BLOCKBITE

AUDIT REPORT

FRANKENCOIN

06.02.2023 – 10.02.2023

Table of Contents

Introduction

The Frankencoin is a collateralized stablecoin that is intended to track the value of the Swiss Franc. It's governance is decentralized, with anyone being able to propose new minting mechanisms and anyone who contributed more than 3% to the stability reserve being able to veto new minting mechanisms.

The audit has taken place from the 06th February 2023 until the 10th February 2023.

Disclaimer: While the review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Scope

In Scope are all Solidity contracts and the test files for said contracts in the git repository https://github.com/Frankencoin-ZCHF/FrankenCoin.

The audit covers the files present at this commit hash 0db42977b331bd25a12e3a6da767b7df18a5d66b.

Fixes until and with the commit 295e802234012983dcce573bf98ced970344ee8c are considered.

Techniques

A comprehensive examination has been performed utilizing Manual Review and Static Analysis techniques. The auditing process pays special attention to the following considerations:

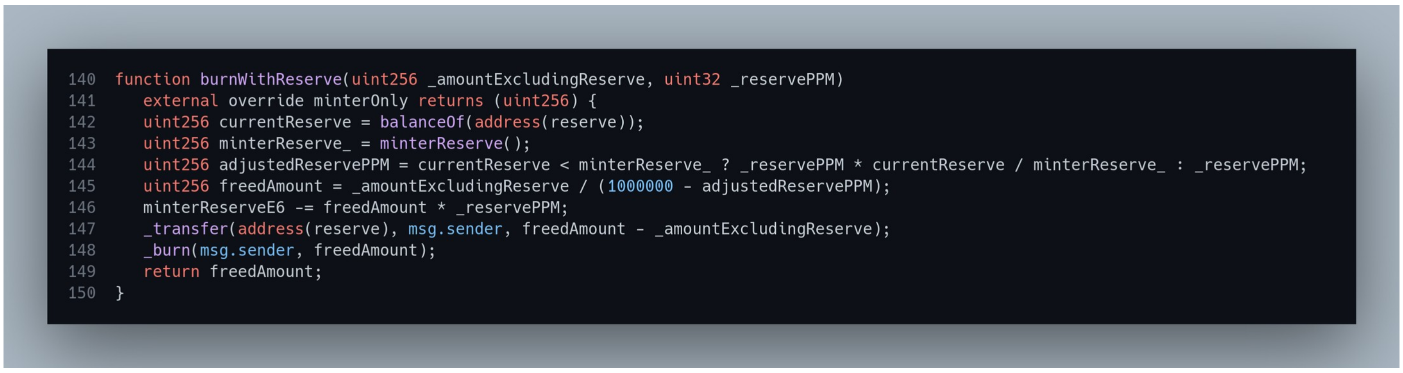
- Testing the smart contracts against both common and uncommon attack vectors
- Assessing the code base to ensure compliance with current best practices and industry standards
- Ensuring contract logic meets the specifications and intentions of the client
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- Thorough line-by-line manual review of the entire code base

Findings

High

H-1 Positions cannot be repaid

Positions use the burnWithReserve() function of the Frankencoin contract to burn ZCHF tokens during their repay routine. Due to a calculation error on line 145 in burnWithReserve() freedAmount is missing 6 decimal places.



```
140 function burnWithReserve(uint256 _amountExcludingReserve, uint32 _reservePPM)
141     external override minterOnly returns (uint256) {
142     uint256 currentReserve = balanceOf(address(reserve));
143     uint256 minterReserve_ = minterReserve();
144     uint256 adjustedReservePPM = currentReserve < minterReserve_ ? _reservePPM * currentReserve / minterReserve_ : _reservePPM;
145     uint256 freedAmount = _amountExcludingReserve / (1000000 - adjustedReservePPM);
146     minterReserveE6 -= freedAmount * _reservePPM;
147     _transfer(address(reserve), msg.sender, freedAmount - _amountExcludingReserve);
148     _burn(msg.sender, freedAmount);
149     return freedAmount;
150 }
```

This error then prevents the subtraction on line 147 from succeeding because it would underflow and therefore revert.

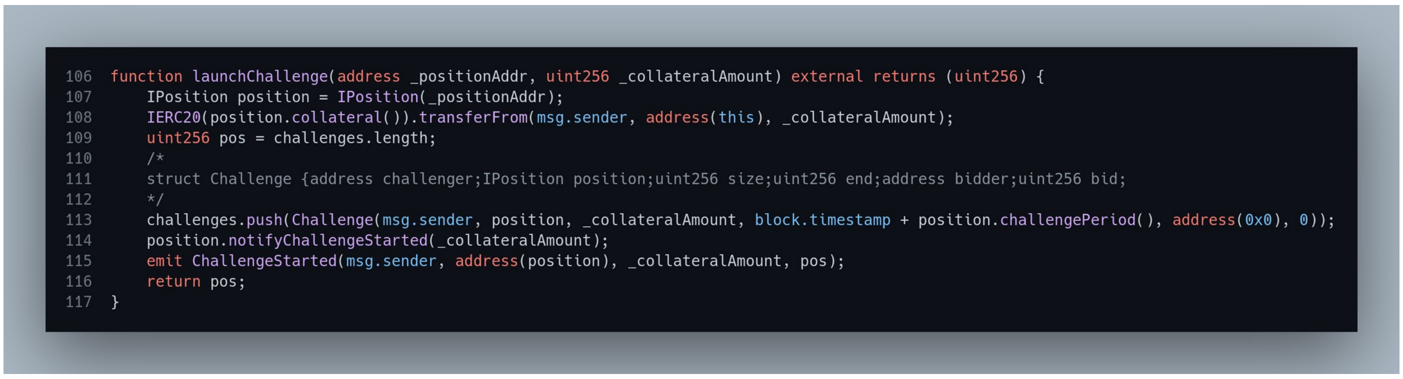
Consider multiplying the result stored in freedAmount with 1’000’000.

