Frax Finance: Fraxlend and Fraxferry

Security Assessment

November 22, 2022

Prepared for:
Sam Kazemian
Frax Finance

Prepared by: **Simone Monica and Robert Schneider**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Frax Finance engaged Trail of Bits to review the security of its Fraxlend and Fraxferry smart contracts. From October 10 to October 24, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with partial knowledge of the system, including access to the source code and some documentation. We performed dynamic testing of the target system, using both automated and manual processes.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	4
Low	5
Informational	2
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	1
Data Validation	8
Undefined Behavior	3

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-FXFERRY-10**
  An unsafe cast operation in the `depart` function makes it possible to create an invalid batch that cannot be processed when run.
- **TOB-FXLEND-3**
  The penalty interest rate is incorrectly applied to a period of time within the loan maturity window.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 6, 2022	Pre-project kickoff call
October 17, 2022	Status update meeting #1
October 24, 2022	Delivery of report draft
October 24, 2022	Report readout meeting
November 22, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of Frax Finance's Fraxlend and Fraxferry smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are proper access controls used in contracts with externally callable functions?
- Could an unapproved borrower mint shares or borrow assets?
- Could an attacker withdraw another user's collateral?
- Are there front-running or denial-of-service opportunities in the system?
- Could the system's funds or an individual user's funds become frozen or stuck?
- Are oracle integrations vulnerable to price manipulation or stale price reporting?
- Could an attacker gain unauthorized access to the whitelist?
- Could bridged assets be prematurely withdrawn?
- Could bridge transactions be faked or replayed?
- Could critical operations still be performed while the system is paused?

Project Targets

The engagement involved a review and testing of the targets listed below.

Fraxlend

Repository	https://github.com/FraxFinance/fraxlend-dev
Version	`2a35898ed1c303f4f878e94b61d80fcd50a3190d`
Type	Solidity
Platform	Ethereum

Fraxferry

Repository	https://github.com/FraxFinance/frax-solidity
Version	`32d1cbde2baaa35c0b8296a14e56dc06e111e345`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`FraxlendPair`**. The `FraxlendPair` contract is the primary point of interaction in the Fraxlend protocol. A single `FraxlendPair` contract represents an isolated lending market through which users borrow a single ERC20 token by depositing a different ERC20 token as collateral. This contract inherits from several others and is responsible for implementing the logic that facilitates borrowing, lending, and liquidation operations within Fraxlend. We reviewed the contract's bookkeeping code to ensure that users cannot borrow more than expected.

- **`FraxlendPairDeployer`**. This contract is responsible for deploying and initializing individual `FraxlendPair` contracts. We reviewed the contract for access control issues and other issues that could result in incorrect deployments.

- **`FraxlendWhitelist`**. The whitelist contract allows an `owner` address to set and update approved addresses for oracles, rate contracts, and deployer contracts. We manually checked the contract to ensure that events are emitted correctly and access controls are properly applied.

- **`Fraxferry`**. This contract is a bridge contract that facilitates the movement of tokens (mainly FRAX and FXS) from one chain to another. It can execute individual or batched transactions on the chain that it is deployed on, and it holds or releases assets when an authorized address submits a valid transaction for it to execute. We reviewed the contract to ensure that it performs correct accounting of users' funds and to investigate ways in which it could stop working properly.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not extensively investigate the `Fraxferry` contract's compatibility with various chains.

- We did not review the Fraxlend swapper contracts, as they were not included in the scope of this audit.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase is written in a version of Solidity that is higher than 0.8 and uses safe arithmetic. While we did not identify issues directly related to arithmetic, we found an instance of unsafe casting that should be reviewed for correctness (TOB-FXFERRY-10). Moreover, both the Fraxlend and Fraxferry systems would benefit from additional documentation and testing of arithmetic invariants.	Satisfactory
Auditing	The Fraxlend protocol emits events for critical administrative functionalities; however, the `Fraxferry` contract's events could be improved with additional details (TOB-FXFERRY-13). Additionally, it is not clear whether Frax Finance uses an incident response plan or monitoring tool.	Moderate
Authentication / Access Controls	The access controls in both systems are sufficient; however, the documentation describing each role in the system lacks detail. `Fraxferry` has roles with critical privileges; the documentation should describe who holds these roles and how they are protected.	Moderate
Complexity Management	The functions are scoped appropriately; each one has a clearly defined action to perform. The Fraxlend contracts contain appropriate inline documentation; however, `Fraxferry`'s documentation does not clearly describe the functionality of each component.	Moderate

