Zellic

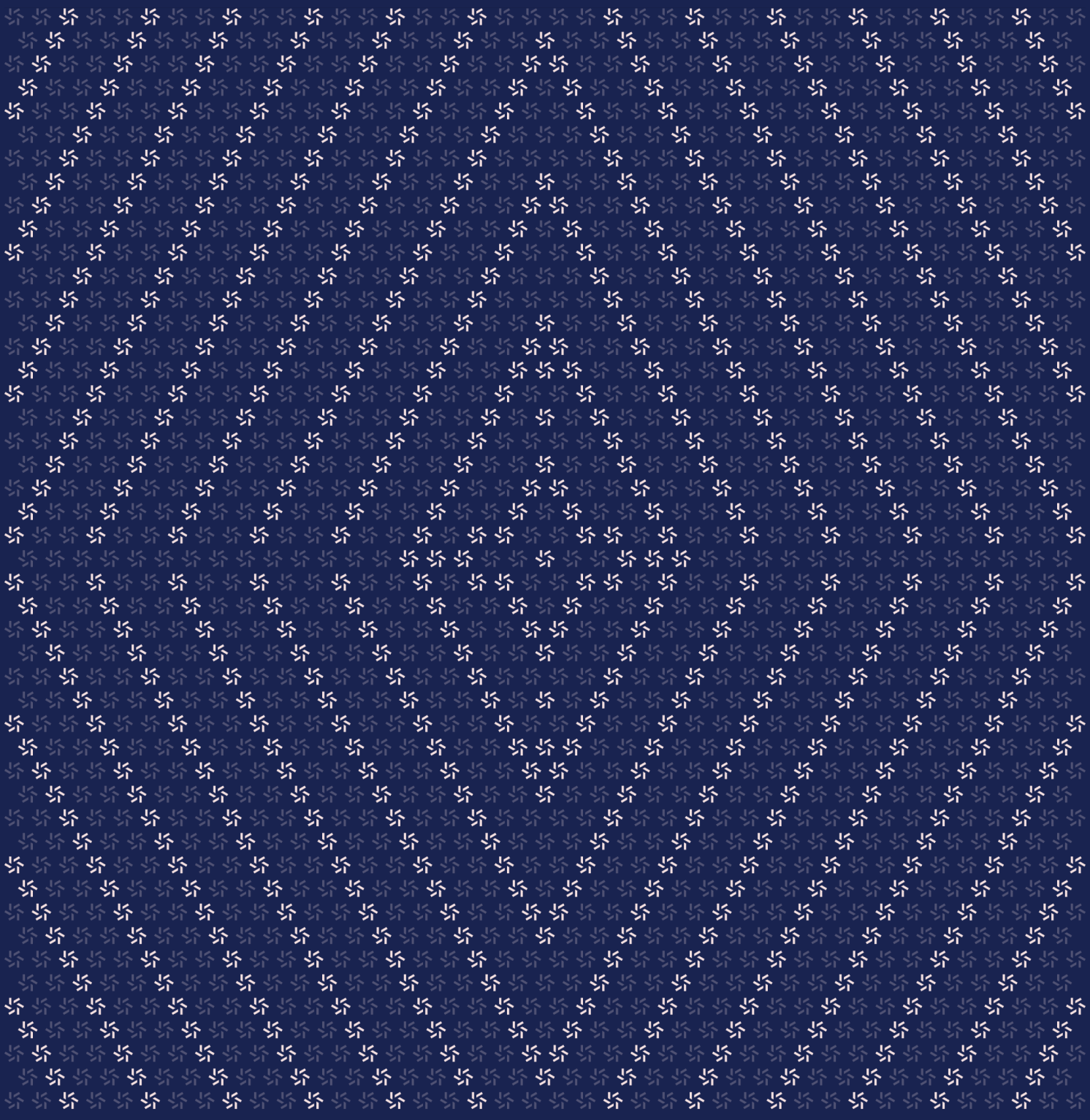
**Prepared for**
Sam K
Thomas C
FraxNet

**Prepared by**
Katerina Belotskaia
Jaeeu Kim
Zellic

**July 7, 2025**

# Frax

## Smart Contract Security Assessment



# Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Overview

## 1.1. Executive Summary

Zellic conducted a security assessment for FraxNet from June 23rd to July 7th, 2025. During this engagement, Zellic reviewed Frax's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any security risks in the current proxy setup or the proposed changes to its layout?
- How does the system interact with third-party message-passing mechanisms, and are these interactions secure?
- Is the minting and redeeming process secure, and is ownership of contracts deployed from the factory appropriately controlled?
- Is it possible for an attacker to mint frxUSD without providing collateral?
- Can a malicious actor exploit or drain contracts like the deposit contract?
- Could someone manipulate the frxUSD stablecoin exchange rate through malicious behavior?

## 1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Off-chain components

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

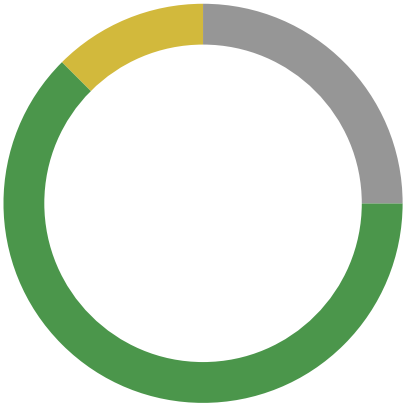
## 1.4. Results

During our assessment on the scoped Frax contracts, we discovered eight findings. No critical issues were found. One finding was of medium impact, five were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of FraxNet in the Discussion section (4. ↗).

### Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 1 |
| Low | 5 |
| Informational | 2 |



# 2. Introduction

## 2.1. About Frax

FraxNet contributed the following description of Frax:

> Frax Finance is building scalable stablecoin infrastructure for the next generation of finance. This audit is for the “FraxNet system” and supporting contracts. A constellation of contracts that service the frxUSD stablecoins mint/redeem system against other stablecoins like USDC and RWA tokens.

