PUBLIC

Code Assessment

of the dEURO Smart Contracts

April 3, 2025

Produced for

dEURO

by

CHAINSECURITY

Contents

1 Executive Summary

Dear dEURO Team,

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

dEURO implements a decentralized protocol to issue dEURO on-chain, a stablecoin that is pegged to the Euro. Each dEURO minted is backed either by collateral assets or other trusted stablecoins pegged to the EURO.

The smart contracts are forked from Frankencoin v2024. This review was limited to the changes applied by dEURO, under the assumption that the Frankencoin codebase does not contain any vulnerabilities.

The most critical subjects covered in our audit are asset solvency, functional correctness and accounting correctness. Security regarding asset solvency was improved, see Interest Accrual Can Lead to Under-Collateralization. Security regarding functional correctness was improved, see Auctions May Never End. Accounting correctness is improvable, see Challengers must calculate virtualPrice themselves.

In summary, we find that the codebase currently provides a good 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	1
• Code Corrected	1
Low-Severity Findings	7
• Code Corrected	2
• Specification Changed	2
• Acknowledged	3

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 dEURO 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	03 Feb 2025	88dbcbc52b0f385753ee74848f3c6c04b33d24a5	Initial Version
2	23 Mar 2025	5b41929fc9924e843b65c1f3cff383c85a75646b	Version with Fixes

