Zellic

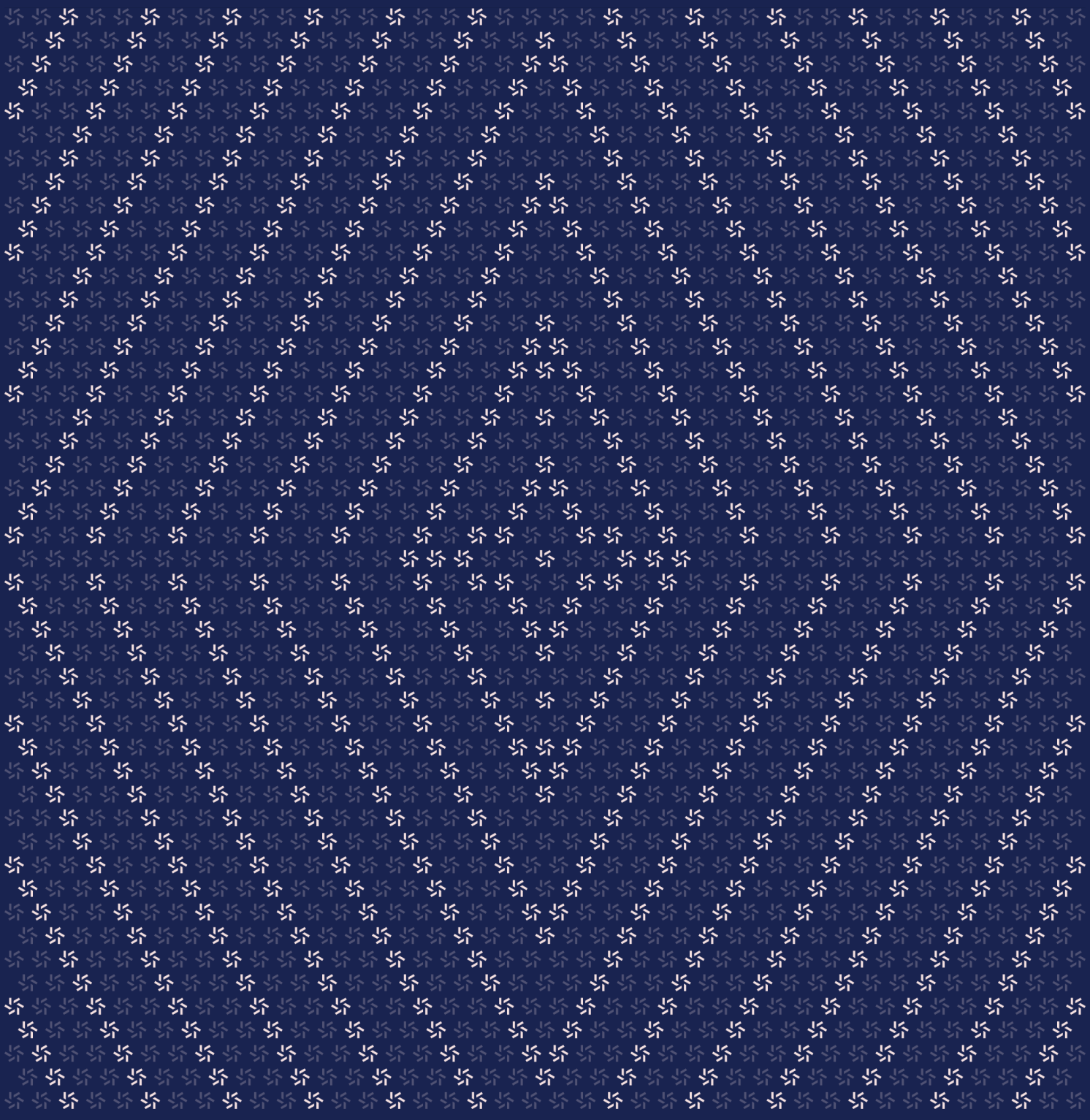
Prepared for
Sam Kazemian
Travis Moore
Frax Finance

Prepared by
Filipe Alves
Chongyu Lv
Zellic

February 13, 2026

Frax (USD Stablecoin)

Smart Contract Security Assessment



Contents

About Zellic **5**

1. Overview **5**

1.1. Executive Summary 6

1.2. Goals of the Assessment 6

1.3. Non-goals and Limitations 6

1.4. Results 7

2. Introduction **7**

2.1. About Frax (USD Stablecoin) 8

2.2. Methodology 8

2.3. Scope 10

2.4. Project Overview 13

2.5. Project Timeline 13

3. Detailed Findings **14**

3.1. BrandedSiloedCustodian passes shares where assets are expected, breaking withdrawals for non-18-decimal collateral 15

3.2. Cross-chain transfers truncate amounts to zero when branded stablecoin uses fewer than 18 decimals 17

3.3. Function `RemoteVaultDeposit.deposit` transfers before truncation, leaving dust in RemoteVaultHop 19

3.4. Unbounded `minters_array` growth risks denial of service when removing minters 21

3.5. Function `FraxNetDepositV3.recoverERC20` uses unsafe `transfer` instead of `safeTransfer`, silently failing for nonstandard tokens 23

