Fraxlend (Frax Finance) contest Findings & Analysis Report

2023-01-13

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 contest 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 contest outlined in this document, C4 conducted an analysis of the Fraxlend (Frax Finance) smart contract system written in Solidity. The audit contest took place between August 12—August 17 2022.

Wardens

136 Wardens contributed reports to the Fraxlend (Frax Finance) contest:

1. 0xA5DF
2. Lambda
3. berndartmueller
4. 0x1f8b
5. auditor0517
6. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder and RoiEvenHaim)
7. cccz
8. panprog
9. IIIIIII
10. __141345__

11. 0x52
12. sseefried
13. rbserver
14. cryptphi
15. brgltd
16. reassor
17. 0xDjango
18. carlitox477
19. _Adam
20. Rolezn
21. pfapostol
22. minhquanym
23. zzzitron
24. minhtrng
25. Deivitto
26. JC
27. yac (t4k, Peep, thebensams, devtooligan, blockdev, usmannk, sjkelleyjr, thraull, and NibblerExpress)
28. 0xSmartContract
29. 0xNazgul
30. oyc_109
31. mics
32. c3phas
33. Bnke0x0
34. ret2basic
35. MiloTruck
36. robee
37. Dravee
38. gogo

39. ReyAdmirado
40. Junnon
41. Waze
42. Aymen0909
43. erictee
44. CodingNameKiki
45. Rohan16
46. ElKu
47. durianSausage
48. TomJ
49. kyteg
50. fatherOfBlocks
51. ignacio
52. LeoS
53. ballx
54. medikko
55. simon135
56. Noah3o6
57. Sm4rty
58. Funen
59. delfin454000
60. sach1r0
61. SaharAP
62. sryysryy
63. SooYa
64. a12jmx
65. cRat1st0s
66. ak1
67. asutorufos

68. Chom
69. djxploit
70. gzeon
71. d3e4
72. Yiko
73. EthLedger
74. ladboy233
75. PaludoX0
76. bin2chen
77. ajtra
78. RoiEvenHaim
79. tabish
80. beelzebufo
81. ayeslick
82. yash90
83. cryptonue
84. dy
85. hyh
86. 0xsolstars (Varun_Verma and masterchief)
87. 0xmatt
88. The_GUILD (David_, Ephraim, LeoGold, and greatsamist)
89. 0xNineDec
90. dipp
91. Chinmay
92. 0xkatana
93. Tomio
94. m_Rassska
95. saian
96. NoamYakov

97. flyx
98. Amithuddar
99. Fitraldys
100. gerdusx
101. Metatron
102. 2997ms
103. newfork01
104. chrisdior4
105. 0xackermann
106. jag
107. mrpathfindr
108. Ruhum
109. Diraco
110. Randyyy
111. zeesaw
112. 0xc0ffEE
113. IgnacioB
114. dharma09
115. hakerbaya
116. nxrblsrpr
117. 0xbepresent
118. find_a_bug
119. francoHacker
120. ltyu

This contest was judged by Justin Goro.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 85 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 94 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 Fraxlend (Frax Finance) contest repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 2,110 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] Any borrower with bad debt can be liquidated multiple times to lock funds in the lending pair

Submitted by panprog, also found by 0xA5DF and Lambda

Leftover shares in `liquidateClean` are only subtracted from pair totals, but not from user's borrowed shares. This means that after `liquidateClean`, borrower's shares will be greater than `0` (leftover shares after liquidations), but the user is still insolvent and can be liquidated again and again (with `_sharesToLiquidate` set to `0`). Each subsequent liquidation will write off the bad debt (reduce pair totals by borrower leftover shares/amounts), but doesn't take anything from liquidator nor borrower (since `_sharesToLiquidate == 0`).

This messes up the whole pair accounting, with total asset amounts reducing and total borrow amounts and shares reducing. This will make it impossible for borrowers to repay debt (or be liquidated), because borrow totals will underflow, and lenders amount to withdraw will reduce a lot (they will share non-existant huge bad debt).

Reducing pair totals scenario:

1. Alice borrows `1000 FRAX` (`1000` shares) against `1.5 ETH` collateral (`1 ETH = 1000, Max LTV = 75%`)
2. ETH drops to `500` very quickly with liquidators being unable to liquidate Alice due to network congestion
3. At ETH = `500`, Alice collateral is worth `750` against `1000 FRAX` debt, making Alice insolvent and in a bad debt
4. Liquidator calls `liquidateClean` for `800` shares, which cleans up all available collateral of `1.5 ETH`.
5. At this point Alice has `200` shares debt with `0` collateral
6. Liquidator repeatedly calls `liquidateClean` with `0` shares to liquidate. Each call pair totals are reduced by `200` shares (and total borrow amount by a corresponding amount).
7. When pair totals reach close to `0`, the pool is effectively locked. Borrowers can't repay, lenders can withdraw severly reduced amounts.

Proof of Concept

Copy this to src/test/e2e/LiquidationBugTest.sol

https://gist.github.com/panprog/cbdc1658d63c30c9fe94127a4b4b7e72

Recommended Mitigation Steps

After the line

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L1012

add

```
        _sharesToLiquidate += _sharesToAdjust;
```

amirnader-ghazvini (Frax) marked as duplicate and commented:

> Duplicate of #112

gititGoro (judge) commented:

> Setting to original in set.

DrakeEvans (Frax) confirmed

[H-02] `liquidate()` doesn't mark off bad debt, leading to a 'last lender to withdraw loses' scenario

Submitted by 0xA5DF, also found by cccz and Lambda

When there's bad debt which wasn't marked off yet, the `totalAssets.amount` is higher than the actual (solvent) amount the pair has, meaning that lenders who redeem their tokens earlier will get more in return, at the expense of lenders who leave later. Therefore bad debt should be marked off as soon as possible, the later

it's done the more interest it accumulates and the higher the chances are that some of the lenders will notice and redeem their shares before the bad debt is subtracted from the total assets amount.

Having the option to liquidate via the `liquidate()` function (which doesn't mark off bad debt) can lead to users using that function and leaving bad debt alongside zero collateral or near-zero collateral (giving no motivation for other users to liquidate the rest).

Marking off the remaining of the bad debt via `liquidateClean()` with 0 shares might be possible (not always, some tokens **don't allow 0 transfers**), however there's no motivation for outside users to do so. And as for the lenders - redeeming their tokens before the bad debt is subtracted from the total amount might be more profitable than staying and marking off the bad debt.

Impact

Some lenders might be able to dodge the loss of the bad debt (+ interest), while the last one(s) will have to absorb the lost of the lenders who left too.

Proof of Concept

Consider the following scenario:

- A pair has 10 lenders with 1K$ from each one (10K total)
- Borrower borrowed that 10K
- The collateral price went down and now it's worth only 7K
- A liquidator notices that and liquidates it via `liquidate()`
- The `totalAssets.amount` is 10K + interest, but the total asset amount is actually less than 7K (subtracting liquidator fees)
- 6 lenders notice that and redeem their shares, getting back their money + interest
- The 7th lender to redeem will only be able to get back part of his money
- The remaining 3 lenders will loose all of their money

Recommended Mitigation Steps

Mark off bad debt when remaining collateral reaches zero or near-zero value.

(if only `liquidateClean()` was available then there would be a motivation to not leave near-zero collateral, but as long as this isn't the case consider marking off also in case of near-zero collateral left).

DrakeEvans (Frax) confirmed

gititGoro (judge) commented:

> Setting to original in set. Severity will be maintained as the wardens couldn't know that only one liquidate function would be included in the final release.

Medium Risk Findings (13)

[M-01] Penalty rate is used for pre-maturity date as well

Submitted by 0xA5DF, also found by 0x52 and rbserver

When `_addInterest()` is called post maturity, the penalty rate is applied since the last time `_addInterest()` was called, including the period before maturity.

Impact

Borrowers will be charged with higher interest rates, when maturity date passes. The longer the time since `_addInterest()` was last called before maturity date the more extra-interest they will be charged.

Sidenote: I think this should be considered High, since it comes at the cost of the assets of the borrowers (even though it stems from unnecessary fees being charged, similar to this Putty bug).

Proof of Concept

The test compares 2 scenarios, the 1st one where `addInterest()` isn't called for 30 days before maturity date, and in the 2nd it isn't called just for 1 day before.

The debt (**not interest**) in the first scenario would be ~50% higher than the second, see test output below.

