decurity

Smart Contract Security Audit Report

FrankenCoin Audit

1. Contents

2. General Information

This report contains information about the results of the security audit of the FrankenCoin (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 16/09/2024 to 25/09/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: Frankencoin-ZCHF/FrankenCoin. Initial review was done for the commit a6db51, with re-testing performed on commit 667267.

The following contracts have been tested:

- contracts/Leadrate.sol
- contracts/MintingHub.sol
- contracts/Position.sol
- contracts/PositionFactory.sol
- contracts/PositionRoller.sol
- contracts/Savings.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Liquidity Token owner/contract.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	**Actor**
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a re-entrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided "as is" and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer's project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have discovered a several critical exploitable security issue as well as high, medium and low-risk issues. These vulnerabilities have been addressed and thoroughly re-tested as part of our process.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of October 22, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
The `buyExpiredCollateral` can be front run	contracts/MintingHub.sol contracts/Position.sol	**Critical**	Fixed
Initializer can be called more than one time	contracts/Position.sol	**Critical**	Fixed
Small challenge time can be abused	contracts/MintingHub.sol	**Critical**	Acknowledged

Issue	Contract	Risk Level	Status
Reward may not be received	contracts/Savings.sol	**High**	Acknowledged
Withdrawing of saving can be blocked	contracts/Savings.sol	**High**	Fixed
Funds can be stolen with a small challenge time	contracts/MintingHub.sol	**High**	Acknowledged
The `expectedPrice` can be abused	contracts/MintingHub.sol	**Medium**	Fixed
Funds may be locked on the contract for more than two weeks	contracts/Savings.sol	**Medium**	Fixed
SafeERC20 should be used		**Medium**	Fixed
minimumCollateral check during roll is not present	contracts/PositionRoller.sol	**Medium**	Fixed
Unnecessary calls to original contract	contracts/Position.sol	**Low**	Fixed
Position without minCollateral may be created	contracts/MintingHub.sol	**Low**	Fixed
Fees are not checked	contracts/MintingHub.sol	**Low**	Acknowledged
Precision loss	contracts/MintingHub.sol	**Low**	Acknowledged
Unnecessary modifier	contracts/MintingHub.sol	**Low**	Fixed

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. The `buyExpiredCollateral` can be front run

Risk Level: **Critical**

Status: Fixed in the commit 502f786.

Contracts:

- contracts/MintingHub.sol
- contracts/Position.sol

Description:

The MintingHub offers a functionality for buying the expired collateral from position though the `buyExpiredCollateral()` that calculates the needed amount of the zchf tokens that should be withdrawn from the buyer though the `expiredPurchasePrice()` function.

The `expiredPurchasePrice()` function accepts the position address and uses it's current price and expiration to calculate the current collateral price in zchf tokens. The price that was calculated will be used to calculate the final amount of the tokens that should be burned and withdrawn from the buyer during the `forceSale()` call to the given position.

Because the excess funds will be sent to the owner of the position it opens the opportunity to frontrun the call of the buyer and manipulate the price of the position.

```
    function forceSale(address buyer, uint256 collAmount, uint256 proceeds)
external onlyHub expired {
        if (minted > 0) {
            uint256 availableReserve = zchf.calculateAssignedReserve(minted,
reserveContribution);
            if (proceeds + availableReserve >= minted) {
                // we can repay everything
                uint256 returnedReserve = zchf.burnFromWithReserve(buyer,
minted, reserveContribution);
                assert(returnedReserve == availableReserve);
                zchf.transferFrom(buyer, owner, proceeds + returnedReserve -
minted);
                _notifyRepaid(minted);
            } else {
                // we can only repay a part
                zchf.transferFrom(buyer, address(this), proceeds);
                uint256 repaid = zchf.burnWithReserve(proceeds,
reserveContribution);
                _notifyRepaid(repaid);
                // nothing to return to the owner
            }
        } else {
            // wire funds directly to owner
            zchf.transferFrom(buyer, owner, proceeds);
        }
```

The position has infinite allowance, so the amount of tokens limited by the buyer balance and by the check in the `_setPrice()`:

```
require(newPrice * minimumCollateral <= bounds * ONE_DEC18); // sanity check
```

Remediation:

Consider not allowing to change the price after the position has expired.