Decentralization	`FraxlendPairDeployer`'s owner has the ability to set the code for pairs to be deployed; however, once deployed, the pair's owner can change only the swapper contracts. `Fraxferry` is a centralized bridge in which users have to trust the Frax Finance team to actually relay the transactions to the target chain.	Weak
Documentation	The Fraxlend contracts have good high-level documentation, a specification indicating the derivations for the formulas used by the system, and good use of NatSpec and inline comments. However, the system's invariants are not specified. `Fraxferry` has limited documentation with some inline comments; we recommend creating diagrams explaining the different phases of the Fraxferry system and technical documentation explaining its functionality.	Moderate
Front-Running Resistance	We did not identify any obvious front-running issues during our review. However, lending platforms that rely on price updates from oracles, like Fraxlend, are particularly vulnerable to front-running issues. Care should be taken to avoid introducing potential sandwiching or timing issues in future updates.	Satisfactory
Low-Level Manipulation	Very little low-level manipulation is performed in the codebase, except for one necessary use of `create2`. It appears to have been implemented correctly, and edge cases are properly accounted for. We identified one issue related to low-level manipulation (TOB-FXFERRY-12).	Satisfactory
Testing and Verification	The Fraxlend system contains a number of unit and integration tests. The codebase would benefit from advanced testing techniques such as fuzzing. Moreover, given the system's reliance on third-party and off-chain integrations, we feel the codebase would benefit from a more robust end-to-end testing setup in which those services can be represented by real components. `Fraxferry` has a limited number of unit and integration tests that should be expanded with fuzzing techniques.	Moderate

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of two-step process for contract ownership changes	Data Validation	**Medium**
2	Missing checks during constructor/initialization	Data Validation	**Low**
3	Incorrect application of penalty fee rate	Undefined Behavior	**Medium**
4	Improper validation of Chainlink data	Data Validation	**Low**
5	Risk of oracle outages	Configuration	**Informational**
6	Unapproved lenders could receive fTokens	Data Validation	**Low**
7	FraxlendPairDeployer can’t deploy contracts of fewer than 13000 bytes	Undefined Behavior	**Medium**
8	setCreationCode fails to overwrite _secondHalf slice if updated code size is less than 13,000 bytes	Undefined Behavior	**Undetermined**
9	Missing checks in setter functions	Data Validation	**Low**
10	Risk of invalid batches due to unsafe cast in depart function	Data Validation	**High**
11	Transactions that were already executed can be canceled	Data Validation	**Low**

12	Lack of contract existence check on low-level call	Data Validation	**Medium**
13	Events could be improved	Auditing and Logging	**Informational**

Detailed Findings

1. Lack of two-step process for contract ownership changes	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FXLEND-1
Target: `@openzeppelin/contracts/access/Ownable.sol`	

Description
The owner of a contract that inherits from the `FraxlendPairCore` contract can be changed through a call to the `transferOwnership` function. This function internally calls the `_setOwner` function, which immediately sets the contract's new owner. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}

function _setOwner(address newOwner) private {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Figure 1.1: OpenZeppelin's `OwnableUpgradeable` contract

Exploit Scenario
Alice, a Frax Finance administrator, invokes the `transferOwnership` function to change the address of an existing contract's owner but mistakenly submits the wrong address. As a result, ownership of the contract is permanently lost.

Recommendations
Short term, implement ownership transfer operations that are executed in a two-step process, in which the owner proposes a new address and the transfer is completed once the new address has executed a call to accept the role.

Long term, identify and document all possible actions that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

2. Missing checks of constructor/initialization parameters

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FXLEND-2
Target: `src/contracts/FraxlendPairCore.sol`	

Description

In the Fraxlend protocol's `constructor` function, various settings are configured; however, two of the configuration parameters do not have checks to validate the values that they are set to.

First, the `_liquidationFee` parameter does not have an upper limit check:

```
    constructor(
        bytes memory _configData,
        bytes memory _immutables,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        uint256 _maturityDate,
        uint256 _penaltyRate,
        bool _isBorrowerWhitelistActive,
        bool _isLenderWhitelistActive
     ) {
      [...]
      cleanLiquidationFee = _liquidationFee;
      dirtyLiquidationFee = (_liquidationFee * 90000) / LIQ_PRECISION; //
90% of clean fee
```

Figure 2.1: The `constructor` function's parameters in `FraxlendPairCore.sol#L193-L194`