At `src/test/e2e/BorrowPairTest.t.sol` I've modified `_fuzzySetupBorrowToken()` to this and then ran `source .env && forge test --fork-url $MAINNET_URL --fork-block-number $DEFAULT_FORK_BLOCK -m testFuzzyMaxLTVBorrowToken -vvv`. And I've also modified `src/test/e2e/BasePairTest.sol` a bit so that each pair can have a different name for salt (see diff below).

```
        function _fuzzySetupBorrowToken(
            uint256 _minInterest,
            uint256 _vertexInterest,
            uint256 _maxInterest,
            uint256 _vertexUtilization,
            uint256 _maxLTV,
            uint256 _liquidationFee,
            uint256 _priceTop,
            uint256 _priceDiv
        ) public {
            asset = IERC20(FIL_ERC20);
            collateral = IERC20(MKR_ERC20);
            oracleMultiply = AggregatorV3Interface(CHAINLINK_MKR_ETH
            oracleMultiply.setPrice(_priceTop, 1e8, vm);
            oracleDivide = AggregatorV3Interface(CHAINLINK_FIL_ETH);
            oracleDivide.setPrice(_priceDiv, 1e8, vm);
            uint256 _oracleNormalization = 1e18;
            (
                address _rateContract,
                bytes memory _rateInitData
            ) = fuzzyRateCalculator(
                    2,
                    _minInterest,
                    _vertexInterest,
                    _maxInterest,
                    _vertexUtilization
                );
            startHoax(COMPTROLLER_ADDRESS);
            setWhitelistTrue();
            vm.stopPrank();

            address[] memory _borrowerWhitelist = _maxLTV >= LTV_PRE
                ? users
```

```
        : new address[](0);
    address[] memory _lenderWhitelist = _maxLTV >= LTV_PREC
        ? users
        : new address[](0);

    uint256 borrowerDebtWrong;
    {
        // wrong calculation, when addInterest isn't called
        deployFraxlendCustom(
            _oracleNormalization,
            _rateContract,
            _rateInitData,
            _maxLTV,
            _liquidationFee,
            block.timestamp + 30 days,
            1000 * DEFAULT_INT,
            _borrowerWhitelist,
            _lenderWhitelist
        );
        pair.updateExchangeRate();
        _borrowTest(_maxLTV, 15e20, 15e23);
        vm.roll(block.number + 1);


        vm.warp(block.timestamp + 29 days);
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 2 days);
        pair.addInterest();



        address borrower = users[2];
        uint256 borrowerShares = pair.userBorrowShares(borrc
        borrowerDebtWrong = pair.toBorrowAmount(borrowerShar
    }
    deploySaltName = "asdf2";
    uint256 borrowerDebtRight;
    {
        // relatively correct calculation, when addInterest
        deployFraxlendCustom(
            _oracleNormalization,
            _rateContract,
            _rateInitData,
            _maxLTV,
            _liquidationFee,
            block.timestamp + 30 days,
```

```
                1000 * DEFAULT_INT,
                _borrowerWhitelist,
                _lenderWhitelist
            );
            pair.updateExchangeRate();
            _borrowTest(_maxLTV, 15e20, 15e23);
            vm.roll(block.number + 1);

            vm.warp(block.timestamp + 29 days);

            pair.addInterest();

            vm.roll(block.number + 1);
            vm.warp(block.timestamp + 2 days);
            pair.addInterest();


            address borrower = users[2];
            uint256 borrowerShares = pair.userBorrowShares(borrc
            borrowerDebtRight = pair.toBorrowAmount(borrowerShar
        }

        // compare final debt between both scenarios
        assertEq(borrowerDebtRight, borrowerDebtWrong);
    }
```

```
diff --git a/src/test/e2e/BasePairTest.sol b/src/test/e2e/BasePa
index 25114d9..27321dc 100644
--- a/src/test/e2e/BasePairTest.sol
+++ b/src/test/e2e/BasePairTest.sol
@@ -67,6 +67,8 @@ contract BasePairTest is FraxlendPairConstants
     PairAccounting public initial;
     PairAccounting public final_;
     PairAccounting public net;
+    string public deploySaltName = "asdf";
+

     // Users
     address[] public users = [vm.addr(1), vm.addr(2), vm.addr(3
@@ -284,10 +286,12 @@ contract BasePairTest is FraxlendPairConst
         address[] memory _approvedLenders
     ) public {
         startHoax(COMPTROLLER_ADDRESS);
+
```

```
+
            {
                pair = FraxlendPair(
                    deployer.deployCustom(
-                       "testname",
+                       deploySaltName,
                        _encodeConfigData(_normalization, _rateCont
                        _maxLTV,
                        _liquidationFee,
```

Output:

```
  Error: a == b not satisfied [uint]
    Expected: 2135773332009600000000
      Actual: 154101798725378977725
```

Recommended Mitigation Steps

When maturity date passes and the last time `_addInterest()` was called was before maturity date, calculate first the interest rate for that interval as normal interest rate, and then calculate penalty rate for the remaining time.

DrakeEvans (Frax) disputed, disagreed with severity and commented:

> This is mitigated by borrower calling `addInterest()` this is the purpose of the public addInterest function. Alternatively they can repay their loan on time as intended.

0xA5DF (warden) commented:

> Generally speaking, I think we should asses severity based on expected user behavior if the issue hasn't been reported. Since there's no warning to users to call `addInterest()` before maturity date, users would simply not do that and will be charged extra interest.

> Alternatively they can repay their loan on time as intended.

> It's indeed a fine charged for not paying on time, but I don't think you can wave off charging a higher fine than intended as non significant. A user might be a few

> minutes late and be charged a penalty rate for a few days/weeks.

DrakeEvans (Frax) acknowledged

gititGoro (judge) decreased severity to Medium and commented:

> It will create an unexpected user experience to be penalized for periods in which a penalty shouldn't apply, especially during high gas eras. If this is communicated to the user, then an acknowledged label is reasonable.

> If there was an incentive for keeper type users to addInterest such as issuing a small % of shares to users who call the public addInterest then this issue can be marked invalid.

> As for the severity, the existence of addInterest function as an intended mechanism to avoid this means that steps can be taken to avoid excessive interest charging. Indeed third parties can build on safe wrappers of these contracts in a downstream convex-like service. In other words, high interest charging on late debt is by no means an inevitable outcome.

> For these two reasons, I'm downgrading severity to 2 and NOT marking invalid.

[M-02] Interest can be significantly lower if `addInterest` isn't called frequently enough

Submitted by 0xA5DF, also found by sseefried

`addInterest()` always multiplies the interest rate by the time delta, and then multiplies the total borrow amount by the results (`newTotalBorrowAmount = interestRate * timeDelta * totalBorrowAmount`). This can lead to different interest amounts, depending on how frequently it's called.

This will mostly affect custom pairs with stable interest rates (e.g. using linear rate with same min/max/vertex interest) which don't have much activity (e.g. whitelisted borrowers and lenders with long term loan).

The higher the interest rate, the higher the difference will be, for example:

- 15% annual rate would turn into 13.9% if called only once a year (compared to once a day)
- 30% will turn into 26.2%
- 50% will turn to 40%
- 100% to ~70%
- 200% to 110%

Impact

Lenders might end up gaining less interest than expected.

Proof of Concept

The following test shows a case of 50% annual rate, comparing calling it once in 3 days and only once after a year, the difference would be about 4.5% of the final debt.

At `src/test/e2e/BorrowPairTest.t.sol` I've modified `_fuzzySetupBorrowToken()` to this and then ran `source .env && forge test --fork-url $MAINNET_URL --fork-block-number $DEFAULT_FORK_BLOCK -m testFuzzyMaxLTVBorrowToken -vvv`. And I've also modified `src/test/e2e/BasePairTest.sol` a bit so that each pair can have a different name for salt (see diff below).

```
    function _fuzzySetupBorrowToken(
            uint256 _minInterest,
            uint256 _vertexInterest,
            uint256 _maxInterest,
            uint256 _vertexUtilization,
            uint256 _maxLTV,
            uint256 _liquidationFee,
            uint256 _priceTop,
            uint256 _priceDiv
        ) public {
            asset = IERC20(FIL_ERC20);
            collateral = IERC20(MKR_ERC20);
            oracleMultiply = AggregatorV3Interface(CHAINLINK_MKR_ETH
            oracleMultiply.setPrice(_priceTop, 1e8, vm);
            oracleDivide = AggregatorV3Interface(CHAINLINK_FIL_ETH);
            oracleDivide.setPrice(_priceDiv, 1e8, vm);
```

```
uint256 _oracleNormalization = 1e18;
(
    address _rateContract,
    bytes memory _rateInitData
) = fuzzyRateCalculator(
        2,
        12864358183, // ~ 50% annual rate - setting this
        12864358183,
        12864358183,
        _vertexUtilization
    );
startHoax(COMPTROLLER_ADDRESS);
setWhitelistTrue();
vm.stopPrank();

address[] memory _borrowerWhitelist = _maxLTV >= LTV_PRE
    ? users
    : new address[](0);
address[] memory _lenderWhitelist = _maxLTV >= LTV_PRECI
    ? users
    : new address[](0);

uint256 borrowerDebtWrong;
{
    // wrong calculation, when addInterest isn't called
    deployFraxlendCustom(
        _oracleNormalization,
        _rateContract,
        _rateInitData,
        _maxLTV,
        _liquidationFee,
        0,
        1000 * DEFAULT_INT,
        _borrowerWhitelist,
        _lenderWhitelist
    );
    pair.updateExchangeRate();
    _borrowTest(_maxLTV, 15e20, 15e23);
    vm.roll(block.number + 1);


    for(uint256 i =0; i < 100; i++){
        vm.warp(block.timestamp + 3 days);
        vm.roll(block.number + 1);
    }
    pair.addInterest();
```

```
        address borrower = users[2];
        uint256 borrowerShares = pair.userBorrowShares(borrc
        borrowerDebtWrong = pair.toBorrowAmount(borrowerShar
    }
    deploySaltName = "asdf2";
    uint256 borrowerDebtRight;
    {
        // relatively correct calculation, when addInterest
        deployFraxlendCustom(
            _oracleNormalization,
            _rateContract,
            _rateInitData,
            _maxLTV,
            _liquidationFee,
            0,
            1000 * DEFAULT_INT,
            _borrowerWhitelist,
            _lenderWhitelist
        );
        pair.updateExchangeRate();
        _borrowTest(_maxLTV, 15e20, 15e23);

        for(uint256 i =0; i < 100; i++){
            vm.warp(block.timestamp + 3 days);
            pair.addInterest();
            vm.roll(block.number + 1);

        }
        pair.addInterest();


        address borrower = users[2];
        uint256 borrowerShares = pair.userBorrowShares(borrc
        borrowerDebtRight = pair.toBorrowAmount(borrowerShar
    }

    // compare final debt between both scenarios
    assertEq(borrowerDebtRight, borrowerDebtWrong);
}
```

```
diff --git a/src/test/e2e/BasePairTest.sol b/src/test/e2e/BasePa
```

```
index 25114d9..27321dc 100644
--- a/src/test/e2e/BasePairTest.sol
+++ b/src/test/e2e/BasePairTest.sol
@@ -67,6 +67,8 @@ contract BasePairTest is FraxlendPairConstants
     PairAccounting public initial;
     PairAccounting public final_;
     PairAccounting public net;
+    string public deploySaltName = "asdf";
+

     // Users
     address[] public users = [vm.addr(1), vm.addr(2), vm.addr(3
@@ -284,10 +286,12 @@ contract BasePairTest is FraxlendPairConst
         address[] memory _approvedLenders
     ) public {
         startHoax(COMPTROLLER_ADDRESS);
+
+
         {
             pair = FraxlendPair(
                 deployer.deployCustom(
-                    "testname",
+                    deploySaltName,
                     _encodeConfigData(_normalization, _rateCont
                     _maxLTV,
                     _liquidationFee,
```

Output:

```
  Error: a == b not satisfied [uint]
    Expected: 2000166246155040000000
      Actual: 209248965574055348373 5
```

Also, here's a JS function I've used to calculate the rate depending on frequency:

```
function calculateInterest(annualRate, frequency){
    let dailyRate = (1+annualRate) ** (1/365);
    let ratePerFrequency = (dailyRate -1) * 365 / frequency;
    let finalRate = (1+ratePerFrequency) ** frequency;
    return finalRate - 1;
}
```

Recommended Mitigation Steps

- Let the users know that in some cases not calling `addInterest()` frequently enough can lead to lower interest rates.
- Consider dividing the interest calculation into small period of times in case a long period has passed since last time `addInterest()` ran.

