Zellic

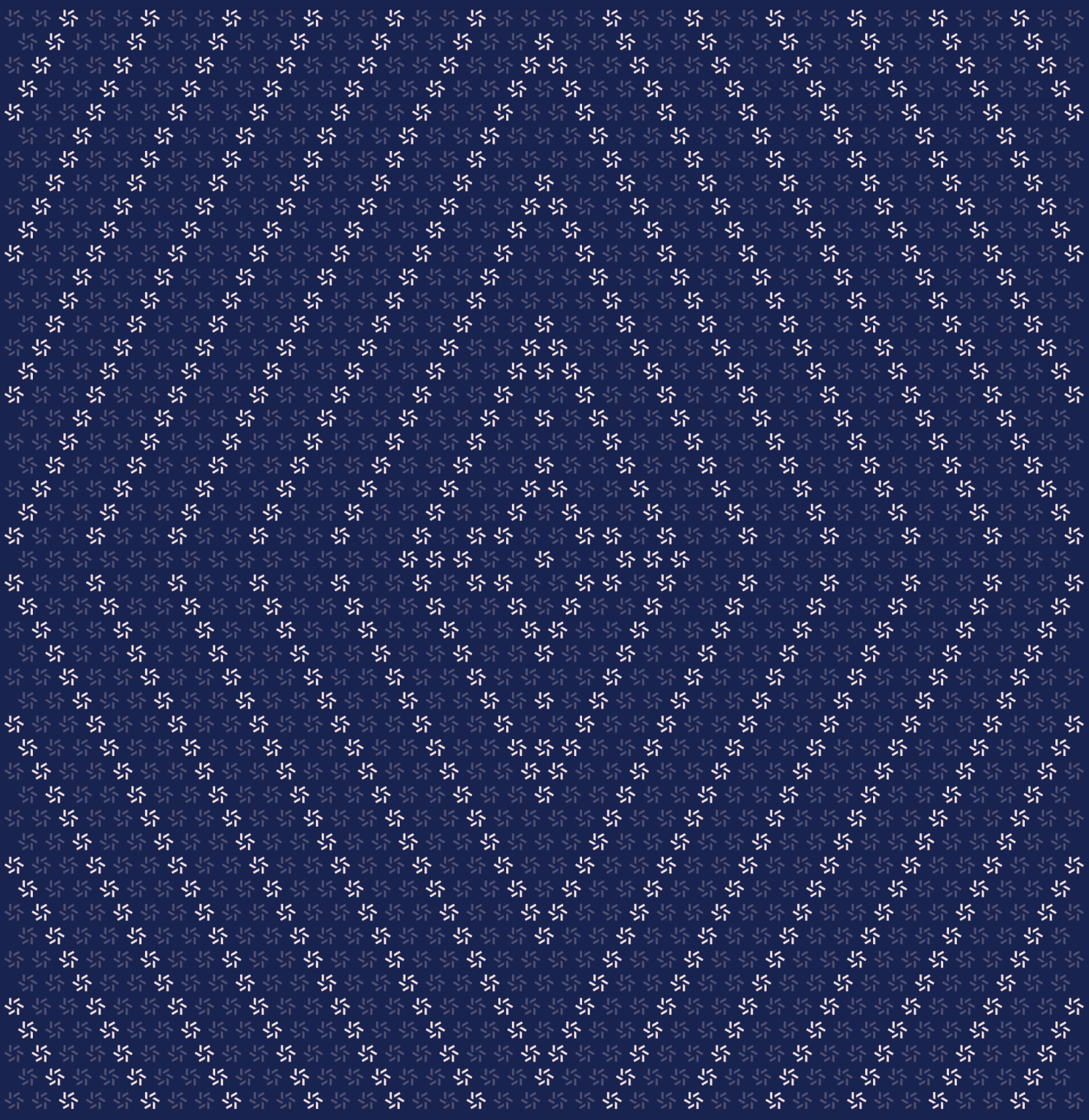
Prepared for
Sam Kazemian
Frax Finance

Prepared by
Hojung Han
Qingying Jie
Zellic

September 24, 2025

Frax0 Mesh

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 Frax Finance from September 9th to September 22nd, 2025. During this engagement, Zellic reviewed Frax0 Mesh'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:

- Could funds get stuck in transit from the RemoteCustodian?
- Could contract upgrades lead to storage-data corruption?
- Could an attacker perform a signature malleability attack?
- Do the contract upgrade scripts have any issues?

1.3. Non-goals and Limitations

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

- Inherited contracts not listed in the scope
- Front-end components
- Infrastructure relating to the project
- Key custody

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

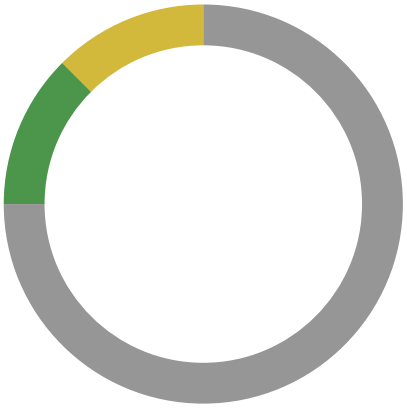
1.4. Results

During our assessment on the scoped Frax0 Mesh contracts, we discovered eight findings. No critical issues were found. One finding was of medium impact, one was 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 Frax Finance in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	6



2. Introduction

2.1. About Frax0 Mesh

Frax Finance contributed the following description of Frax0 Mesh:

> Frax0 mesh is the LayerZero-powered protocol operated by Frax with multiple Frax-designed subprotocols.

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:

Frax0 Mesh Contracts

Type	Solidity
Platform	EVM-compatible
Target	fraxnet-dev
Repository	https://github.com/FraxFinance/fraxnet-dev ↗
Version	`4e10ffdb588d5832eca87255964c5921c5650bfa`

Programs	`FraxtalMinter.sol`
	`RemoteCustodianUsdc.sol`
	`RemoteCustodianWithOracle.sol`
	`RemoteCustodian.sol`
	`CustodianBase.sol`

Target	dev-staked-frax-erc4626
Repository	https://github.com/FraxFinance/dev-staked-frax-erc4626 ↗
Version	`afb47e071b9c8aaaf6e1696bc0694a05e94bec9d`
Programs	`frxUSD/FrxUSD3.sol` `sfrxUSD/StakedFrxUSD3.sol`

Target	dev-fraxchain-contracts
Repository	https://github.com/FraxFinance/dev-fraxchain-contracts ↗
Version	`2c2cc85f4bfd17540bc07a9a78f80347a2c921e2`
Programs	`frxUSD/FrxUSD2.sol` `sfrxUSD/StakedFrxUSD2.sol`

Target	Only changes between d3d9d17...b8d76da
Repository	https://github.com/FraxFinance/frax-oft-upgradeable ↗
Version	`b8d76da3a2bb5ff1fc557c194f8fe6496873959d`

Programs	`contracts/frxUsd/FrxUSDOFTUpgradeable.sol`
	`contracts/frxUsd/SFrxUSDOFTUpgradeable.sol`
	`contracts/FraxOFTMintableAdapterUpgradeable.sol`
	`contracts/FraxOFTUpgradeable.sol`
	`contracts/WFRAXTokenOFTUpgradeable.sol`
	`contracts/modules/*.sol`
	`scripts/ops/V110/destinations/UpgradeV110Destinations.s.sol`
	`scripts/ops/V110/ethereum/UpgradeAdapterEthereum.s.sol`
	`scripts/ops/V110/fraxtal/UpgradeAdapterFraxtal.s.sol`

2.4. Project Overview

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

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Hojung Han
Engineer
hojung@zellic.io ↗

Qingying Jie
Engineer
qingying@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 9, 2025	Kick-off call
September 9, 2025	Start of primary review period
September 22, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect condition for determining dust removal

Target	RemoteCustodian		
Category	Business Logic	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

Before performing a cross-chain transfer with OFT, the transfer amount's precision is converted from local decimals to shared decimals. Any dust generated during this precision conversion will not be transferred.

In the contract RemoteCustodian, the OFT token frxUsd serves as its share token. The functions `_removeDustFrxUsdFloor` and `_removeDustFrxUsdCeil` can remove dust from the share amount. However, whether dust removal is executed depends on the custodian token decimals.

```
function _removeDustFrxUsdFloor(uint256 _amountLD)
    internal view override returns (uint256) {
    if (custodianTknDecimals() > 6) {
      uint256 decimalConversionRate
    = IOFT2(address(frxUsd())).decimalConversionRate();
      return (_amountLD / decimalConversionRate) * decimalConversionRate;
    } else {
      return _amountLD;
    }
}

function _removeDustFrxUsdCeil(uint256 _amountLD)
    internal view override returns (uint256) {
    if (custodianTknDecimals() > 6) {
      uint256 decimalConversionRate
    = IOFT2(address(frxUsd())).decimalConversionRate();
      uint256 out = (_amountLD / decimalConversionRate)
    * decimalConversionRate;
      if (out < _amountLD) out += decimalConversionRate;
      return out;
    } else {
      return _amountLD;
    }
}
```

Impact

If the custodian token decimals are less than or equal to six, the dust in the share amount will not be removed by the contract RemoteCustodian, which may lead to potential issues.

Assume the custodian token has six decimals. The frxUsd uses 18 local decimals and six shared decimals.

In the function `mint`, assume the `mintFee` is zero and suppose the user sets `_sharesOut` to `1.9e12`, which contains dust. In that case, the function `_convertToAssets` will return 2 due to rounding up.