5.2. Initializer can be called more than one time

Risk Level: **Critical**

Status: Fixed in the commits 67cec26 and 3498f08.

Contracts:

- contracts/Position.sol

Description:

`Position` contract has `initialize`, that has the only modifier `onlyOwner`. Initially this function should be called only once, BUT it does not.

```
function initialize(address owner, Position parent, uint40 _expiration)
external onlyOwner {
    if (_expiration > Position(original).expiration()) revert
ExpirationAfterOriginal(); // original might have later expiration than
parent, which is ok
    expiration = _expiration;
    _setOwner(owner);
    price = parent.price();
}
```

The Position contract has an `initialize()` function that is expected to be called only once. However, due to an issue with the custom Ownable contract, this function can be called multiple times, which can lead to a variety of security vulnerabilities.

The problem stems from the `_setOwner()` function, where the comment states that initialization should only happen once, but the check is not correctly implemented:

```
function _setOwner(address newOwner) internal {
    require(newOwner != address(0x0)); // ensure initialization can only done
once
//@audit-issue comment states that init should be called only once, so check
should be set as "oldOwner != address(0x0)"
    address oldOwner = owner;
    owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

The issue lies in the fact that there is no check to prevent the owner from being set multiple times, which allows the `initialize()` function to be called more than once.

Impact of such a scenario is very wide, but here is one of the cases of impact:

If an attacker calls `initialize()` multiple times, they can pass arbitrary contracts as the Position parent. This allows them to manipulate the price variable because the line `price = parent.price();` sets the price based on the parent contract's price. By passing a malicious parent contract, the attacker can set an inflated price, which can then be used to mint an excessive amount of ZCHF tokens by exploiting the Position contract's `mint()` function.

Remediation:

Consider changing the `onlyOwner()` to `onlyHub()` modifier. It will not allow to call `initialize()` multiple times or even 1 time if the position is the original and was not deployed as a clone.

5.3. Small challenge time can be abused

Risk Level: Critical

Status: The assumption is that positions with unreasonable parameters will not pass the governance process. Furthermore, this attack would be very risky for the attacker. If the attacker gets frontrun by another attacker, they will lose their collateral to whoever comes first.

Contracts:

- contracts/MintingHub.sol

Description:

When the position expires, the value of its collateral begins to decrease. After the 2 * challengePeriod, the position can be redeemed for free.

```
function expiredPurchasePrice(IPosition pos) public view returns (uint256) {
    uint256 liqprice = pos.price();
    uint256 expiration = pos.expiration();
    if (block.timestamp <= expiration) {
        return EXPIRED_PRICE_FACTOR * liqprice;
    } else {
        uint256 challengePeriod = pos.challengePeriod();
        uint256 timePassed = block.timestamp - expiration;
        if (timePassed <= challengePeriod) {
            // from 10x liquidation price to 1x in first phase
            uint256 timeLeft = challengePeriod - timePassed;
            return liqprice + (((EXPIRED_PRICE_FACTOR - 1) * liqprice) /
challengePeriod) * timeLeft;
        } else if (timePassed < 2 * challengePeriod) {
            // from 1x liquidation price to 0 in second phase
            uint256 timeLeft = 2 * challengePeriod - timePassed;
            return (liqprice / challengePeriod) * timeLeft;
        } else {
            // get collateral for free after both phases passed
            return 0;
        }
    }
}

/**
 * Buys the desired amount of the collateral asset from the given expired
position using
 * the applicable 'expiredPurchasePrice' in that instant.
 */
function buyExpiredCollateral(IPosition pos, uint256 amount) external {
    uint256 forceSalePrice = expiredPurchasePrice(pos);
    uint256 costs = (forceSalePrice * amount) / 10 ** 18;
    pos.forceSale(msg.sender, amount, costs);
    emit ForcedSale(address(pos), amount, forceSalePrice);
}
```

This can be exploited as follows:

1. The attacker creates a position with a small challengeTime (about 30 seconds).

2. If someone opens a challenge to a position, the attacker backruns the transaction to the next block and averts the challenge. The attacker prevents the execution of the `notifyChallengeSucceeded()` function.