Something along these lines (this shouldn't cost much more gas, as it's mostly math operations):

```
// Calculate interest accrued
if(_deltaTime > THRESHOLD){
    uint256 iterations = min(MAX_ITERATIONS, _deltaT
    uint256 multiplier = 1e36;
    for(uint i=0; i < iterations; i++){
    _    multiplier =  (multiplier * _deltaTime  * _c
    }
    _interestEarned = ( _totalBorrow.amount * multip

}else{
    _interestEarned = (_deltaTime * _totalBorrow.amo
}
```

DrakeEvans (Frax) acknowledged

[M-03] Impossible to `setCreationCode()` with code size less than 13K

Submitted by 0xA5DF

https://github.com/code-423n4/2022-08-frax/blob/92a8d7d331cc718cd64de6b02515b554672fb0f3/src/contracts/FraxlendPairDeployer.sol#L170-L178

https://github.com/code-423n4/2022-08-frax/blob/92a8d7d331cc718cd64de6b02515b554672fb0f3/src/contracts/FraxlendPairDeployer.sol#L207-L210

Vulnerability Details

In case of setting the creation code to less than 13K the creation would fail at 2 points:

- `BytesLib.slice(_creationCode, 0, 13000)` would fail since `slice()` requires that `_bytes.length` would be at least `start + length`.
- `SSTORE2.read(contractAddress2)` at `_deployFirst()` would fail since calling this on address without code would cause a math underflow (due to `pointer.code.length - DATA_OFFSET`)

Impact

In case the code of the pair gets smaller (e.g. optimization, moving part of the code to a library etc.) to 13K or less, it'd be impossible to set it as the new creation code (or in the case of the 2nd issue, it'd be impossible to deploy it).

Proof of Concept

In the following test I try to set creation code to a smaller mock pair. The test was added to `src/test/e2e/FraxlendPairDeployerTest.sol`:

```
    function testChangeCreationCodeBug() public {
        // Setup contracts
        setExternalContracts();
        startHoax(COMPTROLLER_ADDRESS);
        setWhitelistTrue();
        vm.stopPrank();
        console2.log("Mock pair size:", type(MockPair).creationC

        vm.startPrank(deployer.owner());
        deployer.setCreationCode(type(MockPair).creationCode);

        // Set initial oracle prices
        bytes memory _rateInitData = defaultRateInitForLinear();
        deployer.deploy(
            abi.encode(
                address(asset),
                address(collateral),
                address(oracleMultiply),
                address(oracleDivide),
                1e10,
```

```
                address(linearRateContract),
                _rateInitData
            )
        );

    }
```

The mock pair was created as `src/contracts/audit/MockPair.sol`:

```
// SPDX-License-Identifier: ISC
pragma solidity ^0.8.15;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/math/SafeCast.sol";
import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Int
import "../FraxlendPairConstants.sol";
import "../libraries/VaultAccount.sol";
import "../libraries/SafeERC20.sol";
import "../interfaces/IERC4626.sol";
import "../interfaces/IFraxlendWhitelist.sol";
import "../interfaces/IRateCalculator.sol";
import "../interfaces/ISwapper.sol";

/// @title FraxlendPairCore
/// @author Drake Evans (Frax Finance) https://github.com/drakee
/// @notice  An abstract contract which contains the core logic
contract MockPair is FraxlendPairConstants, ERC20, Ownable, Paus
    using VaultAccountingLibrary for VaultAccount;
    using SafeERC20 for IERC20;
    using SafeCast for uint256;

    string public version = "1.0.0";

    // ============================================================
    // Settings set by constructor() & initialize()
    // ============================================================

    // Asset and collateral contracts
    IERC20 internal immutable assetContract;
    IERC20 public immutable collateralContract;
```

```
// Oracle wrapper contract and oracleData
address public immutable oracleMultiply;
address public immutable oracleDivide;
uint256 public immutable oracleNormalization;

// LTV Settings
uint256 public immutable maxLTV;

// Liquidation Fee
uint256 public immutable cleanLiquidationFee;
uint256 public immutable dirtyLiquidationFee;

// Interest Rate Calculator Contract
IRateCalculator public immutable rateContract; // For comple
bytes public rateInitCallData; // Optional extra data from i

// Swapper
mapping(address => bool) public swappers; // approved swappe

// Deployer
address public immutable DEPLOYER_ADDRESS;

// Admin contracts
address public immutable CIRCUIT_BREAKER_ADDRESS;
address public immutable COMPTROLLER_ADDRESS;
address public TIME_LOCK_ADDRESS;

// Dependencies
address public immutable FRAXLEND_WHITELIST_ADDRESS;

// ERC20 token name, accessible via name()
string internal nameOfContract;

// Maturity Date & Penalty Interest Rate (per Sec)
uint256 public immutable maturityDate;
uint256 public immutable penaltyRate;

// ==========================================================
// Storage
// ==========================================================

/// @notice Stores information about the current interest ra
/// @dev struct is packed to reduce SLOADs. feeToProtocolRat
CurrentRateInfo public currentRateInfo;
struct CurrentRateInfo {
```

```
    uint64 lastBlock;
    uint64 feeToProtocolRate; // Fee amount 1e5 precision
    uint64 lastTimestamp;
    uint64 ratePerSec;
}

/// @notice Stores information about the current exchange ra
/// @dev Struct packed to save SLOADs. Amount of Collateral
ExchangeRateInfo public exchangeRateInfo;
struct ExchangeRateInfo {
    uint32 lastTimestamp;
    uint224 exchangeRate; // collateral:asset ratio. i.e. ho
}

// Contract Level Accounting
VaultAccount public totalAsset; // amount = total amount of
VaultAccount public totalBorrow; // amount = total borrow am
uint256 public totalCollateral; // total amount of collatera

// User Level Accounting
/// @notice Stores the balance of collateral for each user
mapping(address => uint256) public userCollateralBalance; //
/// @notice Stores the balance of borrow shares for each use
mapping(address => uint256) public userBorrowShares; // repr
// NOTE: user shares of assets are represented as ERC-20 tok

// Internal Whitelists
bool public immutable borrowerWhitelistActive;
mapping(address => bool) public approvedBorrowers;

bool public immutable lenderWhitelistActive;
mapping(address => bool) public approvedLenders;
event Fine(uint256 line);
// ============================================================
// Initialize
// ============================================================

/// @notice The ```constructor``` function is called on depl
/// @param _configData abi.encode(address _asset, address _c
/// @param _maxLTV The Maximum Loan-To-Value for a borrower
/// @param _liquidationFee The fee paid to liquidators given
/// @param _maturityDate The maturityDate date of the Pair
/// @param _penaltyRate The interest rate after maturity dat
/// @param _isBorrowerWhitelistActive Enables borrower white
/// @param _isLenderWhitelistActive Enables lender whitelist
constructor(
```

```
        bytes memory _configData,
        bytes memory _immutables,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        uint256 _maturityDate,
        uint256 _penaltyRate,
        bool _isBorrowerWhitelistActive,
        bool _isLenderWhitelistActive
    )  ERC20("", "") {
        // Handle Immutables Configuration
        {
            (
                address _circuitBreaker,
                address _comptrollerAddress,
                address _timeLockAddress,
                address _fraxlendWhitelistAddress
            ) = abi.decode(_immutables, (address, address, addre

            // Deployer contract
            DEPLOYER_ADDRESS = msg.sender;
            CIRCUIT_BREAKER_ADDRESS = _circuitBreaker;
            COMPTROLLER_ADDRESS = _comptrollerAddress;
            TIME_LOCK_ADDRESS = _timeLockAddress;
            FRAXLEND_WHITELIST_ADDRESS = _fraxlendWhitelistAddre
        }
        emit Fine(177);
        {
            (
                address _asset,
                address _collateral,
                address _oracleMultiply,
                address _oracleDivide,
                uint256 _oracleNormalization,
                address _rateContract,

            ) = abi.decode(_configData, (address, address, addre

            // Pair Settings
            assetContract = IERC20(_asset);
            collateralContract = IERC20(_collateral);
            currentRateInfo.feeToProtocolRate = DEFAULT_PROTOCOI
            cleanLiquidationFee = _liquidationFee;
            dirtyLiquidationFee = (_liquidationFee * 9000) / LIQ

            if (_maxLTV >= LTV_PRECISION && !_isBorrowerWhitelis
            maxLTV = _maxLTV;
```

```
            // Swapper Settings
            swappers[FRAXSWAP_ROUTER_ADDRESS] = true;

            // Oracle Settings
            {
                IFraxlendWhitelist _fraxlendWhitelist = IFraxler
                // Check that oracles are on the whitelist
                if (_oracleMultiply != address(0) && !_fraxlendW
                    revert NotOnWhitelist(_oracleMultiply);
                }

                if (_oracleDivide != address(0) && !_fraxlendWhi
                    revert NotOnWhitelist(_oracleDivide);
                }

                // Write oracleData to storage
                oracleMultiply = _oracleMultiply;
                oracleDivide = _oracleDivide;
                oracleNormalization = _oracleNormalization;

                // Rate Settings
                if (!_fraxlendWhitelist.rateContractWhitelist(_r
                    revert NotOnWhitelist(_rateContract);
                }
            }

            rateContract = IRateCalculator(_rateContract);
        }
        emit Fine(177);


        // Set approved borrowers whitelist
        borrowerWhitelistActive = _isBorrowerWhitelistActive;

        // Set approved lenders whitlist active
        lenderWhitelistActive = _isLenderWhitelistActive;

        // Set maturity date & penalty interest rate
        maturityDate = _maturityDate;
        penaltyRate = _penaltyRate;
    }


    function initialize(
        string calldata _name,
```

```
            address[] calldata _approvedBorrowers,
            address[] calldata _approvedLenders,
            bytes calldata _rateInitCallData
        ) external {
        }

    }
```

Recommended Mitigation Steps

If creation code length is 13K or less:

- Don't run `BytesLib.slice()`
- At deployment read only from the first address

DrakeEvans (Frax) acknowledged

[M-04] Wrong percent for `FraxlendPairCore.dirtyLiquidationFee`.