However, later when this `_sharesOut` is divided by the `decimalConversionRate` for cross-chain transfer, the receiver would actually end up receiving 1e12 amount of share tokens, while two assets have already been deposited to the contract.

```
function _convertToAssets(uint256 _shares, Math.Rounding _rounding)
    internal view virtual returns (uint256 _assets) {
    CustodianBaseStorage storage $ = _getCustodianBaseStorage();

    _assets = Math.mulDiv(_shares, uint256(10 ** $.custodianTknDecimals),
    uint256(10 ** $.frxUSDDecimals), _rounding);
}

function previewMint(uint256 _sharesOut)
    public view returns (uint256 _assetsIn) {
    _sharesOut = _removeDustFrxUsdCeil(_sharesOut);
    CustodianBaseStorage storage $ = _getCustodianBaseStorage();

    uint256 fee = $.mintFee;
    _assetsIn = _convertToAssets(_sharesOut, Math.Rounding.Ceil);
    if (fee > 0) _assetsIn = Math.mulDiv(_assetsIn, 1e18, (1e18 - fee),
    Math.Rounding.Ceil);
}

function mint(uint256 _sharesOut, uint32 _receiverEid, address _receiver)
    internal returns (uint256 _assetsIn) {
    // [...]
    _assetsIn = previewMint(_sharesOut);

    // Do the minting
    _deposit(msg.sender, _receiverEid, _receiver, _assetsIn, _sharesOut);
}
```

If a user uses the function `deposit` to deposit, depositing two assets should result in receiving a 2e12 amount of share tokens.

```
function _convertToShares(uint256 _assets, Math.Rounding _rounding)
    internal view virtual returns (uint256 _shares) {
    CustodianBaseStorage storage $ = _getCustodianBaseStorage();

    _shares = Math.mulDiv(_assets, uint256(10 ** $.frxUSDDecimals),
    uint256(10 ** $.custodianTknDecimals), _rounding);
}

function previewDeposit(uint256 _assetsIn)
    public view returns (uint256 _sharesOut) {
    CustodianBaseStorage storage $ = _getCustodianBaseStorage();

    uint256 fee = $.mintFee;
    if (fee > 0) _assetsIn = Math.mulDiv(_assetsIn, (1e18 - fee), 1e18,
    Math.Rounding.Floor);
    _sharesOut = _convertToShares(_assetsIn, Math.Rounding.Floor);
    _sharesOut = _removeDustFrxUsdFloor(_sharesOut);
}
```

Recommendations

Consider removing dust from the share amount based on the return value of the function `frxUSDDecimals`.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #69 ↗. They intend to merge these changes into the production branch.

3.2. Inconsistent native fee refund recipient in `_remoteMint`

Target	RemoteCustodian		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In `RemoteCustodian._remoteMint`, when the `_receiverEid` is a local EID and the contract holds sufficient shares, any excess native token is refunded to `_receiver` rather than `msg.sender` (the payer). In the alternate branch (cross-chain), excess is correctly refunded to `msg.sender`. This inconsistent refund destination can return fees to a party different from the actual payer.

Impact

Excess native fee may be refunded to the wrong address, causing misdirected funds and inconsistent user experience.

Recommendations

Always refund the excess to `msg.sender` for all code paths.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #68 ↗. They intend to merge these changes into the production branch.

3.3. Incorrect `Transfer` event parameters

Target	StakedFrxUSD3		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Transfer` event is emitted with `from = msg.sender` instead of the actual debited account (`owner`). In an internal routine like `__transfer(owner, spender, amount)`, the correct emission is `Transfer(owner, spender, amount)`. This diverges from ERC-20 semantics.

Impact

Event consumers may attribute transfers to the wrong sender, causing incorrect accounting and monitoring.

Recommendations

Emit `Transfer(owner, spender, amount)`.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #30 ↗. They intend to merge these changes into the production branch.

3.4. Incorrect EIP-712 domain separator in upgradable proxy

Target	StakedFrxUSD3		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `__domainSeparatorV4()` delegates to Solmate's `DOMAIN_SEPARATOR()`, which caches `INITIAL_DOMAIN_SEPARATOR` in the implementation constructor. In a proxy, this binds the domain to the implementation address, not the proxy, contradicting EIP-712 (that the verifying contract must be the proxy). When `block.chainid == INITIAL_CHAIN_ID`, the stale implementation-bound separator is used.

Impact

EIP-712 signatures (e.g., permit) may be invalid or bound to the wrong contract, impacting verification and interoperability.

Recommendations

Compute the domain separator from `address(this)` at runtime (proxy) or use OZ `EIP712Upgradeable._domainSeparatorV4`; avoid Solmate's cached separator in proxies.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #30 ↗. They intend to merge these changes into the production branch.

3.5. Incorrect proxy contract addresses

Target	UpgradeAdapterEthereum		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The script `UpgradeAdapterEthereum.s.sol` is used to upgrade contracts on Ethereum. However, when calling the function `upgradeExistingLockbox`, the provided `_lockbox` address (i.e., the proxy contract address) corresponds to the one on Fraxtal.

```
function upgradeExistingLockboxes() public {
    upgradeExistingLockbox(fraxtalFrxUsdLockbox, frxUsdMintableLockboxImp);
    upgradeExistingLockbox(fraxtalSFrxUsdLockbox, sfrxUsdMintableLockboxImp);
}

function upgradeExistingLockbox(
    address _lockbox,
    address _implementation
) public {
    bytes memory data = abi.encodeCall(
        ProxyAdmin.upgrade,
        (
            TransparentUpgradeableProxy(payable(_lockbox)),
            _implementation
        )
    );

    bytes32 adminSlot = vm.load(_lockbox, ERC1967Utils.ADMIN_SLOT);
    proxyAdmin = address(uint160(uint256(adminSlot)));
    (bool success, ) = proxyAdmin.call(data);
    require(success, "Upgrade failed");
    // [...]
}
```

Impact

Because the proxy contract address on Fraxtal does not have a contract deployed on Ethereum, the script execution will fail.

Recommendations

Consider using `ethFrxUsdLockbox` and `ethSFrxUsdLockbox` instead.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #106. They intend to merge these changes into the production branch.

3.6. Incorrect caller in script

Target	UpgradeAdapterEthereum		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The new version of the implementation contract requires minting permission for the corresponding tokens. The function `addMinterRole` calls the token contract's function `addMinter` to set the proxy contract address as a minter.

However, the privileged accounts that can add new minters differ between the frxUsd and sfrxUsd tokens: for frxUsd, the owner ↗ has the authority, while for sfrxUsd, it is the `timelockAddress` ↗. Meanwhile, the modifier `prankAndWriteTxs` makes the function `addMinterRole` be called by the same address.

```
modifier prankAndWriteTxs(address who) {
    vm.startPrank(who);
    _;
    vm.stopPrank();

    // [...]
}

function generateComptrollerTxs()
    public prankAndWriteTxs(broadcastConfig.delegate) {
    addMinterRoles();
    upgradeExistingLockboxes();
}

function addMinterRoles() public {
    addMinterRole(frxUsd, ethFrxUsdLockbox);
    addMinterRole(sfrxUsd, ethSFrxUsdLockbox);
}
```

Impact

Because the caller cannot pass the token contract's validation, the script execution will fail.

Recommendations

Call the function `addMinterRole` with the appropriate address.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #106. They intend to merge these changes into the production branch.

3.7. Redundant call to the function `approve`

Target	FraxtalMinter		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `burnFrxUSD` allows the contract's owner or operator to burn the frxUSD tokens held by the contract. It calls the function `approve` of frxUSD to grant itself an allowance, but burning the tokens held by the contract itself does not consume the allowance.

```
function burnFrxUSD(uint256 amount) external {
    if (msg.sender != owner() && msg.sender != operator) revert NeedsAuth();
    if (amount == 0) amount = frxUSD.balanceOf(address(this));
    frxUSD.approve(address(this), amount);
    frxUSD.burn(amount);
}
```

Impact

Redundant calls consume unnecessary gas.

Recommendations

Consider removing the call to the function `approve`.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in PR #67 ↗. They intend to merge these changes into the production branch.

3.8. Centralization risk

Target	CustodianBase		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Accounts with the `DEFAULT_ADMIN_ROLE` in the contract CustodianBase can call several sensitive functions:

- `setMintRedeemFee`, which allows the privileged account to set any fee rate below 100%
- `recoverERC20`, which allows the privileged account to withdraw arbitrary ERC-20 tokens from the contract, including the custodian token
- `recoverETH`, which enables withdrawal of native tokens

Impact

The above functions introduce centralization risks that users should be aware of, as they grant a single point of control over the system.

Recommendations

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

Remediation

This issue has been acknowledged by Frax Finance.

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. Oracle issue in RemoteCustodianWithOracle

Since withdrawals use the oracle price, a participant may deposit custodian tokens to receive shares and then withdraw after a price update to extract additional value. There is also the risk that, under sudden price moves, the contract might not hold enough custodian tokens to satisfy withdrawals.

The Frax Finance team has clarified that the oracle price is cached per block, so a same-block flash loan / sandwich around an oracle update is not possible; an attacker must hold the tokens for at least one block. They also emphasized that this custodian is intended for slowly moving tokens; for more volatile assets, they would set mint/redeem fees that exceed expected per-block price changes.

4.2. Refund recipient for native fee in RemoteHop

We note that in some paths the native fee refund recipient is set to `address(this)` (e.g., `RemoteCustodian._remoteMint()` calls `IOFT(address(frxUsd())).send{ value: fee.nativeFee }(sendParam, fee, address(this))`), which may be concerning given that the implementation contract is nonpayable even if deployed behind a payable proxy. Using `msg.sender` as the refund recipient would avoid the contract needing to handle refunds directly.

The Frax Finance team has explained that the deployed proxy is a payable proxy, so even if the underlying implementation cannot receive native value directly, the proxy address can receive a refund. This pattern was adopted from src/contracts/hop/RemoteHop.sol, which had been previously audited. In addition, since the native value attached to `send()` is based on a fresh `quoteSend()` immediately beforehand, a refund is unlikely in practice.

We confirmed using `address(this)` as the refund recipient is acceptable under this design and has negligible practical difference because refunds are rare. Using `msg.sender` would also be reasonable.

4.3. Hop-fee handling vs `lzCompose` value in RemoteHop

We discussed with the Frax Finance team the handling of the hop fee collected on Fraxtal during L2 → Fraxtal → L2 flows. The current approach intentionally keeps a difference between the quoted fee and the actual value passed, allowing the contract to accrue a small surplus that can be later

withdrawn (e.g., via `recoverETH()`) and allocated to FraxtalMinter to offset on-chain gas costs.

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. Contract: FrxUSD2.sol

Function: `cancelAuthorization(address authorizer, byte[32] nonce, bytes signature)`

This function marks an unused authorization nonce as used, preventing future EIP-3009 transfers with the same nonce.

Inputs

- `authorizer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: The account whose authorization nonce will be canceled.
- `nonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be an unused nonce for the `authorizer`.
  - **Impact**: The specific authorization nonce that will be canceled.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by the `authorizer`.
  - **Impact**: Authorizes the cancellation of the nonce.

Branches and code coverage

Intended branches

- `authorizationState[authorizer][nonce]` is marked as used after a successful call.
  - ☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.