Second, the Fraxlend system can work with one or two oracles; however, there is no check to ensure that at least one oracle is set:

```
   constructor(
        bytes memory _configData,
        bytes memory _immutables,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        uint256 _maturityDate,
        uint256 _penaltyRate,
        bool _isBorrowerWhitelistActive,
        bool _isLenderWhitelistActive
     ) {
      // [...]
      // Oracle Settings
      {
      IFraxlendWhitelist _fraxlendWhitelist =
IFraxlendWhitelist(FRAXLEND_WHITELIST_ADDRESS);
      // Check that oracles are on the whitelist
      if (_oracleMultiply != address(0) &&
!_fraxlendWhitelist.oracleContractWhitelist(_oracleMultiply)) {
        revert NotOnWhitelist(_oracleMultiply);
      }

      if (_oracleDivide != address(0) &&
!_fraxlendWhitelist.oracleContractWhitelist(_oracleDivide)) {
        revert NotOnWhitelist(_oracleDivide);
      }

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

Figure 2.2: The `constructor` function's body in `FraxlendPairCore.sol#L201-L214`

Exploit Scenario

Bob deploys a custom pair with a misconfigured `_configData` argument in which no oracle is set. As a consequence, the exchange rate is incorrect.

Recommendations

Short term, add an upper limit check for the `_liquidationFee` parameter, and add a check for the `_configData` parameter to ensure that at least one oracle is set. The checks can be added in either the `FraxlendPairCore` contract or the `FraxlendPairDeployer` contract.

Long term, add appropriate requirements to values that users set to decrease the likelihood of user error.

3. Incorrect application of penalty fee rate

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FXLEND-3
Target: `src/contracts/FraxlendPairCore.sol`	

Description

A Fraxlend pair can have a maturity date, after which a penalty rate is applied to the interest to be paid by the borrowers. However, the penalty rate is also applied to the amount of time immediately before the maturity date.

As shown in figure 3.1, the `_addInterest` function checks whether a pair is past maturity. If it is, the function sets the new rate (the `_newRate` parameter) to the penalty rate (the `penaltyRate` parameter) and then uses it to calculate the matured interest. The function should apply the penalty rate only to the time between the maturity date and the current time; however, it also applies the penalty rate to the time between the last interest accrual (`_deltaTime`) and the maturity date, which should be subject only to the normal interest rate.

```
function _addInterest()
  // [...]
  uint256 _deltaTime = block.timestamp - _currentRateInfo.lastTimestamp;
  // [...]
  if (_isPastMaturity()) {
      _newRate = uint64(penaltyRate);
  } else {
  // [...]
  // Effects: bookkeeping
  _currentRateInfo.ratePerSec = _newRate;
  _currentRateInfo.lastTimestamp = uint64(block.timestamp);
  _currentRateInfo.lastBlock = uint64(block.number);

  // Calculate interest accrued
  _interestEarned = (_deltaTime * _totalBorrow.amount * _currentRateInfo.ratePerSec)
/ 1e18;
```

Figure 3.1: The `_addInterest` function in `FraxlendPairCore.sol#L406-L494`

Exploit Scenario

A Fraxlend pair's maturity date is 100, the delta time (the last time interest accrued) is 90, and the current time is 105. Alice decides to repay her debt. The `_addInterest` function is executed, and the penalty rate is also applied to the 10 units of time between the last interest accrual and the maturity date. As a result, Alice owes more in interest than she should.

Recommendations

Short term, modify the associated code so that if the `_isPastMaturity` branch is taken and the `_currentRateInfo.lastTimestamp` value is less than `maturityDate` value, the penalty interest rate is applied only for the amount of time after the maturity date.

Long term, identify edge cases that could occur in the interest accrual process and implement unit tests and fuzz tests to validate them.

4. Improper validation of Chainlink data	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FXLEND-4
Target: `src/contracts/FraxlendPairCore.sol`	

Description

The current validation of the values returned by Chainlink's `latestRoundData` function could result in the use of stale data.

The `latestRoundData` function returns the following values: the `answer`, the `roundId` (which represents the current round), the `answeredInRound` value (which corresponds to the round in which the answer was computed), and the `updatedAt` value (which is the timestamp of when the round was updated). An `updatedAt` value of zero means that the round is not complete and should not be used. An `answeredInRound` value that is less than the `roundId` could indicate stale data.

However, the `_updateExchangeRate` function does not check for these conditions.