Submitted by auditor0517, also found by _Adam, 0xA5DF, cccz, minhquanym, minhtrng, and zzzitron

After confirming with the sponsor, `dirtyLiquidationFee` is 90% of `cleanLiquidationFee` like the comment.

But it uses `9% (9000 / 1e5 = 0.09)` and the fee calculation will be wrong here.

Recommended Mitigation Steps

We should change `9000` to `90000`.

```
    dirtyLiquidationFee = (_liquidationFee * 90000) / LIQ_PRECISION;
```

DrakeEvans (Frax) confirmed and commented:

> Confirmed.

gititGoro (judge) commented:

> This issue has a great many duplicates. Reason for setting this report as the original:
>
> 1. It's complete in that it charts the problem and gives a code based mitigation
> 2. It's short and to the point.
> 3. The sponsor's preference was factored in.

[M-05] Liquidator might end up paying much more asset than collateral received

Submitted by 0xA5DF, also found by __141345__

`liquidateClean` doesn't check that the amount of shares the liquidator repays are covered by the collateral available, if the user repays more shares than (equally worth) collateral available - he'll get only the amount of collateral available.

Impact

Liquidator might end up paying assets that are worth much more than the collateral he's received.

Even though it's also the liquidator's responsibility to check that there's enough collateral available, mistakes are commonly done (wrong calculation, typos, deadline too large) and the protocol should prevent those kinds of errors if possible.

Otherwise users might lose trust in the protocol when they lose their funds due to errors.

Proof of Concept

The following test was added to `src/test/e2e/LiquidationPairTest.sol`. In this scenario the user ends up with collateral that's worth only 1/3 of the asset-tokens he paid.

```
    function testCleanLiquidate() public {
```

```
// Setup contracts
defaultSetUp();
// Sets price to 3200 USD per ETH

uint256 _amountToBorrow = 16e20; // 1.5k
uint256 _amountInPool = 15e23; // 1.5m

// collateral is 1.5 times borrow amount
oracleDivide.setPrice(3200, 1, vm);
mineBlocks(1);
(, uint256 _exchangeRate) = pair.exchangeRateInfo();
uint256 _collateralAmount = (_amountToBorrow * _exchange
faucetFunds(asset, _amountInPool);
faucetFunds(collateral, _collateralAmount);
lendTokenViaDeposit(_amountInPool, users[0]);
borrowToken(uint128(_amountToBorrow), _collateralAmount,
uint256 mxltv = pair.maxLTV();
uint256 cleanLiquidationFee = pair.cleanLiquidationFee()
uint256 liquidation_price = ((((1e18 / _exchangeRate) *
liquidation_price /= 4; // Collateral price goes down er
oracleDivide.setPrice(liquidation_price, 1, vm);
mineBlocks(1);
uint128 _shares = pair.userBorrowShares(users[2]).toUint
for (uint256 i = 0; i < 1; i++) {
    pair.addInterest();
    mineOneBlock();
}
vm.startPrank(users[1]);
pair.addInterest();
asset.approve(address(pair), toBorrowAmount(_shares, tru

uint256 liquidatorBalanceBefore = asset.balanceOf(users[

uint256 collateralGained = pair.liquidateClean(_shares,

uint256 liquidatorBalanceAfter = asset.balanceOf(users[1

uint256 balanceDiff = liquidatorBalanceBefore - liquidat

console2.log("Balance diff is: ", balanceDiff);
console2.log("_exchangeRate is: ", _exchangeRate);
console2.log("cleanLiquidationFee is: ", cleanLiquidatic

(, _exchangeRate) = pair.exchangeRateInfo();
// 11/10 is for liquidation fee, 1e18 is for exchange pr
assertEq(collateralGained, balanceDiff *  _exchangeRate
```

```
        assertEq(pair.userBorrowShares(users[2]), 0);
        vm.stopPrank();
    }
```

Output:

```
Balance diff is:  160000011384589113999
_exchangeRate is:  863759326621800
cleanLiquidationFee is:  10000
Error: a == b not satisfied [uint]
  Expected: 7394958035811125166
    Actual: 20730223838923200000
```

Recommended Mitigation Steps

Check that the shares intended to be repaid are worth the collateral + fee, if not reduce the amount of shares to be repaid accordingly.

DrakeEvans (Frax) acknowledged, but disagreed with severity and commented:

> This is by design, the deadline parameter can protect against this. Liquidators are encouraged to provide values such that all collateral is cleared. This does present the risk of slightly (a few wei) overpaying for the collateral. However, they are compensated with a high liquidation fee. They can protect themselves by inputting a deadline to the tx as well to prevent interest accruing and an old tx being replayed.

gititGoro (judge) commented:

> My concern is with transaction ordering. While the deadline can prevent stale transactions, nothing prevents a validator (or whoever orders transactions in the future) from inserting their repayment prior to a well calculated transaction. An MEV drain.

> What protects the victim in this case is the _totalBorrow parameter. It will be lower and so place a lower bound on the overpayment.

> Since liquidators can't protect themselves from MEV, there is potential leakage. Leaving severity at Medium.

[M-06] FraxlendPair#setTimeLock: Allows the owner to reset TIME*LOCK*ADDRESS

Submitted by carlitox477, also found by 0xA5DF, 0xDjango, brgltd, lllllll, and reassor

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L84-L86

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L204-L207

Impact

Allows to reset **TIME*LOCK*ADDRESS** value multiple times by the owner. According to comments in FraxlendPairCore this should act as a constant/immutable value. Given that this value will be defined through function **setTimeLock** in **FraxLendPair** contract this value can be changed whenever the owner wants. This does not seem to be the expected behaviour.

Proof of Concept

The owner can call the function **setTimeLock** whenever they want, which resets the value of **TIME*LOCK*ADDRESS**.

Recommended Mitigation Steps

Add a bool which act as mutex if **TIME*LOCK*ADDRESS** has already been set, and modify **setTimeLock** function in FraxlendPair contract

```
    // In FraxlendPair contract
    bool public timelockSetted;
    function setTimeLock(address _newAddress) external onlyOwner {
            require(!timelockSetted);
            emit SetTimeLock(TIME_LOCK_ADDRESS, _newAddress);
            TIME_LOCK_ADDRESS = _newAddress;
```

```
            timelockeSetted=true;
    }
```

DrakeEvans (Frax) confirmed

[M-07] FraxlendPair.changeFee() doesn't update interest before changing fee.

Submitted by auditor0517

This function is changing the protocol fee that is used during interest calculation here.

But it doesn't update interest before changing the fee so the `_feesAmount` will be calculated wrongly.

Proof of Concept

As we can see during pause() and unpause(), `_addInterest()` must be called before any changes.

But with the changeFee(), it doesn't update interest and the `_feesAmount` might be calculated wrongly.

- At time `T1`, _currentRateInfo.feeToProtocolRate = F1.
- At `T2`, the owner had changed the fee to `F2`.
- At `T3`, _addInterest() is called during `deposit()` or other functions.
- Then during this calculation, `F1` should be applied from `T1` to `T2` and `F2` should be applied from `T2` and `T3`. But it uses `F2` from `T1` to `T2`.

Recommended Mitigation Steps

Recommend modifying `changeFee()` like below.

```
function changeFee(uint32 _newFee) external whenNotPaused {
    if (msg.sender != TIME_LOCK_ADDRESS) revert OnlyTimeLock();
```

```
        if (_newFee > MAX_PROTOCOL_FEE) {
            revert BadProtocolFee();
        }

        _addInterest(); //++++++++++++++++++++++++++++++++

        currentRateInfo.feeToProtocolRate = _newFee;
        emit ChangeFee(_newFee);
    }
```

[DrakeEvans (Frax) confirmed, but disagreed with severity and commented]:

> Disagree with severity as it can be mitigated by calling `addInterest()` beforehand by admin or regular user. But we will address it anyway for convenience. We assume admins are not malicious.

[0xA5DF (warden) commented]:

> Admins might not be malicious, but without knowing about the issue (i.e. if the warden hasn't reported this) they wouldn't call `addInterest()` beforehand, leading to higher interest than intended.

[gititGoro (judge) commented]:

> Maintaining severity as it is a potential leakage.

[M-08] Owner of `FraxlendPair` can set arbitrary time lock contract address to circumvent time lock

Submitted by berndartmueller

The ownership of a deployed Fraxlend pair is transferred to `COMPTROLLER_ADDRESS` on deployment via `FraxlendPairDeployer_deploySecond`. This very owner is able to change the currently used time lock contract address with the `FraxlendPair.setTimeLock` function. A time lock is enforced on the `FraxlendPair.changeFee` function whenever the protocol fee is adjusted.

However, as the Fraxlend pair owner is able to change the time lock contract address to any other arbitrary (contract) address, it is possible to circumvent this timelock without users knowing. By using a custom smart contract without an enforced time lock, the protocol fee can be changed at any time without a proper time lock.

Proof of Concept

FraxlendPair.sol#L206

```
    /// @notice The ```setTimeLock``` function sets the TIME_LOCK ac
    /// @param _newAddress the new time lock address
    function setTimeLock(address _newAddress) external onlyOwner {
        emit SetTimeLock(TIME_LOCK_ADDRESS, _newAddress);
        TIME_LOCK_ADDRESS = _newAddress;
    }
```

Recommended Mitigation Steps

Currently, the owner `COMPTROLLER_ADDRESS` address is trustworthy, however, nothing prevents the above-described scenario. To protect users from sudden protocol fee changes, consider using a minimal time lock implementation directly implemented in the contract without trusting any external contract.

DrakeEvans (Frax) confirmed

[M-09] FraxlendPair.sol is not fully EIP-4626 compliant

Submitted by 0x52, also found by berndartmueller, cryptphi, and Lambda

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L136-L138

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L140-L142

Impact

FraxlendPair.sol is not EIP-4626 compliant, variation from the standard could break composability and potentially lead to loss of funds.

Proof of Concept

According to EIP-4626 method specifications (https://eips.ethereum.org/EIPS/eip-4626)

For maxDeposit:

```
MUST factor in both global and user-specific limits, like if dep
```

For maxMint:

```
MUST factor in both global and user-specific limits, like if mir
```

When FraxlendPair.sol is paused, deposit and mint are both disabled. This means that maxMint and maxDeposit should return 0 when the contract is paused.