3.6. Emergency shutoff and borrow-limit checks in FraxlendLeverOdos are permanently inactive 24

3.7. Unchecked low-level call in `FraxlendLeverOdos.recoverERC20` silently fails on transfer errors 26

3.8. Missing `_disableInitializers` in GeniusDollar constructor 29

3.9. Deprecated tokens incorrectly claim support for nonfunctional mintable interfaces 30

3.10. Encoding backing token address in `CREATE2` salt reduces entropy 32

4. Discussion 33

4.1. No minimum amount validation after `removeDust` in HopV2 deposit flow 34

4.2. Redundant `removeDust` call in HopV2 deposit flow 35

4.3. Refund logic in `RemoteVaultDeposit.deposit` refunds entire contract balance instead of only excess fee 35

4.4. Cross-chain deployment marked as complete before destination chain confirms execution 36

4.5. Custom initialization pattern used instead of OpenZeppelin Initializable 37

4.6. Remediations 38

5. Threat Model 38

5.1. Module: Component #2: Branded Stablecoin / BrandedCustodian 39

5.2. Module: Component #2: Branded Stablecoin / BrandedFactory 43

5.3. Module: Component #2: Branded Stablecoin / BrandedSiloedCustodian 46

5.4. Module: Component #2: Branded Stablecoin / BrandedSiloedCustodianWithOracle 51

5.5. Module: Component #2: Branded Stablecoin / CrossChainRouter 55

5.6. Module: Component #2: Branded Stablecoin / FraxNetDepositV3 58

5.7. Module: Component #3: Odos Fraxlend Leverage Router PR #19 / FraxlendLeverOdos 61

5.8. Module: Component #1: HopV2/HopV2 64

5.9. Module: Component #7: Frax OFT Minter Array / MinterModule 66

5.10. Module: Component #1: HopV2/RemoteHopV2 68

5.11. Module: Component #1: HopV2/RemoteVaultDeposit 70

6. Assessment Results 72

6.1. Observations and recommendations 73

6.2. Disclaimer 74

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 January 23rd to February 12th, 2026. During this engagement, Zellic reviewed Frax (USD Stablecoin)'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:

- Does the invariant that `frxUSD.balanceOf(brandedCustodian) == brandedToken.totalSupply()` hold on all chains?
- For HopV2, is there any kind of address spoofing or forgery of LayerZero origination parameters?
- For frxUSD OFT minters, are there any downstream implications that minting on remote chains can cause?
- Are access-control mechanisms properly implemented across all components to prevent unauthorized actions?
- Can the Branded Stablecoin system maintain proper backing relationships in both frxUSD-backed and siloed collateral configurations?
- Does the Fraxlend Leverage Router correctly handle dynamic swap paths from Odos API without introducing loss of funds?
- Are there edge cases in cross-chain message passing that could lead to funds being stuck or lost?
- Can the freeze and mint functionality extensions maintain system integrity without introducing privilege escalation risks?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Third-party dependencies and external protocols (LayerZero, Odos API) beyond their integration points
- Code outside the specific changes in scope

- Read-only reentrancy for Fraxlend Odos Leverage, as large protocols are built on top of this system and the risk is accepted

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

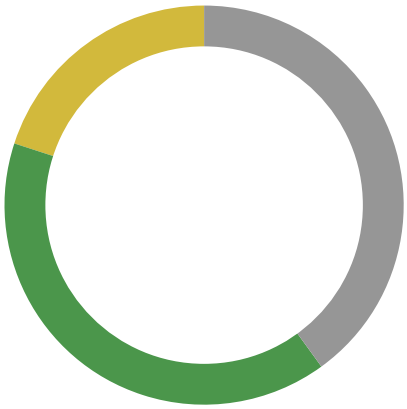
1.4. Results

During our assessment on the scoped Frax (USD Stablecoin) contracts, we discovered 10 findings. No critical issues were found. Two findings were of medium impact, four were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	4
Informational	4



2. Introduction

2.1. About Frax (USD Stablecoin)

Frax Finance contributed the following description of Frax (USD Stablecoin):

> This audit covers seven interrelated components that extend Frax's stablecoin infrastructure and cross-chain capabilities:
>
> **HopV2** enables action-based message passing via the Frax LayerZero mesh network, allowing for cross-chain interactions and coordination.
>
> **Branded Stablecoins** allows issuers to create their own multichain Genius-compatible stablecoins. Clients can choose between two configurations: tokens backed by frxUSD's existing real-world asset (RWA) portfolio, which integrate seamlessly with the existing FraxNet structure, or tokens deployed in a siloed configuration backed by a single approved RWA token.
>
> **Fraxlend Leverage Router** enables dynamic routing of swaps to create leverage within Fraxlend, utilizing paths returned from the Odos API for optimized execution.
>
> **FrxUSD Freezers Array** extends the existing frxUSD layout to allow externally owned accounts (EOAs) other than the owner to freeze accounts, providing additional access control capabilities.
>
> **Native Bridge Deprecation** introduces changes to Frax tokens on Fraxtal to disable native bridge functionality, affecting both token contracts and the underlying bridge infrastructure.
>
> **OFT Minters Array** adds support for multiple authorized minters to frxUSD Omnichain Fungible Token (OFT) implementations, extending the token's operational flexibility.

2.2. Methodology

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

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

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic

tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

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

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

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

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

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

2.3. Scope

The engagement involved a review of the following targets:

Frax (USD Stablecoin) Contracts

Type	Solidity
Platform	EVM-compatible
Target	hop-v2
Repository	https://github.com/fraxfinance/hop-v2 ↗
Version	`3dbbbca723159c06eade93852c64a59ed42f61b1`

Programs	`RemoteAdmin.sol`
	`hop/HopV2.sol`
	`hop/FraxtalHopV2.sol`
	`hop/RemoteHopV2.sol`
	`vault/RemoteVaultDeposit.sol`
	`vault/RemoteVaultHop.sol`

Target	fraxnet-dev
Repository	https://github.com/FraxFinance/fraxnet-dev ↗
Version	`Only changes between d5660fd...850d06d (PR #63)`

Programs	`BrandedStablecoin/BrandedCustodian.sol`
	`BrandedStablecoin/BrandedCustodianRegistry.sol`
	`BrandedStablecoin/BrandedFactory.sol`
	`BrandedStablecoin/BrandedSiloedCustodian.sol`
	`BrandedStablecoin/BrandedSiloedCustodianWithOracle.sol`
	`BrandedStablecoin/CrossChainRouter.sol`
	`BrandedStablecoin/FraxBeaconProxy.sol`
	`BrandedStablecoin/FraxtalBrandedDeployer.sol`
	`BrandedStablecoin/GeniusDollar.sol`
	`BrandedStablecoin/modules/EIP3009Module.sol`
	`BrandedStablecoin/modules/PermitModule.sol`
	`BrandedStablecoin/modules/SignatureModule.sol`
	`BrandedStablecoin/modules/inherited/NoncesUpgradeable.sol`
	`BrandedStablecoin/modules/FreezeThawModule.sol`
	`BrandedStablecoin/modules/PauseModule.sol`
	`CustodianRegistry.sol`
	`FraxNetDepositFactory.sol`
	`FraxNetDepositV3.sol`