Status

Fixed with commit 48062eb360966758153d4369083395c1e271136a

H-2.1 Missing position check leads to unristrcited minting

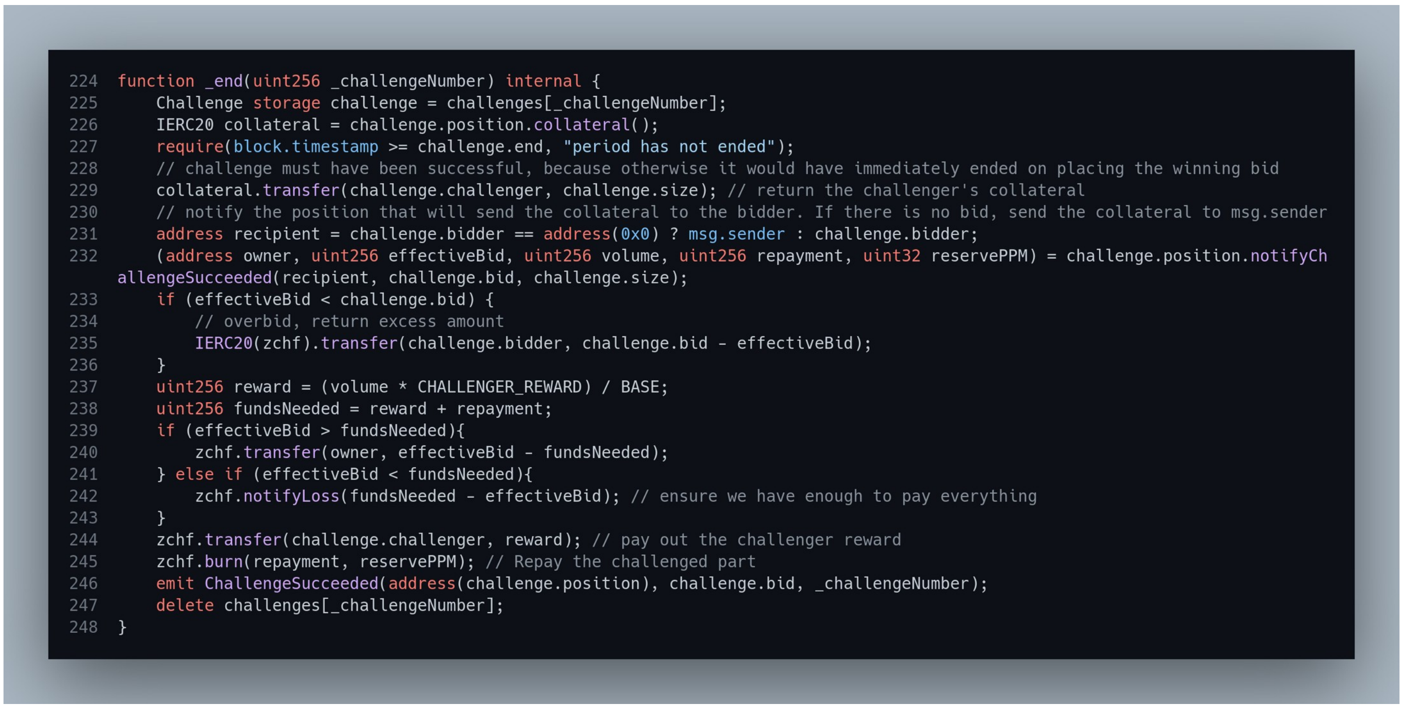
Launching a challenge against a position with launchChallenge() of the MintingHub doesn’t check if the position is registered with the Frankencoin contract nor if the MintingHub is the owner of this position.



```
106 function launchChallenge(address _positionAddr, uint256 _collateralAmount) external returns (uint256) {
107     IPosition position = IPosition(_positionAddr);
108     IERC20(position.collateral()).transferFrom(msg.sender, address(this), _collateralAmount);
109     uint256 pos = challenges.length;
110     /*
111     struct Challenge {address challenger;IPosition position;uint256 size;uint256 end;address bidder;uint256 bid;
112     */
113     challenges.push(Challenge(msg.sender, position, _collateralAmount, block.timestamp + position.challengePeriod(), address(0x0), 0));
114     position.notifyChallengeStarted(_collateralAmount);
115     emit ChallengeStarted(msg.sender, address(position), _collateralAmount, pos);
116     return pos;
117 }
```

This gives the attacker control over the challenge end parameter defined on line 113 but also the collateral used and transferred on line 108. Combined with the greater than or equal check in _end() on line 227 means that a challenge can be started and ended within the same transaction.

Keeping in mind that the attacker has complete control over the return values of the position used in the challenge, the attacker can force the code to land on line 242 within the _end() function with any arbitrary value the attacker wants.



```
224 function _end(uint256 _challengeNumber) internal {
225     Challenge storage challenge = challenges[_challengeNumber];
226     IERC20 collateral = challenge.position.collateral();
227     require(block.timestamp >= challenge.end, "period has not ended");
228     // challenge must have been successful, because otherwise it would have immediately ended on placing the winning bid
229     collateral.transfer(challenge.challenger, challenge.size); // return the challenger's collateral
230     // notify the position that will send the collateral to the bidder. If there is no bid, send the collateral to msg.sender
231     address recipient = challenge.bidder == address(0x0) ? msg.sender : challenge.bidder;
232     (address owner, uint256 effectiveBid, uint256 volume, uint256 repayment, uint32 reservePPM) = challenge.position.notifyChallengeSucceeded(recipient, challenge.bid, challenge.size);
233     if (effectiveBid < challenge.bid) {
234         // overbid, return excess amount
235         IERC20(zchf).transfer(challenge.bidder, challenge.bid - effectiveBid);
236     }
237     uint256 reward = (volume * CHALLENGER_REWARD) / BASE;
238     uint256 fundsNeeded = reward + repayment;
239     if (effectiveBid > fundsNeeded){
240         zchf.transfer(owner, effectiveBid - fundsNeeded);
241     } else if (effectiveBid < fundsNeeded){
242         zchf.notifyLoss(fundsNeeded - effectiveBid); // ensure we have enough to pay everything
243     }
244     zchf.transfer(challenge.challenger, reward); // pay out the challenger reward
245     zchf.burn(repayment, reservePPM); // Repay the challenged part
246     emit ChallengeSucceeded(address(challenge.position), challenge.bid, _challengeNumber);
247     delete challenges[_challengeNumber];
248 }
```

This results in MintingHub calling the Frankencoin contract's notifyLoss() function which transfers ZCHF from the reserves or mints new ZCHF and transfers these to the challenger on line 244.