The current implementations of maxMint and maxDeposit do not follow this specification:

```
function maxDeposit(address) external pure returns (uint256) {
    return type(uint128).max;
}

function maxMint(address) external pure returns (uint256) {
    return type(uint128).max;
}
```

No matter the state of the contract they always return uint128.max, but they should return 0 when the contract is paused.

Recommended Mitigation Steps

maxDeposit and maxMint should be updated to return 0 when contract is paused. Use of the whenNotPaused modifier is not appropriate because that would cause a revert and maxDeposit and maxMint should never revert according to EIP-4626.

DrakeEvans (Frax) confirmed

[M-10] Decimals limitation limits the tokens that can be used

Submitted by CertoraInc

Decimals limitation limits the tokens that can be used.

Proof of Concept

Let's give some name to the decimals of certain numbers: n = decimals of numerator oracle. d = decimals denominator oracle. a = decimals of the asset. c= decimals of the collateral.

Now, the `oracleNormalization = 10 ^(18 + n - d + a - c)`.

And here: https://github.com/code-423n4/2022-08-frax/blob/main/src/contracts/FraxlendPairCore.sol#L536 , `price` has decimals of `36 + n -d`, so `here()` when we calculate `_exchangeRate = _price / oracleNormalization;` it would underflow and revert if `a >18 +c`.

And that's a pretty big limitation on the tokens options. We have USDC which have 6 decimals so all the tokens the their decimals < 24 are not possible to use in this system (with USDC together).

DrakeEvans (Frax) disputed, disagreed with severity and commented:

> Known issue, prevents certain combinations of tokens from being deployed. No high risk as no deployment will occur. No funds at risk, no incorrect functionality. Low at best.

gititGoro (judge) decreased severity to Medium and commented:

> This hasn't been listed as a known issue so it can't be marked invalid but since deployments can't occur, it's a Medium severity.

[M-11] Fraxlend pair deployment can be front-run by a custom pair deployment

Submitted by berndartmueller, also found by lllllll

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L205

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L329-L330

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L253

Impact

The `FraxlendPairDeployer` contract used to deploy Fraxlend pairs prevents deploying pairs with the same `salt` and `_configData`. This makes it vulnerable to front-running pair deployments and prevents deploying honest Fraxlend pairs.

Proof of Concept

FraxlendPairDeployer._deployFirst

```
    function _deployFirst(
        bytes32 _saltSeed,
        bytes memory _configData,
        bytes memory _immutables,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        uint256 _maturityDate,
        uint256 _penaltyRate,
        bool _isBorrowerWhitelistActive,
```

```
        bool _isLenderWhitelistActive
    ) private returns (address _pairAddress) {
        {
            // _saltSeed is the same for all public pairs so duplica
            bytes32 salt = keccak256(abi.encodePacked(_saltSeed, _cc
            require(deployedPairsBySalt[salt] == address(0), "Fraxle

            bytes memory _creationCode = BytesLib.concat(
                SSTORE2.read(contractAddress1),
                SSTORE2.read(contractAddress2)
            );
            bytes memory bytecode = abi.encodePacked(
                _creationCode,
                abi.encode(
                    _configData,
                    _immutables,
                    _maxLTV,
                    _liquidationFee,
                    _maturityDate,
                    _penaltyRate,
                    _isBorrowerWhitelistActive,
                    _isLenderWhitelistActive
                )
            );

            assembly {
                _pairAddress := create2(0, add(bytecode, 32), mload(
            }
            require(_pairAddress != address(0), "FraxlendPairDeploye

            deployedPairsBySalt[salt] = _pairAddress;
        }

        return _pairAddress;
    }
```

A deployed Fraxlend pair address is stored in the `deployedPairsBySalt` mapping using a `salt` as the key. This salt is generated based on a seed `_saltSeed` and `_configData`. Deploying a standard Fraxlend pair with `FraxlendPairDeployer.deploy` uses `_saltSeed = keccak256(abi.encodePacked("public"))` and user-defined `_configData`. The salt is the same for all standard deployments.

FraxlendPairDeployer.deploy

```
function deploy(bytes memory _configData) external returns (addr
    (address _asset, address _collateral, , , , address _rateCor
        _configData,
        (address, address, address, address, uint256, address, b
    );
    string memory _name = string(
        abi.encodePacked(
            "FraxlendV1 - ",
            IERC20(_collateral).safeName(),
            "/",
            IERC20(_asset).safeName(),
            " - ",
            IRateCalculator(_rateContract).name(),
            " - ",
            (deployedPairsArray.length + 1).toString()
        )
    );

    _pairAddress = _deployFirst(
        keccak256(abi.encodePacked("public")),
        _configData,
        abi.encode(CIRCUIT_BREAKER_ADDRESS, COMPTROLLER_ADDRESS,
        DEFAULT_MAX_LTV,
        DEFAULT_LIQ_FEE,
        0,
        0,
        false,
        false
    );

    _deploySecond(_name, _pairAddress, _configData, new address[

    _logDeploy(_name, _pairAddress, _configData, DEFAULT_MAX_LTV
}
```

A whitelisted (and potentially dishonest) user could watch the mempool for new Fraxlend pair deployments and front-run deployments with a custom pair deployment by calling `FraxlendPairDeployer.deployCustom`.

FraxlendPairDeployer.deployCustom

```
    function deployCustom(
        string memory _name,
        bytes memory _configData,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        uint256 _maturityDate,
        uint256 _penaltyRate,
        address[] memory _approvedBorrowers,
        address[] memory _approvedLenders
    ) external returns (address _pairAddress) {
        require(_maxLTV <= GLOBAL_MAX_LTV, "FraxlendPairDeployer: _m
        require(
            IFraxlendWhitelist(FRAXLEND_WHITELIST_ADDRESS).fraxlendI
            "FraxlendPairDeployer: Only whitelisted addresses"
        );

        _pairAddress = _deployFirst(
            keccak256(abi.encodePacked(_name)),
            _configData,
            abi.encode(CIRCUIT_BREAKER_ADDRESS, COMPTROLLER_ADDRESS,
            _maxLTV,
            _liquidationFee,
            _maturityDate,
            _penaltyRate,
            _approvedBorrowers.length > 0,
            _approvedLenders.length > 0
        );

        _deploySecond(_name, _pairAddress, _configData, _approvedBor

        deployedPairCustomStatusByAddress[_pairAddress] = true;

        _logDeploy(_name, _pairAddress, _configData, _maxLTV, _liqui
    }
```

Internally the `FraxlendPairDeployer.deployCustom` uses the same `_deployFirst` function as a standard pair deployment uses. However, the salt is user-definable by using any `_name`, for instance `public`. Then the salt is the same as `_saltSeed = keccak256(abi.encodePacked("public"))`. The user then uses the same `_configData` as the to be deployed pair. But, contrary to a standard pair deployment, a whitelisted user is also able to provide additional configuration, `_maxLTV`, `_liquidationFee` and so on.

The whitelisted user is able to deploy a custom Fraxlend pair with the same salt and the same `_configData` but by using for instance a `_maturityDate < block.timestamp`, this deployed custom pool is unusable (due to the modifier `isNotPastMaturity` reverting. This modifier is used by `FraxlendPairCore.deposit`, `FraxlendPairCore.mint`, `FraxlendPairCore.borrowAsset`, `FraxlendPairCore.addCollateral` and `FraxlendPairCore.leveragedPosition`).

As `FraxlendPairDeployer._deployFirst` checks if a pair is already deployed with a given salt, the honest pair deployment that got front-run reverts. It is now not possible to deploy a pair with the same configuration.

NOTE: The same can be achieved by front-running a pair deployment transaction with a custom pair deployment and with the same name (`FraxlendPairDeployer.deployCustom` allows providing an arbitrary name. `FraxlendPairDeployer._deploySecond` checks the name of the pair and reverts if a pair with that name already exists:

FraxlendPairDeployer._deploySecond

```
    function _deploySecond(
        string memory _name,
        address _pairAddress,
        bytes memory _configData,
        address[] memory _approvedBorrowers,
        address[] memory _approvedLenders
    ) private {
        (, , , , , , bytes memory _rateInitData) = abi.decode(
            _configData,
            (address, address, address, address, uint256, address, b
        );
        require(deployedPairsByName[_name] == address(0), "FraxlendP
        deployedPairsByName[_name] = _pairAddress;
        deployedPairsArray.push(_name);

        // Set additional values for FraxlendPair
        IFraxlendPair _fraxlendPair = IFraxlendPair(_pairAddress);
        _fraxlendPair.initialize(_name, _approvedBorrowers, _approve

        // Transfer Ownership of FraxlendPair
```

```
        _fraxlendPair.transferOwnership(COMPTROLLER_ADDRESS);
    }
```

Recommended Mitigation Steps

Consider validating the `_name` parameter in the `FraxlendPairDeployer.deployCustom` function to not equal the string `public`, thus preventing the usage of the same salt as a standard pair deployment.

Additionally, consider prefixing the name of a custom pair deployment to also prevent front-running this way.

DrakeEvans (Frax) acknowledged

[M-12] Denial of service in globalPause by wrong logic

Submitted by 0x1f8b

The method `globalPause` is not tested and it doesn't work as expected.

Proof of Concept

Because the method returns an array (`_updatedAddresses`) and has never been initialized, when you want to set its value, it fails.

Recipe:

- Call `globalPause` with any valid address.
- The transaction will FAULT.

Affected source code

- FraxlendPairDeployer.sol#L405

Recommended Mitigation Steps

Initialize the `_updatedAddresses` array like shown below:

```
    function globalPause(address[] memory _addresses) external r
        require(msg.sender == CIRCUIT_BREAKER_ADDRESS, "Circuit
        address _pairAddress;
        uint256 _lengthOfArray = _addresses.length;
+       _updatedAddresses = new address[](_lengthOfArray);
        for (uint256 i = 0; i < _lengthOfArray; ) {
            _pairAddress = _addresses[i];
            try IFraxlendPair(_pairAddress).pause() {
                _updatedAddresses[i] = _addresses[i];
            } catch {}
            unchecked {
                i = i + 1;
            }
        }
    }
```

DrakeEvans (Frax) confirmed, but disagreed with severity and commented:

> Valid, disagree with severity. This is a convenience function. Pause can still be called on the pairs themselves individually. No user funds at risk, and no users can even touch this function, additionally no loss of functionality in the pairs themselves. Only result is admin having to revert and spin up tx individually.

gititGoro (judge) decreased severity to Medium and commented:

> Well caught! But definitely a Medium severity, given the existence of per pair pausing.