Target	frax-flashloan
Repository	https://github.com/FraxFinance/frax-flashloan ↗
Version	`Only changes between ea8b368...8e840e0 (PR #19)`
Programs	`src/FraxlendLeverOdos.sol`

Target	frax-tokens
Repository	https://github.com/FraxFinance/frax-tokens ↗
Version	`Only changes between 7f652a4...6c1c472 (PR #13)`
Programs	`ethereum/frxUSD/versioning/FrxUSD2.sol` `fraxtal/frxUSD/versioning/FrxUSD2.sol`

Target	frax-tokens
Repository	https://github.com/FraxFinance/frax-tokens ↗
Version	`Only changes between b917a98...9d05cbd`

Programs	`fpi/FPI.sol`
	`fpis/FPIS.sol`
	`frxBTC/FrxBTC.sol`
	`frxUSD/FrxUSD.sol`
	`frxUSD/versioning/FrxUSD2.sol`
	`frxUSD/versioning/FrxUSD3.sol`
	`sfrxETH/SfrxETH.sol`
	`sfrxUSD/SfrxUSD.sol`
	`sfrxUSD/versioning/SfrxUSD.sol`
	`sfrxUSD/versioning/SfrxUSD2.sol`
	`shared/ERC20ExWrappedPPOM.sol`
	`shared/ERC20PermitPermissionedNonBridgeableMintable.sol`

Target	optimism-fraxtal-cgt
Repository	https://github.com/FraxFinance/optimism-fraxtal-cgt ↗
Version	`Only changes between fc2f948...920b12d (PR #41)`
Programs	`L1/L1StandardBridge.sol` `frax/FraxTokenHandler.sol`

Target	frax-oft-upgradeable
Repository	https://github.com/FraxFinance/frax-oft-upgradeable ↗
Version	`705aea54c1072072b7b22e5cf777baf71a905f50`
Programs	`contracts/modules/MinterModule.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.8 person-weeks. The assessment was conducted by two consultants over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filipe Alves
Engineer
filipe@zellic.io ↗

Chongyu Lv
Engineer
chongyu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 23, 2026	Start of primary review period
January 26, 2026	Kick-off call
February 12, 2026	End of primary review period

3. Detailed Findings

3.1. BrandedSiloedCustodian passes shares where assets are expected, breaking withdrawals for non-18-decimal collateral

Target	fraxnet-dev/BrandedSiloedCustodian		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `BrandedSiloedCustodian.withdraw` function in the Branded Stablecoin system (Component #2) converts the requested asset amount into frxUSD-denominated units via `previewWithdraw(_assetsOut)` then passes the resulting `frxUsdNeeded` value to `withdrawInheritedExternal`.

```
// fraxnet-dev/src/BrandedStablecoin/BrandedSiloedCustodian.sol#L114
function withdraw(
  uint256 _assetsOut,
  address _receiver,
  address _owner
) public virtual override returns (uint256 _sharesIn) {
  uint frxUsdNeeded = previewWithdraw(_assetsOut);
  // [...]
  _sharesIn = this.withdrawInheritedExternal(frxUsdNeeded, _receiver,
    address(this));
}
```

The `withdrawInheritedExternal` function forwards this value directly to `FrxUSDCustodian.withdraw`, which expects its first parameter to be an asset amount, not an frxUSD amount.

```
// fraxnet-dev/src/BrandedStablecoin/BrandedSiloedCustodian.sol#L161
function withdrawInheritedExternal(
  uint256 _sharesOut,
  address _receiver,
  address _owner
) external returns (uint256 assetIn) {
  // [...]
  assetIn = FrxUSDCustodian.withdraw(_sharesOut, _receiver, _owner);
}
```

The `FrxUSDCustodian.withdraw` function interprets the first parameter as `_assetsOut` and

validates it against `maxWithdraw`, which returns the maximum withdrawable amount in asset-denominated units.

```
// fraxnet-dev/src/deps/FrxUSDCustodian.sol#L431
function withdraw(uint256 _assetsOut, address _receiver, address _owner)
    public virtual returns (uint256 _sharesIn) {
  // [...]
  uint256 _maxAssets = maxWithdraw(_owner);

  // Revert if you are trying to withdraw too many asset tokens
  if (_assetsOut > _maxAssets) {
    revert ERC4626ExceededMaxWithdraw(_owner, _assetsOut, _maxAssets);
  }

  // See how many shares are needed
  _sharesIn = previewWithdraw(_assetsOut);

  // Do the withdrawal
  _withdraw(msg.sender, _receiver, _owner, _assetsOut, _sharesIn);
}
```

This creates a unit mismatch: `frxUsdNeeded` is denominated in frxUSD (18 decimals), but `FrxUSDCustodian.withdraw` treats it as an asset amount. When the collateral token uses fewer decimals than frxUSD, for example USDC with six decimals, the value passed is inflated by a factor of 1e12.

Impact

Withdrawals from BrandedSiloedCustodian revert when the backing collateral has fewer decimals than frxUSD. For a collateral token with six decimals, `previewWithdraw` scales the amount by 1e12, producing a value that exceeds the `maxWithdraw` check in FrxUSDCustodian. This renders the `withdraw` function completely nonfunctional for any siloed deployment backed by a non-18-decimal token such as USDC.

Recommendations

Pass `_assetsOut` directly to `withdrawInheritedExternal` instead of the frxUSD-converted value, consistent with the implementation in `BrandedSiloedCustodianWithOracle.withdraw`.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit f212df58 ↗.