```
function _updateExchangeRate() internal returns (uint256 _exchangeRate) {
     // [...]
     uint256 _price = uint256(1e36);
     if (oracleMultiply != address(0)) {
         (, int256 _answer, , , ) =
AggregatorV3Interface(oracleMultiply).latestRoundData();
         if (_answer <= 0) {
             revert OracleLTEZero(oracleMultiply);
         }
         _price = _price * uint256(_answer);
     }

     if (oracleDivide != address(0)) {
         (, int256 _answer, , , ) =
AggregatorV3Interface(oracleDivide).latestRoundData();
         if (_answer <= 0) {
             revert OracleLTEZero(oracleDivide);
         }
         _price = _price / uint256(_answer);
     }

     // [...]
 }
```

Figure 4.1: The `_updateExchangeRate` function in `FraxlendPairCore.sol#L513-L544`

Exploit Scenario

Chainlink is not updated correctly in the current round, and Eve, who should be liquidated with the real collateral asset price, is not liquidated because the price reported is outdated and is higher than it is in reality.

Recommendations

Short term, have `_updateExchangeRate` perform the following sanity check: `require(updatedAt != 0 && answeredInRound == roundId)`. This check will ensure that the round has finished and that the pricing data is from the current round.

Long term, when integrating with third-party protocols, make sure to accurately read their documentation and implement the appropriate sanity checks.

5. Risk of oracle outages

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-FXLEND-5
Target: `src/contracts/FraxlendPairCore.sol`	

Description

Under extreme market conditions, the Chainlink oracle may cease to work as expected, causing unexpected behavior in the Fraxlend protocol.

Such oracle issues have occurred in the past. For example, during the LUNA market crash, the Venus protocol was exploited because Chainlink stopped providing up-to-date prices. The interruption occurred because the price of LUNA dropped below the minimum price (`minAnswer`) allowed by the LUNA/USD price feed on the BNB chain. As a result, all oracle updates reverted. Chainlink's automatic circuit breakers, which will pause price feeds during extreme market conditions, could pose similar problems.

Note that these kinds of events cannot be tracked on-chain. If a price feed is paused, `updatedAt` will still be greater than zero, and `answeredInRound` will still be equal to `roundId`.

Therefore, the Frax Finance team should implement an off-chain monitoring solution to detect any anomalous behavior exhibited by Chainlink oracles.

Recommendations

Short term, implement an off-chain monitoring solution that checks for the following conditions and issues alerts if they occur, as they may be indicative of abnormal market events:

- An asset price that is approaching the `minAnswer` or `maxAnswer` value
- The suspension of a price feed by an automatic circuit breaker
- Any large deviations in the price of an asset

6. Unapproved lenders could receive fTokens	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FXLEND-6
Target: `src/contracts/FraxlendPairCore.sol`	

Description

A Fraxlend custom pair can include a list of approved lenders; these are the only lenders who can deposit the underlying asset into the given pair and receive the corresponding fTokens. However, the system does not perform checks when users transfer fTokens; as a result, approved lenders could send fTokens to unapproved addresses.

Although unapproved addresses can only redeem fTokens sent to them—meaning this issue is not security-critical—the ability for approved lenders to send fTokens to unapproved addresses conflicts with the currently documented behavior.

```
function deposit(uint256 _amount, address _receiver)
     external
     nonReentrant
     isNotPastMaturity
     whenNotPaused
     approvedLender(_receiver)
     returns (uint256 _sharesReceived)
{...}
```

Figure 6.1: The `deposit` function in `FraxlendPairCore.sol#L587-L594`

Exploit Scenario

Bob, an approved lender, deposits 100 asset tokens and receives 90 fTokens. He then sends the fTokens to an unapproved address, causing other users to worry about the state of the protocol.

Recommendations

Short term, override the `_beforeTokenTransfer` function by applying the `approvedLender` modifier to it. Alternatively, document the ability for approved lenders to send fTokens to unapproved addresses.

Long term, when applying access controls to token owners, make sure to evaluate all the possible ways in which a token can be transferred and document the expected behavior.

7. FraxlendPairDeployer cannot deploy contracts of fewer than 13,000 bytes	
Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FXLEND-7
Target: `src/contracts/FraxlendPairDeployer.sol`	

Description

The `FraxlendPairDeployer` contract, which is used to deploy new pairs, does not allow contracts that contain less than 13,000 bytes of code to be deployed.

To deploy new pairs, users call the `deploy` or `deployCustom` function, which then internally calls `_deployFirst`. This function uses the `create2` opcode to create a contract for the pair by concatenating the bytecode stored in `contractAddress1` and `contractAddress2`.

The `setCreationCode` function, which uses solmate's SSTORE2 library to store the bytecode for use by `create2`, splits the bytecode into two separate contracts (`contractAddress1` and `contractAddress2`) if the `_creationCode` size is greater than 13,000.