```
function notifyChallengeSucceeded(
    address _bidder,
    uint256 _size
) external onlyHub returns (address, uint256, uint256, uint32) {
    challengedAmount -= _size;
    uint256 colBal = _collateralBalance();
    if (colBal < _size) {
        _size = colBal;
    }
    uint256 repayment = colBal == 0 ? 0 : (minted * _size) / colBal; // for
enormous colBal, this could be rounded to 0, which is ok
    _notifyRepaid(repayment); // we assume the caller takes care of the actual
repayment

    // Give time for additional challenges before the owner can mint again. In
particular,
    // the owner might have added collateral only seconds before the challenge
ended, preventing a close.
    _restrictMinting(3 days);

    uint256 newBalance = _sendCollateral(_bidder, _size); // transfer
collateral to the bidder and emit update

    emit MintingUpdate(newBalance, price, minted);

    return (owner, _size, repayment, reserveContribution);
}
```

3. After 3 days, the position will be able to mint tokens.
4. The attacker creates a clone of this position with almost expired time and mint `_initialMint` with `_initialCollateral`.
5. After 60 seconds (2*`challengeTime`), an attacker can redeem the collaterals for free and repeat the exploit.

```

function clone(address owner, address parent, uint256 _initialCollateral,
uint256 _initialMint, uint40 expiration) public validPos(parent) returns
(address) {
        // @audit-issue we can create an expired position
        address pos = POSITION_FACTORY.clonePosition(parent, expiration);
        zchf.registerPosition(pos);
        IPosition child = IPosition(pos);
        IERC20 collateral = child.collateral();
        collateral.transferFrom(msg.sender, pos, _initialCollateral); //
collateral must still come from sender for security
        emit PositionOpened(owner, address(pos), parent, address(collateral));
        child.mint(owner, _initialMint);
        Ownable(address(child)).transferOwnership(owner);
        return address(pos);
    }
```

6. The attacker will be able to mint `limit` from the origin position tokens for free.

Remediation:

Consider enforcing a minimum challenge time.

5.4. Reward may not be received

Risk Level: High

Status: The assumption is that there normally is enough equity capital to pay out interest on savings. In cases where equity capital is depleted, it is ok to make savers forfeit their accumulated interests.

Contracts:

- contracts/Savings.sol

Description:

The issue in the Savings contract's `refresh()` function lies in how rewards are handled when the contract does not have enough funds to pay out the interest earned by users. If the difference between ticks is large, and the equity contract does not have enough ZCHF tokens to cover the reward, users will lose all their collected rewards.

```
function refresh(address accountOwner) internal returns (Account storage) {
        Account storage account = savings[accountOwner];
        uint64 ticks = currentTicks();
        if (ticks > account.ticks) {
            uint192 earnedInterest = uint192((uint256(ticks - account.ticks) *
account.saved) / 1000000 / 365 days);
            if (earnedInterest > 0 && zchf.balanceOf(address(equity)) >=
earnedInterest) {
                //@audit user will not recieve any rewards? if not enough
funds
                zchf.transferFrom(address(equity), address(this),
earnedInterest); // collect interest as you go
                account.saved += earnedInterest;
            }
            account.ticks = ticks;
        }
        return account;
    }
```

If a frontrunner calls `refresh()` and drains the reward pool, users calling refresh afterward will lose their pending rewards since the `account.ticks` is updated regardless of whether the user receives any rewards. Even if the reward pool is replenished later, users who missed out due to insufficient funds will not be able to claim their lost rewards, as their `account.ticks` will have already been updated.

Remediation:

Do not update `account.ticks` if there are not enough funds to pay for reward, or update it partially (if contract send a piece of total desired rewards).

5.5. Withdrawing of saving can be blocked

Risk Level: High

Status: Fixed in the commit f42227. The team has accepted the risk of griefing. The withdrawal may be blocked if someone decides to add funds to the user's balance.

Contracts:

- contracts/Savings.sol

Location: Function: `save()`.

Description:

The `save()` function in the Savings contract allows anyone to deposit tokens on behalf of any user, which increments the `balance.ticks` of the recipient's savings account. Additionally, the function can be called with an amount of 0, which still updates the ticks. This creates a vulnerability where an attacker can continuously call `save()` on a victim's account, artificially inflating the ticks and thereby extending the lock period for the victim's funds.