3.2. Cross-chain transfers truncate amounts to zero when branded stablecoin uses fewer than 18 decimals

Target	fraxnet-dev/CrossChainRouter		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The CrossChainRouter contract in the Branded Stablecoin system (Component #2) hardcodes `1e12` as the dust-removal factor for LayerZero cross-chain transfers across five separate functions: `send`, `mintAndSend`, `quote`, `quoteSend`, and the LayerZero-compatible `send` overload. This constant assumes that all branded stablecoins use 18 decimals, since the LayerZero OFT standard uses six shared decimals and the dust factor is derived from the difference between the token's decimals and the shared decimals (18 - 6 = 12).

The following snippet from `send` illustrates the pattern, which is repeated identically in all five functions.

```
// fraxnet-dev/src/BrandedStablecoin/CrossChainRouter.sol#L82
function send(uint32 dstEid, address recipient, uint256 amount)
    public payable {
    bytes32 router = bytes32(uint(uint160(address(this))));
    if (eidToNonDeterministicAddress[dstEid] != bytes32(0)) router
    = eidToNonDeterministicAddress[dstEid];
    amount = (amount / 1e12) * 1e12; // Remove lzDust
    coin.transferFrom(msg.sender, address(this), amount);
    // [...]
}
```

This assumption holds for the GeniusDollar contract, which inherits OpenZeppelin's default 18 decimals, and is correct for all branded stablecoins deployed today (the time of writing). However, the dependency is entirely implicit and not enforced at the code level. The BrandedFactory contract exposes `setDollarImplementation`, which allows the admin to change the underlying implementation used for new branded stablecoin deployments. Additionally, existing token proxies can be upgraded via their respective ProxyAdmin contracts. Neither path enforces any constraint on the new implementation's `decimals` return value.

```
// fraxnet-dev/src/BrandedStablecoin/BrandedFactory.sol#L172
function setDollarImplementation(address newDollarImplementation)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
```

```
        emit DollarImplementationSet(dollarImplementationLocal,
        newDollarImplementation);
        dollarImplementationLocal = newDollarImplementation;
    }
```

Impact

While no branded stablecoins with non-18 decimals exist today, the consequences of this assumption breaking would be severe. If a branded stablecoin with fewer than 18 decimals were deployed or an existing token were upgraded to a lower-decimal implementation, the integer division `amount / 1e12` would truncate nonzero amounts to zero. The subsequent multiplication by 1e12 would yield zero, causing the contract to transfer zero tokens to the destination chain while the caller loses the full cross-chain messaging fee paid via `msg.value`. Although the likelihood of this scenario is low given current operational constraints, the potential for permanent loss of funds warrants enforcing the 18-decimal assumption at the code level rather than relying solely on deployment practices.

Recommendations

Derive the dust-removal factor dynamically from `coin.decimals()` rather than hardcoding 1e12. The factor should be computed as `10 ** (coin.decimals() - sharedDecimals)`, where `sharedDecimals` is six per the LayerZero OFT standard.

Remediation

This issue has been acknowledged by Frax Finance.

3.3. Function `RemoteVaultDeposit.deposit` transfers before truncation, leaving dust in RemoteVaultHop

Target	hop-v2/RemoteVaultDeposit		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `RemoteVaultDeposit.deposit` function in the HopV2 vault system (Component #1) transfers the full `_amount` from the caller to RemoteVaultHop before dust removal occurs. The `removeDust` operation happens later inside `RemoteVaultHop.deposit`, which means the difference between the original amount and the dust-removed amount remains stranded in the RemoteVaultHop contract.

```
// hop-v2/src/contracts/vault/RemoteVaultDeposit.sol#L148
function deposit(uint256 _amount, address _to) public payable {
    RemoteVaultDepositStorage storage $ = _getRemoteVaultDepositStorage();

    SafeERC20.safeTransferFrom(IERC20($.ASSET), msg.sender, address($.REMOTE_
        VAULT_HOP), _amount);
    RemoteVaultHop(payable($.REMOTE_VAULT_HOP)).deposit{ value: msg.value }(
        _amount,
        $.VAULT_CHAIN_ID,
        $.VAULT_ADDRESS,
        _to
    );
    if (address(this).balance > 0) {
        (bool success, ) = msg.sender.call{ value: address(this).balance
    }("");
        require(success, "Refund failed");
    }
}
```

The full `_amount` is transferred to RemoteVaultHop, but `RemoteVaultHop.deposit` then truncates the amount via `removeDust`, discarding the dust portion from the cross-chain transfer while the corresponding tokens have already been moved.

```
// hop-v2/src/contracts/vault/RemoteVaultHop.sol#L131
function deposit(uint256 _amount, uint32 _remoteEid, address _remoteVault,
    address _to) external payable {
```

```
    RemoteVaultHopStorage storage $ = _getRemoteVaultHopStorage();

    if ($.remoteVaultHops[_remoteEid] == address(0)) revert InvalidChain();
    if (address($.depositToken[_remoteEid][_remoteVault]) != msg.sender)
    revert InvalidCaller();

    _amount = removeDust(_amount);
    uint256 fee = quote(_amount, _remoteEid, _remoteVault);
    if (msg.value < fee) revert InsufficientFee();
    // [...]
```

Impact

For example, if a user deposits 1,001 wei, 1 wei of dust will remain stranded in the RemoteVaultHop contract after `removeDust` truncates the amount. While individual dust amounts are negligible, they accumulate over time across many deposits. The `RemoteVaultHop.recover` function can eventually recover the accumulated dust, but this requires manual administrator intervention.

Recommendations

Consider calling `removeDust` before transferring assets from the caller so that only the dust-free amount is bridged and no excess tokens remain in the contract.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit `a11eb7fe`↗.