[M-13] No incentives to write off bad debt when remaining collateral is very small

Submitted by Lambda

https://github.com/code-423n4/2022-08-frax/blob/b58c9b72f5fe8fab81f7436504e7daf60fd124e3/src/contracts/FraxlendPairCore.sol#L989

https://github.com/code-423n4/2022-08-frax/blob/b58c9b72f5fe8fab81f7436504e7daf60fd124e3/src/contracts/Fraxlend

PairCore.sol#L1002

Impact

When a position only has a bit of collateral left (e.g., because a previous `liquidateClean` call provided just too little shares or the exchange rate just changed), there is no incentive anymore to call `liquidateClean` again. The maximum amount that a user will get is the remaining collateral balance, which does not cover the gas fees in such cases.

Therefore, the bad debt is never written off, meaning that the mechanism does not work in such cases and these pairs become toxic (as the last user will have to pay for the leftover borrow shares).

Recommended Mitigation Steps

To avoid such scenarios, it should not be possible to remove almost all of the collateral. Consider introducing an upper limit, e.g. 95%. Above this limit, no dirty liquidations would be allowed.

DrakeEvans (Frax) acknowledged, but disagreed with severity and commented:

> This is inherent in high gas fee environments. There are significant incentives to avoid this with the clean liquidation fee being 11% higher than the dirty liquidation fee. To create this scenario the liquidator would need to do a dirty liquidation first.

gititGoro (judge) commented:

> This sort of thing seems unavoidable when gas is present. Aave documentation refers to residual untouched balances as 'dust'. If the pair is worth recapitalizing then the gas hit will be worth it. It doesn't appear that there are problems with incentives here.

> And there isn't an obvious QA improvement.

> The mitigation adds a gas overhead for rare, mostly harmless boundary cases which does not seem to be prudent when deploying to mainnet.

> Value is technically leaked but acknowledgement from sponsor seems appropriate in this case.

Low Risk and Non-Critical Issues

For this contest, 85 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0x1f8b** received the top score from the judge.

The following wardens also submitted reports: pfapostol, Rolezn, yac, brgltd, Deivitto, 0xNazgul, IIIIIII, mics, oyc_109, Bnke0x0, reassor, 0xDjango, MiloTruck, bin2chen, robee, c3phas, rbserver, Junnon, gogo, RoiEvenHaim, ReyAdmirado, Waze, tabish, Aymen0909, CodingNameKiki, auditor0517, ElKu, beelzebufo, kyteg, fatherOfBlocks, Lambda, ballx, LeoS, berndartmueller, erictee, ret2basic, TomJ, 0x52, ayeslick, cccz, yash90, Funen, 0xA5DF, JC, Dravee, medikko, cryptonue, cryptphi, delfin454000, sach1r0, simon135, Sm4rty, sryysryy, durianSausage, dy, minhquanym, Rohan16, zzzitron, a12jmx, hyh, ak1, asutorufos, CertoraInc, Chom, 0xsolstars, __141345__, 0xSmartContract, djxploit, SooYa, _Adam, 0xmatt, cRat1st0s, The_GUILD, Yiko, EthLedger, gzeon, ignacio, Noah3o6, 0xNineDec, d3e4, dipp, ladboy233, PaludoX0, *and* SaharAP.

Low Risk Issues

[L-01] Outdated compiler

The pragma version used is:

```
pragma solidity ^0.8.15;
```

The minimum required version must be 0.8.16; otherwise, contracts will be affected by the following **important bug fixes**:

0.8.16

- Code Generation: Fix data corruption that affected ABI-encoding of calldata values represented by tuples: structs at any nesting level; argument lists of external functions, events and errors; return value lists of external functions. The

32 leading bytes of the first dynamically-encoded value in the tuple would get zeroed when the last component contained a statically-encoded array.

Apart from these, there are several minor bug fixes and improvements.

[L-02] `SafeERC20` mismatch logics

The `SafeERC20` library says:

> @title SafeERC20 provides helper functions for safe transfers as well as safe metadata access

But the reality is that *metada access* is not secure.

The `SafeERC20.safeTransfer` and `SafeERC20.safeTransferFrom` methods perform a safe check that the address to call is a contract by calling functionCall, this checks doesn't appear in the methods `safeSymbol`, `safeName` and `safeDecimals`, so some of the `SafeERC20` library methods are prone to phantom methods attacks.

Proof of Concept

If you try to call the `SafeERC20` methods with the following token `0x0000000000000000000000000000000000000000` (or any EOA) you can see that the transaction is successful, you have to check that the address is a valid contract and maybe check that the call returns a certain data. Otherwise it is possible to call it and lose tokens or produce undesired errors.

This reminds me of this attack https://certik.medium.com/qubit-bridge-collapse-exploited-to-the-tune-of-80-million-a7ab9068e1a0 where we can see:

Affected source code:

- SafeERC20.sol#L39-L58

Recommended changes

- Use `functionStaticCall` from OpenZeppelin.

```
+import "@openzeppelin/contracts/utils/Address.sol";
...
library SafeERC20 {
+    using Address for address;

    /// @notice Provides a safe ERC20.symbol version which retur
    /// @param token The address of the ERC-20 token contract.
    /// @return (string) Token symbol.
    function safeSymbol(IERC20 token) internal view returns (str
-       (bool success, bytes memory data) = address(token).stati
+       (bool success, bytes memory data) = address(token).funct
        return success ? returnDataToString(data) : "???";
    }

    /// @notice Provides a safe ERC20.name version which returns
    /// @param token The address of the ERC-20 token contract.
    /// @return (string) Token name.
    function safeName(IERC20 token) internal view returns (strin
-       (bool success, bytes memory data) = address(token).stati
+       (bool success, bytes memory data) = address(token).funct
        return success ? returnDataToString(data) : "???";
    }

    /// @notice Provides a safe ERC20.decimals version which ret
    /// @param token The address of the ERC-20 token contract.
    /// @return (uint8) Token decimals.
    function safeDecimals(IERC20 token) internal view returns (u
-       (bool success, bytes memory data) = address(token).stati
+       (bool success, bytes memory data) = address(token).funct
        return success && data.length == 32 ? abi.decode(data,
    }
```

[L-03] Lack of ACK during owner change

It's possible to lose the ownership under specific circumstances.

Because of human error it's possible to set a new invalid owner. When you want to change the owner's address it's better to propose a new owner, and then accept this ownership with the new wallet.

Affected source code:

- FraxlendWhitelist.sol#L30
- FraxlendPairDeployer.sol#L44

[L-04] Constant salt

It's not possible to use `public` as pair name, if you are using the same default values as was used in `deploy`.

Becuase the salt for deploy is *"public"*, it is not possible to use the same as a name.

Affected source code:

- FraxlendPairDeployer.sol#L329
- FraxlendPairDeployer.sol#L372

Recommended changes

```
-    keccak256(abi.encodePacked(_name)),
+    keccak256(abi.encodePacked("custom", _name)),
```

[L-05] Bypass `TimeLock` restrictions

Esto permite que llamadas como la que se muestran a continuacion, que se encuentran limitadas al contrato de `TIME_LOCK_ADDRESS`, sea posible ser llamadas por el owner, evitando cualquier limitación de tiempo, debido a que en primer lugar no se comprueba que la direccion establecida en `setTimeLock` sea un contrato, y segundo, porque podria establecer cualquier contrato, y en la misma llamada hacer la llamada limitada al `TimeLock`.

The `FraxlendPair` contract has a `setTimeLock` method that allows you to modify the `TimeLock` contract (`TIME_LOCK_ADDRESS`), this call is protected by the `onlyOwner` modifier.

This allows the owner to call methods like the one shown below (`changeFee`), which are limited to the `TIME_LOCK_ADDRESS` contract, avoiding any time constraints, since

the established address in `setTimeLock` is not checked to be a contract, you could set any contract, and in the same transaction you can call the protected method.

```
        function changeFee(uint32 _newFee) external whenNotPaused {
            if (msg.sender != TIME_LOCK_ADDRESS) revert OnlyTimeLock
```

This can facilitate rogue pool attacks.

Affected source code:

- FraxlendPair.sol#L204-L222

[L-06] Division before multiple can lead to precision errors

Because Solidity integer division may truncate, it is often preferable to do multiplication before division to prevent precision loss.

Affected source code:

- FraxlendPairCore.sol#L315

Recommended changes

```
        uint256 internal constant LTV_PRECISION = 1e5; // 5 decimals
        uint256 internal constant EXCHANGE_PRECISION = 1e18;
        ...
    +   uint256 internal constant SOLVENT_PRECISION = EXCHANGE_PREC
        ...

        function _isSolvent(address _borrower, uint256 _exchangeRate
            if (maxLTV == 0) return true;
            uint256 _borrowerAmount = totalBorrow.toAmount(userBorrc
            if (_borrowerAmount == 0) return true;
            uint256 _collateralAmount = userCollateralBalance[_borrc
            if (_collateralAmount == 0) return false;

    -       uint256 _ltv = (((_borrowerAmount * _exchangeRate) / EXC
    +       uint256 _ltv = (_borrowerAmount * _exchangeRate) / (_col
            return _ltv <= maxLTV;
```

```
        }
```

[L-07] `Ownable` and `Pausable`

The contract `FraxlendPairCore` is `Ownable` and `Pausable`, so the owner could resign while the contract is paused, causing a Denial of Service. Owner resignation while the contract is paused should be avoided.

Affected source code:

- FraxlendPairCore.sol#L46

Non-Critical Issues

[N-01] Use `encode` instead of `encodePacked` for hashig

Use of `abi.encodePacked` is safe, but unnecessary and not recommended. `abi.encodePacked` can result in hash collisions when used with two dynamic arguments (string/bytes).

There is also discussion of removing `abi.encodePacked` from future versions of Solidity (ethereum/solidity#11593), so using `abi.encode` now will ensure compatibility in the future.

Affected source code:

- FraxlendPairDeployer.sol#L204

[N-02] Wrong comment of `toBorrowAmount` function

The function `toBorrowAmount` is wrongly commented.