## 2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

**Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

**Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

**Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

**Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

## 2.3. Scope

The engagement involved a review of the following targets:

### Frax Contracts

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Target** | fraxnet-dev |
| **Repository** | https://github.com/FraxFinance/fraxnet-dev/ ↗ |
| **Version** | `d0e4ef06dda7aac4da3890f6e9a68d98fd4def82` |
| **Programs** | `FraxNetDepositFactory.sol`<br>`FraxNetDeposit.sol`<br>`FraxProxy.sol`<br>`FraxNetDepositV2.sol`<br>`Constants.sol`<br>`FraxBeacon.sol`<br>`RWARedemptionCoordinator.sol`<br>`deps/FrxUSD.sol`<br>`deps/frxUsd_fraxtal.sol`<br>`deps/FrxUSDCustodian.sol`<br>`interfaces/IRemoteHop.sol`<br>`interfaces/AggregatorV3Interface.sol`<br>`interfaces/ITokenMessenger.sol`<br>`interfaces/IFrxUSDCustodian.sol`<br>`interfaces/IProxy.sol`<br>`interfaces/IRWAUSDCRedeemer.sol`<br>`interfaces/IRWAIssuer.sol`<br>`interfaces/IMessageTransmitter.sol` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3 person-weeks. The assessment was conducted by two consultants over the course of 1.5 calendar weeks.

### Contact Information

The following project managers were associated with the engagement:

**Jacob Goreski**
Engagement Manager
jacob@zellic.io

**Chad McDonald**
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

**Katerina Belotskaia**
Engineer
kate@zellic.io

**Jaeeu Kim**
Engineer
jaeeu@zellic.io

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **June 23, 2025** | Start of primary review period |
| **June 24, 2025** | Kick-off call |
| **July 7, 2025** | End of primary review period |

# 3. Detailed Findings

## 3.1. Missing slippage check in `shuffleToRwa`

| Target | FrxUSDCustodian | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Medium |

### Description

While the protocol performs a slippage check in the `redeemRWA` function of the RWARedemptionCoordinator contract after an RWA-to-USDC swap, there is no equivalent check in the `shuffleToRwa` function of the FrxUSDCustodian contract after a USDC-to-RWA swap.

```
function shuffleToRwa(uint256 amount) public {
    IRWAIssuer rwaIssuer
    = IRWAIssuer(0x43415eB6ff9DB7E26A15b704e7A3eDCe97d31C4e);
    address rwaCustodian = 0x5fbAa3A3B489199338fbD85F7E3D444dc0504F33;

    if (!isApprovedOperator[msg.sender]) revert NotOperator();
    if (custodianTkn.balanceOf(address(this)) - amount < minAfterShuffle)
    revert AmountTooHigh();

    custodianTkn.approve(address(rwaIssuer), amount);
    rwaIssuer.subscribe(rwaCustodian, amount, address(custodianTkn));
}
```

### Impact

The lack of a slippage check leaves the swap outcome unbounded, potentially resulting in an unfavorable swap result.

### Recommendations

We recommend adding a slippage check to the `shuffleToRwa` function to ensure the swap result is within an acceptable threshold, consistent with how slippage is handled in `redeemRWA`.

### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit 036bdc5b ↗.

## 3.2. Frozen tokens cannot be burned by owner

| | | | |
|---|---|---|---|
| **Target** | FrxUSD | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

The `_update` function is invoked during all actions, including transfers, minting, and burning tokens. It has been extended to block any activity involving frozen sender or receiver addresses.

```
function _update(address from, address to, uint256 value) internal override {
    if (isFrozen[from]) revert IsFrozen();
    if (isFrozen[to]) revert IsFrozen();
    if (isPaused) revert IsPaused();
    super._update(from, to, value);
}
```

Additionally, the FrxUSD contract implements the `burnMany` and `burn` functions that allow the contract owner to burn tokens from arbitrary accounts. However, since `_update` is also called internally during these owner-only operations, frozen tokens cannot be burned unless the account is first unfrozen.

### Impact

This behavior might be unintended, especially if the owner is expected to have full control over burning tokens regardless of freeze status. For example, if assets frozen due to a hack need to be wiped, the operation cannot be executed.

### Recommendations

Consider modifying the `_update` function to allow burning frozen tokens when the caller is the owner account.

### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit `59ba68bf` ↗.

### 3.3. The `maxSlippage` is not validated during initialization

| Target | RWARedemptionCoordinator | | |
|---|---|---|---|
| **Category** | Code Maturity | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

The `initialize` function of the RWARedemptionCoordinator contract does not validate the value of the `maxSlippage` parameter. As a result, it is possible to initialize the contract with a `maxSlippage` value that exceeds the `PERCENT_SCALE` (1e5), which may lead to incorrect slippage checks.

```
function initialize(uint256 _maxSlippage) external {
  _transferOwnership(msg.sender);
  if (wasInitialized) revert AlreadyInitialized();
  wasInitialized = true;
  maxSlippage = _maxSlippage;
}
```

#### Impact

Incorrect `maxSlippage` values can cause swaps to fail unintentionally, preventing the successful execution of the `redeemRWA` function, even if the RWA-to-USDC tokens' swap result is valid.

#### Recommendations

Add a check in the `initialize` function to ensure that `maxSlippage <= PERCENT_SCALE`.

#### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit `72ef1668`↗.

### 3.4. Incorrect address emitted in `ProcessedRedemption` event for Solana

| | | | |
|---|---|---|---|
| **Target** | FraxNetDeposit, FraxNetDepositV2 | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

The `processRedemption` function is responsible for redeeming USDC tokens via USDCustodian or the RWARedeemer contracts then transferring them to the `targetAddress` in `targetEid` using the CCTP protocol and emitting the appropriate event.