```
    function save(address owner, uint192 amount) public {
        Account storage balance = refresh(owner);
        zchf.transferFrom(msg.sender, address(this), amount);
        uint64 ticks = currentTicks();
        uint64 oldExtraTicks = ticks >= balance.ticks ? 0 : balance.ticks -
ticks;
        uint64 newExtraTicks = uint64(currentRatePPM) * INTEREST_DELAY;
        uint64 weightedAverage = uint64(
            (oldExtraTicks * balance.saved + newExtraTicks * amount) /
(balance.saved + amount)
        );
        balance.saved += amount;
        balance.ticks += weightedAverage;
    }
```

In the `withdraw()` function, users are only allowed to withdraw their funds after a certain "timelock" period, determined by `account.ticks > currentTicks()`. However, due to this vulnerability, an attacker can manipulate the lock duration by calling `save()` multiple times, locking the victim's funds for an extended period.

```
function withdraw(address target, uint192 amount) external returns (uint256) {
        Account storage account = refresh(msg.sender);
        //@audit-issue greefing
        if (account.ticks > currentTicks()) {
            //@audit-issue if ticks rate will insrease while whose 14 days,
user will be able to withdraw in less than 14 days
            // or if rate will go down, user will lock funds for longer thatn
14 days.
            revert FundsLocked(uint40(account.ticks - currentTicks() /
currentRatePPM));
        } else if (amount >= account.saved) {
            amount = account.saved;
            delete savings[msg.sender];
        } else {
            account.saved -= amount;
        }
        zchf.transfer(target, amount);
        return amount;
    }
```

If `save()` will be called in the same block after the victim made his "deposit" the `weightedAverage` will be equal to the `oldExtraTicks` which was equal to the `uint64(currentRatePPM) * INTEREST_DELAY` after first call.

```
weightedAverage = (oldExtraTicks * balance.saved + newExtraTicks * amount) /
(balance.saved + amount)
weightedAverage = oldExtraTicks * balance.saved / balance.saved =
oldExtraTicks
```

The next call after this call will be increase the saved ticks twice. Then four times. So the ticks value will be growing exponentially.

Remediation:

To mitigate this vulnerability, consider restricting the `save()` function so that users can only deposit to their own savings account, preventing external manipulation of their ticks. Additionally, you can introduce a minimum amount requirement for the `save()` function to ensure meaningful deposits.

5.6. Funds can be stolen with a small challenge time

Risk Level: High

Status: This error can only happen if the victim chooses an unreasonably short challenge time. The resolution is to enforce reasonable values in the frontend. Since this value is subject to changes over time, declining as the ecosystem gets more professional, it would be difficult to hardcode a minimal value in the smart contracts today.

Contracts:

- contracts/MintingHub.sol

Description:

When opening a position, you can specify any value of challenge seconds:

```
function openPosition(
      address _collateralAddress,
      uint256 _minCollateral,
      uint256 _initialCollateral,
      uint256 _mintingMaximum,
      uint40 _initPeriodSeconds,
      uint40 _expirationSeconds,
      uint40 _challengeSeconds,
      uint24 _riskPremium,
      uint256 _liqPrice,
      uint24 _reservePPM
  ) public returns (address) {
      require(_riskPremium <= 1000000);
      require(CHALLENGER_REWARD <= _reservePPM && _reservePPM <= 1000000);
      require(IERC20(_collateralAddress).decimals() <= 24); // leaves 12
digits for price
      require(_initialCollateral >= _minCollateral, "must start with min
col");
```