3.4. Unbounded `minters_array` growth risks denial of service when removing minters

Target	frax-oft-upgradeable/MinterModule		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `MinterModule._addMinter` function in the Frax OFT Minter Array module (Component #7) causes `minters_array` to grow indefinitely, as removed entries are only zeroed out rather than properly deleted. The `_removeMinter` function performs an O(n) linear scan over the array, and its gas consumption increases proportionally with the array length. When the array becomes sufficiently large, this may result in a denial-of-service condition.

```
// frax-oft-upgradeable/contracts/modules/MinterModule.sol#L34
function _addMinter(address minter_address) internal {
   MinterModuleStorage storage $ = _getMinterModuleStorage();
   if ($.minters[minter_address]) revert AlreadyExists();
   $.minters[minter_address] = true;
  $.minters_array.push(minter_address);
   emit MinterAdded(minter_address);
}

function _removeMinter(address minter_address) internal {
    MinterModuleStorage storage $ = _getMinterModuleStorage();
    if (!$.minters[minter_address]) revert AddressNonexistant();
    delete $.minters[minter_address];
    for (uint256 i = 0; i < $.minters_array.length; i++) {
        if ($.minters_array[i] == minter_address) {
           $.minters_array[i] = address(0);
            break;
        }
    }
    emit MinterRemoved(minter_address);
}
// [...]
```

While `_addMinter` is restricted to admin-only access, we recommend establishing explicit boundaries at the code level rather than relying solely on operational constraints. This ensures

correctness regardless of how the contract is administered.

Impact

As the length of `minters_array` increases, `MinterModule._removeMinter` may revert due to being out of gas, leading to a denial of service.

Recommendations

Consider adopting the same approach used for the `freezers` array in the FreezeThawModule contract, using `EnumerableSet::AddressSet` to avoid the denial-of-service risk caused by O(n) array scanning.

Remediation

This issue has been acknowledged by Frax Finance.

3.5. Function `FraxNetDepositV3.recoverERC20` uses unsafe `transfer` instead of `safeTransfer`, silently failing for nonstandard tokens

Target	fraxnet-dev/FraxNetDepositV3		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `FraxNetDepositV3.recoverERC20` function, part of the Branded Stablecoin system (Component #2), uses `transfer` without checking the return value. Some ERC-20 tokens do not revert on failure but instead return `false`, and nonstandard implementations such as USDT return `void` instead of a boolean. In both cases, the transfer silently fails.

```
// fraxnet-dev/src/FraxNetDepositV3.sol#L350
function recoverERC20(address tokenAddress, uint256 tokenAmount,
    address recipient) external {
    if (msg.sender != address(factory)) revert Forbidden();
    IERC20(tokenAddress).transfer(recipient, tokenAmount);
}
```

Impact

The `FraxNetDepositV3.recoverERC20` function is intended to recover ERC-20 tokens accidentally sent to this contract. However, it cannot recover tokens with nonstandard `transfer` implementations such as USDT, as the call silently fails without reverting.

Recommendations

Use the SafeERC20 library ↗ implementation from OpenZeppelin.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit 59750197 ↗.

3.6. Emergency shutoff and borrow-limit checks in FraxlendLeverOdos are permanently inactive

Target	frax-flashloan/FraxlendLeverOdos		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The FraxlendLeverOdos contract, part of the Odos Fraxlend Leverage Router (Component #3), implements two safety mechanisms: an emergency shutoff flag and a per-pair borrow-limit check. Both are evaluated in the `swapExactTokensForTokens` function before executing any swap.

```
// frax-flashloan/src/FraxlendLeverOdos.sol#L20
contract FraxlendLeverOdos is AccessControl, ReentrancyGuard {
    using BytesLib for bytes;
    bool public shutoff;
    bytes32 public constant PATH_MANAGER = keccak256("PATH_MANAGER");
    bytes32 public constant PATH_SENTINEL = keccak256("PATH_SENTINEL");
    address public immutable odosRouter;
    mapping(address => bool) public needsExternalBorrowCheck;
// [...]
// frax-flashloan/src/FraxlendLeverOdos.sol#L40
function swapExactTokensForTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
) external nonReentrant returns (uint256[] memory amounts) {
    if (shutoff) revert ShutOff();
    if (needsExternalBorrowCheck[msg.sender]) {
        // [...]
```

However, the contract provides no setter functions for either `shutoff` or `needsExternalBorrowCheck`. Since `shutoff` defaults to `false` and `needsExternalBorrowCheck` defaults to `false` for all addresses, both safety mechanisms are permanently inactive. The existing FraxlendLever contract includes setter functions for these variables, but they were not carried over to the FraxlendLeverOdos implementation.

Impact

The emergency shutoff mechanism cannot be activated, meaning the contract has no way to pause swap operations in the event of an exploit or market anomaly. Similarly, borrow-limit enforcement is permanently disabled, as no pair can be configured to require external borrow checks. In an emergency scenario, the only recourse would be to deploy a new contract entirely.

Recommendations

Add setter functions with appropriate access control for both state variables, consistent with the existing FraxlendLever implementation.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit 2ecc2a7d ↗.

3.7. Unchecked low-level call in `FraxlendLeverOdos.recoverERC20` silently fails on transfer errors

Target	frax-flashloan/FraxlendLeverOdos		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `recoverERC20` function in FraxlendLeverOdos (Component #3, Odos Fraxlend Leverage Router) uses an unchecked low-level `call` to transfer tokens without verifying the return value. This causes token transfers to fail silently, preventing the admin from recovering tokens even when the function executes successfully.

The problematic implementation directly invokes a low-level call without capturing or checking its return value.

```
// frax-flashloan/src/FraxlendLeverOdos.sol#L151
function recoverERC20(address token, uint256 amount)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
    token.call(abi.encodeWithSignature("transfer(address,uint256)",
    msg.sender, amount));
}
```

The `call` method returns a boolean indicating success, but this return value is discarded. As a result, the function does not revert when the token transfer fails, whether due to insufficient balance, a nonstandard ERC-20 implementation such as USDT that returns `void`, or a token that returns `false` instead of reverting on failure.