However, when the `targetEid` corresponds to 30168 (Solana), the `targetUsdcAtaAddress` is used instead of `targetAddress` as an actual `recipient`, but the emitted event still logs `targetAddress`.

#### Impact

Consumers of on-chain events may interpret the emitted data incorrectly, since the logged address does not correspond to the real `recipient` address.

#### Recommendations

Update the `processRedemption` function to emit the appropriate `recipient` address.

```
function processRedemption(uint256 amount) external {
  // [...]
 bytes32 recipient;
  if (targetEid == 30_101) {
   recipient = targetAddress;
    USDC.transfer(address(uint160(uint256(targetAddress))), usdcOut);
  } else {
    // [...]
    }
   bytes32 recipient = targetAddress;
   recipient = targetAddress;
    if (targetEid == 30168) recipient = targetUsdcAtaAddress;
    // [...]
  }
```

```
    emit ProcessedRedemption(targetEid, recipient, amount, usdcOut);
    emit ProcessedRedemption(targetEid, targetAddress, amount, usdcOut);
}
```

### Remediation

This issue has been acknowledged by FraxNet.

### 3.5. Incorrect fee checking in the `setMintRedeemFee` function

| Target | FrxUSDCustodian | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

The `setMintRedeemFee` function allows the owner to set the mint and redeem fees. However, the current implementation checks if either fee is less than `1e18`, which is not a valid check for ensuring that the fees are fractions of the underlying asset.

```
function setMintRedeemFee(uint256 _mintFee, uint _redeemFee)
    public onlyOwner {
  require(_mintFee < 1e18 || _redeemFee < 1e18, "Fee must be a fraction of
    underlying");
  mintFee = _mintFee;
  redeemFee = _redeemFee;
}
```

#### Impact

This check allows for the possibility of setting one fee to a fraction while the other is set to a value greater than or equal to 1e18, which would not be a valid fraction of the underlying asset. This could lead to unexpected behavior in the minting and redeeming processes.

#### Recommendations

Use `AND` instead of `OR` in the require statement to ensure both fees are fractions of the underlying asset.

#### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit `1faa7367`↗.

### 3.6. Lack of zero-address check in the `initialize` function

| | | | |
|---|---|---|---|
| **Target** | FrxUSDCustodian, FrxUSD | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

In the `initialize` function, ownership is set by calling `_transferOwnership` with the provided `owner` address. However, there is no validation to ensure that this address is not the zero address.

```
function initialize(address _owner, string memory _name,
    string memory _symbol) public {
    require(owner() == address(0), "Already initialized");
    _transferOwnership(_owner);
    // [...]
}

function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

#### Impact

Assigning the zero address as the owner can lead to a permanently unowned contract, preventing any admin actions that require ownership. While it may be unlikely in normal usage, failing to validate this introduces unnecessary risk and may lead to misconfigurations during deployment.

#### Recommendations

We recommend adding a check in `initialize` to ensure the `owner` is not the zero address.

#### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit `dfc64fc3`.

## 3.7. Inconsistent `frxUSDMinted` check

| Target | FrxUSDCustodian | | |
|---|---|---|---|
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The `maxDeposit` and `maxMint` functions return 0 when the current `frxUSDMinted` is greater than or equal to the `mintCap`.

```
function maxDeposit(address _addr) public view returns (uint256 _maxAssetsIn)
    {
    // See how much custodianTkn you would need to exchange for 100% of the
    frxUSD available under the cap
    if (frxUSDMinted >= mintCap) _maxAssetsIn = 0;
    else _maxAssetsIn = previewMint(mintCap - frxUSDMinted);
}

function maxMint(address _addr) public view returns (uint256 _maxSharesOut) {
    // See how much frxUSD is actually available in the contract
    if (frxUSDMinted >= mintCap) _maxSharesOut = 0;
    else _maxSharesOut = mintCap - frxUSDMinted;
  }
```

However, the `_deposit` function, which is used in both `deposit` and `mint`, performs a revert only if `frxUSDMinted` is strictly greater than the `mintCap`.

```
function _deposit(address _caller, address _receiver, uint256 _assets,
    uint256 _shares) internal nonReentrant {
    [...]
    // frxUSD minted accounting
    frxUSDMinted += _shares;
    if (frxUSDMinted > mintCap) revert MintCapExceeded(_receiver, _shares,
    mintCap);

    emit Deposit(_caller, _receiver, _assets, _shares);
  }
```

### Impact

This creates a slight inconsistency: `maxDeposit` returns 0 (implying deposits are no longer allowed), but a subsequent deposit transaction may still succeed if `frxUSDMinted == mintCap`.

### Recommendations

We recommend changing the check to revert when `frxUSDMinted >= mintCap` in the `_deposit` function. This ensures consistent behavior across functions.

### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit 8365e22b ↗.

## 3.8. Shared interface creates confusion between contracts with different functionality

| | | | |
|---|---|---|---|
| **Target** | RWARedemptionCoordinator, FraxNetDeposit, FraxNetDepositV2 | | |
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The IRWAUSDCRedeemer interface includes the `redeem` and `redeemRWA` functions.