Affected source code:

- FraxlendPair.sol#L187-L190

Recommended changes

```
-   /// @notice The ```toBorrtoBorrowAmountowShares``` function
+   /// @notice The ```toBorrowAmount``` function converts a giv
    /// @param _shares Shares of borrow
    /// @param _roundUp Whether to roundup during division
    function toBorrowAmount(uint256 _shares, bool _roundUp) exte
        return totalBorrow.toAmount(_shares, _roundUp);
    }
```

[N-03] Confusing variables as to how they are stored

The `FraxlendPairCore` contract constructor takes two arguments, `_configData` and `_immutables`, however, there are immutables in `_configData` and variables in `_immutables`.

This may seem confusing, since for example one expects the `TIME_LOCK_ADDRESS` to be immutable as it is defined in the `_immutables` variable, however it is possible to change it with setTimeLock.

Affected source code:

- FraxlendPairCore.sol#L174
- FraxlendPairCore.sol#L190-L191
- FraxlendPairCore.sol#L193-L197

gititGoro (judge) commented:

> Very good report quality! There were some invalid points:

- L-02. Misconfigured or misbehaving tokens are out of scope
- L-06. The precision cap is intentional. This issue was reported as a Medium risk bug by another warden and marked invalid.
- L-07. Owners resigning seems beyond the scope of the audit. FraxLend will be managed by a DAO like other Frax products. Even without explicit documentation, for a Frax product, this should be the assumed default. This is if you were unable to contact the sponsor and verify explicitly that a human

controlled EOA would be the owner, a very uncommon circumstance for a large mainnet dapp.

Gas Optimizations

For this contest, 94 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: JC, Deivitto, 0xSmartContract, ajtra, 0x1f8b, c3phas, oyc_109, ret2basic, __141345__, Bnke0x0, Chinmay, MiloTruck, Dravee, 0xkatana, ReyAdmirado, gogo, Rolezn, Tomio, erictee, Waze, pfapostol, Aymen0909, Rohan16, durianSausage, _Adam, TomJ, ignacio, 0xDjango, m_Rassska, medikko, Noah3o6, rbserver, robee, simon135, LeoS, Sm4rty, brgltd, 0xA5DF, reassor, saian, 0xNazgul, mics, kyteg, NoamYakov, flyx, SaharAP, Amithuddar, fatherOfBlocks, Fitraldys, ballx, gerdusx, Junnon, Metatron, 2997ms, carlitox477, cRat1st0s, delfin454000, newfork01, SooYa, sryysryy, chrisdior4, CodingNameKiki, Funen, 0xackermann, ak1, asutorufos, d3e4, djxploit, gzeon, jag, mrpathfindr, Ruhum, sach1r0, ElKu, Diraco, Randyyy, Yiko, a12jmx, zeesaw, 0xc0ffEE, Chom, EthLedger, IgnacioB, Lambda, dharma09, hakerbaya, ladboy233, nxrblsrpr, PaludoX0, 0xbepresent, find_a_bug, francoHacker, *and* ltyu.

Summary

	Issue	Instances
[G-01]	String should only be generated once, and saved	1
[G-02]	Remove or replace unused state variables	1
[G-03]	Multiple `address` /ID mappings can be combined into a single `mapping` of an `address` /ID to a `struct` , where appropriate	1
[G-04]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	11
[G-05]	Using `storage` instead of `memory` for structs/arrays saves gas	16
[G-06]	State variables should be cached in stack variables rather than re-reading them from storage	2

	Issue	Instances
[G-07]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	2
[G-08]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	3
[G-09]	`<array>.length` should not be looked up in every loop of a `for` -loop	7
[G-10]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	8
[G-11]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	8
[G-12]	Optimize names to save gas	6
[G-13]	Using `bool` s for storage incurs overhead	9
[G-14]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	9
[G-15]	Splitting `require()` statements that use `&&` saves gas	3
[G-16]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	13
[G-17]	Using `private` rather than `public` for constants, saves gas	8
[G-18]	Use custom errors rather than `revert()` / `require()` strings to save gas	9
[G-19]	Functions guaranteed to revert when called by normal users can be marked `payable`	7

Total: 124 instances over 19 issues

[G-01] String should only be generated once, and saved

There is 1 instance of this issue:

File: /src/contracts/FraxlendPair.sol

```
81:         return string(abi.encodePacked("FraxlendV1 - ", col
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L81

[G-02] Remove or replace unused state variables

Saves a storage slot. If the variable is assigned a non-zero value, saves Gsset (**20000 gas**). If it's assigned a zero value, saves Gsreset (**2900 gas**). If the variable remains unassigned, there is no gas savings unless the variable is `public`, in which case the compiler-generated non-payable getter deployment cost is saved. If the state variable is overriding an interface's public function, mark the variable as `constant` or `immutable` so that it does not use a storage slot

There is 1 instance of this issue:

```
File: src/contracts/FraxlendPairCore.sol

51:         string public version = "1.0.0";
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L51

[G-03] Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: src/contracts/FraxlendPairCore.sol

127       mapping(address => uint256) public userCollateralBalar
128       /// @notice Stores the balance of borrow shares for ea
129:      mapping(address => uint256) public userBorrowShares; /
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L127-L129

[G-04] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved

There are 11 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol
```

```
/// @audit _configData
/// @audit _immutables
151       constructor(
152           bytes memory _configData,
153           bytes memory _immutables,
154           uint256 _maxLTV,
155           uint256 _liquidationFee,
156           uint256 _maturityDate,
157           uint256 _penaltyRate,
158           bool _isBorrowerWhitelistActive,
159:          bool _isLenderWhitelistActive

/// @audit _path
1062      function leveragedPosition(
1063          address _swapperAddress,
1064          uint256 _borrowAmount,
1065          uint256 _initialCollateralAmount,
1066          uint256 _amountCollateralOutMin,
1067          address[] memory _path
1068      )
1069          external
1070          isNotPastMaturity
1071          nonReentrant
1072          whenNotPaused
1073          approvedBorrower
1074          isSolvent(msg.sender)
1075:         returns (uint256 _totalCollateralBalance)
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L151-L159

```
File: src/contracts/FraxlendPairDeployer.sol

/// @audit _configData
310:      function deploy(bytes memory _configData) external ret

/// @audit _name
/// @audit _configData
/// @audit _approvedBorrowers
/// @audit _approvedLenders
355       function deployCustom(
```

```
356             string memory _name,
357             bytes memory _configData,
358             uint256 _maxLTV,
359             uint256 _liquidationFee,
360             uint256 _maturityDate,
361             uint256 _penaltyRate,
362             address[] memory _approvedBorrowers,
363             address[] memory _approvedLenders
364:        ) external returns (address _pairAddress) {

/// @audit _addresses
398:        function globalPause(address[] memory _addresses) exte
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L310

```
File: src/contracts/FraxlendPair.sol

/// @audit _configData
/// @audit _immutables
45          constructor(
46              bytes memory _configData,
47              bytes memory _immutables,
48              uint256 _maxLTV,
49              uint256 _liquidationFee,
50              uint256 _maturityDate,
51              uint256 _penaltyRate,
52              bool _isBorrowerWhitelistActive,
53              bool _isLenderWhitelistActive
54          )
55              FraxlendPairCore(
56                  _configData,
57                  _immutables,
58                  _maxLTV,
59                  _liquidationFee,
60                  _maturityDate,
61                  _penaltyRate,
62                  _isBorrowerWhitelistActive,
63                  _isLenderWhitelistActive
64              )
65              ERC20("", "")
66              Ownable()
```

```
67:             Pausable()
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L45-L67

[G-05] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There are 16 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

419:             CurrentRateInfo memory _currentRateInfo = currentI

426:             VaultAccount memory _totalAsset = totalAsset;

427:             VaultAccount memory _totalBorrow = totalBorrow;

517:             ExchangeRateInfo memory _exchangeRateInfo = exchar

592:             VaultAccount memory _totalAsset = totalAsset;

611:             VaultAccount memory _totalAsset = totalAsset;

666:             VaultAccount memory _totalAsset = totalAsset;
```

```
682:          VaultAccount memory _totalAsset = totalAsset;

708:          VaultAccount memory _totalBorrow = totalBorrow;

884:          VaultAccount memory _totalBorrow = totalBorrow;

926:          VaultAccount memory _totalBorrow = totalBorrow;

965:          VaultAccount memory _totalBorrow = totalBorrow;

1204:         VaultAccount memory _totalBorrow = totalBorrow;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L419

```
File: src/contracts/FraxlendPairDeployer.sol

123:          string[] memory _deployedPairsArray = deployedPair
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L123

```
File: src/contracts/FraxlendPair.sol

236:          VaultAccount memory _totalAsset = totalAsset;

237:          VaultAccount memory _totalBorrow = totalBorrow;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L236

[G-06] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 2 instances of this issue:

```
File: src/contracts/FraxlendPairDeployer.sol

/// @audit DEFAULT_MAX_LTV on line 332
/// @audit DEFAULT_LIQ_FEE on line 333
342:           _logDeploy(_name, _pairAddress, _configData, DEFAU
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L342

[G-07] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves 113 gas

There are 2 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

773:           totalCollateral += _collateralAmount;

815:           totalCollateral -= _collateralAmount;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L773

[G-08] Add `unchecked {}` for subtractions where the

operands cannot underflow because of a previous `require()` or `if` -statement

```
require(a <= b); x = b - a => require(a <= b); unchecked { x = b - a
}
```

There are 3 instances of this issue:

```
File: src/contracts/LinearInterestRate.sol

/// @audit if-condition on line 86
88:                 _newRatePerSec = uint64(_vertexInterest + (((
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L88

```
File: src/contracts/VariableInterestRate.sol

/// @audit if-condition on line 68
69:                 uint256 _deltaUtilization = ((MIN_UTIL - _util

/// @audit if-condition on line 75
76:                 uint256 _deltaUtilization = ((_utilization - M
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/VariableInterestRate.sol#L69

[G-09] `<array>.length` should not be looked up in every loop of a `for` -loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)

- calldata arrays use `CALLDATALOAD` (3 gas)

Caching the length changes each of these to a `DUP<N>` (3 gas), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 7 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

265:          for (uint256 i = 0; i < _approvedBorrowers.length;

270:          for (uint256 i = 0; i < _approvedLenders.length; i
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L265