```
    function setCreationCode(bytes calldata _creationCode) external onlyOwner {
        bytes memory _firstHalf = BytesLib.slice(_creationCode, 0, 13000);
        contractAddress1 = SSTORE2.write(_firstHalf);
        if (_creationCode.length > 13000) {
            bytes memory _secondHalf = BytesLib.slice(_creationCode, 13000,
_creationCode.length - 13000);
            contractAddress2 = SSTORE2.write(_secondHalf);
        }
    }
```

Figure 7.1: The `setCreationCode` function in `FraxlendPairDeployer.sol#L173-L180`

The first problem is that if the `_creationCode` size is less than 13,000, `BytesLib.slice` will revert with the `slice_outOfBounds` error, as shown in figure 7.2.

```
    function slice(
         bytes memory _bytes,
         uint256 _start,
         uint256 _length
     ) internal pure returns (bytes memory) {
         require(_length + 31 >= _length, "slice_overflow");
         require(_bytes.length >= _start + _length, "slice_outOfBounds");
```

Figure 7.2: The `BytesLib.slice` function from the `solidity-bytes-utils` library

Assuming that the first problem does not exist, another problem arises from the use of `SSTORE2.read` in the `_deployFirst` function (figure 7.3). If the creation code was less than 13,000 bytes, `contractAddress2` would be set to `address(0)`. This would cause the `SSTORE2.read` function's `pointer.code.length - DATA_OFFSET` computation, shown in figure 7.4, to underflow, causing the `SSTORE2.read` operation to panic.

```
function _deployFirst(
   // [...]
 ) private returns (address _pairAddress) {
     {
         // [...]
         bytes memory _creationCode = BytesLib.concat(
             SSTORE2.read(contractAddress1),
             SSTORE2.read(contractAddress2)
         );
```

Figure 7.3: The `_deployFirst` function in `FraxlendPairDeployer.sol#L212-L231`

```
   uint256 internal constant DATA_OFFSET = 1;
   function read(address pointer) internal view returns (bytes memory) {
        return readBytecode(pointer, DATA_OFFSET, pointer.code.length -
DATA_OFFSET);
    }
```

Figure 7.4: The `SSTORE2.read` function from the solmate library

Exploit Scenario

Bob, the `FraxlendPairDeployer` contract's owner, wants to set the creation code to be a contract with fewer than 13,000 bytes. When he calls `setCreationCode`, it reverts.

Recommendations

Short term, make the following changes:

- In `setCreationCode`, in the line that sets the `_firstHalf` variable, replace `13000` in the third argument of `BytesLib.slice` with `min(13000, _creationCode.length)`.

- In `_deployFirst`, add a check to ensure that the `SSTORE2.read(contractAddress2)` operation executes only if `contractAddress2` is not `address(0)`.

Alternatively, document the fact that it is not possible to deploy contracts with fewer than 13,000 bytes.

Long term, improve the project's unit tests and fuzz tests to check that the functions behave as expected and cannot unexpectedly revert.

8. setCreationCode fails to overwrite _secondHalf slice if updated code size is less than 13,000 bytes

Severity: **Undetermined**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FXLEND-8
Target: `src/contracts/FraxlendPairDeployer.sol`	

Description

The `setCreationCode` function permits the owner of `FraxlendPairDeployer` to set the bytecode that will be used to create contracts for newly deployed pairs. If the `_creationCode` size is greater than 13,000 bytes, it will be split into two separate contracts (`contractAddress1` and `contractAddress2`). However (assuming that TOB-FXLEND-7 were fixed), if a `FraxlendPairDeployer` owner were to change the creation code from one of greater than 13,000 bytes to one of fewer than 13,000 bytes, `contractAddress2` would not be reset to `address(0)`; therefore, `contractAddress2` would still contain the second half of the previous creation code.

```
  function setCreationCode(bytes calldata _creationCode) external onlyOwner {
       bytes memory _firstHalf = BytesLib.slice(_creationCode, 0, 13000);
       contractAddress1 = SSTORE2.write(_firstHalf);
       if (_creationCode.length > 13000) {
           bytes memory _secondHalf = BytesLib.slice(_creationCode, 13000,
_creationCode.length - 13000);
           contractAddress2 = SSTORE2.write(_secondHalf);
       }
   }
```

Figure 8.1: The `setCreationCode` function in `FraxlendPairDeployer.sol#L173-L180`

Exploit Scenario

Bob, `FraxlendPairDeployer`'s owner, changes the creation code from one of more than 13,000 bytes to one of less than 13,000 bytes. As a result, `deploy` and `deployCustom` deploy contracts with unexpected bytecode.

Recommendations

Short term, modify the `setCreationCode` function so that it sets `contractAddress2` to `address(0)` at the beginning of the function.

Long term, improve the project's unit tests and fuzz tests to check that the functions behave as expected and cannot unexpectedly revert.

9. Missing checks in setter functions	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FXFERRY-9
Target: `Fraxferry.sol`	

Description

The `setFee` and `setMinWaitPeriods` functions do not have appropriate checks.

First, the `setFee` function does not have an upper limit, which means that the `Fraxferry` owner can set enormous fees. Second, the `setMinWaitPeriods` function does not require the new value to be at least one hour. A minimum waiting time of less than one hour would invalidate important safety assumptions. For example, in the event of a reorganization on the source chain, the minimum one-hour waiting time ensures that only transactions after the reorganization are ferried (as described in the code comment in figure 9.1).