```
interface IRWAUSDCRedeemer {
  function redeem(uint256 amount) external;
  function redeemRWA(uint256 amount) external returns (uint256);
  function redeem(uint256 amount, uint256 minAmountOut)
    external returns (uint256);
}
```

This interface is implemented by two distinct contracts with different responsibilities:

1. The RedemptionIdle contract (rwaUSDCRedeemer), used in the RWARedemptionCoordinator contract, which supports the `redeem` function.
2. The RWARedemptionCoordinator contract (rwaRedeemer), used in FraxNetDeposit and FraxNetDepositV2, which supports only the `redeemRWA` function.

Although both contracts use the same IRWAUSDCRedeemer interface, they do not share the same functionality.

### Impact

This can lead to confusion, and it reduces clarity when reading or maintaining code.

### Recommendations

Split the current IRWAUSDCRedeemer interface into two separate interfaces: one containing `redeem` functions, and one containing the `redeemRWA` function.

This improves code clarity and reduces the chance of misuse or incorrect assumptions.

### Remediation

This issue has been acknowledged by FraxNet, and a fix was implemented in commit caad8f2a.

# 4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

## 4.1. Missing events

Several functions across different contracts perform key state changes but do not emit any events. Emitting events in such functions would improve transparency and make it easier to monitor contract behavior off chain.

The following functions are missing event emissions:

- `setMintRedeemFee()` in FrxUSDCustodian.sol
- `shuffleToRwa()` in FrxUSDCustodian.sol
- `burnMany()` in FrxUSD.sol
- `burn()` in FrxUSD.sol
- `recoverERC20()` in FraxNetDeposit.sol, FraxNetDepositV2.sol, and FraxNetDepositFactory.sol
- `setAtaIfNotPreviouslySet()` in FraxNetDepositV2.sol
- `redeemRWA()` in RWARedemptionCoordinator.sol

We recommend adding event emissions to the functions listed above.

## 4.2. Unused function

The `scaleDownTo1e6` function in the FraxNetDeposit and FraxNetDepositV2 contracts is currently unused in the codebase. While it may be intended for future use, keeping unused functions in production contracts can add unnecessary complexity.

Consider removing the function if it is not needed.

## 4.3. Low value share redemptions result in loss of funds

In the FrxUSDCustodian contract, the `_convertToAssets` function rounds down the result of a conversion. As a result, if a user attempts to redeem fewer than 1e12 frxUSD, the returned USDC amount will be zero. Since the shares are still burned, this leads to a silent loss of funds.

This behavior aligns with the ERC-4626 specification, which permits rounding, but it may be unexpected for users who attempt to redeem small amounts of frxUSD. While not a critical issue,

this behavior should either be guarded against at the contract level (by rejecting redemptions where `previewRedeem(_sharesIn) == 0`) or be clearly documented and handled in the front end to avoid confusion and unintended value loss.

# 5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

## 5.1. Module: FraxBeacon.sol

### Function: `setImplementation(address _newImplementation)`

This function is used to set the implementation address for the beacon. It is only callable by the owner of the contract.

#### Inputs

- `_newImplementation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the new implementation contract that will be used by the beacon.

#### Branches and code coverage

**Intended branches**

- Set the implementation address to the new one.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

## 5.2. Module: FraxNetDepositV2.sol

### Function: `initialize(uint32 _targetEid, byte[32] _targetAddress, byte[32] _targetUsdcAtaAddress)`

This function is used to initialize the contract with the target chain ID and recipient address. It is designed to be called in proxy upgrades.

### Inputs

- `_targetEid`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of EID for LayerZero.
- `_targetAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient on the target chain, in `bytes32` format.
- `_targetUsdcAtaAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: ATA address of the associated target address, in `bytes32` format, used in Solana.

### Branches and code coverage

**Intended branches**

- Set the target chain ID and target addresses.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the factory.
  - ☐ Negative test
- Revert if the contract is already initialized.
  - ☑ Negative test
- Revert if the target EID is 30168 (Solana) but the target USDC ATA address is not set.
  - ☐ Negative test

### Function: `processDeposit(uint256 amount)`

This function is used to process USDC deposits as well as mint frxUSD and send it to a target address. It can only be called by the factory's operator.

### Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of USDC to process for deposit.

### Branches and code coverage

#### Intended branches

- Get the custodian and remote hop contract from the factory.
  - ☑ Test coverage
- Approve and deposit USDC to mint frxUSD.
  - ☑ Test coverage
- Transfer frxUSD directly to the target address if target EID is `30101` (Ethereum mainnet).
  - ☑ Test coverage
- Transfer cross-chain for other chains using LayerZero otherwise.
  - ☑ Test coverage
- Return unused ETH (for gas) back to the sender.
  - ☑ Test coverage

#### Negative behavior

- Revert if the caller is not an operator.
  - ☐ Negative test
- Revert if the USDC balance of this contract is insufficient for the deposit.
  - ☑ Negative test
- Revert if the unused ETH balance transferring has failed.
  - ☐ Negative test

### Function call analysis

- `this.factory.isOperator(msg.sender)`
  - The operators should be trusted addresses that can call this function.
- `FraxNetDeposit.USDC.balanceOf(address(this))`
  - The USDC address should be the correct USDC token address.
- `this.factory.getCustodianAndHop()`

    - This function returns a tuple of two contracts (custodian and remote hop). These contract addresses should be valid.
- `frxUSDCustodian.deposit(amount, address(this))`
    - This function should mint frxUSD from USDC and return the amount of frxUSD minted.
- `FraxNetDeposit.frxUSD.transfer(address payable(uint160(uint256(this.targetAddress))), frxUSDOut)`
    - This function should transfer the minted frxUSD to the target address.
- `remoteHop.sendOFT{value: msg.value}(FraxNetDeposit.oft, this.targetEid, this.targetAddress, frxUSDOut)`
    - This function should send the frxUSD to the target address on the target chain using LayerZero's OFT mechanism.

### Function: `processRedemption(uint256 amount)`

This function is used to process USDC redemptions, conditionally redeeming via the USDC custodian or the RWA redeemer contract based on the amount. It can only be called by the factory's operator.

### Inputs

- `amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of frxUSD to redeem for USDC.