For the solidity smart contracts, the compiler version `0.8.26` was chosen. The following files in the folder `contracts` are in scope:

```
gateway/FrontendGateway.sol
gateway/MintingHubGateway.sol
gateway/SavingsGateway.sol
impl/ERC3009.sol
MintingHubV2/MintingHub.sol
MintingHubV2/Position.sol
MintingHubV2/PositionFactory.sol
MintingHubV2/PositionRoller.sol
utils/DEPSWrapper.sol
utils/MathUtil.sol
DecentralizedEURO.sol
Equity.sol
LeadRate.sol
Savings.sol
StablecoinBridge.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 or stablecoins in the system were not in the scope of this code assessment. Moreover, third party libraries are assumed to behave correctly and according to their specification.

In this report, we assume Frankencoin v2024 is safe and the review is focused on the changes applied by dEURO to the smart contracts from Frankencoin. Therefore, any bug present in Frankencoin v2024 might still be present in dEURO. Finally, the soundness of the financial model was not evaluated.

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.

dEURO is an oracle-free stablecoin protocol, which was forked from Frankencoin v2024.

This system overview focuses on the parts that have changed between dEURO and Frankencoin. For an overview of the full functionality, refer to the Frankencoin v2024 audit report and the Frankencoin v1 audit report.

The changes made in the dEURO contracts are as follows:

1. ZCHF was renamed to dEURO.
2. Frankencoin was renamed to DecentralizedEURO.
3. FPS was renamed to nDEPS (native Decentralized Protocol Share).
4. nDEPS now cost 10_000 times less than the FPS for Frankencoin.
5. In the Equity smart contract, the valuation factor was adjusted from 3 to 5.
6. The ERC20 token has been completely converted to standard OpenZeppelin V5.
7. The ERC165 token standard has been added to dEURO.
8. The ERC3009 token standard added to dEURO.
9. The Equity internal exchange fee (issuance fee) was increased from 0.3% to 2%.
10. Minters are no longer authorized to execute `sendFrom` and `burnFrom` from any address.
11. In Savings, the lock-up and interest earning delay of 3 days has been removed.
12. The WFPS wrapper has been renamed to DEPS.
13. The StablecoinBridge now supports stablecoins with a configurable number of decimals (not just 18). dEURO plans to use multiple bridges to multiple EUR stablecoins.

Additionally, the largest changes are the removal of up-front interest in Position, and the new FrontendGateway, MintingHubGateway, and SavingsGateway, which enable a referral mechanism. A new limitation has been added in function `buyExpiredCollateral` which now requires that enough collateral (worth at least 1000 dEuro) remains in the position if it is not fully liquidated.

The impact the changes have on the profitability of the system have been described here: Factors influencing system profitability.

2.2.1 Position

In dEURO, Positions no longer charge the interest for the loan duration upfront. Instead, they allow minting up to the `limit` amount of dEURO to the user and minter reserve (in Frankencoin the upfront interest is also counted towards the limit). The interest now accrues over time and is paid when the position's debt is paid down (e.g. when closing the position). The interest is accounted as a profit to the equity at the time it is paid. If the user repays their position before the expiry, they will no longer need to pay interest on the repaid part. Interest is only paid on principal, there is never interest on interest.

The interest paid by the position is computed as the sum of the lead rate interest and the `riskPremiumPPM`. When new dEuro are minted, the interest rate for the whole amount is recorded in the state variable `fixedAnnualRatePPM`. If the lead rate goes down, borrowers have an incentive to mint a small amount of additional dEuro, so they pay less interest. In case the lead rate goes up, borrowers have an incentive to mint from a new position such that the existing minted amount pays the lower interest.

2.2.2 FrontendGateway

The FrontendGateway introduces a referral system to dEURO. There are three types of referrals:

1. Savings referrals: When a user deposits into Savings, the referrer receives a percentage reward whenever the user earns interest.
2. Position referrals: When a user opens a position, the referrer receives a percentage reward whenever the user pays interest.
3. Investment referrals: When a user invests or redeems in the Equity, the referrer receives a percentage reward on the invested or redeemed amount.

Savings rewards are always paid to the last frontend code used by the user. In contrast, position rewards are always paid to the frontendcode that was used when creating the position and cannot be changed later.

The referral rewards are paid in dEURO by the equity of the system (which belongs to the nDEPS holders). The user pays the same fees whether or not they have a referrer. However, a user could refer themselves to receive rewards for their own actions.

The percentage of the referral rewards can be voted on by the nDEPS holders. The mechanism used is identical to that in the LeadRate contract. Anyone with at least 2% of the nDEPS votes can propose new referral reward percentages. If nobody makes a different proposal for 7 days, the rate change can be executed by anyone. If someone wants to cancel a change, they can do so by making a proposal to set the rate back to the previous value.

Anyone can register as a referrer by calling the `registerFrontendCode` function. This will set them as the owner of their chosen `bytes32` code. Claiming the zero code is not allowed. When a user interacts with the system, they can provide the frontend code to accrue rewards to it. The owner can then claim the rewards using `withdrawRewards()`. The owner of a frontend code can transfer it a new owner using `transferFrontendCode()`.

The FrontendGateway exposes the following functions, with which a user can provide a frontend code and take an action in the same call:

- `invest`
- `redeem`
- `unwrapAndSell`

2.2.3 MintingHubGateway

The MintingHubGateway is derived from the MintingHub contract and exposes new versions of the `openPosition` and `clone` functions. They are exactly the same as in MintingHub, except that they take an additional `frontendCode` argument. After opening the new position, the call `FrontendGateway.registerPosition()` to register the referrer for the position. This will be used to allocate the position referral rewards whenever interest is paid for the position.