- ☑ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid or not signed by `authorizer`.
  - ☐ Negative test

Function: `permit(address owner, address spender, uint256 value, uint256 deadline, bytes signature)`

This function allows users to set `_allowances[owner][spender] = value` using an off-chain signature.

Inputs

- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: Grants allowance to `spender`.
- `spender`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `spender` encoded in the `signature`.
  - **Impact**: Receives allowance from `owner`.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `value` encoded in the `signature`.
  - **Impact**: The amount of allowance to set.
- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `block.timestamp` must be less than or equal to `deadline`.
  - **Impact**: Permit validity period.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by `owner`.
  - **Impact**: Authorizes the permit action.

Branches and code coverage

Intended branches

- This function can successfully update `_allowances[owner][spender]` to `value`.
  - ☐ Test coverage
- `_nonces[owner]` is incremented by 1 after a successful call.
  - ☐ Test coverage

Negative behavior

- Reverts with `ERC2612ExpiredSignature` if `block.timestamp` is greater than `deadline`.
  - ☐ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid or not signed by `owner`.
  - ☐ Negative test

Function: `receiveWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, byte[32] nonce, bytes signature)`

This function executes an ERC-20 transfer to the caller with a signed authorization. It marks `authorizationState[from][nonce]` as used and transfers `value` amount of tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: Source of the tokens to be transferred.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the caller and equal to the `to` encoded in the `signature`.
  - **Impact**: Destination of the tokens to be transferred.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `value` encoded in the `signature` and must be less than or equal to the `from` account's balance.
  - **Impact**: Amount of tokens to be transferred.
- `validAfter`
  - **Control**: Fully controlled by the caller.

  - **Constraints**: Must be less than the current block timestamp.
  - **Impact**: Start time of the authorization window.
- `validBefore`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than the current block timestamp.
  - **Impact**: End time of the authorization window.
- `nonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be unused for the `from` address.
  - **Impact**: Prevents replay attacks.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by `from`.
  - **Impact**: Authorizes the transfer action.

Branches and code coverage

Intended branches

- The transfer succeeds and tokens are moved from `from` to the caller.
  - ☑ Test coverage
- `authorizationState[from][nonce]` is marked as used after a successful call.
  - ☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.
  - ☑ Negative test
- Reverts with `InvalidAuthorization` if the current time is before or equal to `validAfter`.
  - ☑ Negative test
- Reverts with `ExpiredAuthorization` if the current time is after or equal to `validBefore`.
  - ☑ Negative test
- Reverts with `InvalidPayee` if the `to` address does not match the caller.
  - ☑ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid.
  - ☑ Negative test

Function: `transferWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, byte[32] nonce, bytes signature)`

This function executes an ERC-20 transfer with a signed authorization. It marks `authorizationState[from][nonce]` as used and transfers `value` amount of tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: Source of the tokens to be transferred.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `to` encoded in the `signature`.
  - **Impact**: Destination of the tokens to be transferred.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `value` encoded in the `signature` and must be less than or equal to the `from` account's balance.
  - **Impact**: Amount of tokens to be transferred.
- `validAfter`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the `block.timestamp`.
  - **Impact**: Start time of the authorization window.
- `validBefore`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than the `block.timestamp`.
  - **Impact**: End time of the authorization window.
- `nonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be unused for the `from` address.
  - **Impact**: Prevents replay attacks.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by `from`.

- **Impact**: Authorizes the transfer action.

Branches and code coverage

Intended branches

- The transfer succeeds, and tokens are moved from `from` to `to`.
  - ☑ Test coverage
- `authorizationState[from][nonce]` is marked as used after a successful call.
  - ☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.
  - ☑ Negative test
- Reverts with `InvalidAuthorization` if the current time is before or equal to `validAfter`.
  - ☑ Negative test
- Reverts with `ExpiredAuthorization` if the current time is after or equal to `validBefore`.
  - ☑ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid.
  - ☑ Negative test

5.2. Contract: StakedFrxUSD2.sol

Function: `cancelAuthorization(address authorizer, byte[32] nonce, bytes signature)`

This function marks an unused authorization nonce as used, preventing future EIP-3009 transfers with the same nonce.

Inputs

- `authorizer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: The account whose authorization nonce will be canceled.
- `nonce`
  - **Control**: Fully controlled by the caller.

    - **Constraints**: Must be an unused nonce for the `authorizer`.
    - **Impact**: The specific authorization nonce that will be canceled.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid EIP-712 signature signed by the `authorizer`.
    - **Impact**: Authorizes the cancellation of the nonce.

Branches and code coverage

Intended branches

- `authorizationState[authorizer][nonce]` is marked as used after a successful call.
    - ☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.
    - ☑ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid or not signed by `authorizer`.
    - ☐ Negative test

Function: `permit(address owner, address spender, uint256 value, uint256 deadline, bytes signature)`

This function allows users to set `_allowances[owner][spender] = value` using an off-chain signature.

Inputs

- `owner`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be the signer of the `signature`.
    - **Impact**: Grants allowance to the `spender`.
- `spender`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be equal to the `spender` encoded in the `signature`.
    - **Impact**: Receives allowance from the `owner`.

- `value`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be equal to the `value` encoded in the `signature`.
    - **Impact**: The amount of allowance to set.
- `deadline`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The `block.timestamp` must be less than or equal to `deadline`.
    - **Impact**: Permit validity period.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid EIP-712 signature signed by `owner`.
    - **Impact**: Authorizes the permit action.

Branches and code coverage

Intended branches

- This function can successfully update `_allowances[owner][spender]` to `value`.
    - ☐ Test coverage
- `_nonces[owner]` is incremented by 1 after a successful call.
    - ☐ Test coverage

Negative behavior

- Reverts with `ERC2612ExpiredSignature` if `block.timestamp` is greater than `deadline`.
    - ☐ Negative test
- Reverts with `InvalidSignature` if the `signature` is not valid or not signed by `owner`.
    - ☐ Negative test

Function: `receiveWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, byte[32] nonce, bytes signature)`

This function executes an ERC-20 transfer to the caller with a signed authorization. It marks `authorizationState[from][nonce]` as used and transfers `value` amount of tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: Source of the tokens to be transferred.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the caller and equal to the `to` encoded in the `signature`.
  - **Impact**: Destination of the tokens to be transferred.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `value` encoded in the `signature` and must be less than or equal to the `from` account's balance.
  - **Impact**: Amount of tokens to be transferred.
- `validAfter`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the current block timestamp.
  - **Impact**: Start time of the authorization window.
- `validBefore`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than the current block timestamp.
  - **Impact**: End time of the authorization window.
- `nonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be unused for the `from` address.
  - **Impact**: Prevents replay attacks.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by `from`.
  - **Impact**: Authorizes the transfer action.

Branches and code coverage

Intended branches

- The transfer succeeds and tokens are moved from `from` to the caller.

☑ Test coverage

- `authorizationState[from][nonce]` is marked as used after a successful call.

☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.

☑ Negative test

- Reverts with `InvalidAuthorization` if the current time is before or equal to `validAfter`.

☑ Negative test

- Reverts with `ExpiredAuthorization` if the current time is after or equal to `validBefore`.

☑ Negative test

- Reverts with `InvalidPayee` if the `to` address does not match the caller.

☑ Negative test

- Reverts with `InvalidSignature` if the `signature` is not valid.

☑ Negative test

Function: `transferWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, byte[32] nonce, bytes signature)`

This function executes an ERC-20 transfer with a signed authorization. It marks `authorizationState[from][nonce]` as used and transfers `value` amount of tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the signer of the `signature`.
  - **Impact**: Source of the tokens to be transferred.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the `to` encoded in the `signature`.
  - **Impact**: Destination of the tokens to be transferred.
- `value`
  - **Control**: Fully controlled by the caller.

  - **Constraints**: Must be equal to the `value` encoded in the `signature` and must be less than or equal to the `from` account's balance.
  - **Impact**: Amount of tokens to be transferred.
- `validAfter`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the `block.timestamp`.
  - **Impact**: Start time of the authorization window.
- `validBefore`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than the `block.timestamp`.
  - **Impact**: End time of the authorization window.
- `nonce`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be unused for the `from` address.
  - **Impact**: Prevents replay attacks.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid EIP-712 signature signed by `from`.
  - **Impact**: Authorizes the transfer action.

Branches and code coverage

Intended branches

- The transfer succeeds, and tokens are moved from `from` to `to`.
  - ☑ Test coverage
- `authorizationState[from][nonce]` is marked as used after a successful call.
  - ☑ Test coverage

Negative behavior

- Reverts with `UsedOrCanceledAuthorization` if the `nonce` has already been used or canceled.
  - ☑ Negative test
- Reverts with `InvalidAuthorization` if the current time is before or equal to `validAfter`.
  - ☑ Negative test
- Reverts with `ExpiredAuthorization` if the current time is after or equal to `validBefore`.
  - ☑ Negative test

- Reverts with `InvalidSignature` if the `signature` is not valid.
  - ☑ Negative test

5.3. Contract: FrxUSD3.sol

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

This is an owner-only function that burns `_amount` tokens from `_owner`. If `_amount == 0`, it burns the entire balance of `_owner`.

Inputs

- `_owner`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Source account to burn from.
- `_amount`
  - **Control**: N/A.
  - **Constraints**: If 0, replaced with `balanceOf(_owner)` — otherwise, must be <= `balanceOf(_owner)`.
  - **Impact**: Decreases `balanceOf[_owner]` and `totalSupply`.

Branches and code coverage

Intended branches

- When `_amount == 0`, the full balance of `_owner` is burned.
  - ☐ Test coverage
- `_amount` is burned from `_owner`.
  - ☐ Test coverage

Negative behavior

- `msg.sender` not being the owner reverts with `Ownable: caller is not the owner`.
  - ☐ Negative test
- Burn exceeding balance reverts via arithmetic underflow in `_burn`.
  - ☐ Negative test

Function: `cancelAuthorization(address authorizer, bytes32 nonce, signature)`

This function cancels a previously unused authorization by marking `nonce` as used for `authorizer`. It prevents future EIP-3009 transfers with the same `nonce`.

Inputs

- `authorizer`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the cancellation.
  - **Impact**: Sets `isAuthorizationUsed[authorizer][nonce] = true`.
- `nonce`
  - **Control**: N/A.
  - **Constraints**: Must be currently unused.
  - **Impact**: Becomes unusable for future authorizations.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(authorizer, nonce)`.
  - **Impact**: Proves cancel intent.