```
File: src/contracts/FraxlendPair.sol

289:          for (uint256 i = 0; i < _lenders.length; i++) {

308:          for (uint256 i = 0; i < _borrowers.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L289

```
File: src/contracts/FraxlendWhitelist.sol

51:           for (uint256 i = 0; i < _addresses.length; i++) {

66:           for (uint256 i = 0; i < _addresses.length; i++) {

81:           for (uint256 i = 0; i < _addresses.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/Fraxlend

Whitelist.sol#L51

[G-10] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop

There are 8 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

265:          for (uint256 i = 0; i < _approvedBorrowers.length;

270:          for (uint256 i = 0; i < _approvedLenders.length; i
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L265

```
File: src/contracts/FraxlendPair.sol

289:          for (uint256 i = 0; i < _lenders.length; i++) {

308:          for (uint256 i = 0; i < _borrowers.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L289

```
File: src/contracts/FraxlendWhitelist.sol

51:           for (uint256 i = 0; i < _addresses.length; i++) {
```

```
66:             for (uint256 i = 0; i < _addresses.length; i++) {
```

```
81:             for (uint256 i = 0; i < _addresses.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendWhitelist.sol#L51

```
File: src/contracts/libraries/SafeERC20.sol
```

```
27:                 for (i = 0; i < 32 && data[i] != 0; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/libraries/SafeERC20.sol#L27

[G-11] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs **3 gas**

There are 8 instances of this issue:

```
File: src/contracts/FraxlendPairDeployer.sol
```

```
205:                 require(deployedPairsBySalt[salt] == address(0
```

```
228:                 require(_pairAddress != address(0), "FraxlendP
```

```
253:             require(deployedPairsByName[_name] == address(0),
```

```
365:             require(_maxLTV <= GLOBAL_MAX_LTV, "FraxlendPairDe
```

```
366             require(
367                 IFraxlendWhitelist(FRAXLEND_WHITELIST_ADDRESS)
368                 "FraxlendPairDeployer: Only whitelisted addres
369:             );
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L205

```
File: src/contracts/LinearInterestRate.sol

57            require(
58                _minInterest < MAX_INT && _minInterest <= _ver
59                "LinearInterestRate: _minInterest < MAX_INT &&
60:           );

61            require(
62                _maxInterest <= MAX_INT && _vertexInterest <=
63                "LinearInterestRate: _maxInterest <= MAX_INT &
64:           );

65            require(
66                _vertexUtilization < MAX_VERTEX_UTIL && _verte
67                "LinearInterestRate: _vertexUtilization < MAX_
68:           );
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L57-L60

[G-12] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted

There are 6 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol
```

```
/// @audit initialize(), addInterest(), updateExchangeRate(), bc
46:   abstract contract FraxlendPairCore is FraxlendPairConstant
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L46

```
File: src/contracts/FraxlendPairDeployer.sol

/// @audit deployedPairsLength(), getAllPairAddresses(), getCust
44:   contract FraxlendPairDeployer is Ownable {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L44

```
File: src/contracts/FraxlendPair.sol

/// @audit assetsPerShare(), assetsOf(), getConstants(), toBorrc
41:   contract FraxlendPair is FraxlendPairCore {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L41

```
File: src/contracts/FraxlendWhitelist.sol

/// @audit setOracleContractWhitelist(), setRateContractWhitelis
30:   contract FraxlendWhitelist is Ownable {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendWhitelist.sol#L30

```
File: src/contracts/LinearInterestRate.sol
```

```
/// @audit getConstants(), requireValidInitData(), getNewRate()
32:   contract LinearInterestRate is IRateCalculator {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L32

```
File: src/contracts/VariableInterestRate.sol

/// @audit getConstants(), requireValidInitData(), getNewRate()
33:   contract VariableInterestRate is IRateCalculator {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/VariableInterestRate.sol#L33

[G-13] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past

There are 9 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

78:       mapping(address => bool) public swappers; // approved
```

```
133:      bool public immutable borrowerWhitelistActive;

134:      mapping(address => bool) public approvedBorrowers;

136:      bool public immutable lenderWhitelistActive;

137:      mapping(address => bool) public approvedLenders;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L78

```
File: src/contracts/FraxlendPairDeployer.sol

94:       mapping(address => bool) public deployedPairCustomStat
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L94

```
File: src/contracts/FraxlendWhitelist.sol

32:       mapping(address => bool) public oracleContractWhitelis

35:       mapping(address => bool) public rateContractWhitelist;

38:       mapping(address => bool) public fraxlendDeployerWhitel
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendWhitelist.sol#L32

[G-14] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**

There are 9 instances of this issue:

```
File: src/contracts/FraxlendPairDeployer.sol

130:                     i++;

158:                     i++;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L130

```
File: src/contracts/FraxlendPair.sol

289:             for (uint256 i = 0; i < _lenders.length; i++) {

308:             for (uint256 i = 0; i < _borrowers.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L289

```
File: src/contracts/FraxlendWhitelist.sol

51:              for (uint256 i = 0; i < _addresses.length; i++) {

66:              for (uint256 i = 0; i < _addresses.length; i++) {

81:              for (uint256 i = 0; i < _addresses.length; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendWhitelist.sol#L51

```
File: src/contracts/libraries/SafeERC20.sol

24:                      i++;
```

```
27:                 for (i = 0; i < 32 && data[i] != 0; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/libraries/SafeERC20.sol#L24

[G-15] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas**

There are 3 instances of this issue:

```
File: src/contracts/LinearInterestRate.sol

57              require(
58                  _minInterest < MAX_INT && _minInterest <= _ver
59                  "LinearInterestRate: _minInterest < MAX_INT &&
60:             );

61              require(
62                  _maxInterest <= MAX_INT && _vertexInterest <=
63                  "LinearInterestRate: _maxInterest <= MAX_INT &
64:             );

65              require(
66                  _vertexUtilization < MAX_VERTEX_UTIL && _verte
67                  "LinearInterestRate: _vertexUtilization < MAX_
68:             );
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L57-L60

[G-16] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html
Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed

There are 13 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

/// @audit uint64 _newRate
421:               _newRate = _currentRateInfo.ratePerSec;

/// @audit uint64 _newRate
448:                   _newRate = uint64(penaltyRate);

/// @audit uint64 _newRate
456:                   _newRate = IRateCalculator(rateContract).g

/// @audit uint128 _amountToAdjust
1005:                      _amountToAdjust = (_totalBorrow.toAmou
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L421

```
File: src/contracts/FraxlendPair.sol

/// @audit uint64 _DEFAULT_INT
175:           _DEFAULT_INT = DEFAULT_INT;

/// @audit uint16 _DEFAULT_PROTOCOL_FEE
176:           _DEFAULT_PROTOCOL_FEE = DEFAULT_PROTOCOL_FEE;
```

```
/// @audit uint128 _shares
240:          if (_shares == 0) _shares = uint128(balanceOf(addr
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L175

```
File: src/contracts/libraries/SafeERC20.sol

/// @audit uint8 i
27:               for (i = 0; i < 32 && data[i] != 0; i++) {
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/libraries/SafeERC20.sol#L27

```
File: src/contracts/LinearInterestRate.sol

/// @audit uint64 _newRatePerSec
85:               _newRatePerSec = uint64(_minInterest + ((_util

/// @audit uint64 _newRatePerSec
88:               _newRatePerSec = uint64(_vertexInterest + (((_

/// @audit uint64 _newRatePerSec
90:               _newRatePerSec = uint64(_vertexInterest);
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L85

```
File: src/contracts/VariableInterestRate.sol

/// @audit uint64 _newRatePerSec
71:               _newRatePerSec = uint64((_currentRatePerSec *

/// @audit uint64 _newRatePerSec
```

```
78:                 _newRatePerSec = uint64((_currentRatePerSec *
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/VariableInterestRate.sol#L71

[G-17] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

There are 8 instances of this issue:

```
File: src/contracts/FraxlendPairCore.sol

64:         uint256 public immutable oracleNormalization;

67:         uint256 public immutable maxLTV;

70:         uint256 public immutable cleanLiquidationFee;

71:         uint256 public immutable dirtyLiquidationFee;

95:         uint256 public immutable maturityDate;

96:         uint256 public immutable penaltyRate;

133:        bool public immutable borrowerWhitelistActive;

136:        bool public immutable lenderWhitelistActive;
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairCore.sol#L64

[G-18] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

There are 9 instances of this issue:

```
File: src/contracts/FraxlendPairDeployer.sol

205:              require(deployedPairsBySalt[salt] == address(0

228:              require(_pairAddress != address(0), "FraxlendP

253:          require(deployedPairsByName[_name] == address(0),

365:          require(_maxLTV <= GLOBAL_MAX_LTV, "FraxlendPairDe

366           require(
367               IFraxlendWhitelist(FRAXLEND_WHITELIST_ADDRESS)
368               "FraxlendPairDeployer: Only whitelisted addres
369:          );

399:          require(msg.sender == CIRCUIT_BREAKER_ADDRESS, "Ci
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L205

```
File: src/contracts/LinearInterestRate.sol

57            require(
58                _minInterest < MAX_INT && _minInterest <= _ver
59                "LinearInterestRate: _minInterest < MAX_INT &&
60:           );

61            require(
62                _maxInterest <= MAX_INT && _vertexInterest <=
63                "LinearInterestRate: _maxInterest <= MAX_INT &
64:           );
```

```
65              require(
66                  _vertexUtilization < MAX_VERTEX_UTIL && _verte
67                  "LinearInterestRate: _vertexUtilization < MAX_
68:             );
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/LinearInterestRate.sol#L57-L60

[G-19] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 7 instances of this issue:

```
File: src/contracts/FraxlendPairDeployer.sol

170:      function setCreationCode(bytes calldata _creationCode)
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPairDeployer.sol#L170

```
File: src/contracts/FraxlendPair.sol

204:      function setTimeLock(address _newAddress) external onl

234:      function withdrawFees(uint128 _shares, address _recipi
```

```
274:     function setSwapper(address _swapper, bool _approval)
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendPair.sol#L204

```
File: src/contracts/FraxlendWhitelist.sol

50:     function setOracleContractWhitelist(address[] calldata

65:     function setRateContractWhitelist(address[] calldata _

80:     function setFraxlendDeployerWhitelist(address[] calldat
```

https://github.com/code-423n4/2022-08-frax/blob/c4189a3a98b38c8c962c5ea72f1a322fbc2ae45f/src/contracts/FraxlendWhitelist.sol#L50

gititGoro (judge) commented:

> Very good report. Could have scored much higher if gas before and after numbers were provided. Nothing invalid.

Disclosures

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

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