```
** - Reorgs on the source chain. Avoided, by only returning the transactions on the
source chain that are at least one hour old.
** - Rollbacks of optimistic rollups. Avoided by running a node.
** - Operators do not have enough time to pause the chain after a fake proposal.
Avoided by requiring a minimal amount of time between sending the proposal and
executing it.

// [...]

function setFee(uint _FEE) external isOwner {
    FEE=_FEE;
    emit SetFee(_FEE);
 }

 function setMinWaitPeriods(uint _MIN_WAIT_PERIOD_ADD, uint
_MIN_WAIT_PERIOD_EXECUTE) external isOwner {
    MIN_WAIT_PERIOD_ADD=_MIN_WAIT_PERIOD_ADD;
    MIN_WAIT_PERIOD_EXECUTE=_MIN_WAIT_PERIOD_EXECUTE;
    emit SetMinWaitPeriods(_MIN_WAIT_PERIOD_ADD, _MIN_WAIT_PERIOD_EXECUTE);
 }
```

Figure 9.1: The `setFee` and `setMinWaitPeriods` functions in `Fraxferry.sol#L226-L235`

Exploit Scenario

Bob, `Fraxferry`'s owner, calls `setMinWaitPeriods` with a `_MIN_WAIT_PERIOD_ADD` value lower than 3,600 (one hour), invalidating the waiting period's protection regarding chain reorganizations.

Recommendations

Short term, add an upper limit check to the `setFee` function; add a check to the `setMinWaitPeriods` function to ensure that `_MIN_WAIT_PERIOD_ADD` and `_MIN_WAIT_PERIOD_EXECUTE` are at least 3,600 (one hour).

Long term, make sure that configuration variables can be set only to valid values.

10. Risk of invalid batches due to unsafe cast in depart function

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FXFERRY-10
Target: `Fraxferry.sol`	

Description

The `depart` function performs an unsafe cast operation that could result in an invalid batch.

Users who want to send tokens to a certain chain use the various `embark*` functions. These functions eventually call `embarkWithRecipient`, which adds the relevant transactions to the `transactions` array.

```
function embarkWithRecipient(uint amount, address recipient) public notPaused {
    // [...]

transactions.push(Transaction(recipient,amountAfterFee,uint32(block.timestamp)));
 }
```

Figure 10.1: The `embarkWithRecipient` function in `Fraxferry.sol#L127-L135`

At a certain point, the captain role calls `depart` with the `start` and `end` indices within transactions to specify the transactions inside of a batch. However, the `depart` function performs an unsafe cast operation when creating the new batch; because of this unsafe cast operation, an end value greater than 2 `**` 64 would be cast to a value lower than the `start` value, breaking the invariant that end is greater than or equal to `start`.

```
  function depart(uint start, uint end, bytes32 hash) external notPaused isCaptain {
      require ((batches.length==0 && start==0) || (batches.length>0 &&
start==batches[batches.length-1].end+1),"Wrong start");
      require (end>=start,"Wrong end");
      batches.push(Batch(uint64(start),uint64(end),uint64(block.timestamp),0,hash));
      emit Depart(batches.length-1,start,end,hash);
   }
```

Figure 10.2: The `depart` function in `Fraxferry.sol#L155-L160`

If the resulting incorrect batch is not disputed by the crew member roles, which would cause the system to enter a paused state, the first officer role will call `disembark` to actually execute the transactions on the target chain. However, the `disembark` function's third check, highlighted in figure 10.3, on the invalid transaction will fail, causing the transaction to revert and the system to stop working until the incorrect batch is removed with a call to `removeBatches`.

