reaches a suitable level before they can call the function.

However, the position owner can call the function at any time, regardless of the price. This is because the owner will pay the price to themselves, so there is no loss even if they pay a high price.

This lets the owner repay their position and withdraw their collateral after expiry, at no extra cost.

Acknowledged:

Frankencoin acknowledges the issue and states:

> Impact reduced thanks to resolution of buyExpiredCollateral Can Disincentivize Challenging.

7.5 Unrealized Interest Not Considered in Equity

`Informational` `Version 1` `Acknowledged`

CS-ZCHF2-018

The interest earned by depositing Frankencoin to a savings account is paid from the equity of the system. Note that earned interest is only realized once a user's savings account is refreshed. A refresh is triggered upon calling `save()`, `withdraw()` or can be triggered explicitly by on any address to compound interest.

This implies that there will be an amount of Frankencoins that are attributed to the equity as long as there are savings accounts with unrealized interest. These tokens should technically not be accounted for in the equity anymore. The equity's balance influences the price per FPS token. As FPS tokens are cheaper when there is less equity in the system it could be favourable for FPS buyers to refresh users' savings accounts in case there are savers with a significant amount of unrealized earned interest.

While there is unrealized interest, the FPS price will be slightly higher than it should. Anyone using `redeem()` to turn their FPS into ZCHF will receive a slightly higher rate than they should. However, it

seems unlikely that the price difference will be more than the 0.3% fee that is charged when using `invest` or `redeem`. As a result, it is unlikely that an arbitrage will be possible due to the mispricing.

Acknowledged:

Frankencoin acknowledges the issue.

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   Deployment of Position Roller

`Note` `Version 1`

The PositionRoller and the new MintingHub must both be granted the `minter` role to be used as intended. To receive the `minter` role, they must not be vetoed by governance. The contracts must be proposed as minters separately, so it could happen that one of them is vetoed and the other is not.

A vetoed roller would still have been set as the `roller` in the constructor of the MintingHub, and pass the `ownerOrRoller` checks in its positions. However, it would be unable to provide flashloans.

If the PositionRoller is vetoed by governance, the MintingHub referencing it should be vetoed as well. Otherwise, the system will be in an unintended state.

8.2   Governance Should Change Lead Rate Gradually

`Note` `Version 1`

The Leadrate in the system can be adjusted by governance. Governance should not change the lead rate too quickly for the following reasons:

1. Users will see an interest rate change coming. They can mint with the old interest rate before the change is applied.

2. Positions pay their interest upfront, but the interest to savers is paid over time. If the interest rate is increased, it is possible for savers to receive more interest than was paid by the positions.

3. Reducing the interest rate by a large factor will cause savers to be locked in the Savings module for longer than they expected.

8.3   Higher Reserve Factor Implicitly Increases Interest Rate

`Note` `Version 1`

When proposing a position, the user must choose a reserve factor between zero and one-hundred percent. A higher reserve factor will make the position less risky for the system, as it will be able to absorb more losses before becoming undercollateralized.

Due to the way the interest rate is calculated, a higher reserve factor will also increase the "lead" interest rate for the position. This is because the lead rate is the minimum rate charged on the full minted amount of the position. However, the user only receives a part of the minted tokens. The user can choose to pay an additional risk premium, but they must always pay at least the lead rate.

Consider the following examples when the leadrate is `5%`, expiration is in one year, and the minted amount is `100`:

1. Reserve: `20%`. The user receives `80` tokens. They pay `5` interest. Effective interest rate: `6.25%`.

2. Reserve: `40%`. The user receives `60` tokens. They pay `5` interest. Effective interest rate: `8.33%`.

Users and governance should be aware of how the interest is applied, and factor the reserve into their calculations when choosing an appropriate risk premium.

8.4 Integrations Must Check Withdraw Return Value

Note Version 1

The `withdraw` function takes an `amount` parameter. If the amount deposited is smaller than the requested `amount`, the full balance is withdrawn instead and the withdrawn amount is returned. Integrations must check the return value and not assume that the function would revert if the requested `amount` is not available.

8.5 Savings Could Remain Locked Indefinitely

Note Version 1

The Savings module uses account ticks for time weighted per-second interest computation. When depositing funds via `save()`, the account's ticks are updated, which incurs a lockup period on the account during which the user can not withdraw their savings.

The contract disallows users to deposit if the system's interest rate could be set to 0 within the next three days as this is the maximum lockup period in case the interest rate does not change during lockup. This check is performed with the intention to guard the user from having their funds locked indefinitely.

However, there is a scenario where users could be locked indefinitely. An account is considered as locked if `account.ticks > currentTicks()` holds. The `currentTicks()` are linearly dependent on the current interest rate. Halving the interest rate would hence result in double the time until the system ticks and account ticks are equal and the funds become unlocked.

Locking an account indefinitely could happen as follows:

1. The current interest rate is lowered just after a user deposited savings - while the user's account is in a locked state (*currentTicks() < account.ticks*).

2. As the system ticks accumulate w.r.t. the new, smaller interest rate, the funds will be locked for longer (inversely proportional to the rate change). E.g. if the interest rate is reduced to one third, the lock will now be nine days instead of three.

3. The `currentRatePPM` is set to 0 (possible to happen seven days after last rate change), while *currentTicks() < account.ticks* still holds.

4. The user's savings account are now locked indefinitely, as the system no longer accumulates ticks.

In order to unlock funds, governance could allow to set the interest rate back to a non-zero value.

To prevent this, governance should ensure that the interest rate is not reduced by a large percentage shortly before setting it to zero.