### Branches and code coverage

**Intended branches**

- Get the RWA redeemer and USDC custodian from the factory.
    - ☑ Test coverage
- Redeem all via USDC custodian if the target amount is less than the RWA redemption and USDC redemption threshold.
    - ☑ Test coverage
- Redeem partially via USDC custodian and the rest via RWA redeemer otherwise.
    - ☑ Test coverage
- Transfer USDC directly to the target address if target EID is `30101` (Ethereum mainnet).

☑ Test coverage

- Transfer cross-chain for other chains using LayerZero otherwise.

☑ Test coverage

- Use the CCTP token messenger V2 for cross-chain transfer if target EID is 30332 (Sonic) or 30183 (Linea mainnet).

☑ Test coverage

- Use the CCTP token messenger for cross-chain transfer otherwise.

☑ Test coverage

- Use the ATA address for the target address if target EID is 30168 (Solana).

☑ Test coverage

**Negative behavior**

- Revert if the caller is not an operator.

☐ Negative test

- Revert if the frxUSDC balance of this contract is insufficient for the redemption.

☑ Negative test

- Revert if the CCTP domain is not configured for the target chain.

☑ Negative test

- Revert if the ATA account is not initialized for Solana.

☑ Negative test

## Function call analysis

- `this.factory.isOperator(msg.sender)`
  - The operators should be trusted addresses that can call this function.
- `FraxNetDeposit.frxUSD.balanceOf(address(this))`
  - The frxUSD address should be the correct frxUSD token address.
- `FraxNetDeposit.USDC.balanceOf(address(usdcCustodian))`
  - The USDC address should be the correct USDC token address.
- `this.factory.mdwrComboView()`
  - This function should return the maximum RWA redeemable and USDC redeemable amounts.
- `usdcCustodian.redeem(amount, address(this), address(this))`
  - This function should redeem USDC from the custodian and return the amount of USDC redeemed.

- `rwaRedeemer.redeemRWA(amount)`
    - This function should redeem RWA from the redeemer and return the amount of USDC redeemed.
- `FraxNetDeposit.USDC.transfer(address payable(uint160(uint256(this.targetAddress))), usdcOut)`
    - This function should transfer the redeemed USDC to the target address.
- `this.factory.cctpTokenMessenger(), this.factory.cctpTokenMessengerV2()`
    - This function should return the CCTP token messenger address for the target chain.
- `this.factory.eidToCCTPDomain(targetEid)`
    - This function should return the CCTP domain for the target chain mapped correctly.
- `ITokenMessenger(cctpTokenMessenger).depositForBurn(usdcOut, cctpDestinationDomain, recipient, address(FraxNetDeposit.USDC))`
    - This function should send the USDC to the target address on the target chain using LayerZero's CCTP mechanism.

### Function: `recoverERC20(address tokenAddress, uint256 tokenAmount, address recipient)`

This function is used to recover any ERC-20 tokens that were accidentally sent to this contract. It can only be called by the factory.

#### Inputs

- `tokenAddress`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the ERC-20 token to recover.
- `tokenAmount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of the ERC-20 token to recover.
- `recipient`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address to send the recovered tokens to.

### Branches and code coverage

**Intended branches**

- Transfer the specified amount of tokens to the recipient.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the factory.
  - ☐ Negative test

## 5.3. Module: FraxNetDeposit.sol

### Function: `initialize(uint32 _targetEid, byte[32] _targetAddress, byte[32] _targetUsdcAtaAddress)`

This function is used to initialize the contract with the target chain ID and recipient address. It is designed to be called in proxy upgrades.

### Inputs

- `_targetEid`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of endpoint ID (EID) for LayerZero.
- `_targetAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient on the target chain, in `bytes32` format.
- `_targetUsdcAtaAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: ATA address of the associated target address, in `bytes32` format, used in Solana.

### Branches and code coverage

**Intended branches**

- Set the target chain ID and target addresses.

☑ Test coverage

**Negative behavior**

- Revert if the caller is not the factory.
  - ☐ Negative test
- Revert if the contract is already initialized.
  - ☑ Negative test
- Revert if the target EID is `30168` (Solana) but the target USDC ATA address is not set.
  - ☐ Negative test

### Function: `processDeposit(uint256 amount)`

This function is used to process USDC deposits as well as mint frxUSD and send it to a target address. It can only be called by the factory's operator.

### Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of USDC to process for deposit.

### Branches and code coverage

**Intended branches**

- Get the custodian and remote hop contract from the factory.
  - ☑ Test coverage
- Approve and deposit USDC to mint frxUSD.
  - ☑ Test coverage
- Transfer frxUSD directly to the target address if target EID is `30101` (Ethereum mainnet).
  - ☑ Test coverage
- Transfer cross-chain for other chains using LayerZero otherwise.
  - ☑ Test coverage
- Return unused ETH (for gas) back to the sender.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not an operator.
  - ☐ Negative test
- Revert if the USDC balance of this contract is insufficient for the deposit.
  - ☑ Negative test
- Revert if the unused ETH balance transferring has failed.
  - ☐ Negative test

### Function call analysis

- `this.factory.isOperator(msg.sender)`
  - The operators should be trusted addresses that can call this function.
- `FraxNetDeposit.USDC.balanceOf(address(this))`
  - The USDC address should be the correct USDC token address.
- `this.factory.getCustodianAndHop()`
  - This function returns a tuple of two contracts (custodian and remote hop). These contract addresses should be valid.
- `frxUSDCustodian.deposit(amount, address(this))`
  - This function should mint frxUSD from USDC and return the amount of frxUSD minted.
- `FraxNetDeposit.frxUSD.transfer(address payable(uint160(uint256(this.targetAddress))), frxUSDOut)`
  - This function should transfer the minted frxUSD to the target address.
- `remoteHop.sendOFT{value: msg.value}(FraxNetDeposit.oft, this.targetEid, this.targetAddress, frxUSDOut)`
  - This function should send the frxUSD to the target address on the target chain using LayerZero's OFT mechanism.

### Function: `processRedemption(uint256 amount)`

This function is used to process USDC redemptions, conditionally redeeming via the USDC custodian or the RWA redeemer contract based on the amount. It can only be called by the factory's operator.

### Inputs

- `amount`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Amount of frxUSD to redeem for USDC.

## Branches and code coverage

**Intended branches**

- Get the RWA redeemer and USDC custodian from the factory.
  - ☑ Test coverage
- Redeem via USDC custodian if the target amount is less than the RWA redemption threshold.