Additionally, the MintingHubGateway has a new function `notifyInterestPaid`, which is called by positions whenever interest is paid, to trigger the rewards.

2.2.4 SavingsGateway

The SavingsGateway is derived from the Savings contract. It overwrites the `refresh` function to add a `FrontendGateway.updateSavingsRewards()` call after collecting interest. This is used to allocate savings rewards.

Additionally, it adds the new functions `save`, `adjust` and `withdraw`, which take the additional parameter `frontendCode`. This code gets sent to the frontend gateway which updates the savings code for the caller (`msg.sender`). The caller's saving code is updated also when saving for another account.

The SavingsGateway requires the `minter` role to be able to pay rewards from the equity.

2.2.5 Roles and Trust Model

The `minter` role is privileged in the system. Any `minter` in the DecentralizedEURO contract is assumed to be fully trusted, with the power to mint unlimited dEURO. Initially, only the MintingHub, StablecoinBridge, and FrontendGateway contracts should have the `minter` role. DecentralizedEURO 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 nDEPS holders that own 51% of the total voting power in the Equity contract always behave in the best interests of the system. nDEPS holders are trusted to continuously monitor and veto proposals for adding untrusted minters in DecentralizedEURO or opening positions with unfair parameters in the MintingHub. Similarly, 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. In case a minority of nDEPS 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.

FrontendGateway has a `owner` role that is trusted only to configure the contract correctly after deployment by calling `init()`. The role is automatically renounced afterwards.

The external stablecoins supported in StablecoinBridge are considered to be non-malicious, behave according to the specifications and maintain their peg to the Euro. These stablecoins should be ERC20-compliant with no special features (e.g., rebasing, transfer hooks, capping of the transferred amount, fees on transfer, etc.) and revert or return false 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).

2.2.6 Changes in Version 2

- The `virtualPrice` has been introduced, which is used for challenges and post-expiry liquidations instead of the Position's `price`. The `virtualPrice` is defined as: `max((principal + (interest / 1 - reserveContribution)) / collateralBalance, price)`. This means the position's liquidation price will slowly increase as interest accrues, and it can be reduced by adding collateral. However, it can never be lower than `price`. If there is an ongoing challenge, the `virtualPrice` at the beginning of the earliest challenge is pinned and only reset once all challenges have ended.
- The collateral in a position is now required to be sufficient to cover the accrued interest and the interest reserve, in addition to the principal. This requirement is enforced on the internal function `_checkCollateral`, which is called whenever the position owner mints new tokens, withdraws collateral, or adjusts the position's `price`.
- A minimum challenge period of `1 day` has been introduced. This means that it is impossible to configure Positions to use shorter challenge periods. Long challenge periods increase the risk of a collateral falling significantly in value before it can be liquidated. Note that a liquidation may take up to 2 full challenge periods to complete.

- In PositionRoller, the `repay` parameter in `roll()` now takes the total amount of debt to repay, not just the principal. Previously the interest was added on top.
- In PositionRoller, the `rollFullyWithExpiration` function now attempts to roll the interest of the `source` position into the `target` position as principal. This will lead to an increase in principal.

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 Open 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

5.1 Challengers Must Calculate virtualPrice Themselves

Correctness Low Version 2 Acknowledged

CS-dEUR-021

There is no minimum remaining amount check when calling `bid()` on a challenge in MintingHub. As a result, anyone could challenge their own position and then bid on it, leaving just a 1 wei challenge. As challenging pins the `virtualPrice` to the value it had when the challenge started, this will keep the virtualPrice from increasing as interest accrues. Anyone can update the virtualPrice to be correct again by completing the challenge, however they need to pay the gas for this and will not receive any compensation if the challenge amount is very small. Additionally, if someone is monitoring the `virtualPrice` or `expiredPurchasePrice` of positions, they may not notice that it is outdated and could be increased by finishing the previous challenge.

Challengers monitoring positions must be aware of this behavior and calculate the `virtualPrice` themselves, as the public `virtualPrice` function may not be up to date due to ongoing small challenges. Similarly, liquidators should compute `expiredPurchasePrice` themselves since it relies on `virtualPrice` which can be outdated.

Acknowledged:

dEURO has acknowledged this behavior, but has decided not to change it in the current version of the system. A fix will be considered in a future version.

5.2 No Slippage Protection in buyExpiredCollateral

Design Low Version 2 Acknowledged

CS-dEUR-022

As of Version 2, the `expiredPurchasePrice` in `buyExpiredCollateral()` is calculated based on the `virtualPrice`, not the `price`. If there is a call to `buyExpiredCollateral()` that only buys a part of the total collateral, this can increase the `virtualPrice`, as more collateral than debt can be removed from the position. As a result, `buyExpiredCollateral()` may buy the collateral at a higher price than expected, if it is called multiple times.

Acknowledged:

dEURO has acknowledged this behavior, but has decided not to change it in the current version of the system. A fix will be considered in a future version.

5.3 Limited Support for Referrals in PositionRoller

Design Low Version 1 Acknowledged

CS-dEUR-007

The FrontendGateway introduces referral rewards for positions. However, the PositionRoller does not support this functionality in all scenarios. If a user wants to roll a position that has been created from a different MintingHub, the new position will be registered with an empty code:

```
bytes32 frontendCode = IMintingHubGateway(target.hub()).GATEWAY().
    getPositionFrontendCode(address(source));
```

The position `source` does not exist in the gateway of `target.hub()` if `source` and `target` are created by different minting hubs, hence the empty code is used for the new position.

Acknowledged:

dEURO is aware of this limitation but has decided to leave the code unchanged for this iteration.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open 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

- Auctions May Never End Code Corrected
- Interest Accrual Can Lead to Under-Collateralization Code Corrected

Medium-Severity Findings	1

- Minter Reserves Updated Incorrectly Code Corrected

Low-Severity Findings	4

- Inconsistent Values Used for Infinite Allowance Code Corrected
- Incorrect Code Comments Specification Changed
- Max Interest Check Is Ineffective Specification Changed
- Zero-value Parameter Always Passed to mintWithReserve Code Corrected

Informational Findings	1

- Integer Division Before Multiplication Code Corrected

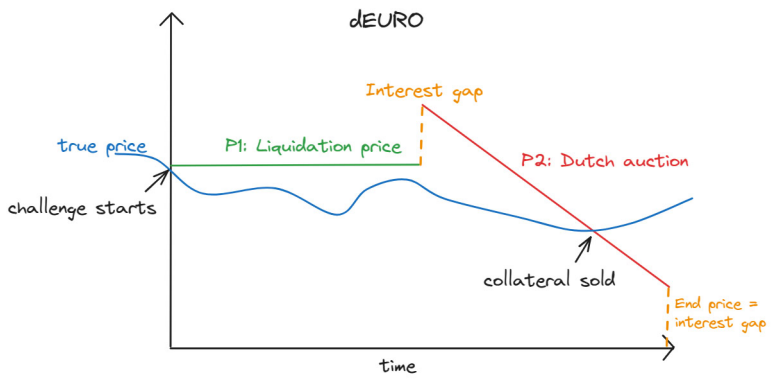
6.1 Auctions May Never End

Design High Version 1 Code Corrected

CS-dEUR-001

When a position is challenged or expired, an auction is started. Once the second phase of the auction is started, the price for the collateral declines linearly from the liquidation price down to zero. However, in Version 1 of the codebase, the amount required to repay the accrued interest is added on top of the auction price. The full interest (in proportion to how much of the collateral is liquidated) must always be paid by the bidder. This effectively increases the auction's start and end price.

The change in auction price is illustrated here with the "auction gap":



In the case where the interest alone has a higher value than the collateral of the position, the auction price will reach zero. However, bidders will still not be incentivized to bid. This is because the cost of the interest does not decrease over time. As a result, the position will never be liquidated, and the auction will never end.

This can happen if the collateral price drops too quickly, or if the interest is a large percentage of the total debt.

Consider these examples:

Example 1: Collateral price drops too quickly

- Current collateral price: 100
- Liquidation price: 1000
- Collateral: 1
- ReserveRequirement: 20%
- Principal: 1000 (800 to user, 200 to reserve)
- Interest: 150
- Current auction price: 0 (final price)

This auction will never end as it is not profitable to buy the 1 collateral by paying back the 150 interest (which the bidder must do even though the auction price is zero).

Example 2: Interest is a large percentage of total debt

- Current collateral price: 900
- Liquidation price: 1000
- Collateral: 1
- ReserveRequirement: 20%
- Principal: 1000 (800 to user, 200 to reserve)
- Interest: 950
- Current auction price: 0 (final price)

This auction will never end as it is not profitable to buy the 1 collateral by paying back the 950 interest (which the bidder must do even though the auction price is zero).

In these cases, there are 3 problems:

1. The remaining collateral is not sold
2. Bad debt is not correctly accounted
3. If it was a challenge (not position expiry), the challenger will not receive their collateral back (unless they bid themselves and take a loss). The challenger will also not receive a reward.

Even if the auction does end, the challenger's reward may be very small, as they only receive a percentage of the repaid principal, not the repaid interest. As interest is repaid first, the repaid principal can be small even when the collateral value is substantial.

In summary, the separate accounting of the interest changes the auction dynamics and can cause liquidations that never end. It can also cause positions that should be liquidated due to being undercollateralized but cannot be challenged, see Interest accrual can lead to under-collateralization.

Code corrected:

The liquidation functionalities have been refactored to remove the separate accounting of a position's interest in the liquidation process. Now the same price is applied to the principal and the interest to be repaid. In Version 1, partial liquidations were designed to pay a share of both interest and principal of the position. In Version 2, partial liquidations are in line with usual debt repayments, thus the interest of a position is paid first and then the principal. If the collateral price drops quickly, the position may cause bad debt, but the position gets liquidated as expected.

On bidding, the function `Position.notifyChallengeSucceeded()` pays the debt as follows:

```
_notifyInterestPaid(interestToPay);
_notifyRepaid(principalToPay);
```

Similarly, `Position.forceSale()` repays the debt in the same order when a position is liquidated after expiry.

6.2 Interest Accrual Can Lead to Under-Collateralization

Design High Version 1 Code Corrected

CS-dEUR-002

The function `Position._checkCollateral()` ensures that the position has sufficient collateral to back its debt principal given the liquidation price of the position:

```
function _checkCollateral(uint256 collateralReserve, uint256 atPrice) internal view {
    ...
    if (relevantCollateral * atPrice < principal * ONE_DEC18) {
        revert InsufficientCollateral(...);
    }
}
```

However, the total debt of the position is principal plus interest (tracked in a dedicated state var `accruedInterest`). There is no check that the accrued interest is also backed by sufficient collateral. For positions with long durations or high interest rates, it is possible that they become undercollateralized, as the total debt including interest could become larger than the collateral value. Essentially, the position's reserve is reduced by the accrued unpaid interest. If the interest becomes greater than the position's reserve, it will create bad debt for the system, while still not being liquidatable. The liquidation mechanism relies on the total debt amount being overcollateralized (with a reserve amount as buffer).

In the Version 1 design, reserve amounts must be increased by the maximum amount of interest that can be accrued of the position lifetime, in order to guarantee the same liquidation buffer as in Frankencoin. However, the maximum interest depends on the variable lead rate.

Code corrected:

In Version 2, the `_checkCollateral` function has been updated to include the accrued interest in the collateral check. Now, the function checks that there is sufficient collateral at `price` to `principal + (interest / (1 - reserveContribution))`. This ensures that the principal and interest are both overcollateralized by the reserve contribution factor.

6.3 Minter Reserves Updated Incorrectly

Correctness | Medium | Version 1 | Code Corrected

CS-dEUR-003

In DecentralizedEURO, the `burnFromWithReserveNet` function burns tokens from a payer and updates the minter reserves. In the case where the reserve of the position is fully remaining, it behaves as expected. However, in the case where the reserves have been used to cover bad debt in the system, it does not.

The function looks as follows:

```
function burnFromWithReserveNet(
    address payer,
    uint256 amountExcludingReserve,
    uint32 reservePPM
) external override minterOnly returns (uint256) {
    uint256 freedAmount = calculateFreedAmount(amountExcludingReserve, reservePPM); // Add reserve portion
    minterReserveE6 -= freedAmount * reservePPM; // reduce reserve requirements by original ratio
    _transfer(address(reserve), payer, freedAmount - amountExcludingReserve); // collect assigned reserve
    _burn(payer, freedAmount); // burn the freed amount
    return freedAmount;
}
```

In order to "reduce reserve requirements by original ratio", as stated in the comment, any missing reserves must be covered by the payer in addition to the amountExcludingReserve. However, here we do not enforce that the missing reserves are covered.

They are optionally covered after returning from the call in `_repayPrincipalNet()`, but only if there are extra funds available. However, even if the funds are available there, they are burnt without updating `minterReserveE6`, which still leads to incorrect reserve accounting.

In `burnFromWithReserveNet()`, if the reserve is not fully remaining, we should not pay out the full reserve left. Instead, we should retain the reserve portion of the principal that will be left after the call. Currently we transfer the full remaining reserve to the payer, even if there will be principal left after the call.

Consider the following example:

1. A position has `reservePPM = 20%` and `principal = 1'000` dEURO
2. Half of the position's reserves have been used to cover bad debt, so the remaining reserves are 10%
3. `burnFromWithReserveNet()` is called to repay the full amountExcludingReserve of `800` dEURO
4. The `freedAmount` will be `900` dEURO, and the payer will receive `100` dEURO from the reserves, which is all the remaining reserve
5. The `minterReserveE6` will be reduced by `900 * 20% = 180` dEURO.
6. `900` dEURO will be burnt, leaving the position with `100` principal. The `minterReserveE6` indicates that there should be `20 / 2 = 10` dEURO left in the reserves, but there are none.
7. Optionally, if there are funds available, `_repayPrincipleNet` will burn up to `100 dEURO` from the payer to reduce the principal to `0`. However, the `minterReserveE6` is not updated, and the reserves are still incorrect.

In summary, the `minterReserve` and amount transferred from reserves are incorrect when the reserves are no longer fully present.

Code corrected:

The function `burnFromWithReserveNet()` has been removed in Version 2 and the function `_repayPrincipalNet()` has been refactored to compute the available reserves in line with the functionalities `mint` and `repay`.

6.4 Inconsistent Values Used for Infinite Allowance

Design | Low | Version 1 | Code Corrected

CS-dEUR-005

The function `ERC20._spendAllowance()` interprets the value `type(uint256).max` as infinite allowance and a call to `_approve()` is ommitted. However, the function `allowance()` implemented in DecentralizedEURO uses the value `2**255` for infinite allowances:

```
if (spender == address(reserve)) {
    return 1 << 255;
}

if (
    (isMinter(spender) || isMinter(getPositionParent(spender))) &&
    (isMinter(owner) || positions[owner] != address(0) || owner == address(reserve))
) {
    return 1 << 255;
}
```

Since `2**255` is different from `type(uint256).max`, the function `_spendAllowance()` will actually call `_approve()` which writes a new allowance in the storage and emits an event.

Code corrected:

The function `allowance()` in DecentralizedEURO has been revised to be in line with `ERC20._spendAllowance()`. Both functions now interpret the value `type(uint256).max` as infinite approval.

6.5 Incorrect Code Comments

Design | Low | Version 1 | Specification Changed

CS-dEUR-006

- The comment in PositionRoller states that the `repay` param is the amount to flash loan. However, the `repay` should actually be the `principal` to repay. The flash loan amount will be `repay + interest`.
- The comment in FrontEndGateway describing the feeRate states that `10 = 1%` in PPM, but actually `10'000 = 1%`.
- The inline code comment in the function `_calculateShares()` states the initial deposit mints `1_000_000` shares, but the code actually mints `10_000_000`.
- The comment in Equity states: `the supply is proportional to the cubic root of the reserve` but the fifth root is actually used.

- The comment in function `forceSale()` states: "Proceeds are used to repay 'principal' and if any remains, the 'interest'". This comment can be misleading as the implementation first repays `propInterest`, then the principal is paid.

Specification changed:

All comments have been updated to reflect the implementation. The last one is no longer relevant as the code has been updated.

6.6 Max Interest Check Is Ineffective

Design | Low | Version 1 | Specification Changed

CS-dEUR-008

In MintingHub, the `bid()` function has a param `maxInterest` where the bidder can specify a maximum interest amount they are willing to repay when bidding.

However, this parameter does not appear to be very useful, as the interest rate is known in advance, so the interest is predictable. The interest paid per collateral cannot be increased by being frontrun by another actor, so it is unclear in which situation the bidder would want to specify a maximum interest amount.

If the bidder does want to rely on the check, it can be circumvented. They may be frontrun by another bidder making a partial liquidation. If this reduces the collateral available, they can receive less collateral than expected but still have the same `maxInterest`. This effectively increases the interest they are willing to pay per collateral.

Specification changed:

In Version 2 the interest is no longer paid separately from the principal. The `maxInterest` parameter has been removed.

6.7 Zero-value Parameter Always Passed to mintWithReserve

Design | Low | Version 1 | Code Corrected

CS-dEUR-009

The function `mintWithReserve()` is called only when minting dEuro from a position and the value for `_feesPPM` is always set to `0`. Therefore, the event `Profit` is always emitted with the field `amount` set to `0`.

Code corrected:

The unused parameter has been removed from the function `mintWithReserve()`.

6.8 Integer Division Before Multiplication

Informational | Version 1 | Code Corrected

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.

Code corrected:

The code has been updated to perform multiplication before division.

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 Frontend Gateway Supports One Hub

Informational **Version 1**

CS-dEUR-023

The current implementation of FrontendGateway supports only one MintingHub. If a new MintingHub is deployed in the future, a new frontend gateway needs to be deployed.

7.2 Interest Rate Can Overflow

Informational **Version 1**

CS-dEUR-014

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

```
function _fixRateToLeadrate(uint24 _riskPremiumPPM) internal {
    fixedAnnualRatePPM = IMintingHub(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.