```
function disembark(BatchData calldata batchData) external notPaused isFirstOfficer {
    Batch memory batch = batches[executeIndex++];
    require (batch.status==0,"Batch disputed");
    require (batch.start==batchData.startTransactionNo,"Wrong start");
    require (batch.start+batchData.transactions.length-1==batch.end,"Wrong size");
    require (block.timestamp-batch.departureTime>=MIN_WAIT_PERIOD_EXECUTE,"Too
soon");
    // [...]
 }
```

Figure 10.3: The `disembark` function in `Fraxferry.sol#L162-L178`

Exploit Scenario

Bob, `Fraxferry`'s captain, calls `depart` with an `end` value greater than 2 `**` 64, which is cast to a value less than `start`. As a consequence, the system becomes unavailable either because the crew members called `disputeBatch` or because the `disembark` function reverts.

Recommendations

Short term, replace the unsafe cast operation in the depart function with a safe cast operation to ensure that the `end >= start` invariant holds.

Long term, implement robust unit tests and fuzz tests to check that important invariants hold.

11. Transactions that were already executed can be canceled

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FXFERRY-11
Target: `Fraxferry.sol`	

Description

The `Fraxferry` contract's owner can call the `jettison` or `jettisonGroup` functions to cancel a transaction or a series of transactions, respectively. However, these functions incorrectly use the `executeIndex` variable to determine whether the given transaction has already been executed. As a result, it is possible to cancel an already executed transaction.

The problem is that `executeIndex` tracks executed batches, not executed transactions. Because a batch can contain more than one transaction, the check in the `_jettison` function (figure 11.1) does not work correctly.

```
function _jettison(uint index, bool cancel) internal {
    require (index>=executeIndex,"Transaction already executed");
    cancelled[index]=cancel;
    emit Cancelled(index,cancel);
 }

 function jettison(uint index, bool cancel) external isOwner {
    _jettison(index,cancel);
 }

 function jettisonGroup(uint[] calldata indexes, bool cancel) external isOwner {
    for (uint i=0;i<indexes.length;++i) {
       _jettison(indexes[i],cancel);
    }
 }
```

Figure 11.1: The `_jettison`, `jettison`, and `jettisonGroup` functions in `Fraxferry.sol#L208-L222`

Note that canceling a transaction that has already been executed does not cancel its effects (i.e., the tokens were already sent to the receiver).

Exploit Scenario

Two batches of 10 transactions are executed; `executeIndex` is now 2. Bob, `Fraxferry`'s owner, calls `jettison` with an index value of 13 to cancel one of these transactions. The call to `jettison` should revert, but it is executed correctly. The emitted `Cancelled` event shows that a transaction that had already been executed was canceled, confusing the off-chain monitoring system.

Recommendations

Short term, use a different index in the `jettison` and `jettisonGroup` functions to track executed transactions.

Long term, implement robust unit tests and fuzz tests to check that important invariants hold.

12. Lack of contract existence check on low-level call	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FXFERRY-12
Target: `Fraxferry.sol`	

Description

The `execute` function includes a low-level `call` operation without a contract existence check; `call` operations return `true` even if the `_to` address is not a contract, so it is important to include contract existence checks alongside such operations.

```
// Generic proxy
 function execute(address _to, uint256 _value, bytes calldata _data) external
isOwner returns (bool, bytes memory) {
    (bool success, bytes memory result) = _to.call{value:_value}(_data);
    return (success, result);
 }
```

Figure 12.1: The `execute` function in `Fraxferry.sol#L274-L278`

The Solidity documentation includes the following warning:

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 12.2: A snippet of the Solidity documentation detailing unexpected behavior related to `call`

Exploit Scenario

Bob, `Fraxferry`'s owner, calls `execute` with `_to` set to an address that should be a contract; however, the contract was self-destructed. Even though the contract at this address no longer exists, the operation still succeeds.

Recommendations

Short term, implement a contract existence check before the `call` operation in the `execute` function. If the `call` operation is expected to send ETH to an externally owned address, ensure that the check is performed only if the `_data.length` is not zero.

Long term, carefully review the Solidity documentation, especially the "Warnings" section.

13. Events could be improved

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-FXFERRY-13
Target: `Fraxferry.sol`	

Description

The events declared in the `Fraxferry` contract could be improved to be more useful to users and monitoring systems.

Certain events could be more useful if they used the `indexed` keyword. For example, in the `Embark` event, the `indexed` keyword could be applied to the `sender` parameter.

Additionally, `SetCaptain`, `SetFirstOfficier`, `SetFee`, and `SetMinWaitPeriods` could be more useful if they emitted the previous value in addition to the newly set one.