Branches and code coverage

Intended branches

- `nonce` is marked used.
  - ☑ Test coverage

Negative behavior

- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.
  - ☑ Negative test
- Invalid signature reverts with `InvalidSignature`.
  - ☐ Negative test

Function: `freeze(address _owner)`

This is an owner-only function that freezes an account. A frozen account cannot transfer while paused checks apply in `_update`.

Inputs

- `_owner`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Sets `isFrozen[account] = true` and emits `AccountFrozen(account)`.

Branches and code coverage

Intended branches

- Freeze updates `isFrozen[_owner] = true`.
  - ☐ Test coverage

Negative behavior

- Caller not being the owner reverts with `Ownable: caller is not the owner`.
  - ☐ Negative test

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

This is a minter-only function that burns `b_amount` of tokens from `b_address`. It internally wraps `burnFrom`, reducing `balanceOf[b_address]` and `totalSupply`. It emits `TokenMinterBurned(b_address, msg.sender, b_amount)`.

Inputs

- `b_address`
  - **Control**: N/A.
  - **Constraints**: `allowance[b_address][msg.sender]` and `balanceOf[b_address] >= b_amount` must hold.
  - **Impact**: Source account to burn from.
- `b_amount`
  - **Control**: N/A.
  - **Constraints**: Must be <= `balanceOf[b_address]`.
  - **Impact**: Decreases `balanceOf[b_address]` and `totalSupply`.

Branches and code coverage

Intended branches

- `totalSupply` and `balanceOf[b_address]` decrease by `b_amount`.
  - ☐ Test coverage

Negative behavior

- `msg.sender` not being a minter reverts with `"Only minters"`.
  - ☐ Negative test
- Insufficient allowance or balance reverts in `burnFrom`.
  - ☐ Negative test

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

This function mints `m_amount` of tokens to `m_address`. It is restricted to addresses in `minters`.

Inputs

- `m_address`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Recipient of newly minted tokens.
- `m_amount`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Increases `totalSupply` and `balanceOf[m_address]`.

Branches and code coverage

Intended branches

- `totalSupply` and `balanceOf[m_address]` increase by `m_amount`.
  - ☐ Test coverage

Negative test

- Caller not in `minters` reverts with `"Only minters"`.
  - ☐ Negative test
- Mint to zero address reverts via ERC-20 logic.

☐ Negative test

Function: `permit(address owner, address spender, uint256 value, uint256 deadline, signature)`

This is an EIP-2612 permit with ERC-1271 support. It sets `allowance[owner][spender] = value` using an off-chain signature. It emits `Approval(owner, spender, value)`.

Inputs

- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the permit.
  - **Impact**: Grants allowance to `spender`.
- `spender`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Can spend up to `value` on behalf of `owner`.
- `value`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Allowance to set.
- `deadline`
  - **Control**: N/A.
  - **Constraints**: `block.timestamp <= deadline` must hold.
  - **Impact**: Permit validity window.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(owner,spender,value,nonce,deadline)`.
  - **Impact**: Proves approval intent.

Branches and code coverage

Intended branches

- `allowance[owner][spender]` updated to `value`.

☑ Test coverage

- `nonces[owner]` increments by 1.
  - ☐ Test coverage
- ERC-1271 contract wallets are accepted via `isValidSignatureNow`.
  - ☐ Test coverage

Negative behavior

- `block.timestamp > deadline` reverts with `"Permit: expired deadline"`.
  - ☐ Negative test
- Invalid signature (including incorrect spender/value/nonce/deadline) reverts with `InvalidSignature`.
  - ☐ Negative test

Function: `receiveWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, bytes32 nonce, signature)`

This function executes a gasless transfer to `msg.sender` authorized by `from` per EIP-3009. Enforces `to == msg.sender` to prevent front-running. It marks `nonce` as used and transfers tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the authorization.
  - **Impact**: Balance decreases by `value`.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must equal `msg.sender` and the `to` encoded in the signed authorization.
  - **Impact**: Recipient of `value`.
- `value`
  - **Control**: N/A.
  - **Constraints**: Must be <= `balanceOf[from]`.
  - **Impact**: Transfer amount.
- `validAfter`, `validBefore`
  - **Control**: N/A.

  - **Constraints**: `block.timestamp > validAfter` and `block.timestamp < validBefore` must hold.
  - **Impact**: Authorization time window.
- `nonce`
  - **Control**: N/A.
  - **Constraints**: Must be unused for `from`.
  - **Impact**: Prevents replay.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(from,to,value,validAfter,validBefore,nonce)`.
  - **Impact**: Proves authorization.

Branches and code coverage

Intended branches

- Authorization `nonce` is marked used exactly once.
  - ☑ Test coverage
- Transfer succeeds and updates balances.
  - ☑ Test coverage

Negative behavior

- `to != msg.sender` reverts with `InvalidPayee(caller, payee)`.
  - ☑ Negative test
- `block.timestamp <= validAfter` reverts with `InvalidAuthorization`.
  - ☑ Negative test
- `block.timestamp >= validBefore` reverts with `ExpiredAuthorization`.
  - ☑ Negative test
- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.
  - ☑ Negative test
- Signature mismatch reverts with `InvalidSignature`.
  - ☑ Negative test

Function: `thaw(address _owner)`

This is an owner-only function that unfreezes an account. Thawed accounts resume normal transfer behavior.

Inputs

- `_owner`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Sets `isFrozen[account] = false` and emits `AccountThawed(account)`.

Branches and code coverage

Intended branches

- Thaw updates `isFrozen[_owner] = false`.
  - ☐ Test coverage

Negative behavior

- Caller not being the owner reverts with `Ownable: caller is not the owner`.
  - ☐ Negative test

Function: `transferWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, bytes32 nonce, signature)`

This function executes a gasless ERC-20 transfer authorized by `from` per EIP-3009. It marks `nonce` as used and transfers tokens from `from` to `to`.

Inputs

- `from`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the authorization.
  - **Impact**: Balance decreases by `value`.
- `to`
  - **Control**: N/A.

    - **Constraints**: Must equal the `to` encoded in the signed authorization.
    - **Impact**: Balance increases by `value`.
- `value`
    - **Control**: N/A.
    - **Constraints**: Must be <= `balanceOf[from]`.
    - **Impact**: Transfer amount.
- `validAfter`, `validBefore`
    - **Control**: N/A.
    - **Constraints**: `block.timestamp > validAfter` and `block.timestamp < validBefore` must hold.
    - **Impact**: Authorization time window.
- `nonce`
    - **Control**: N/A.
    - **Constraints**: Must be unused for `from`.
    - **Impact**: Prevents replay.
- `signature`
    - **Control**: N/A.
    - **Constraints**: Must be a valid EIP-712 signature over `(from,to,value,validAfter,validBefore,nonce)`.
    - **Impact**: Proves authorization.

Branches and code coverage

Intended branches

- Authorization `nonce` is marked used exactly once.
    - ☑ Test coverage
- Transfer succeeds and updates balances.
    - ☑ Test coverage

Negative behavior

- `block.timestamp <= validAfter` reverts with `InvalidAuthorization`.
    - ☑ Negative test
- `block.timestamp >= validBefore` reverts with `ExpiredAuthorization`.
    - ☑ Negative test
- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.
    - ☑ Negative test

- Signature mismatch reverts with `InvalidSignature`.

  ☑ Negative test

5.4. Contract: StakedFrxUSD3.sol

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

This is an owner-only function that burns `_amount` tokens from `_owner`. If `_amount == 0`, it burns the entire balance of `_owner`.

Inputs

- `_owner`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Source account to burn from.
- `_amount`
  - **Control**: N/A.
  - **Constraints**: If 0, replaced with `balanceOf(_owner)` — otherwise must be <= `balanceOf(_owner)`.
  - **Impact**: Decreases `balanceOf[_owner]` and `totalSupply`.

Branches and code coverage

Intended branches

- When `_amount == 0`, the full balance of `_owner` is burned.

  ☐ Test coverage
- `_amount` is burned from `_owner`.

  ☐ Test coverage

Negative behavior

- `msg.sender` not being the owner reverts with `Ownable: caller is not the owner`.

  ☐ Negative test
- Burn exceeding balance reverts via arithmetic underflow in `_burn`.

  ☐ Negative test

Function: `cancelAuthorization(address authorizer, bytes32 nonce, signature)`

This function cancels a previously unused authorization by marking `nonce` as used for `authorizer`. It prevents future EIP-3009 transfers with the same `nonce`.

Inputs

- `authorizer`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the cancellation.
  - **Impact**: Sets `isAuthorizationUsed[authorizer][nonce] = true`.
- `nonce`
  - **Control**: N/A.
  - **Constraints**: Must be currently unused.
  - **Impact**: Becomes unusable for future authorizations.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(authorizer, nonce)`.
  - **Impact**: Proves cancel intent.

Branches and code coverage

Intended branches

- `nonce` is marked used.
  - ☑ Test coverage

Negative behavior

- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.
  - ☑ Negative test
- Invalid signature reverts with `InvalidSignature`.
  - ☐ Negative test

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

This function burns `b_amount` shares from `b_address`. It is restricted to addresses in `minters`. It internally wraps the vault burn path, decreasing `balanceOf[b_address]` and `totalSupply`. It emits `TokenMinterBurned(b_address, msg.sender, b_amount)`.

Inputs

- `b_address`
  - **Control**: N/A.
  - **Constraints**: `balanceOf[b_address] >= b_amount` must hold.
  - **Impact**: Source account burned from.
- `b_amount`
  - **Control**: N/A.
  - **Constraints**: Must be <= `balanceOf[b_address]`.
  - **Impact**: Decreases `balanceOf[b_address]` and `totalSupply`.

Branches and code coverage

Intended branches

- `totalSupply` and `balanceOf[b_address]` decreases by `b_amount`.
  - ☐ Test coverage
- `TokenMinterBurned(b_address, msg.sender, b_amount)` is emitted.
  - ☐ Test coverage

Negative behavior

- Caller not in `minters` reverts with `OnlyMinters`.
  - ☐ Negative test
- Burn exceeding balance reverts via arithmetic underflow.
  - ☐ Negative test

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

This function mints `m_amount` shares to `m_address`. It is restricted to addresses in `minters`.

Inputs

- `m_address`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Recipient of newly minted shares.
- `m_amount`
  - **Control**: N/A.