7.3 Liquidations Send Rewards to Position's Frontend

Informational **Version 1**

CS-dEUR-010

When a position is liquidated, either via normal challenges or force sales, a call to `_notifyInterestPaid()` is triggered. This function calls into gateway to record the paid interest and reward the frontend used to open the position. The reward is paid even if the position caused bad debt in the system.

7.4 Non-indexed Events

Informational **Version 1**

CS-dEUR-024

Several events declared in `IFrontendGateway` do not index any event parameters. They could index the relevant event parameters to allow integrators and dApps to quickly search for these and simplify UIs.

7.5 Owner Can Bid at Any Expired Collateral Price

Informational **Version 1**

CS-dEUR-015

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 if no challenges are ongoing, 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.

7.6 Possible Loss of Precision in StablecoinBridge

Informational **Version 1**

CS-dEUR-016

The function `StablecoinBridge.mintTo()` transfers from the caller the full `amount` passed as argument. However, `targetAmount` of dEURO is minted to the recipient. Theoretically, if the external token uses more decimals than dEURO, `targetAmount` might be slightly lower than `amount` due to loss of precision.

7.7 Possible to Claim Frontend Codes in a New FrontendGateway

Informational **Version 1**

CS-dEUR-017

The contract `FrontendGateway` allows anyone to claim free frontend codes. If another gateway with the same functionality is deployed in the future, an attacker can claim the existing frontend codes. Thus, frontend providers should consider that they might not own the same code across all gateways.

7.8 Rate Change Limitation on FrontendGateway

Informational **Version 1**

CS-dEUR-018

The FrontendGateway contract has three different rates (`feeRate`, `savingsFeeRate`, and `mintingFeeRate`) that can be changed by the governance. Proposals are subject to a delay of 7 days. The contract uses a single variable `changeTimeLock` to store when a proposal for a fee rate is applied. If a proposal for one rate is pending (e.g., `feeRate`) and a new proposal for another rate is published (e.g., `savingsFeeRate`), the clock is reset for both proposals.

7.9 Rounding Errors in Kamikaze Function

Informational Version 1

CS-dEUR-019

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.

7.10 Unrealized Interest Not Considered in Equity

Informational Version 1

CS-dEUR-020

The interest earned by depositing dEURO 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 dEuro 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 nDEPS token. As nDEPS tokens are cheaper when there is less equity in the system it could be favourable for nDEPS 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 nDEPS price will be slightly higher than it should. Anyone using `redeem()` to turn their nDEPS into dEuro will receive a slightly higher rate than they should. However, it seems unlikely that the price difference will be more than the 2% 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.

Additionally, outstanding interest that is expected to be paid by positions when they are closed is also not considered. When an interest payment is made, it will be accounted as a profit towards the equity, which can suddenly increase the nDEPS price.

7.11 Gas Optimizations

Informational Version 1 Code Partially Corrected

CS-dEUR-011

We provide a non-exchaustive list of optimizations to make the codebase more efficient in terms of gas:

1. The function `MintingHubGateway.clone()` uses the modifier `validPos` which is redundant due to `MintingHub.clone()` executing the same modifier.

2. The function `Position.initialize()` performs an external call to get parent's `riskPremiumPPM` although the variable is immutable.
3. The function `FrontendGateway.registerPosition()` could return early if a zero code is provided.

Version 2:

4. The function `_collateralBalance()` is executed multiple times when the position's price is adjusted.
5. The function `getCollateralRequirement()` can be marked as `external`.
6. The function `_repayPrincipalNet()` performs redundant SLOADs when reading `reserveContribution`.

Code partially corrected:

The optimizations described in points 1 and 3 have been implemented.

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 Custom Logic Required to Protect Against Slippage

Note Version 2

Users calling `buyExpiredCollateral()` should be aware of the behavior described in no slippage protection in buyExpiredCollateral and may want to implement their own slippage protection logic, to prevent buying at a higher price than expected. Calling the function through a dark mempool such as Flashbots Protect can also help reduce the chances of this problem arising, as it prevents frontrunning.