```
      require(_minCollateral * _liqPrice >= 5000 ether * 10 ** 18); // must
start with at least 5000 ZCHF worth of collateral
      IPosition pos = IPosition(
          POSITION_FACTORY.createNewPosition(
              msg.sender,
              address(zchf),
              _collateralAddress,
              _minCollateral,
              _mintingMaximum,
              _initPeriodSeconds,
              _expirationSeconds,
              _challengeSeconds, // @audit arbitrary challenge time can be
setted
              _riskPremium,
              _liqPrice,
              _reservePPM
          )
      );
```

Immediately after opening a position, a challenge may come to it.

If the time of the challenge (phase) is 0 or small enough, the challenge will be considered completed.

```
function bid(uint32 _challengeNumber, uint256 size, bool
postponeCollateralReturn) external {
        Challenge memory _challenge = challenges[_challengeNumber];
        (uint256 liqPrice, uint40 phase) =
_challenge.position.challengeData();
        size = _challenge.size < size ? _challenge.size : size; // cannot bid
for more than the size of the challenge

        if (block.timestamp <= _challenge.start + phase) {
            _avertChallenge(_challenge, _challengeNumber, liqPrice, size);
            emit ChallengeAverted(address(_challenge.position),
_challengeNumber, size);
        } else {
            _returnChallengerCollateral(_challenge, _challengeNumber, size,
postponeCollateralReturn);
            // @audit-issue after 2*challengeTime from the start, the price
will be reset to zero
            (uint256 transferredCollateral, uint256 offer) =
_finishChallenge(_challenge, liqPrice, phase, size);
            emit ChallengeSucceeded(address(_challenge.position),
_challengeNumber, offer, transferredCollateral, size);
        }
    }
```

As a result, the challenger will receive all of the collateral for free.

Remediation:

Consider enforcing a minimum challenge time.

5.7. The `expectedPrice` can be abused

Risk Level: Medium

Status: Fixed in the commit c38c1c24.

Contracts:

- contracts/MintingHub.sol

Description:

The `challenge()` function checks if the current position price matches the expected price. However, an attacker can front-run this transaction and change the position's price by a small amount (e.g., 1 wei lower), causing the challenge to revert with the `UnexpectedPrice()` error.

```
function challenge(
        address _positionAddr,
        uint256 _collateralAmount,
        uint256 expectedPrice
    ) external validPos(_positionAddr) returns (uint256) {
        IPosition position = IPosition(_positionAddr);
        if (position.price() != expectedPrice) revert UnexpectedPrice();
```

At the same time, the minting of tokens will not be blocked, since the price is lowered.

```
function _adjustPrice(uint256 newPrice) internal noChallenge {
    if (newPrice > price) {
        _restrictMinting(3 days);
    } else {
        _checkCollateral(_collateralBalance(), newPrice);
    }
    _setPrice(newPrice, minted + availableForMinting());
}
```

It is possible to set a very high price for a position, which will be limited only by a sanity check. Revert all challenges for 3 days and get the opportunity to mint tokens at a high price.

```
function _setPrice(uint256 newPrice, uint256 bounds) internal {
      require(newPrice * minimumCollateral <= bounds * ONE_DEC18); // sanity
check
      price = newPrice;
  }
```

Remediation:

Consider introducing a timeout mechanism that prevent the price from being changed multiple times within a short period.

5.8. Funds may be locked on the contract for more than two weeks

Risk Level: Medium

Status: Fixed in the commit 667267.

Contracts:

- contracts/Savings.sol

Description:

In the Savings contract, when a user deposits tokens through the `save()` function, the current interest rate (`currentRatePPM`) is multiplied by a constant `INTEREST_DELAY` (14 days) to calculate the lock period for withdrawing tokens. However, if the `currentRatePPM` decreases during the lock period, the user will have to wait longer than the expected 14 days to withdraw funds, as it will take more time for `account.ticks` to become less than `currentTicks()`.