Consider the following changes to the code base:

- Adding a check that only registered positions which are owned by the MintingHub can be used in challenges
- Doing the calculations of Position.notifyChallengeSucceeded() in MintingHub and use the repay() function to inform the position about the debt change
- Copy on line 225 the challenge into memory (saves gas described in I-3) and move the deletion of the challenge from line 247 to line 226
- Don't allow challenges to be started and ended in the same block

Status

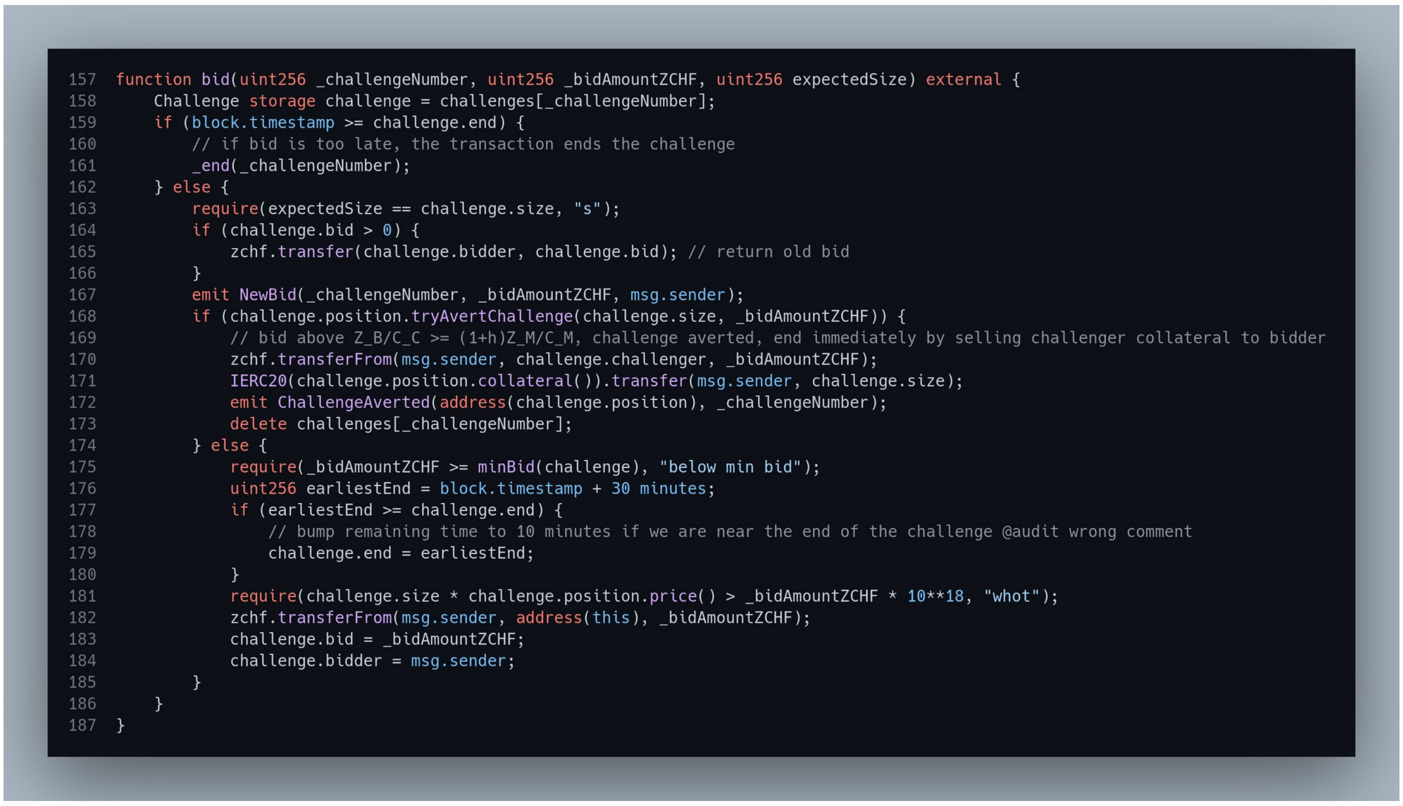
Fixed with commit 5b0659ec39a3e7b67599c16d4eb4d5ca1e83474e.

H-2.2 Missing position check and reentrancy leads to total loss of all challenge funds

The same entrypoint that leads to H-2.1 can also be used to drain all challenge funds stored in the MintingHub.

When a challenge is started or a new bid is placed for a challenge the funds at stake (collateral or ZCHF from bids) are stored in the MintingHub.

So considering that an attacker has full control over a position used in a challenge the attacker can start bidding with the bid() function on the challenge.



```
157 function bid(uint256 _challengeNumber, uint256 _bidAmountZCHF, uint256 expectedSize) external {
158     Challenge storage challenge = challenges[_challengeNumber];
159     if (block.timestamp >= challenge.end) {
160         // if bid is too late, the transaction ends the challenge
161         _end(_challengeNumber);
162     } else {
163         require(expectedSize == challenge.size, "s");
164         if (challenge.bid > 0) {
165             zchf.transfer(challenge.bidder, challenge.bid); // return old bid
166         }
167         emit NewBid(_challengeNumber, _bidAmountZCHF, msg.sender);
168         if (challenge.position.tryAvertChallenge(challenge.size, _bidAmountZCHF)) {
169             // bid above Z_B/C_C >= (1+h)Z_M/C_M, challenge averted, end immediately by selling challenger collateral to bidder
170             zchf.transferFrom(msg.sender, challenge.challenger, _bidAmountZCHF);
171             IERC20(challenge.position.collateral()).transfer(msg.sender, challenge.size);
172             emit ChallengeAverted(address(challenge.position), _challengeNumber);
173             delete challenges[_challengeNumber];
174         } else {
175             require(_bidAmountZCHF >= minBid(challenge), "below min bid");
176             uint256 earliestEnd = block.timestamp + 30 minutes;
177             if (earliestEnd >= challenge.end) {
178                 // bump remaining time to 10 minutes if we are near the end of the challenge @audit wrong comment
179                 challenge.end = earliestEnd;
180             }
181             require(challenge.size * challenge.position.price() > _bidAmountZCHF * 10**18, "whot");
182             zchf.transferFrom(msg.sender, address(this), _bidAmountZCHF);
183             challenge.bid = _bidAmountZCHF;
184             challenge.bidder = msg.sender;
185         }
186     }
187 }
```

The attacker's position can first force the code to follow the positive path of the if on line 168 leading to line 171. Here the position switches the address returned by the collateral() function from some dummy token to something more valuable (e.g. the address of WETH).