8.2 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.

8.3 Factors Influencing System Profitability

Note Version 1

The system's equity has ongoing revenues and costs:

- Revenues: Leadrate + risk premium interest paid by positions.
- Costs: Interest paid to savers + referral rewards

Additionally, it has one-off revenues and costs:

- Revenues: Profitable liquidations + fees paid to propose positions and minters
- Costs: Bad debt from unprofitable liquidations

The profitability of the system depends on the balance between these revenues and costs. As only a part of all existing dEUROs can be deposited into the savings module at any time (some will always be in the minter reserve and in the equity), dEUROs minted through positions should generally generate more interest than is paid to savers of those coins. However, the referrals introduced with the FrontendGateway are paid from the system's equity in addition to the savings. The savings referreals increase the savings paid, while the position referrals reduce the position interest earned. This means they reduce the profitability of the system. If the referral rewards are too high, the system may become unprofitable, shrinking the equity buffer over time.

Additionally, the stablecoin bridge can mint dEUROs that do not generate any revenue for the system, but can still earn interest in the savings module. This is an additional cost to the equity. As a result, it is important that the bridge limits are set in such a way that the ratio of outstanding dEUROs from the bridges compared to the amount created through the MintingHub is not too high. In case this ratio does become too high, it will likely require reducing the leadrate paid to savers.

Governance must ensure that the parameters governing the system's profitability, such as the referral rewards, the leadrate, risk premiums for positions, and bridge limits, are balanced in such a way that the system does not become too unprofitable over long periods of time. The equity buffer must remain large enough to backstop any liquidations that may incur bad debt, otherwise the overcollateralization and peg stability of the dEURO may be at risk.

nDEPS holders are economically incentivized to set parameters that maximize the system's equity over the duration they hold their nDEPS.

8.4 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. The interest of a position is fixed when minting but can be paid till expiry. If the interest rate is increased, it is possible for savers to receive more interest than was paid by the positions.

8.5 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.

8.6 Integrations Must Check Withdraw Return Value

Note Version 1

The function `Savings.withdraw()` takes an `amount` parameter. If this amount is larger than the saved amount, the full balance is withdrawn instead, and the actual 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.

8.7 Market Risk Taken by Challengers

Note Version 1

The auction process takes some time depending on the `challengePeriod` of the position. During this time, the price of the collateral could change, hence challengers face market risks until the auction completes.

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.

8.8 Minimum Collateral Is Never Adjusted

Note Version 1

The `minimumCollateral` for a position is immutable. On creation, it is enforced that the minimum is `> 5000` dEURO. 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. dEURO limit is still large.
3. Someone clones the position many times using minCollateralAmount (which is significantly under `5000` dEURO). 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`.

8.9 Positions With Quickly Collapsing Collateral Must Be Challenged

Note Version 1

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` dEURO which are undercolalteralized and will lead to a loss to the system.

This situation can be avoided if there is an attentive nDEPS 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, nDEPS 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 nDEPS Equity).

8.10 Potential DoS Attack Against Positions

Note Version 2

An attacker who controls two consecutive blocks could potentially place any healthy position into a 1-day cooldown period. This can be done by initiating a challenge at block *i* and then averting the challenge at block *i* + *1*. While a position is in cooldown, the owner cannot mint new tokens, adjust the liquidation price, or withdraw collateral. Additionally, a position on cooldown cannot be cloned. The same can be done if the attacker does not control consecutive blocks. However, they will need to compete for averting the challenge, will cause them significant costs.

8.11 Rolling a Position Compounds Interest

Note Version 2

The function `rollFullyWithExpiration()` rolls an old position into a new one and converts the whole debt (principal + interest) into principal to the new position. This conversion causes a one-time compounding of the interest in the new position. Hence, the new position pays interest on both principal and interest from the old position.

8.12 restructureCapTable May Take Multiple Blocks

Note **Version 1**

If the system has incurred losses and equity is less than 1000 dEURO, the function `restructureCapTable` allows existing shareholders to restructure the system. Any shareholder with more than 2% of the voting power can step in and bootstrap the system by paying for the losses (which can be significantly higher than 1000 dEURO) and become the only nDEPS shareholder. As the function `restructureCapTable` iterates through all nDEPS 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.