Additionally, the `swapExactTokensForTokens` function contains similar issues where standard `transferFrom` and `approve` calls could be replaced with safe alternatives.

```
// frax-flashloan/src/FraxlendLeverOdos.sol#L40
function swapExactTokensForTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
) external nonReentrant returns (uint256[] memory amounts) {
    // [...]
```

```
    IERC20(path[0]).transferFrom(msg.sender, address(this), amountIn);
    IERC20(path[0]).approve(odosRouter, amountIn);
    // [...]
}
```

Impact

When an admin attempts to recover tokens using `recoverERC20`, the transaction will succeed even if the token transfer fails. This creates confusion during emergency recovery scenarios, as the admin receives no indication that the recovery attempt was unsuccessful. The tokens remain stuck in the contract, and the admin may incorrectly assume they were successfully recovered.

Recommendations

Replace the low-level `call` with OpenZeppelin's SafeERC20 library to ensure token transfers are properly validated.

```
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
    SafeERC20.sol";

 contract FraxlendLeverOdos is AccessControl, ReentrancyGuard {
    using SafeERC20 for IERC20;
      // [...]

     function recoverERC20(address token, uint256 amount)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
        token.call(abi.encodeWithSignature("transfer(address,uint256)", msg.
            sender, amount));
        IERC20(token).safeTransfer(msg.sender, amount);
     }
```

Additionally, consider updating the `swapExactTokensForTokens` function to use `safeTransferFrom` and `forceApprove` for improved safety and compatibility with nonstandard tokens.

```
function swapExactTokensForTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
) external nonReentrant returns (uint256[] memory amounts) {
    // [...]
```

```
    IERC20(path[0]).transferFrom(msg.sender, address(this), amountIn);
    IERC20(path[0]).approve(odosRouter, amountIn);
    IERC20(path[0]).safeTransferFrom(msg.sender, address(this), amountIn);
    IERC20(path[0]).forceApprove(odosRouter, amountIn);
      // [...]
}
```

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit 12a375c1 ↗.

3.8. Missing `_disableInitializers` in GeniusDollar constructor

Target	fraxnet-dev/GeniusDollar		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The GeniusDollar contract, deployed as part of the Branded Stablecoin system (Component #2), is an implementation contract that uses OpenZeppelin's Proxy library. According to OpenZeppelin's documentation ↗, implementation contracts should call the `_disableInitializers` function in the constructor to prevent an uninitialized implementation contract from being used:

> Do not leave an implementation contract uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed.

However, the GeniusDollar contract does not take this action.

Impact

While all users may have the potential to assume control of an uninitialized implementation contract, this vulnerability is confined solely to the implementation contract itself.

Recommendations

Invoke `_disableInitializers()` in the constructor of the GeniusDollar contract.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit 39ba159e ↗.

3.9. Deprecated tokens incorrectly claim support for nonfunctional mintable interfaces

Target	frax-tokens/ERC20ExWrappedPPOM		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The deprecated token contracts, updated as part of the native bridging deprecation (Component #5), claim to support the `IOptimismMintableERC20` and `ILegacyMintableERC20` interfaces through their `supportsInterface` implementation, but the required `mint` and `burn` functions always revert. This violates the ERC-165 standard, which requires contracts to return `true` only for interfaces they actually implement.

The `supportsInterface` function in ERC20ExWrappedPPOM declares support for interfaces that are no longer functional.

```
// frax-tokens/src/contracts/fraxtal/shared/ERC20ExWrappedPPOM.sol#L268
function supportsInterface(bytes4 _interfaceId)
    external pure virtual returns (bool) {
    bytes4 iface1 = type(IERC165).interfaceId;
    // Interface corresponding to the legacy L2StandardERC20.
    bytes4 iface2 = type(ILegacyMintableERC20).interfaceId;
    // Interface corresponding to the updated OptimismMintableERC20 (this
    contract).
    bytes4 iface3 = type(IOptimismMintableERC20).interfaceId;
    return _interfaceId == iface1 || _interfaceId == iface2 || _interfaceId ==
    iface3;
}
```

However, the corresponding `mint` and `burn` functions unconditionally revert, making the interface claims incorrect.

```
// frax-tokens/src/contracts/fraxtal/shared/ERC20ExWrappedPPOM.sol#L282
function mint(address, uint256)
    external virtual override(IOptimismMintableERC20, ILegacyMintableERC20) {
    revert Deprecated();
}

function burn(address, uint256)
```

```
    external virtual override(IOptimismMintableERC20, ILegacyMintableERC20) {
    revert Deprecated();
}
```

According to the ERC-165 specification, a contract must return `true` only for interface identifiers it actually implements with functional behavior. Since `mint` and `burn` no longer work correctly, the contract does not legitimately implement these interfaces.

The same issue exists in ERC20PermitPermissionedNonBridgeableMintable.

Impact

External contracts or bridges that rely on ERC-165 interface detection may incorrectly assume these tokens support minting and burning operations. When attempting to call these functions, transactions will revert with a `Deprecated` error, potentially causing integration failures or unexpected behavior in protocols that dynamically detect supported interfaces.

Recommendations

Remove the deprecated interface identifiers from the `supportsInterface` function to accurately reflect the contract's capabilities.

```
function supportsInterface(bytes4 _interfaceId)
    external pure virtual returns (bool) {
    bytes4 iface1 = type(IERC165).interfaceId;
    bytes4 iface2 = type(ILegacyMintableERC20).interfaceId;
    bytes4 iface3 = type(IOptimismMintableERC20).interfaceId;
    return _interfaceId == iface1 || _interfaceId == iface2 || _interfaceId ==
        iface3;
    return _interfaceId == iface1;
}
```

Apply the same fix to ERC20PermitPermissionedNonBridgeableMintable.

Remediation

This issue has been acknowledged by Frax Finance, and a fix was implemented in commit de163f9d ↗.