  - ☑ Test coverage
- Redeem via RWA redeemer otherwise.
  - ☑ Test coverage
- Transfer USDC directly to the target address if target EID is `30101` (Ethereum mainnet).
  - ☑ Test coverage
- Transfer cross-chain for other chains using LayerZero otherwise.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not an operator.
  - ☐ Negative test
- Revert if the frxUSDC balance of this contract is insufficient for the redemption.
  - ☑ Negative test
- Revert if the CCTP domain is not configured for the target chain.
  - ☑ Negative test
- Revert if the ATA account is not initialized for Solana.
  - ☑ Negative test

## Function call analysis

- `this.factory.isOperator(msg.sender)`
  - The operators should be trusted addresses that can call this function.
- `FraxNetDeposit.frxUSD.balanceOf(address(this))`
  - The frxUSD address should be the correct frxUSD token address.

- `FraxNetDeposit.USDC.balanceOf(address(usdcCustodian))`
    - The USDC address should be the correct USDC token address.
- `usdcCustodian.redeem(amount, address(this), address(this))`
    - This function should redeem USDC from the custodian and return the amount of USDC redeemed.
- `rwaRedeemer.redeemRWA(amount)`
    - This function should redeem RWA from the redeemer and return the amount of USDC redeemed.
- `FraxNetDeposit.USDC.transfer(address payable(uint160(uint256(this.targetAddress))), usdcOut)`
    - This function should transfer the redeemed USDC to the target address.
- `this.factory.cctpTokenMessenger()`
    - This function should return the CCTP token messenger address for the target chain.
- `this.factory.eidToCCTPDomain(targetEid)`
    - This function should return the CCTP domain for the target chain mapped correctly.
- `ITokenMessenger(cctpTokenMessenger).depositForBurn(usdcOut, cctpDestinationDomain, recipient, address(FraxNetDeposit.USDC))`
    - This function should send the USDC to the target address on the target chain using LayerZero's CCTP mechanism.

### Function: `recoverERC20(address tokenAddress, uint256 tokenAmount, address recipient)`

This function is used to recover any ERC-20 tokens that were accidentally sent to this contract. It can only be called by the factory.

#### Inputs

- `tokenAddress`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the ERC-20 token to recover.
- `tokenAmount`
    - **Control**: Arbitrary.
    - **Constraints**: None.

- **Impact**: Amount of the ERC-20 token to recover.

- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to send the recovered tokens to.

### Branches and code coverage

**Intended branches**

- Transfer the specified amount of tokens to the recipient.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the factory.
  - ☐ Negative test

## 5.4. Module: FrxUSDCustodian.sol

### Function: `deposit(uint256 _assetsIn, address _receiver)`

This function allows users to deposit a specified amount of underlying asset tokens into the contract and receive shares in return. The user must ensure that they have approved the transfer of their asset tokens to this contract before calling this function.

It is designed to mint frxUSD shares based on the amount of underlying asset tokens deposited.

### Inputs

- `_assetsIn`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of underlying asset tokens to deposit.
- `_receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address that will receive the generated shares.

### Branches and code coverage

**Intended branches**

- Check max and preview deposit.
  - ☑ Test coverage
- Invoke `_deposit` to mint shares.
  - ☑ Test coverage

**Negative behavior**

- Revert if the deposit exceeds the maximum allowed amount.
  - ☐ Negative test

## Function: `initialize(address _owner, uint256 _mintCap, uint256 _mintFee, uint256 _redeemFee)`

This function is used to initialize the contract. It sets the owner, mint cap, mint fee, and redeem fee. It is designed to be called in proxy upgrades.

### Inputs

- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the owner who will have control over the contract.
- `_mintCap`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The maximum amount of frxUSD that can be minted by this contract.
- `_mintFee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The fee charged for minting frxUSD, expressed in 18 decimals.
- `_redeemFee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The fee charged for redeeming frxUSD, expressed in 18 decimals.

### Branches and code coverage

**Intended branches**

- Set the owner, mint cap, mint fee, and redeem fee.
  - ☑ Test coverage

**Negative behavior**

- Revert if the contract is already initialized.
  - ☑ Negative test

## Function: `mint(uint256 _sharesOut, address _receiver)`

This function allows users to mint a specified amount of shares in exchange for underlying asset tokens. The user must ensure that they have approved the transfer of their asset tokens to this contract before calling this function.

It is designed to mint frxUSD shares based on the amount of underlying asset tokens provided.

### Inputs

- `_sharesOut`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of shares to mint.
- `_receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address that will receive the minted shares.

### Branches and code coverage

**Intended branches**

- Check max and preview mint.
  - ☑ Test coverage
- Invoke `_deposit` to mint shares.
  - ☑ Test coverage

**Negative behavior**

- Revert if the mint exceeds the maximum allowed amount.
  - ☐ Negative test

### Function: `recoverERC20(address _tokenAddress, uint256 _tokenAmount)`

This function allows the owner to recover ERC-20 tokens that may have been accidentally sent to the contract. It transfers the specified amount of tokens from the contract to the owner's address.

#### Inputs

- `_tokenAddress`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address of the ERC-20 token to recover.
- `_tokenAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to recover from the contract.

#### Branches and code coverage

**Intended branches**

- Transfer the specified amount of tokens to the owner's address.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the owner of the contract.
  - ☐ Negative test

### Function: `redeem(uint256 _sharesIn, address _receiver, address _owner)`

This function allows users to redeem a specified amount of shares for underlying asset tokens. The user must ensure that they have approved the transfer of their shares to this contract before calling this function.

It is designed to redeem frxUSD shares for underlying asset tokens based on the amount specified.

### Inputs

- `_sharesIn`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The number of shares to redeem.
- `_receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address that will receive the underlying asset tokens.
- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: Must be `msg.sender`.
  - **Impact**: The owner of the shares being redeemed, which must match the `msg.sender`.