```
  event Embark(address sender, uint index, uint amount, uint amountAfterFee, uint
timestamp);
   event Disembark(uint start, uint end, bytes32 hash);
   event Depart(uint batchNo,uint start,uint end,bytes32 hash);
   event RemoveBatch(uint batchNo);
   event DisputeBatch(uint batchNo, bytes32 hash);
   event Cancelled(uint index, bool cancel);
   event Pause(bool paused);
   event OwnerNominated(address newOwner);
   event OwnerChanged(address previousOwner,address newOwner);
   event SetCaptain(address newCaptain);
   event SetFirstOfficer(address newFirstOfficer);
   event SetCrewmember(address crewmember,bool set);
   event SetFee(uint fee);
   event SetMinWaitPeriods(uint minWaitAdd,uint minWaitExecute);
```

Figure 13.1: Events declared in `Fraxferry.sol#L83-L96`

Recommendations

Short term, add the `indexed` keyword to any events that could benefit from it; modify events that report on setter operations so that they report the previous values in addition to the newly set values.

Summary of Recommendations

The Frax Finance Fraxlend and Fraxferry contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that Frax Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Further identify system invariants throughout the codebase and incorporate Echidna into the development process to perform extensive smart contract fuzzing to test such invariants.
- Add more robust testing and further documentation for the `Fraxferry` contract.
- Review all the uses of type casting in the codebase, such as the one described in TOB-FXFERRY-10, to identify any uses that could be unsafe.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Some variables can be refactored as constants.** Use constants wherever possible, which is safer and can reduce gas costs for the system and its users.

```
uint256 public DEFAULT_LIQ_FEE = 10000;
```

Figure C.1: The DEFAULT_LIQ_FEE variable in `FraxlendPairDeployer#L51`

```
uint256 public DEFAULT_MAX_LTV = 75000;
```

Figure C.2: The DEFAULT_MAX_LTV variable in `FraxlendPairDeployer#L49`

```
uint256 public GLOBAL_MAX_LTV = 1e8;
```

Figure C.3: The GLOBAL_MAX_LTV variable in `FraxlendPairDeployer#L50`

```
string public version = “1.0.0”;
```

Figure C.4: The `version` variable in `FraxlendPairCore#L51`

- **Some variables can be refactored as immutables.** Use immutables wherever possible, which is safer and can reduce gas costs for the system and its users.

```
address public CIRCUIT_BREAKER_ADDRESS;
```

Figure C.5: The CIRCUIT_BREAKER_ADDRESS variable in `FraxlendPairDeployer#L57`

```
address public COMPTROLLER_ADDRESS;
```

Figure C.6: The COMPTROLLER_ADDRESS variable in `FraxlendPairCore#L58`

- **Floating pragmas allow contracts to be compiled by different versions of the compiler, which can introduce bugs.** Consider reducing the use of various compiler versions to avoid unexpected interactions between older and newer versions of the compilers.

```
pragma solidity ^0.8.16;
```

Figure C.7: The `pragma` declaration in `FraxlendPair#L2`

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Potential System Invariants

Frax Finance's Fraxlend system relies on several invariants that must be maintained to ensure the system's integrity. In this section, we have attempted to identify several of these invariants for the Frax Finance development team to consider; we recommend that the Frax Finance team add any missing invariants to this list and examine their implications within the codebase. Additionally, we recommend thoroughly testing these invariants using both common and advanced testing techniques, such as fuzzing, symbolic execution, unit tests, and end-to-end tests.

Global Properties

- `totalBorrow.shares` is equal to the sum of the entries in `userBorrowShares[]`.
- `totalCollateral` is equal to the sum of the entries in `userCollateralBalance[]`.
- `totalBorrow.amount` is greater than or equal to `totalBorrow.shares`.
- `totalAsset.amount` is greater than or equal to `totalAsset.shares`.

Deposit

- If `_amount` equals zero, the shares received equal zero.
- The `msg.sender` account's asset balance always decreases by `_amount`.
- The system account's asset balance always increases by `_amount`.

Redeem

- If `_shares` equals zero, the assets received equal zero.
- The `msg.sender` account's share balance always decreases by `_shares`.
- The receiver account's asset balance always increases by `_amountToReturn`.

Borrow Assets

- If `_borrowAmount` does not equal zero, `userBorrowShares[msg.sender]` increases by `_shares`.
- `totalBorrow.shares` increases by `_shares`.
- The system account's asset balance decreases by `_borrowAmount` if `msg.sender` is not the system account address.

- The receiver account's asset balance increases by `_borrowAmount` if `msg.sender` is not the system account address.

Repay Assets

- If `_shares` does not equal zero, the `msg.sender` account's asset balance decreases by `_amountToRepay` if `msg.sender` is not the system account address.

- The system account's asset balance increases by `_amountToRepay` if `msg.sender` is not the system account address.

F. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
  - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Frax Finance will compensate users affected by an issue (if any).**
  - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
  - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
  - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.