- **Constraints**: N/A.
- **Impact**: Increases `totalSupply` and `balanceOf[m_address]`.

Branches and code coverage

Intended branches

- `totalSupply` and `balanceOf[m_address]` increase by `m_amount`.
  - ☐ Test coverage
- `TokenMinterMinted(msg.sender, m_address, m_amount)` is emitted.
  - ☐ Test coverage

Negative behavior

- Caller not in `minters` reverts with `OnlyMinters` (StakedFrxUSD2).
  - ☑ Negative test
- Mint to zero address reverts via ERC-20 logic.
  - ☐ Test coverage

Function: `permit(address owner, address spender, uint256 value, uint256 deadline, signature)`

This is an EIP-2612 permit with ERC-1271 support. It sets `allowance[owner][spender] = value` using an off-chain signature. It emits `Approval(owner, spender, value)`.

Inputs

- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the permit.
  - **Impact**: Grants allowance to `spender`.
- `spender`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Can spend up to `value` on behalf of `owner`.
- `value`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Allowance to set.

- `deadline`
  - **Control**: N/A.
  - **Constraints**: `block.timestamp <= deadline` must hold.
  - **Impact**: Permit validity window.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(owner,spender,value,nonce,deadline)`.
  - **Impact**: Proves approval intent.

Branches and code coverage

Intended branches

- `nonces[owner]` increments by 1.
  - ☑ Test coverage
- ERC-1271 contract wallets are accepted via `isValidSignatureNow`.
  - ☑ Test coverage
- `allowance[owner][spender]` updated to `value`.
  - ☑ Test coverage

Negative behavior

- `block.timestamp > deadline` reverts with `"Permit: expired deadline"`.
  - ☐ Negative test
- Invalid signature (including incorrect `spender`/`value`/`nonce`/`deadline`) reverts with `InvalidSignature`.
  - ☐ Negative test

Function: `receiveWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, bytes32 nonce, signature)`

This function executes a gasless transfer to `msg.sender` authorized by `from` per EIP-3009. it enforces that `to` equals `msg.sender` to prevent front-running, marks `nonce` as used, and transfers shares from `from` to `to`.

Inputs

- `from`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the authorization.
  - **Impact**: Balance decreases by `value`.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must equal `msg.sender` and must equal the `to` encoded in the signed authorization.
  - **Impact**: Recipient of `value`.
- `value`
  - **Control**: N/A.
  - **Constraints**: Must be <= `balanceOf[from]`.
  - **Impact**: Transfer amount.
- `validAfter`, `validBefore`
  - **Control**: N/A.
  - **Constraints**: `block.timestamp > validAfter` and `block.timestamp < validBefore` must hold.
  - **Impact**: Authorization time window.
- `nonce`
  - **Control**: N/A.
  - **Constraints**: Must be unused for `from`.
  - **Impact**: Prevents replay.
- `signature`
  - **Control**: N/A.
  - **Constraints**: Must be a valid EIP-712 signature over `(from,to,value,validAfter,validBefore,nonce)`.
  - **Impact**: Proves authorization.

Branches and code coverage

Intended branches

- `to == msg.sender` is enforced.
  - ☑ Test coverage
- Authorization `nonce` is marked used exactly once.
  - ☑ Test coverage

- Transfer succeeds and updates balances.

  ☑ Test coverage

Negative behavior

- `to != msg.sender` reverts with `InvalidPayee(caller, payee)`.

  ☑ Negative test
- `block.timestamp <= validAfter` reverts with `InvalidAuthorization`.

  ☑ Negative test
- `block.timestamp >= validBefore` reverts with `ExpiredAuthorization`.

  ☑ Negative test
- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.

  ☑ Negative test
- Signature mismatch reverts with `InvalidSignature`.

  ☑ Negative test

Function: `sync()`

This function updates the vault's reward state by persisting the current time-accrued price per share. Specifically, it computes the instantaneous `pricePerShare` from `pricePerShareStored * exp(pricePerShareIncPerSecond * (block.timestamp - lastSync))` then writes this value back into `pricePerShareStored` and sets `lastSync = block.timestamp`.

As for the role of `pricePerShareStored`, it is the anchored rate used by all accounting views. After `sync`, subsequent calls to `totalAssets()` and `convertToAssets(shares)` reflect the newly accrued yield because `totalAssets() = pricePerShare(now) * totalSupply / 1e18` and `convertToAssets(shares) = shares * totalAssets() / totalSupply`. Thus, user balances in asset terms increase over time without minting new shares; `sync` makes that accrual explicit in storage so previews and conversions use the up-to-date rate.

Inputs

- N/A.

Branches and code coverage

Intended branches

- `pricePerShareStored` is set to `previewPricePerShare()`.

  ☐ Test coverage

- `lastSync` is set to `block.timestamp`.

  - ☐ Test coverage

Negative behavior

- N/A.

Function: `transferWithAuthorization(address from, address to, uint256 value, uint256 validAfter, uint256 validBefore, bytes32 nonce, signature)`

This function executes a gasless transfer authorized by `from` per EIP-3009. It marks `nonce` as used and transfers shares from `from` to `to` using the token's internal transfer.

Inputs

- `from`
  - **Control**: N/A.
  - **Constraints**: Must be the signer of the authorization.
  - **Impact**: Balance decreases by `value`.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must equal the `to` encoded in the signed authorization.
  - **Impact**: Balance increases by `value`.
- `value`
  - **Control**: N/A.
  - **Constraints**: Must be <= `balanceOf[from]`.
  - **Impact**: Transfer amount.
- `validAfter`, `validBefore`
  - **Control**: N/A.
  - **Constraints**: `block.timestamp > validAfter` and `block.timestamp < validBefore` must hold.
  - **Impact**: Authorization time window.
- `nonce`
  - **Control**: N/A.
  - **Constraints**: Must be unused for `from`.
  - **Impact**: Prevents replay.
- `signature`

- **Control**: N/A.
- **Constraints**: Must be a valid EIP-712 signature over `(from,to,value,validAfter,validBefore,nonce)`.
- **Impact**: Proves authorization.

Branches and code coverage

Intended branches

- Authorization `nonce` is marked used exactly once.
  - ☑ Test coverage
- Transfer succeeds and updates `balanceOf[from]` and `balanceOf[to]`.
  - ☑ Test coverage
- `Transfer` event is emitted.
  - ☐ Test coverage

Negative behavior

- `block.timestamp <= validAfter` reverts with `InvalidAuthorization`.
  - ☑ Negative test
- `block.timestamp >= validBefore` reverts with `ExpiredAuthorization`.
  - ☑ Negative test
- Reuse of `nonce` reverts with `UsedOrCanceledAuthorization`.
  - ☑ Negative test
- Signature not matching `(from,to,value,validAfter,validBefore,nonce)` reverts with `InvalidSignature`.
  - ☑ Negative test

5.5. Contract: FraxtalMinter.sol

Function: `lzCompose(address _oft, bytes32 /*_guid*/, bytes _message, address /*Executor*/, bytes /*Executor Data*/)`

This function handles incoming LayerZero composed messages, validates the sender and token, deduplicates by message hash, enforces allowed remote custodians, updates per-remote mint caps, optionally mints frxUSD to top up balance, and forwards the minted tokens locally or cross-chain.

Inputs

- `_oft`
  - **Control**: N/A.
  - **Constraints**: Must equal `oft` (immutable token address for this minter).
  - **Impact**: Originating OFT (token) address.
- `_message`
  - **Control**: N/A.
  - **Constraints**: This has the following constraints.
    - `srcEid = OFTComposeMsgCodec.srcEid(_message)` must be a supported LayerZero EID.
    - `composeFrom = OFTComposeMsgCodec.composeFrom(_message)` must be allowlisted (`eidToAllowedRemoteCustodians[srcEid][composeFrom] == true`).
    - `nonce = OFTComposeMsgCodec.nonce(_message)` must yield a unique `messageHash`.
    - `recipient` decoded via `composeMsg(_message)` must encode a valid EVM address when `_dstEid == 30255`.
    - `_dstEid` decoded via `composeMsg(_message)` must be a valid LayerZero EID.
    - `amount` decoded via `composeMsg(_message)` must not exceed the remaining per-remote mint cap.
  - **Impact**: Encodes source/destination metadata and mint/forwarding parameters.

Branches and code coverage

Intended branches

- Tops up local frxUSD balance if needed — if the current balance is lower than the requested amount, the contract mints the shortfall to itself.

  ☑ Test coverage
- If the destination EID points to Fraxtal, the contract transfers frxUSD directly to the recipient address on Fraxtal. Otherwise, the contract prepares a cross-chain send with the destination chain and recipient, normalizes the minimum amount after dust removal, quotes the required native fee, authorizes the token for transfer, and instructs the OFT to deliver the funds to the destination while paying the quoted fee.

  ☑ Test coverage

Negative behavior

- Reverts with `NotEndpoint` if `msg.sender != ENDPOINT`.
  - ☑ Negative test
- Reverts with `Paused` if `paused == true`.
  - ☑ Negative test
- Reverts with `InvalidOFT` if `_oft != oft`.
  - ☑ Negative test
- Returns early if `messageProcessed[keccak256(abi.encode(_oft, srcEid, nonce, composeFrom))] == true` (duplicate message).
  - ☑ Negative test
- Reverts with `InvalidRemoteHop` if `eidToAllowedRemoteCustodians[srcEid][composeFrom] == false`.
  - ☑ Negative test
- Reverts with `MintCapExceeded` if `currentMinted + amount > mintCap` for the `(srcEid, composeFrom)` pair.
  - ☑ Negative test

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

This is an owner-only function that recovers arbitrary ERC-20 tokens held by the contract and sends them to a specified recipient.

Inputs

- `tokenAddress`
  - **Control**: N/A.
  - **Constraints**: Must be a valid ERC-20 token contract address.
  - **Impact**: Token contract to transfer from this contract.
- `recipient`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Destination address receiving the tokens.
- `tokenAmount`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to transfer.

Branches and code coverage

Intended branches

- Transfers `tokenAmount` of `tokenAddress` to `recipient` using `IERC20(tokenAddress).transfer(recipient, tokenAmount)`.
  - ☑ Test coverage

Function: `recoverETH(address recipient, uint256 tokenAmount)`

This is an owner-only function to recover native ETH held by the contract by sending it to a specified recipient via a low-level call.