3.10. Encoding backing token address in CREATE2 salt reduces entropy

Target	fraxnet-dev/BrandedFactory		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The BrandedFactory contract in the Branded Stablecoin system (Component #2) deploys branded stablecoins across multiple EVM chains using CREATE2 with a salt that encodes operational data when isolated backing is enabled. Specifically, when the first four bytes of the salt equal type(uint32).max, the contract interprets the salt as containing a backing token address, leaving only eight bytes (64 bits) of effective entropy.

```
// fraxnet-dev/src/BrandedStablecoin/BrandedFactory.sol#L256
if (bytes4(salt) == bytes4(type(uint32).max)) {
    if (block.chainid == 1) {
        address backing = address(
            uint160((uint256(salt &
    0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000000000000000) << 32)
    >> 96)
        );
        // [...]
    }
    BrandedCustodian(custodian).setUpIsolatedBacking(siloedCustodian, router);
}
```

The salt structure becomes the following when isolated backing is enabled:

- Bytes 0–3: 0xFFFFFFFF (marker for isolated backing)
- Bytes 4–23: Backing token address (20 bytes)
- Bytes 24–31: Available entropy (8 bytes, 64 bits)

Impact

Encoding operational data into CREATE2 salts reduces the cryptographic strength of the salt and weakens defense in depth. While not directly exploitable in the current implementation (as of the time of writing), this pattern unnecessarily couples addressing logic with application logic. For reference, the Profanity vulnerability ↗ demonstrated how reduced entropy (32 bits in that case) made private key recovery feasible through brute-force. The context here differs significantly —

`CREATE2` salt collisions do not compromise private keys, and 64 bits of entropy remain computationally infeasible to brute-force with current hardware — but the parallel illustrates why maximizing entropy is a best practice.

Recommendations

Consider passing the backing token address as a separate parameter rather than encoding it into the salt. This preserves full salt entropy while maintaining cleaner separation between addressing logic and application configuration.

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. No minimum amount validation after `removeDust` in HopV2 deposit flow

The `HopV2.removeDust` function in the HopV2 vault system (Component #1) truncates token amounts before cross-chain transfers, but no subsequent check verifies that the resulting amount meets a caller-specified minimum. Adding such a validation would prevent deposits where dust removal reduces the amount below an acceptable threshold. LayerZero's OFTCore implementation ↗ includes this pattern:

```
// LayerZero-v2/packages/layerzero-v2/evm/oapp/contracts/oft/OFTCore.sol#L360
function _debitView(
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 /*_dstEid*/
) internal view virtual returns (uint256 amountSentLD,
    uint256 amountReceivedLD) {
    // @dev Remove the dust so nothing is lost on the conversion between chains
    with different decimals for the token.
    amountSentLD = _removeDust(_amountLD);
    // @dev The amount to send is the same as amount received in the default
    implementation.
    amountReceivedLD = amountSentLD;

    // @dev Check for slippage.
    if (amountReceivedLD < _minAmountLD) {
        revert SlippageExceeded(amountReceivedLD, _minAmountLD);
    }
}
```

Without this check, a user whose deposit is truncated to zero by dust removal would still pay the cross-chain messaging fee while receiving nothing on the destination chain. We recommend adding a minimum-amount slippage check after `removeDust`, consistent with LayerZero's own cross-chain transfer pattern.

4.2. Redundant `removeDust` call in HopV2 deposit flow

In the HopV2 vault system (Component #1), the first call to `removeDust` occurs in `RemoteVaultHop.deposit`, which truncates `_amount` before passing it to `sendOFT`:

```
// hop-v2/src/contracts/vault/RemoteVaultHop.sol#L131
function deposit(uint256 _amount, uint32 _remoteEid, address _remoteVault,
    address _to) external payable {
    // [...]
   _amount = removeDust(_amount);
    // [...]
   HOP_.sendOFT{ value: fee }(
        $.OFT, _remoteEid,
        bytes32(uint256(uint160($.remoteVaultHops[_remoteEid]))),
        _amount, _remoteGas, hopComposeMessage
    );
    // [...]
```

However, `HopV2.sendOFT` calls `removeDust` again on the already truncated value, making the second call redundant:

```
// hop-v2/src/contracts/hop/HopV2.sol#L122
function sendOFT(address _oft, uint32 _dstEid, bytes32 _recipient,
    uint256 _amountLD, uint128 _dstGas, bytes memory _data)
    public payable virtual {
    // [...]
   _amountLD = removeDust(_oft, _amountLD);
    // [...]
```

The second call is a no-op that wastes gas without affecting correctness, and there is no security impact. The `removeDust` in `sendOFT` is intentional for callers other than `RemoteVaultHop`, but it is redundant in this specific flow. Consider removing the `removeDust` call in `RemoteVaultHop.deposit` since `sendOFT` already handles it.

4.3. Refund logic in `RemoteVaultDeposit.deposit` refunds entire contract balance instead of only excess fee

The `RemoteVaultDeposit.deposit` function in the HopV2 vault system (Component #1) refunds the entire ETH balance of the contract to the caller, rather than only the excess fee from the current transaction.

```
// hop-v2/src/contracts/vault/RemoteVaultDeposit.sol#L148
function deposit(uint256 _amount, address _to) public payable {
    // [...]
    RemoteVaultHop(payable($.REMOTE_VAULT_HOP)).deposit{ value: msg.value }(
        _amount, $.VAULT_CHAIN_ID, $.VAULT_ADDRESS, _to
    );
    if (address(this).balance > 0) {
        (bool success, ) = msg.sender.call{ value: address(this).balance
    }("");
        require(success, "Refund failed");
    }
}
```

The excess-fee refund already occurs correctly in `RemoteVaultHop.deposit`, which returns any overpayment to `msg.sender` (the RemoteVaultDeposit contract). RemoteVaultDeposit then forwards its entire balance to the caller, using `address(this).balance` as a proxy for the refunded amount.

The semantics here are inaccurate, the intent is to refund the excess cross-chain fee, but the implementation refunds whatever ETH the contract holds. Under normal operation, these values coincide, but if ETH were accidentally sent to RemoteVaultDeposit outside of a deposit call, the next depositor would receive it as part of their refund. While there is no direct security impact, the code does not accurately express its intent. An alternative would be to have `RemoteVaultHop.deposit` return the excess-fee amount explicitly, so RemoteVaultDeposit can refund exactly that value.