Then the MintingHub transfers the returned token with the amount defined in challenge.size to the attacker.

To completely drain the MintingHub the attacker can either redo the steps for each collateral or define a small challenge size and use the reentrancy present in the bid() function. The mentioned reentrancy is caused by a call to the collateral() function of the challenge and after this call to switch the state (deleting the challenge) on line 173.

Consider the following changes to the code base:

- Adding a check that only registered positions which are owned by the MintingHub can be used in challenges
- Copy the challenge into memory (saves gas described in I-3) and move the challenge deletion from line 173 to line 169

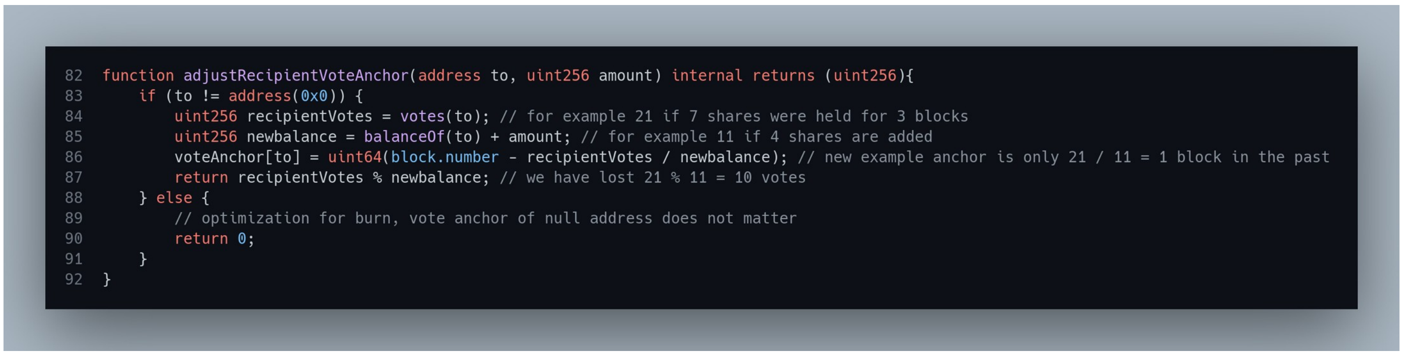
Status

Fixed with commit 5b0659ec39a3e7b67599c16d4eb4d5ca1e83474e.

Medium

M-1.1 Vote reduction in Equity

The Equity contract calls the function adjustRecipientVoteAnchor() before each transfer of it's FPS token to adjust the voteAnchor for the recipient.



```
function adjustRecipientVoteAnchor(address to, uint256 amount) internal returns (uint256){
    if (to != address(0x0)) {
        uint256 recipientVotes = votes(to); // for example 21 if 7 shares were held for 3 blocks
        uint256 newbalance = balanceOf(to) + amount; // for example 11 if 4 shares are added
        voteAnchor[to] = uint64(block.number - recipientVotes / newbalance); // new example anchor is only 21 / 11 = 1 block in the past
        return recipientVotes % newbalance; // we have lost 21 % 11 = 10 votes
    } else {
        // optimization for burn, vote anchor of null address does not matter
        return 0;
    }
}
```

Adjusting the voteAnchor on each transfer make not only transfers more costly but also allows the attacker to control the anchor set for another user.

This makes it possible for an attacker to reduce the amount of votes a specific user has and impacts the possiblity for a user to veto.

In tests performed during the audit we were able with only 200 wei of ZCHF and the gas costs associated with a loop looping 200 times to move the anchor by 200 blocks.

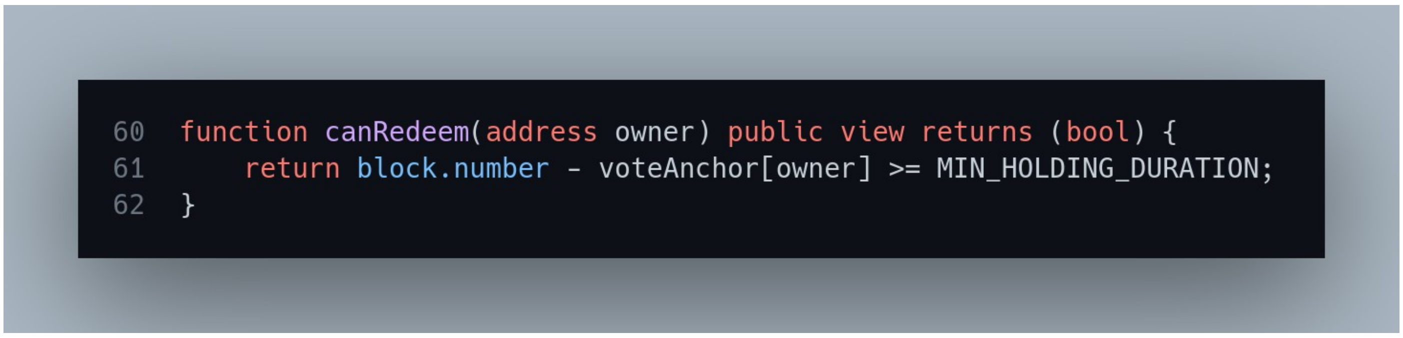
Consider to go with either one token = one vote or something similar to Curve where tokens are locked to receive voting power (https://resources.curve.fi/governance/vote-locking-boost) and give users with less funds more votes.