```
contract Savings is Leadrate {
        ...
        function save(address owner, uint192 amount) public {
        Account storage balance = refresh(owner);
        zchf.transferFrom(msg.sender, address(this), amount);
        uint64 ticks = currentTicks();
        uint64 oldExtraTicks = ticks >= balance.ticks ? 0 : balance.ticks -
ticks;
        uint64 newExtraTicks = uint64(currentRatePPM) * INTEREST_DELAY;
        uint64 weightedAverage = uint64(
            (oldExtraTicks * balance.saved + newExtraTicks * amount) /
(balance.saved + amount)
        );
        balance.saved += amount;
        balance.ticks += weightedAverage;
    }
```

```
        ...
    }
```

If the `currentRatePPM` decreases during the lock period, the user is forced to wait longer for their funds to be unlocked. For example, if the `currentRatePPM` is halved, the user might have to wait up to 28 days instead of 14 days, since the ticks accumulate more slowly with a lower interest rate.

Remediation:

Instead of basing the withdrawal lock on `currentRatePPM`, make the lock dependent on a fixed timestamp. This will ensure that the lock duration remains consistent, even if the interest rate changes.

5.9. SafeERC20 should be used

Risk Level: Medium

Status: Generally, it is the responsibility of the governance system to not allow freak tokens as collateral. Nonetheless, we have added a check to ensure that collateral assets fail on invalid transfers.

Description:

The protocol directly interacts with IERC20 functions. Since the IERC20 interface requires a boolean return value, attempting to `transfer()`, `approve()` , and `transferFrom()` ERC20s with missing return values will revert. This means that Frankencoin won't support several popular ERC20s, including USDT and BNB tokens on the Ethereum chain.

Remediation:

Consider using a safe version of the mentioned functions from `SafeERC20` library such as `safeTransfer()`, `safeTransferFrom()`, `forceApprove()` that handles a case when functions may not return values.

References:

- https://github.com/d-xo/weird-erc20

5.10. minimumCollateral check during roll is not present

Risk Level: Medium

Status: Fixed in the commit d712153.

Contracts:

- contracts/PositionRoller.sol

Description:

In the `rollFullyWithExpiration()` function, if the amount available for minting in the target position is less than the repay amount, the `collateralToWithdraw` might end up being less than the current collateral in the source position. If the remaining collateral becomes less than the `minimumCollateral`, calling `withdrawCollateral()` may revert with `InsufficientCollateral()` due to a safeguard against leaving dust amounts in the position.

```
    function _withdrawCollateral(address target, uint256 amount) internal
noChallenge returns (uint256) {
        if (block.timestamp <= cooldown && !isClosed()) revert Hot();
        uint256 balance = _sendCollateral(target, amount);
        _checkCollateral(balance, price);
        if (balance < minimumCollateral && balance > 0) revert
InsufficientCollateral(); // Prevent dust amounts
        return balance;
    }
```

This check prevents dust amounts of collateral from remaining in the position, but in some scenarios, it can also cause the transaction to revert, making it impossible to withdraw the collateral.

Remediation:

To prevent the transaction from reverting due to dust amounts of collateral, consider adjusting the withdrawal amount in `rollFullyWithExpiration()`.

5.11. Unnecessary calls to original contract

Risk Level: Low

Status: Fixed in the commit 087f940.

Contracts:

- contracts/Position.sol

Description:

The `globalLimit()` function unnecessarily calls the original contract to retrieve the `limit`, even though the limit is immutable and does not change. This introduces unnecessary gas costs and complexity.

```
    function globalLimit() external view returns (uint256) { //@audit-issue
it's not necessary to call the original contract. limit is immutable
        if (address(this) == original) {
            return limit;
        } else {
            return Position(original).globalLimit();
        }
    }
```

Remediation:

Since the `limit` is immutable, it can directly return limit without needing to make the external call to the original contract. This will reduce gas usage and improve efficiency.

5.12. Position without minCollateral may be created

Risk Level: Low

Status: Fixed in the commit 103edb9.

Contracts:

- contracts/MintingHub.sol

Description:

When cloning a position, you can specify any owner of the position, as well as an arbitrary value of collateral for the new position.