### Branches and code coverage

#### Intended branches

- Check max and preview redeem.
  - ☑ Test coverage
- Invoke `_withdraw` to redeem frxUSD shares.
  - ☑ Test coverage

#### Negative behavior

- Revert if the redemption exceeds the maximum allowed amount.
  - ☐ Negative test

## Function: `setApprovedOperator(address _operator, bool _status)`

This function allows the owner to set the approval status of a custodian operator. The operator can be an address that is allowed to perform certain actions on behalf of the contract.

### Inputs

- `_operator`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: The address of the operator whose status is being updated.

- `_status`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The new status for the operator, which can be either approved or not approved.

### Branches and code coverage

**Intended branches**

- Set the approval status of the operator.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the owner of the contract.
  - ☐ Negative test

## Function: `setMinAfterShuffle(uint256 _minAfterShuffle)`

This function is used to set the minimum amount of `custodianTkn` that must remain in the contract after a call to `shuffleToRwa`. It is only callable by the owner of the contract.

### Inputs

- `_minAfterShuffle`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The minimum amount of `custodianTkn` that must remain in the contract after the shuffle operation.

### Branches and code coverage

**Intended branches**

- Set the `minAfterShuffle` variable.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the owner of the contract.
  - ☐ Negative test

### Function: `setMintCap(uint256 _mintCap)`

This function allows the owner to set a cap on the total amount of frxUSD that can be minted. The mint cap is a limit on the total supply of frxUSD that can be created by this contract.

### Inputs

- `_mintCap`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The new mint cap to be set for the contract, which limits the total amount of frxUSD that can be minted.

### Branches and code coverage

**Intended branches**

- Set the mint cap for frxUSD minting.
  - ☑ Test coverage

**Negative behavior**

- Revert if the mint cap is set to a value that exceeds the current total supply of frxUSD.
  - ☐ Negative test

### Function: `setMintRedeemFee(uint256 _mintFee, uint256 _redeemFee)`

This function allows the owner to set the mint and redeem fees for the contract. The fees are specified as fractions of the underlying asset, represented in 1e18 format (e.g., 0.01 = 1%).

### Inputs

- `_mintFee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The mint fee to be set, expressed as a fraction of the underlying asset (1e18 format).

- `_redeemFee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The redemption fee to be set, expressed as a fraction of the underlying asset (1e18 format).

### Branches and code coverage

**Intended branches**

- Set the mint and redeem fees.
  - ☑ Test coverage

**Negative behavior**

- Revert if either fee is not a fraction of the underlying asset.
  - ☐ Negative test

## Function: `shuffleToRwa(uint256 amount)`

This function allows an approved operator to shuffle a specified amount of `custodianTkn` into RWA by subscribing to the RWA issuer. It ensures that the contract retains a minimum amount of `custodianTkn` after the shuffle operation.

### Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: The amount must not exceed the balance of `custodianTkn` in the contract minus `minAfterShuffle`.
  - **Impact**: The amount of `custodianTkn` to be shuffled into RWA.

### Branches and code coverage

**Intended branches**

- Get the `rwaIssuer` and `rwaCustodian` addresses.
  - ☑ Test coverage
- Approve the `rwaIssuer` to spend `custodianTkn`.
  - ☑ Test coverage

- Call the `subscribe` function on the `rwaIssuer`.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not an approved operator.
  - ☐ Negative test
- Revert if the amount exceeds the balance of `custodianTkn` minus `minAfterShuffle`.
  - ☐ Negative test

### Function call analysis

- `rwaIssuer.subscribe(rwaCustodian, amount, address(this.custodianTkn))`
  - This function should get RWA tokens in exchange for `custodianTkn` and is expected to succeed.

### Function: `withdraw(uint256 _assetsOut, address _receiver, address _owner)`

This function allows users to withdraw a specified amount of underlying asset tokens from the contract. The user must ensure that they have approved the transfer of their shares to this contract before calling this function.

It is designed to redeem frxUSD shares for underlying asset tokens based on the amount specified.

### Inputs

- `_assetsOut`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of underlying asset tokens to withdraw.
- `_receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address that will receive the withdrawn asset tokens.
- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: Must be `msg.sender`.
  - **Impact**: The owner of the shares being redeemed, which must match the

`msg.sender`.

### Branches and code coverage

**Intended branches**

- Check max and preview withdraw.
  - ☑ Test coverage
- Invoke `_withdraw` to redeem frxUSD shares.
  - ☑ Test coverage

**Negative behavior**

- Revert if the withdrawal exceeds the maximum allowed amount.
  - ☐ Negative test

## 5.5. Module: FrxUSD.sol

### Function: `addMinter(address minter_address)`

This function is used to add a new minter to the contract. It is only callable by the owner of the contract.

### Inputs

- `minter_address`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the new minter to be added.

### Branches and code coverage

**Intended branches**

- Set the minter address in the mapping.
  - ☐ Test coverage
- Push the minter address to the `minters_array`.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the minter address is zero.
  - ☐ Negative test
- Revert if the minter address already exists in the mapping.
  - ☐ Negative test

### Function: `burnMany(address[] _owners, uint256[] _amounts)`

This function is used to burn balances from a set of accounts. It is only callable by the owner of the contract.

### Inputs

- `_owners`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of addresses of the accounts whose balances will be burned.
- `_amounts`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of amounts corresponding to the balances to be burned from each account in `_owners`.

### Branches and code coverage

**Intended branches**

- Burn balances from each account in the `_owners` array.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the lengths of `_owners` and `_amounts` do not match.
  - ☐ Negative test

### Function: `burn(address _owner, uint256 _amount)`

This function is used to burn the balance from a given account. It is only callable by the owner of the contract.

### Inputs

- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the account whose balance will be burned.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: If `_amount` is 0, the entire balance will be burned.