Inputs

- `recipient`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Address to receive the ETH.
- `tokenAmount`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Amount of ETH (in WEI) to forward.

Branches and code coverage

Intended branches

- Performs `payable(recipient).call{ value: tokenAmount }("")` and expects success.
  - ☑ Test coverage

5.6. Contract: RemoteCustodianUsdc.sol

Function: `deposit(uint256 _assetsIn, uint32 _receiverEid, address _receiver)`

This function inherits the external function `deposit` of the contract RemoteCustodian to transfer USDC in and mint shares (frxUSD) locally or via OFT hop to Fraxtal.

Inputs

- `_assetsIn`
  - **Control**: N/A.
  - **Constraints**: Must be <= `maxDeposit(_receiver)`.
  - **Impact**: Amount of USDC deposited.
- `_receiverEid`
  - **Control**: N/A.
  - **Constraints**: Valid LayerZero EID.
  - **Impact**: Destination chain for frxUSD mint.
- `_receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Recipient of shares.

Branches and code coverage

Intended branches

- Pulls custodian tokens in `custodianTkn().safeTransferFrom(msg.sender, address(this), _assetsIn)`.
  - ☑ Test coverage
- Local path — if `_receiverEid == localEid()` and `frxUSD.balanceOf(address(this)) >= _sharesOut`, transfers frxUSD to `_receiver` and refunds any excess `msg.value`.
  - ☑ Test coverage
- Remote path — otherwise, builds OFT send (compose message encodes recipient, `dstEid`, amount), quotes fees and calls `IOFT.send{value: fee.nativeFee}`, and refunds surplus ETH.
  - ☑ Test coverage
- Minted accounting increases via `_addToFrxUsdMinted(_sharesOut)` and remains <= `mintCap()`.
  - ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxDeposit` if `_assetsIn > maxDeposit(_receiver)`.
  - ☐ Negative test
- Reverts with `"Transfer failed"` if local-path ETH refund to `_receiver` fails.
  - ☐ Negative test

- Reverts with `InsufficientFee` if `msg.value` is less than `quoteSend.nativeFee + quoteHop(_receiverEid)` on remote path.
  - ☐ Negative test
- Reverts with `RefundFailed` if ETH refund to the sender fails.
  - ☐ Negative test
- Reverts with `MintCapExceeded` if cumulative minted exceeds `mintCap()`.
  - ☑ Negative test

Function: `depositWithAuthorization(uint32 _receiverEid, bytes _receiveWithAuthorization)`

This function adds a gasless deposit path using EIP-3009 `receiveWithAuthorization` to pull USDC from the signer and mint shares.

Inputs

- `_receiverEid`
  - **Control**: N/A.
  - **Constraints**: `bytes32(_receiveWithAuthorization[160:192])` must equal `keccak256(abi.encode(_receiverEid, _receiveWithAuthorization[0:160]))`.
  - **Impact**: Binds EID in the authorization.
- `_receiveWithAuthorization`
  - **Control**: N/A.
  - **Constraints**: This has the following constraints.
    - `_receiveWithAuthorization[32:64]` must equal `address(this)`.
    - `bytes32(_receiveWithAuthorization[160:192])` must equal `keccak256(abi.encode(_receiverEid, _receiveWithAuthorization[0:160]))`.
    - `_receiveWithAuthorization[64:96]` must not be greater than `maxDeposit(abi.decode(_receiveWithAuthorization[0:32], (address)))`.
    - The token call must succeed.
  - **Impact**: Encodes receiver and asset amount.

Branches and code coverage

Intended branches

- Pulls custodian tokens via token's 3009 call: `IERC20(custodianTkn).receiveWithAuthorization(...)`.

  ☑ Test coverage
- Local path — if `_receiverEid == localEid()` and `frxUSD.balanceOf(address(this)) >= _sharesOut`, transfers frxUSD to `_receiver` and refunds any excess `msg.value`.

  ☑ Test coverage
- Remote path — otherwise, builds OFT send (compose message encodes recipient, `dstEid`, amount), quotes fees and calls `IOFT.send{value: fee.nativeFee}`, and refunds surplus ETH.

  ☑ Test coverage
- Minted accounting increases via `_addToFrxUsdMinted(_sharesOut)` and remains `<= mintCap()`.

  ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxDeposit` if `_assetsIn > maxDeposit(_receiver)`.

  ☐ Negative test
- Reverts with `TransferFailedEIP3009` if `_receiveWithAuthorization[32:64] != address(this)`.

  ☐ Negative test
- Reverts with `NonceNotValid` if `bytes32(_receiveWithAuthorization[160:192]) != keccak256(abi.encode(_receiverEid, _receiveWithAuthorization[0:160]))`.

  ☐ Negative test
- Reverts with `TransferFailedEIP3009` if the token call fails.

  ☐ Negative test
- Reverts with `"Transfer failed"` if local-path ETH refund to `_receiver` fails.

  ☐ Negative test
- Reverts with `InsufficientFee` if `msg.value` is less than `quoteSend.nativeFee + quoteHop(_receiverEid)` on remote path.

  ☐ Negative test
- Reverts with `RefundFailed` if ETH refund to the sender fails.

  ☐ Negative test
- Reverts with `MintCapExceeded` if cumulative minted exceeds `mintCap()`.

  ☐ Negative test

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

This function is inherited from RemoteCustodian. It rounds `_sharesOut` up to remove dust if needed, checks `maxMint`, computes `_assetsIn = previewMint(_sharesOut)`, and then calls `_deposit`.

Inputs

- `_sharesOut`
  - **Control:** N/A.
  - **Constraints:** Must be <= `maxMint(_receiver)`.
  - **Impact:** Shares to mint (rounded up for dust constraints).
- `_receiver`
  - **Control:** N/A.
  - **Constraints:** N/A.
  - **Impact:** Address receiving frxUSD (local path) or on destination chain via OFT (remote path).

Branches and code coverage

Intended branches

- Rounds shares up to meet L0 dust constraints via `_removeDustFrxUsdCeil(_sharesOut)`.
  - ☑ Test coverage
- Validates `_sharesOut <= maxMint(_receiver)`.
  - ☑ Test coverage
- Pulls custodian tokens in `custodianTkn().safeTransferFrom(msg.sender, address(this), _assetsIn)`.
  - ☑ Test coverage
- Local path — since the public entry sets `_receiverEid = localEid()`, if `frxUSD.balanceOf(address(this)) >= _sharesOut`, then it transfers frxUSD to `_receiver` and refunds any excess `msg.value`.
  - ☑ Test coverage
- Remote path — otherwise, builds OFT send (compose message encodes recipient, `dstEid`, amount), quotes fees and calls `IOFT.send{value: fee.nativeFee}`, and refunds surplus ETH.
  - ☑ Test coverage
- Minted accounting increases via `_addToFrxUsdMinted(_sharesOut)` and remains <= `mintCap()`.

- ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxMint` if `_sharesOut > maxMint(_receiver)`.
  - ☐ Negative test
- Reverts with `"Transfer failed"` if local-path ETH refund to `_receiver` fails.
  - ☐ Negative test
- Reverts with `InsufficientFee` if `msg.value` is less than `quoteSend.nativeFee + quoteHop(localEid())` on remote path.
  - ☐ Negative test
- Reverts with `RefundFailed` if ETH refund to the sender fails.
  - ☐ Negative test
- Reverts with `MintCapExceeded` if cumulative minted exceeds `mintCap()`.
  - ☑ Negative test

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

This function is inherited from RemoteCustodian. It validates ownership, checks `maxRedeem`, computes `_assetsOut = previewRedeem(_sharesIn)`, and then calls `_withdraw`.

Inputs

- `_sharesIn`
  - **Control:** N/A.
  - **Constraints:** Must be <= `maxRedeem(_owner)`.
  - **Impact:** Shares to redeem.
- `_receiver`
  - **Control:** N/A.
  - **Constraints:** N/A.
  - **Impact:** Address receiving custodian tokens.
- `_owner`
  - **Control:** N/A.
  - **Constraints:** Must equal `msg.sender`.
  - **Impact:** Shares owner.

Branches and code coverage

Intended branches

- Pulls frxUSD from caller — `frxUsd().safeTransferFrom(msg.sender, address(this), _sharesIn)`.
  - ☑ Test coverage
- Updates minted accounting — if `frxUsdMinted() < _sharesIn`, then `_resetFrxUsdMinted()` — otherwise, `_subtractFromFrxUsdMinted(_sharesIn)`.
  - ☑ Test coverage
- Transfers custodian tokens to `_receiver` — `custodianTkn().safeTransfer(_receiver, _assetsOut)`.
  - ☑ Test coverage

Negative behavior

- Reverts with `TokenOwnerShouldBeSender` if `_owner != msg.sender`.
  - ☐ Negative test
- Reverts with `ERC4626ExceededMaxRedeem` if `_sharesIn > maxRedeem(_owner)`.
  - ☐ Negative test

Function: `redeemWithAuthorization(bytes _receiveWithAuthorization)`

This function is inherited from RemoteCustodian — gasless frxUSD redemption using EIP-3009 and the `_redeem3009` helper.

Inputs

- `_receiveWithAuthorization`
  - **Control**: N/A.
  - **Constraints**: `_receiveWithAuthorization[32:64]` must equal `address(this)`, `_receiveWithAuthorization[64:96]` must not be greater than `maxRedeem(abi.decode(_receiveWithAuthorization[0:32], (address)))`, and the token call must succeed.
  - **Impact**: Encodes `_receiver` and `_sharesIn`.

Branches and code coverage

Intended branches

- Calls `_redeem3009`, which enforces `to == address(this)` and pulls frxUSD via 3009.

☑ Test coverage

- Updates minted accounting — if `frxUsdMinted() < _sharesIn`, then `_resetFrxUsdMinted()` — otherwise, `_subtractFromFrxUsdMinted(_sharesIn)`.

☑ Test coverage

- Transfers custodian tokens to `_receiver` — `custodianTkn().safeTransfer(_receiver, _assetsOut)`.

☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxRedeem` if `_sharesIn > maxRedeem(_receiver)`.

☐ Negative test

- Reverts with `TransferFailedEIP3009` if `_receiveWithAuthorization[32:64] != address(this)`.

☐ Negative test

- Reverts with `TransferFailedEIP3009` if the token call fails.

☐ Negative test

Function: `sweepExcessToMainnet(uint256 amount)`

This is a permissioned function (requires `OPERATOR_ROLE`) that sweeps excess USDC to Ethereum Mainnet using CCTP V2.