Status

Fixed with commit 7c3701c6186f41bac62759a30b6c7c19844f9cea.

M-1.2 Griefing by preventing users to redeem

The same problem mentioned in M-1.1 where an attacker can move the voteAnchor also poses an issue in canRedeem().



```
function canRedeem(address owner) public view returns (bool) {
    return block.number - voteAnchor[owner] >= MIN_HOLDING_DURATION;
}
```

In this function the voteAnchor is used to determine if a user is allowed to redeem their locked ZCHF. Allowing an attacker to move the voteAnchor also allows the attacker to prevent a user from redeeming the locked tokens.

Consider having another mapping to store the first time a user locks tokens and check the minimum holding duration with this timestamp.

Status

Fixed with commit 7c3701c6186f41bac62759a30b6c7c19844f9cea.

M-2 Position locked because of stuck challenge

Tokens like USDT have a blacklist which prohibit certain addresses to be the source or the recipient of transfers. In case the challenger ends up on the blacklist between creating the challenge and the challenge end, the challenge cannot be ended.

Because the transfer of the collateral on line 229 from the MintingHub to the challenger reverts.

This leaves not only the challenge in a stuck state but also locks the Position which is used by the challenge. The reason for this is because a Position cannot adjust the price, mint, repay the debt nor withdraw the collateral while a challenge is active. Leaving the challenger, the bidder and the position creator as a victim.

Consider adding a timeout to a challenge which is defined after the end where a locked challenge can be considered failed and deleted while the collateral and bid is consider as a win to the system.

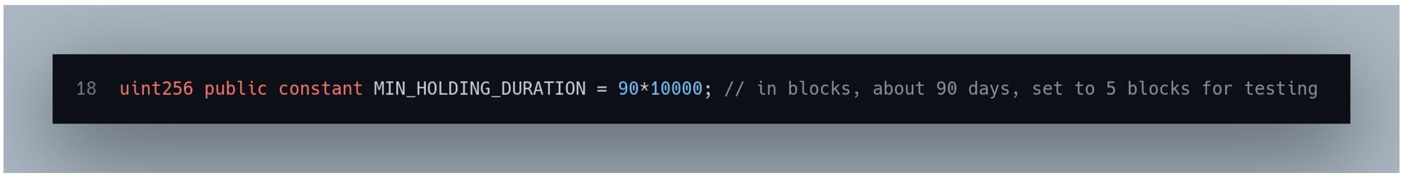
Status

Fixed with commit fcf81e82aaf73ad00747bc5e80dba0245e1c2060.

Low

L-1 MIN_HOLDING_DURATION doesn't match tests and comments

MIN_HOLDING_DURATION is currently set to 90 * 10000. The constant is later used to compare with the current block.number and decides if the user hold their stake long enough.



```
uint256 public constant MIN_HOLDING_DURATION = 90*10000; // in blocks, about 90 days, set to 5 blocks for testing
```

Taking an average block time of around 12 seconds (Source: https://ethereum.org/en/developers/docs/blocks/#block-time) this results in around 125 days instead of the mentioned 90 days.

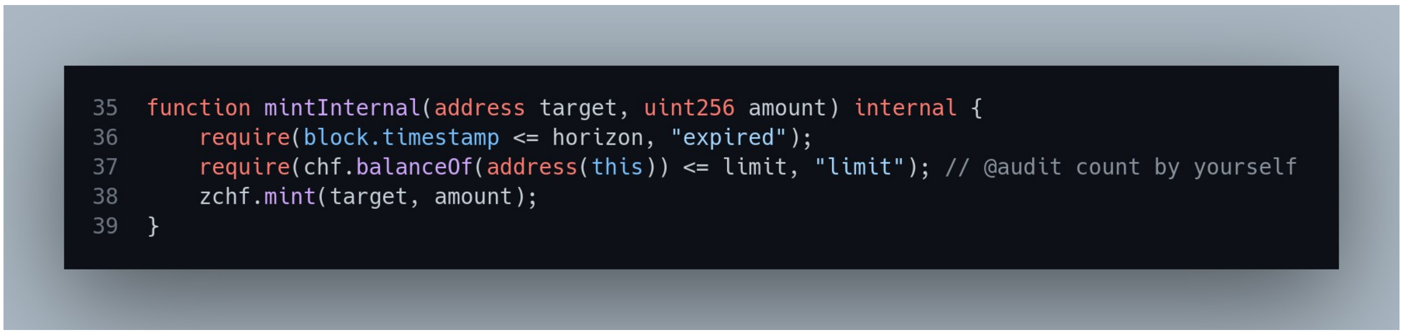
Consider changing the comment and tests to test for 125 days or lower the factor of 10'000 to 7'200.

Status

Fixed with commit f45517a27abe693ea451cebc5e87d68a7d288218.

L-2 DOS in StablecoinBridge minting

mintInternal() in the StablecoinBridge contract relies on the balance of itself in the source token (in this case XCHF). An attacker can now transfer XCHF to the bridge using transfer() on XCHF instead of the bridge functions.



```
function mintInternal(address target, uint256 amount) internal {
    require(block.timestamp <= horizon, "expired");
    require(chf.balanceOf(address(this)) <= limit, "limit"); // @audit count by yourself
    zchf.mint(target, amount);
}
```

This doesn't mint new ZCHF but is added to the limit of the bridge.

This behaviour allows an attacker to transfer more and more XCHF until the limit is reached and no user can mint more ZCHF through the bridge.

Consider keeping track of the minted ZCHF in the stablecoin bridge.

Status

Acknowledged by the client.

L-3 Check returns of ERC20 transfer and transferFrom

Some ERC20 tokens don’t follow the ERC20 EIP (https://eips.ethereum.org/EIPS/eip-20) and don’t throw/revert if a transfer fails but rather return false. One example of such a contract is ZRX (https://etherscan.io/address/0xe41d2489571d322189246dafa5ebde1f4699f498).

Consider using the Openzeppelin SafeERC20 utility (https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#SafeERC20) and replace all transfers and transferFrom with safeTransfer respectively safeTransferFrom.