```
function clone(address owner, address parent, uint256 _initialCollateral,
uint256 _initialMint, uint40 expiration) public validPos(parent) returns
(address) {
    address pos = POSITION_FACTORY.clonePosition(parent, expiration);
    zchf.registerPosition(pos);
    IPosition child = IPosition(pos);
    IERC20 collateral = child.collateral();
    // @audit we can create pos with _initialCollateral < minCollateral
    collateral.transferFrom(msg.sender, pos, _initialCollateral); //
collateral must still come from sender for security
    emit PositionOpened(owner, address(pos), parent, address(collateral));
    child.mint(owner, _initialMint);
    Ownable(address(child)).transferOwnership(owner);
    return address(pos);
}
```

However, there is no check to ensure that the `_initialCollateral` value meets the minimum collateral requirement. This could allow a user to create a position with collateral less than the minimum required, potentially spamming the system with invalid positions.

Remediation:

Consider checking the minimum value of the collateral.

5.13. Fees are not checked

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/MintingHub.sol

Description:

When calculating the fee, the `riskPremiumPPM` value is used as an additional fee.

```
function calculateFee(uint256 exp) public view returns (uint24) {
    uint256 time = block.timestamp < start ? start : block.timestamp;
    uint256 timePassed = exp - time;
    // Time resolution is in the range of minutes for typical interest rates.
    uint256 feePPM = (timePassed * annualInterestPPM()) / 365 days;
    return uint24(feePPM > 1000000 ? 1000000 : feePPM); // fee cannot exceed
100%
}
```

However, it is possible to set it to 0 when creating a position.

```
function openPosition(
        ...
) public returns (address) {
        ...
    IPosition pos = IPosition(
        POSITION_FACTORY.createNewPosition(
            msg.sender,
            address(zchf),
            _collateralAddress,
            _minCollateral,
            _mintingMaximum,
            _initPeriodSeconds,
            _expirationSeconds,
            _challengeSeconds,
            _riskPremium, // @audit it is possible to set 0 and bypass the fee
            _liqPrice,
            _reservePPM
        )
    );
```

Remediation:

Consider adding a restriction on the minimum of the `riskPremiumPPM` value.

5.14. Precision loss

Risk Level: Low

Status: Acknowledged

Contracts:

- contracts/MintingHub.sol

Location: Function: `expiredPurchasePrice` ` `_calculatePrice`.

Description:

The calculation of price in `_calculatePrice()` and `expiredPurchasePrice()` first performs division before multiplication which could lead to a loss of precision due to integer truncation:

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

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

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

Remediation:

To avoid loss of precision, the multiplication should be performed before the division:

```
return liqPrice * timeLeft / phase2 ;

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

return liqprice * timeLeft / challengePeriod;
```

5.15. Unnecessary modifier

Risk Level: Low

Status: Fixed in the commit 7b2381a.

Contracts:

- contracts/MintingHub.sol

Description:

The `clone()` function contains the `validPos()` modifier to validate the passed position, but it calls another `clone()` function that also includes the same `validPos()` modifier. This results in a redundant check, as the `validPos()` modifier is invoked twice for the same logic, causing unnecessary overhead.

```
   function clone(address parent, uint256 _initialCollateral, uint256
_initialMint, uint40 expiration) public validPos(parent) returns (address) {
        return clone(msg.sender, parent, _initialCollateral, _initialMint,
expiration);
   }
```

In this case, the `validPos(parent)` check is applied in both the outer and inner `clone()` function calls, which is redundant.

Remediation:

Remove the `validPos()` modifier from the outer `clone()` function to avoid the redundant check. The inner `clone()` function will still perform the necessary `validPos()` validation, eliminating the redundancy and improving efficiency.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.