Inputs

- `amount`
  - **Control**: N/A.
  - **Constraints**: Must leave post-sweep balance >= `targetMinimumBalance`.
  - **Impact**: USDC amount to sweep to mainnet.

Branches and code coverage

Intended branches

- Approves the CCTP messenger to pull funds — `custodianTkn().approve(cctpTokenMessengerLocal, amount)`.

☑ Test coverage

- Calls `ITokenMessenger.depositForBurn` with `amount = amount`, `dstDomain = MAINNET_CCTP_DOMAIN`, and `recipient = bytes32(uint(uint160(MAINNET_CUSTODIAN)))` as well as `burnToken =`

`address(custodianTkn())`, `dstCaller = bytes32(0)`, `fee = 0`, and `minFinality = 2000`.

- ☑ Test coverage

Negative behavior

- Reverts with `TargetMinNotSet` if `targetMinimumBalance == 0`.
  - ☑ Negative test
- Reverts with `AmountTooHigh` if post-sweep balance would fall below target minimum.
  - ☑ Negative test

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

This function is inherited from RemoteCustodian. It computes `_sharesIn = previewWithdraw(_assetsOut)` and transfers the custodian tokens out after pulling frxUSD in.

Inputs

- `_assetsOut`
  - **Control:** N/A.
  - **Constraints:** Must be <= `maxWithdraw(_owner)`.
  - **Impact:** Custodian tokens to withdraw.
- `_receiver`
  - **Control:** N/A.
  - **Constraints:** N/A.
  - **Impact:** Address receiving custodian tokens.
- `_owner`
  - **Control:** N/A.
  - **Constraints:** Must equal `msg.sender`.
  - **Impact:** Shares owner.

Branches and code coverage

Intended branches

- Pulls frxUSD from caller — `frxUsd().safeTransferFrom(msg.sender, address(this), _sharesIn)`.
  - ☑ Test coverage

- Updates minted accounting — if `frxUsdMinted() < _sharesIn`, then `_resetFrxUsdMinted()` — otherwise, `_subtractFromFrxUsdMinted(_sharesIn)`.

  ☑ Test coverage
- Transfers custodian tokens to `_receiver` — `custodianTkn().safeTransfer(_receiver, _assetsOut)`.

  ☑ Test coverage

Negative behavior

- Reverts with `TokenOwnerShouldBeSender` if `_owner != msg.sender`.

  ☐ Negative test
- Reverts with `ERC4626ExceededMaxWithdraw` if `_assetsOut > maxWithdraw(_owner)`.

  ☐ Negative test

5.7. Contract: RemoteCustodianWithOracle.sol

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

This function overrides the external function `deposit` of contract RemoteCustodian, adding the `updateOracle` modifier to cache and apply the oracle price to ERC-4626 conversions. The remote variant `deposit(uint256 _assetsIn, uint32 _receiverEid, address _receiver)` shares the same behavior.

Inputs

- `_assetsIn`
  - **Control**: N/A.
  - **Constraints**: Must be <= `maxDeposit(_receiver)`.
  - **Impact**: Amount of custodian tokens deposited.
- `_receiverEid` (remote variant only)
  - **Control**: N/A.
  - **Constraints**: Valid LayerZero EID.
  - **Impact**: Destination chain for frxUSD mint.
- `_receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Recipient of the ERC-4626 shares (frxUSD units).

Branches and code coverage

Intended branches

- `updateOracle` caches `lastSavedOraclePrice` and `lastOracleUpdate` at function entry.
  - ☑ Test coverage
- `previewDeposit(_assetsIn)` uses oracle-adjusted `_convertToShares` to compute `_sharesOut`.
  - ☑ Test coverage
- Pulls custodian tokens in `custodianTkn().safeTransferFrom(msg.sender, address(this), _assetsIn)`.
  - ☑ Test coverage
- Local path — if `_receiverEid == localEid()` and `frxUSD.balanceOf(address(this)) >= _sharesOut`, transfer frxUSD to `_receiver` and refund any excess `msg.value`.
  - ☑ Test coverage
- Remote path — otherwise, build OFT `SendParam` (compose message encodes recipient, `dstEid`, amount), quote fees, send via `IOFT.send{value: fee.nativeFee}`, and refund surplus ETH.
  - ☑ Test coverage
- Minted accounting increases via `_addToFrxUsdMinted(_sharesOut)` and remains `<= mintCap()`.
  - ☑ Test coverage
- Emits `Deposit(sender, receiver, assets, shares)` with expected values.
  - ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxDeposit` if `_assetsIn > maxDeposit(_receiver)`.
  - ☐ Negative test
- Reverts with `"Transfer failed"` if local-path ETH refund to `_receiver` fails.
  - ☐ Negative test
- Reverts with `InsufficientFee()` if `msg.value < (quoteSend.nativeFee + quoteHop(_receiverEid))` on remote path.
  - ☐ Negative test
- Reverts with `RefundFailed()` if ETH refund to the sender fails.
  - ☐ Negative test
- Reverts with `MintCapExceeded` if cumulative minted exceeds `mintCap()`.
  - ☐ Negative test

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

This function overrides `RemoteCustodian.mint` by adding `updateOracle`. It rounds `_sharesOut` up to remove dust per base logic, then computes `_assetsIn = previewMint(_sharesOut)` using oracle-adjusted conversion, and calls `_deposit`.

Inputs

- `_sharesOut`
  - **Control:** N/A.
  - **Constraints:** `_sharesOut <= maxMint(_receiver)` after dust-ceil.
  - **Impact:** Shares to mint (frxUSD units).
- `_receiver`
  - **Control:** N/A.
  - **Constraints:** N/A.
  - **Impact:** Recipient of minted shares.

Branches and code coverage

Intended branches

- `updateOracle` caches `lastSavedOraclePrice` and `lastOracleUpdate` at function entry.
  - ☑ Test coverage
- Rounds shares up to meet L0 dust constraints via `_removeDustFrxUsdCeil`.
  - ☑ Test coverage
- Pulls custodian tokens in `custodianTkn().safeTransferFrom(msg.sender, address(this), _assetsIn)`.
  - ☑ Test coverage
- Mints shares by calling `_remoteMint(localEid(), _receiver, _sharesOut)`.
  - Local path — if local and balance suffice, transfers frxUSD to `_receiver` and refunds any excess `msg.value`.
  - Remote path — otherwise, quotes fees and sends compose message via `IOFT.send{value: fee.nativeFee}` and refunds surplus ETH.
    - ☑ Test coverage
- Minted accounting increases via `_addToFrxUsdMinted(_sharesOut)` and remains `<= mintCap()`.
  - ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxMint` if `_sharesOut > maxMint(_receiver)`.
  - ☐ Negative test
- Reverts with `"Transfer failed"` if local-path ETH refund to `_receiver` fails.
  - ☐ Negative test
- Reverts with `InsufficientFee` if `msg.value` is less than `quoteSend.nativeFee + quoteHop(localEid())` on the remote path.
  - ☐ Negative test
- Reverts with `RefundFailed` if ETH refund to the sender fails on remote path.
  - ☐ Negative test
- Reverts with `MintCapExceeded` if cumulative minted exceeds `mintCap()`.
  - ☐ Negative test

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

This function overrides `RemoteCustodian.redeem` by adding `updateOracle`. Validates ownership and limits, computes `_assetsOut = previewRedeem(_sharesIn)` using oracle-adjusted conversions, and calls `_withdraw`.

Inputs

- `_sharesIn`
  - **Control:** N/A.
  - **Constraints:** Must be <= `maxRedeem(_owner)`.
  - **Impact:** Shares to redeem.
- `_receiver`
  - **Control:** N/A.
  - **Constraints:** N/A.
  - **Impact:** Address receiving custodian tokens.
- `_owner`
  - **Control:** N/A.
  - **Constraints:** Must equal `msg.sender`.
  - **Impact:** Shares owner.

Branches and code coverage

Intended branches

- `updateOracle` caches `lastSavedOraclePrice` and `lastOracleUpdate` at function entry.
  - ☑ Test coverage
- Validates ownership — requires `_owner == msg.sender`.
  - ☑ Test coverage
- Checks `maxRedeem(_owner)` and computes `_assetsOut = previewRedeem(_sharesIn)` using oracle-adjusted `_convertToAssets`.
  - ☑ Test coverage
- Pulls frxUSD from caller — `frxUsd().safeTransferFrom(msg.sender, address(this), _sharesIn)`.
  - ☑ Test coverage
- Decreases minted accounting — `_subtractFromFrxUsdMinted(_sharesIn)` or `_resetFrxUsdMinted()` when necessary.
  - ☑ Test coverage
- Transfers custodian tokens out to `_receiver` and emits `Withdraw` with expected values.
  - ☑ Test coverage

Negative behavior

- Reverts with `TokenOwnerShouldBeSender` if `_owner != msg.sender`.
  - ☐ Negative test
- Reverts with `ERC4626ExceededMaxRedeem` if `_sharesIn > maxRedeem(_owner)`.
  - ☐ Negative test

Function: `redeemWithAuthorization(bytes _receiveWithAuthorization)`

This function overrides `RemoteCustodian.redeemWithAuthorization` by adding `updateOracle`. It decodes the receiver and `_sharesIn` from the EIP-3009 payload, enforces max limits, computes oracle-adjusted `_assetsOut = previewRedeem(_sharesIn)`, and then calls `_redeem3009`.

Inputs

- `_receiveWithAuthorization`
  - **Control**: N/A.
  - **Constraints**: `_receiveWithAuthorization[32:64]` must equal `address(this)`, `_receiveWithAuthorization[64:96]` must not be greater than `maxRedeem(abi.decode(_receiveWithAuthorization[0:32], (address)))`, and the token call must succeed.
  - **Impact**: Determines the receiver and `_sharesIn`.