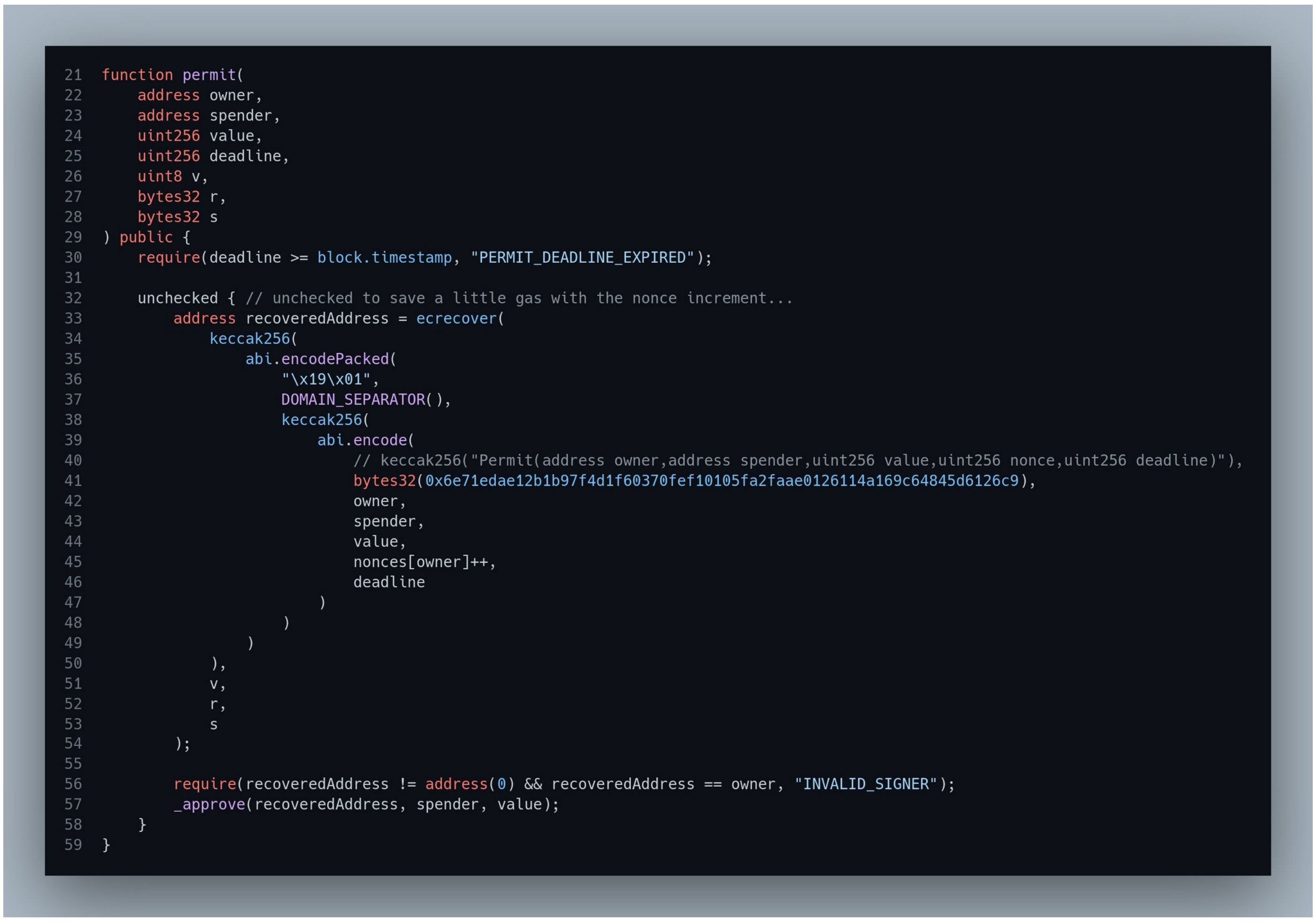
Status

Added to the clients governance guidelines.

L-4 Improper Verification of Cryptographic Signature

The EIP-2612 (EIP-20 Permit extension) implementation ERC20PermitLight uses ecrecover without checking if v is either 27 or 28 and that s is in the lower half order of the elliptic curve.

An attacker can slightly modify the values v,r and s and still produce a valid signature (Signature Malleability). This problem got fixed



```solidity
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public {
    require(deadline >= block.timestamp, "PERMIT_DEADLINE_EXPIRED");

    unchecked { // unchecked to save a little gas with the nonce increment...
        address recoveredAddress = ecrecover(
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"),
                            bytes32(0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9),
                            owner,
                            spender,
                            value,
                            nonces[owner]++,
                            deadline
                        )
                    )
                )
            ),
            v,
            r,
            s
        );

        require(recoveredAddress != address(0) && recoveredAddress == owner, "INVALID_SIGNER");
        _approve(recoveredAddress, spender, value);
    }
}
```

Consider using Openzepplin's ECDSA library or add the proper checks.

Sources:

- https://omniscia.io/platypus-finance-governance-staking/manual-review/Ptp-PTP
- https://swcregistry.io/docs/SWC-117

Status

Acknowledged by the client.

L-5 Suggestion spamming

The function suggestMinter() in the contract Frankencoin doesn’t have protections against spamming attacks. Also previously denied minters can be suggested again, because denyMinter() function deletes the suggestion instead of modifying the suggestions state.

This could lead to a spamming attack proposing a lot of minters leading to slip one through the cracks. Comparable to a MFA Fatigue Attack.

An application fee is already in place to make such attacks more costly.

Consider adding more safe guards like only a limited amount of suggestions per address, minimum balance of ZCHF or FPS to hold by the proposer or allowing to deny suggestions in bulk to make the deny process easier for such attacks.