  - **Impact**: Amount of balance to burn from the specified account.

### Branches and code coverage

**Intended branches**

- Burn the balance from the specified account.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `freezeMany(address[] _owners)`

This function is used to freeze a set of accounts. It is only callable by the owner of the contract.

### Inputs

- `_owners`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of addresses of the accounts to be frozen.

### Branches and code coverage

**Intended branches**

- Freeze each account in the `_owners` array by setting `isFrozen` to true.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `freeze(address _owner)`

This function is used to freeze a given account. It is only callable by the owner of the contract.

### Inputs

- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the account to be frozen.

### Branches and code coverage

**Intended branches**

- Freeze the account by setting `isFrozen` to true.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `initialize(address _owner, string _name, string _symbol)`

This function is used to initialize the contract. It is designed to be called in proxy upgrade.

### Inputs

- `_owner`
  - **Control**: Arbitrary.
  - **Constraints**: Should not be the zero address.
  - **Impact**: Addresses the owner of the contract.
- `_name`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the token, used in the ERC-20 standard.
- `_symbol`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Symbol of the token, used in the ERC-20 standard.

### Branches and code coverage

**Intended branches**

- Set the ownership of the contract to the caller.
  - ☑ Test coverage
- Set the name and symbol of the token.
  - ☑ Test coverage

**Negative behavior**

- Revert if the owner is already set.
  - ☐ Negative test

## Function: `minter_burn_from(address b_address, uint256 b_amount)`

This function is used by minters to burn tokens from a specified address. It is only callable by authorized minters.

### Inputs

- `b_address`
  - **Control**: Arbitrary.
  - **Constraints**: None.

    - **Impact**: Address of the account from which tokens will be burned.
- `b_amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of tokens to burn from the specified address.

### Branches and code coverage

**Intended branches**

- Burn tokens from the specified address with allowance checks.
    - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not a minter.
    - ☐ Negative test

## Function: `minter_mint(address m_address, uint256 m_amount)`

This function is used by minters to mint new tokens to a specified address. It is only callable by authorized minters.

### Inputs

- `m_address`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the account to which tokens will be minted.
- `m_amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of tokens to mint to the specified address.

### Branches and code coverage

**Intended branches**

- Mint new tokens to the specified address.
    - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not a minter.
  - ☐ Negative test

### Function: `pause()`

This function is used to pause the contract. It is only callable by the owner of the contract.

### Branches and code coverage

**Intended branches**

- Pause the contract by setting `isPaused` to true.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `removeMinter(address minter_address)`

This function is used to remove a minter from the contract. It is only callable by the owner of the contract.

### Inputs

- `minter_address`
  - **Control**: Arbitrary.
  - **Constraints**: Should not be the zero address.
  - **Impact**: Address of the minter to be removed.

### Branches and code coverage

**Intended branches**

- Delete the minter address from the mapping.
  - ☐ Test coverage
- Set the minter address to `0x0` in the `minters_array`.

- ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the minter address is zero.
  - ☐ Negative test

### Function: `thawMany(address[] _owners)`

This function is used to unfreeze a set of accounts. It is only callable by the owner of the contract.

### Inputs

- `_owners`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of addresses of the accounts to be unfrozen.

### Branches and code coverage

**Intended branches**

- Unfreeze each account in the `_owners` array by setting `isFrozen` to false.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `thaw(address _owner)`

This function is used to unfreeze an account. It is only callable by the owner of the contract.

### Inputs

- `_owner`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Address of the account to be unfrozen.

### Branches and code coverage

**Intended branches**

- Unfreeze the account by setting `isFrozen` to false.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

### Function: `unpause()`

This function is used to unpause the contract. It is only callable by the owner of the contract.

### Branches and code coverage

**Intended branches**

- Unpause the contract by setting `isPaused` to false.
  - ☐ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☐ Negative test

## 5.6. Module: RWARedemptionCoordinator.sol

### Function: `initialize(uint256 _maxSlippage)`

This function is used to initialize the RWARedemptionCoordinator contract, setting the maximum slippage for RWA redemptions and transferring ownership to the caller. It is designed to be called in proxy upgrades.

### Inputs

- `_maxSlippage`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Value of the maximum slippage for RWA redemptions, denominated in 1e5.

### Branches and code coverage

#### Intended branches

- Set the ownership of the contract to the caller.
  - ☑ Test coverage
- Set the maximum slippage for RWA redemptions.
  - ☑ Test coverage

#### Negative behavior

- Revert if the contract has already been initialized.
  - ☑ Negative test
- Revert if the maximum slippage exceeds 100% (1e5).
  - ☐ Negative test

## Function: `setMaxSlippage(uint256 newSlippage)`

This function is used to set the maximum slippage for RWA redemptions. It is only callable by the owner of the contract.

### Inputs

- `newSlippage`
  - **Control**: Arbitrary.
  - **Constraints**: Should not exceed 100% (1e5).
  - **Impact**: Value of the new maximum slippage for RWA redemptions, denominated in 1e5.

### Branches and code coverage

#### Intended branches

- Update the maximum slippage for RWA redemptions.
  - ☑ Test coverage

**Negative behavior**

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new slippage exceeds 100% (1e5).
  - ☑ Negative test

# 6. Assessment Results

During our assessment on the scoped Frax contracts, we discovered eight findings. No critical issues were found. One finding was of medium impact, five were of low impact, and the remaining findings were informational in nature.

## 6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.