Branches and code coverage

Intended branches

- Oracle cache is updated via `updateOracle`.
  - ☑ Test coverage
- Decodes `_receiver` and `_sharesIn` from the payload and computes `_assetsOut = previewRedeem(_sharesIn)` using oracle-adjusted conversions.
  - ☑ Test coverage
- Calls `_redeem3009` to pull frxUSD via 3009 and perform redemption.
  - ☑ Test coverage
- Decreases minted accounting via `_subtractFromFrxUsdMinted(_sharesIn)` or `_resetFrxUsdMinted()`.
  - ☑ Test coverage
- Transfers custodian tokens out to `_receiver`.
  - ☑ Test coverage

Negative behavior

- Reverts with `ERC4626ExceededMaxRedeem` if `_sharesIn > maxRedeem(_receiver)`.
  - ☐ Negative test
- Reverts with `TransferFailedEIP3009` if `_receiveWithAuthorization[32:64] != address(this)`.
  - ☐ Negative test
- Reverts with `TransferFailedEIP3009` if the token call fails.
  - ☐ Negative test

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

This function overrides `RemoteCustodian.withdraw` by adding `updateOracle`. It computes `_sharesIn = previewWithdraw(_assetsOut)` using oracle-adjusted conversions and executes `_withdraw` after checking ownership and max limits.

Inputs

- `_assetsOut`
  - **Control:** N/A.
  - **Constraints:** Must be <= `maxWithdraw(_owner)`.

    - **Impact:** Underlying custodian tokens to withdraw.
- `_receiver`
    - **Control:** N/A.
    - **Constraints:** N/A.
    - **Impact:** Asset recipient.
- `_owner`
    - **Control:** N/A.
    - **Constraints:** Must equal `msg.sender`.
    - **Impact:** Shares owner.

Branches and code coverage

Intended branches

- `updateOracle` caches `lastSavedOraclePrice` and `lastOracleUpdate` at function entry.
    - ☑ Test coverage
- Validates ownership — requires `_owner == msg.sender`.
    - ☑ Test coverage
- Checks `maxWithdraw(_owner)` and computes `_sharesIn = previewWithdraw(_assetsOut)` using oracle-adjusted `_convertToShares`.
    - ☑ Test coverage
- Pulls frxUSD from the caller — `frxUsd().safeTransferFrom(msg.sender, address(this), _sharesIn)`.
    - ☑ Test coverage
- Decreases minted accounting — `_subtractFromFrxUsdMinted(_sharesIn)` or `_resetFrxUsdMinted()` when necessary.
    - ☑ Test coverage
- Transfers custodian tokens out to `_receiver` and emits `Withdraw` with expected values.
    - ☑ Test coverage

Negative behavior

- Reverts with `TokenOwnerShouldBeSender` if `_owner != msg.sender`.
    - ☐ Negative test
- Reverts with `ERC4626ExceededMaxWithdraw` if `_assetsOut > maxWithdraw(_owner)`.
    - ☐ Negative test

5.8. Contract: FraxOFTMintableAdapterUpgradeable.sol

The contract FraxOFTMintableAdapterUpgradeable is used to upgrade the logic of the previous contract FraxOFTAdapterUpgradeable. It newly inherits the contract SupplyTrackingModule, making it easier to track incoming and outgoing token transfers across different chains.

Function: `recover()`

This function sends the remaining `innerToken` in the contract to the contract owner. This function can be called by anyone.

Branches and code coverage

Intended branches

- This function can be successfully executed.
  - ☑ Test coverage
- The contract balance becomes zero after the call.
  - ☑ Test coverage
- The contract owner receives the tokens.
  - ☑ Test coverage

Function: `_credit(address _to, uint256 _amountLD, uint32 _srcEid)`

This function overrides the internal function `_credit` of contract OFTAdapterUpgradeable, adding updates to `totalTransferFromSum` and `totalTransferFrom[_srcEid]` and replacing the token transfer with a mint operation.

Inputs

- `_to`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The address to credit the tokens to.
- `_amountLD`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The amount to credit in local decimals.
- `_srcEid`

- **Control**: N/A.
- **Constraints**: N/A.
- **Impact**: The source chain ID.

Branches and code coverage

Intended branches

- The `totalTransferFromSum` is updated correctly.
  - ☑ Test coverage
- The `totalTransferFrom[_srcEid]` is updated correctly.
  - ☑ Test coverage
- The recipient balance increases.
  - ☑ Test coverage
- The total supply of the `innerToken` increases.
  - ☑ Test coverage

Function: `_debit(uint256 _amountLD, uint256 _minAmountLD, uint32 _dstEid)`

This function overrides the internal function `_debit` of contract OFTAdapterUpgradeable, adding updates to `totalTransferToSum` and `totalTransferTo[_dstEid]` and replacing the token transfer with a burn operation.

Inputs

- `_amountLD`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The amount to send in local decimals.
- `_minAmountLD`
  - **Control**: N/A.
  - **Constraints**: It must not be greater than the `_amountLD` after dust removal.
  - **Impact**: The minimum amount to send in local decimals.
- `_dstEid`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The destination chain ID.

Branches and code coverage

Intended branches

- The `totalTransferToSum` is updated correctly.
  - ☑ Test coverage
- The `totalTransferTo[_dstEid]` is updated correctly.
  - ☑ Test coverage
- The caller balance decreases.
  - ☑ Test coverage
- The total supply of the `innerToken` decreases.
  - ☑ Test coverage

Negative behavior

- Reverts if `_minAmountLD` is greater than the `_amountLD` after dust removal.
  - ☐ Negative test

5.9. Contract: FraxOFTUpgradeable.sol

The contract FraxOFTUpgradeable has additionally inherited the contract EIP3009Module and contract PermitModule on top of its original inheritance, enabling approval and transfer via signatures.

During the upgrade, the function `initializeV110` needs to be called to initialize the contract EIP712Upgradeable, which will be used to generate the sign messages.

5.10. Contract: FrxUSDOFTUpgradeable.sol

The contract FrxUSDOFTUpgradeable has additionally inherited the contract EIP3009Module, contract PermitModule, contract FreezeThawModule, and contract PauseModule on top of its original inheritance.

During the upgrade, the function `initializeV110` needs to be called to initialize the contract EIP712Upgradeable, which will be used to generate the sign messages.

Function: `addFreezer(address account)`

This function allows the owner to add a freezer role to an account.

Inputs

- `account`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Must not be a freezer.
  - **Impact**: The account to add as a freezer.

Branches and code coverage

Intended branches

- The function can successfully add a new freezer.
  - ☑ Test coverage
- The length of the set `freezers` increases by 1.
  - ☑ Test coverage

Negative behavior

- Reverts if the account is already a freezer.
  - ☑ Negative test
- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `burnMany(address[] accounts, uint256[] amounts)`

This function allows the owner to batch-burn balances from multiple accounts.

Inputs

- `accounts`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must match the length of `amounts`.
  - **Impact**: An array of accounts whose balances will be burned.
- `amounts`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must match the length of `accounts`.
  - **Impact**: An array of amounts corresponding to the balances to be burned. If an amount is `0`, the entire balance of the corresponding account will be burned.

Branches and code coverage

Intended branches

- The function is successfully executed.
  - ☑ Test coverage
- The balance updates for each account are as expected.
  - ☐ Test coverage

Negative behavior

- Reverts if the lengths of `accounts` and `amounts` do not match.
  - ☑ Negative test
- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `burn(address account, uint256 amount)`

This function allows the owner to burn a specified amount of tokens from a given account. If the amount is set to `0`, the entire balance of the account will be burned.

Inputs

- `account`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The account whose balance will be burned.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of balance to burn.

Branches and code coverage

Intended branches

- The function can burn partial balance from the account when `amount` is greater than `0`.
  - ☑ Test coverage
- The function can burn the entire balance from the account when `amount` is `0`.

☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.

  ☑ Negative test

Function: `freezeMany(address[] accounts)`

This function allows the owner or a freezer to freeze multiple accounts at once.

Inputs

- `accounts`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The accounts to freeze.

Branches and code coverage

Intended branches

- The function can successfully freeze multiple accounts by the owner.

  ☑ Test coverage
- The function can successfully freeze multiple accounts by a freezer.

  ☑ Test coverage
- The length of the set `frozen` increases by the number of accounts frozen.

  ☑ Test coverage

Negative behavior

- Reverts if the caller is neither the owner nor a freezer.

  ☑ Negative test

Function: `freeze(address account)`

This function allows the owner or a freezer to freeze an account.

Inputs

- `account`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The account to freeze.

Branches and code coverage

Intended branches

- The function can successfully freeze an account by the owner.
  - ☑ Test coverage
- The function can successfully freeze an account by a freezer.
  - ☑ Test coverage
- The length of the set `frozen` increases by 1.
  - ☑ Test coverage
- The function does not revert when the account is already frozen.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is neither the owner nor a freezer.
  - ☑ Negative test

Function: `removeFreezer(address account)`

This function allows the owner to remove a freezer role from an account.

Inputs

- `account`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Must be a freezer.
  - **Impact**: The account to remove as a freezer.

Branches and code coverage

Intended branches

- The function can successfully remove an existing freezer.
  - ☑ Test coverage
- The length of the set `freezers` decreases by 1.
  - ☑ Test coverage

Negative behavior

- Reverts if the account is not a freezer.
  - ☑ Negative test
- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `thawMany(address[] accounts)`

This function allows the owner to unfreeze multiple accounts at once.

Inputs

- `accounts`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: The accounts to unfreeze.

Branches and code coverage

Intended branches

- The function can successfully unfreeze multiple accounts.
  - ☑ Test coverage
- The length of the set `frozen` decreases by the number of accounts unfrozen.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `thaw(address account)`

This function allows the owner to unfreeze an account.

Inputs

- `account`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: The account to unfreeze.

Branches and code coverage

Intended branches

- The function can successfully unfreeze an account.
  - ☑ Test coverage
- The length of the set `frozen` decreases by 1.
  - ☑ Test coverage
- The function does not revert when the account is not frozen.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

5.11. Contract: SFrxUSDOFTUpgradeable.sol

The contract SFrxUSDOFTUpgradeable has additionally inherited the contract EIP3009Module and contract PermitModule on top of its original inheritance, enabling approval and transfer via signatures.

During the upgrade, the function `initializeV110` needs to be called to initialize the contract EIP712Upgradeable, which will be used to generate the sign messages.

5.12. Contract: WFRAXTokenOFTUpgradeable.sol

The contract WFRAXTokenOFTUpgradeable has additionally inherited the contract EIP3009Module and contract PermitModule on top of its original inheritance, enabling approval and transfer via signatures.

During the upgrade, the function `initializeV110` needs to be called to initialize the contract EIP712Upgradeable, which will be used to generate the sign messages.

6. Assessment Results

During our assessment on the scoped Frax0 Mesh contracts, we discovered eight findings. No critical issues were found. One finding was of medium impact, one was 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.