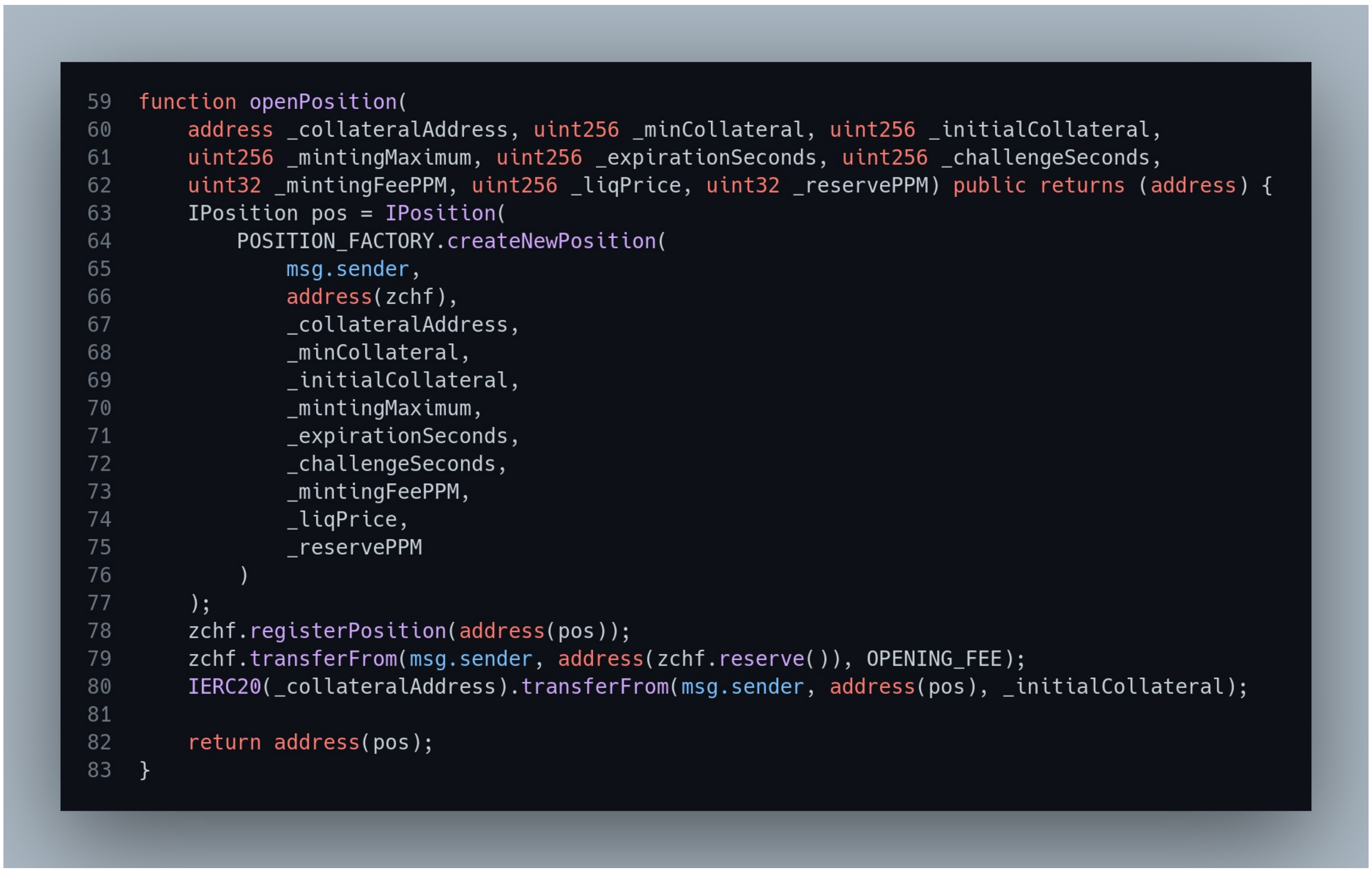
Status

Acknowledged by the client.

L-6 Missing checks for transfer fees

Tokens like USDT can activate a transfer fee. This means that funds received could be lower than expected.

Taking the function openPosition() of the MintingHub as an example, this means that the actual received collateral for a position could be lower than defined with _initialCollateral leading to an undercollateralized position.



```solidity
function openPosition(
    address _collateralAddress, uint256 _minCollateral, uint256 _initialCollateral,
    uint256 _mintingMaximum, uint256 _expirationSeconds, uint256 _challengeSeconds,
    uint32 _mintingFeePPM, uint256 _liqPrice, uint32 _reservePPM) public returns (address) {
    IPosition pos = IPosition(
        POSITION_FACTORY.createNewPosition(
            msg.sender,
            address(zchf),
            _collateralAddress,
            _minCollateral,
            _initialCollateral,
            _mintingMaximum,
            _expirationSeconds,
            _challengeSeconds,
            _mintingFeePPM,
            _liqPrice,
            _reservePPM
        )
    );
    zchf.registerPosition(address(pos));
    zchf.transferFrom(msg.sender, address(zchf.reserve()), OPENING_FEE);
    IERC20(_collateralAddress).transferFrom(msg.sender, address(pos), _initialCollateral);

    return address(pos);
}
```

Consider using the positions balance of the collateral instead of _initialCollateral.

Status

Added to the clients governance guidelines.

Informational

I-1 Failing tests

At the current state 3 tests are failing. 2 of them because of the wrong value mentioned in L-1.

Having a good test suite and a good test coverage is key to find bugs and other issues. Especially in smart contracts which makes it hard to fix bugs because of their immutable nature.

Consider fixing these tests and add more to increase the code coverage.

I-2 Insecure approval implementation on ERC20

The implemented ERC20 contracts are following the guidelines and are compatible with the standard.

Unfortunately the ERC20 token standard has a flaw issuing allowances and allowing an attacker to spend more than expected by the user. This is caused by the nature of allowances where as they are not atomic but override the previous value.

As an example consider Alice allows Bob to spend 100 ZCHF, but later wants to reduce it to 50 ZCHF.

Alice creates transaction 1 and sets the allowance to 100 ZCHF for Bob. Then Alice creates transaction 2 and sets the allowance to 50 ZCHF for Bob. Bob sees transaction 2 in the mempool and front-runs it to spend the 100 ZCHF. Then Alice's second transaction gets approved and Bobs spends an additional 50 ZCHF.

A more detailed explanation can be found here:
https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit

To prevent this problem a best practice has emerged to add 2 new functions to the ERC20 token contracts:

- increaseAllowance(address spender, uint256 addedValue) → bool
- decreaseAllowance(address spender, uint256 subtractedValue) → bool

A reference implementation by Openzepplin can be found here:

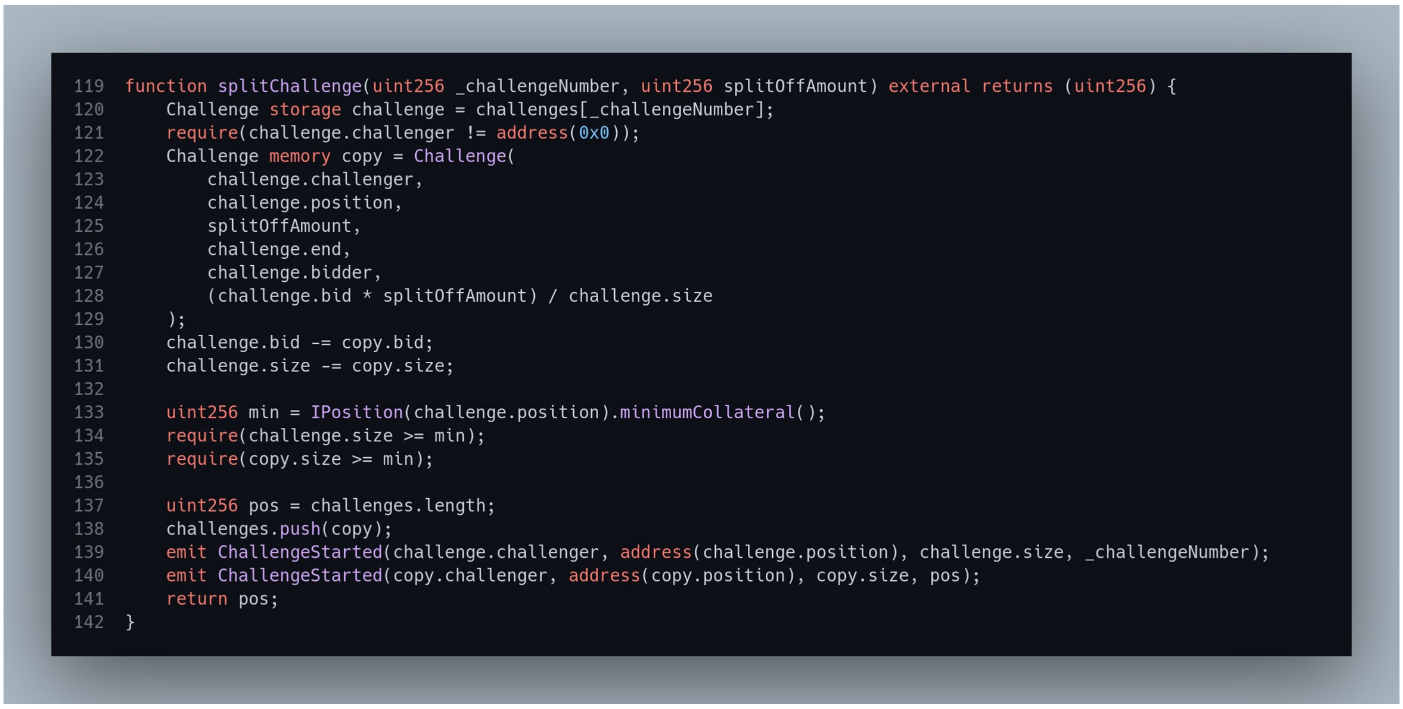
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/dfcc1d16c5efd0fd2a7abac56680810c861a9cd3/contracts/token/ERC20/ERC20.sol#L177-L206

Consider implementing this best practice.

I-3 Storage vs memory pointers

Copying structs from storage to memory is cheaper as soon as there are more reads on the storage pointer as there are properties in the struct.

To make an example with the function splitChallenge() in the MintingHub contract.



```
119 function splitChallenge(uint256 _challengeNumber, uint256 splitOffAmount) external returns (uint256) {
120     Challenge storage challenge = challenges[_challengeNumber];
121     require(challenge.challenger != address(0x0));
122     Challenge memory copy = Challenge(
123         challenge.challenger,
124         challenge.position,
125         splitOffAmount,
126         challenge.end,
127         challenge.bidder,
128         (challenge.bid * splitOffAmount) / challenge.size
129     );
130     challenge.bid -= copy.bid;
131     challenge.size -= copy.size;
132 
133     uint256 min = IPosition(challenge.position).minimumCollateral();
134     require(challenge.size >= min);
135     require(copy.size >= min);
136 
137     uint256 pos = challenges.length;
138     challenges.push(copy);
139     emit ChallengeStarted(challenge.challenger, address(challenge.position), challenge.size, _challengeNumber);
140     emit ChallengeStarted(copy.challenger, address(copy.position), copy.size, pos);
141     return pos;
142 }
```

The function accesses 12 times a property of the challenge storage pointer defined on line 120. This leads to 12 SLOADs which are quite costly in gas. The Challenge struct only has 6 properties which copied to memory on line 120 would lead to 6 SLOADs, this reduces the amount of gas used nearly by half.

To keep in mind is, that writes to pointers in memory (e.g. lines 130 and 131) are **not** reflected in storage.

Consider copying structs into memory as soon as there are more reads than the struct has properties.

I-4 Events

Events are a crucial part in showing and informing the user and other off-chain systems what is currently going on.

Consider adding more events to reflect and log state changes in the system. For example when a totalVotesAtAnchor for a user changes, a user redeems their ZCHF from the reserves, etc.

I-5 Custom Errors

In version 0.8.4 Solidity added custom Errors and the revert keyword. This allows for cheap reverting with custom error messages and dynamic data in them.

Consider replacing the require statements, especially the one with longer strings, with custom Errors to safe gas.

Solidity release notes: https://github.com/ethereum/solidity/releases/tag/v0.8.4

I-6 Lock pragma to fixed compiler version

It is seen as a best practice to use a fixed version of Solidity instead of a more broad solidity pragma. This allows the developers and auditors to test for specific bugs that different versions of Solidity have.

Consider locking down the pragma definition to a fixed version. Preferably 0.8.4 or above so that custom errors (I-5) can be used.

Disclaimer

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

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

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

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

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