4.4. Cross-chain deployment marked as complete before destination chain confirms execution

The `FraxtalBrandedDeployer.deployCrossChainBrandedStablecoin` function, part of the Branded Stablecoin deployment pipeline (Component #2), marks deployments as completed immediately after sending the cross-chain message, before the target chain has executed the deployment logic.

```
// fraxnet-dev/src/BrandedStablecoin/FraxtalBrandedDeployer.sol#L212
function deployCrossChainBrandedStablecoin(
    string memory name, string memory symbol, bytes32 salt
) external payable returns (address coin, address custodian, address router) {
    // [...]
    for (uint i; i < len; ++i) {
        // [...]
        } else {
            // deploy remote
```

```
            SendParam memory sendParam = _generateSendParam(eid,
    eidToLocalFactory[eid], name, symbol, salt);
            MessagingFee memory fee = IOFT(localOFT).quoteSend(sendParam,
    false);
            IOFT(localOFT).send{ value: fee.nativeFee }(sendParam, fee,
    address(this));
            totalFee += fee.nativeFee;
            config.isDeployed[eid] = true;
        }
        // [...]
    }
}
```

The function sets `config.isDeployed[eid] = true` immediately after calling `IOFT(localOFT).send`, which only confirms that the message was submitted to the cross-chain messaging system, not that the target chain has successfully executed the deployment. If the target-chain execution fails due to insufficient gas, executor failure, or factory reversion, the deployment status would be incorrectly marked as complete and subsequent retry attempts via `deploySingle` would revert with `TokenAlreadyDeployed()`, requiring manual administrator intervention through `manuallySetDeploymentForEid`.

The protocol has indicated this is intentional. The primary failure mode outside of insufficient gas would be a `CREATE2` collision, already covered by the local deployment in the same loop. Implementing a confirmation mechanism would require additional cross-chain message exchanges and significant complexity. The current approach is a reasonable trade-off given that deployment failures on target chains are rare and recoverable through administrator intervention.

4.5. Custom initialization pattern used instead of OpenZeppelin Initializable

Multiple contracts in the Branded Stablecoin system (Component #2) use a custom initialization pattern with a `wasInitialized` boolean flag rather than OpenZeppelin's Initializable contract.

```
// fraxnet-dev/src/BrandedStablecoin/BrandedCustodian.sol#L94
constructor() Ownable(msg.sender) {
    wasInitialized = true;
}

function initialize(address _brandedCoin, address _localFrxUSD,
    address _owner, uint256 _mintCap, uint256 _mintFee, uint256 _redeemFee)
    public {
    if (wasInitialized) revert InitializeFailed();
    // [...]
```

```
}
```

This approach maintains consistency with the iterative storage slot layout used throughout the codebase, avoiding the complexity of mixing OpenZeppelin's namespaced storage with sequential slots. The trade-off is that the custom pattern does not provide the additional protections built into OpenZeppelin's Initializable, such as versioned reinitialization and implementation locking. The protocol has confirmed this is an intentional design choice.

The GeniusDollar proxy is an exception — it uses OpenZeppelin's namespaced storage to keep lower storage slots clear, allowing issuers who upgrade their stablecoin to adopt an iterative layout without conflicts.

4.6. Remediations

Frax Finance has acknowledged all the discussion points in this section (section 4. ↗).

Additionally, point 4.3. ↗ was addressed in commit `0ebcb98` ↗ by adding return values to track actual fees charged in `RemoteVaultHop.deposit` and `RemoteVaultHop.redeem`, allowing `RemoteVaultDeposit` to refund exactly `msg.value - fee` rather than the entire contract balance.

5. Threat Model

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

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

5.1. Module: Component #2: Branded Stablecoin / BrandedCustodian

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

This function deposits the underlying `custodianTkn` (local frxUSD) into the BrandedCustodian and mints the branded stablecoin to `_receiver`, enforcing fees, mint caps, and isolated-backing caller restrictions when enabled.

Inputs

- `_assetsIn`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxDeposit(_receiver)`, the caller must have sufficient `custodianTkn` balance and allowance for `SafeERC20.safeTransferFrom`, and if `isolatedBackingEnabled` is true, then `msg.sender` must be `router` or `individualCustodian`.
  - **Impact**: The amount of frxUSD transferred into the custodian and used to compute `_sharesOut` (after applying `mintFee` via `previewDeposit` and decimals conversion).
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)`).
  - **Impact**: The address that receives the newly minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `deposit` should call `SafeERC20.safeTransferFrom` to transfer `_assetsIn` `custodianTkn` from `msg.sender` to `address(this)`, then call `brandedStablecoin.mint` to mint `sharesOut` shares to `_receiver`, emitting `Deposit`.
- ☑ The `deposit` function should increase `brandedCoinMinted` by `sharesOut` and increase `custodianTkn.balanceOf(address(this))` by `_assetsIn`.

- ☑ When `isolatedBackingEnabled` is enabled, `deposit` should succeed when called by `router`.
- ☑ When `isolatedBackingEnabled` is enabled and `block.chainid == 1`, `deposit` should succeed when called by `individualCustodian`.

Negative behavior

- ☑ Revert with `ERC4626ExceededMaxDeposit` when `_assetsIn > maxDeposit(_receiver)`.
- ☑ Revert with `NotIsolatedBacking` when `isolatedBackingEnabled` is enabled and `msg.sender` is not `router` or `individualCustodian`.
- ☐ Revert when `_receiver == address(0)`.

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

This function mints an exact amount of branded stablecoin shares by pulling the required `custodianTkn` (local frxUSD) from the caller, including the `mintFee`, and minting `_sharesOut` shares to `_receiver`. It is subject to the `mintCap` and isolated-backing restrictions.

Inputs

- `_sharesOut`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxMint(_receiver)`, the caller must have sufficient `custodianTkn` balance and allowance for the computed `previewMint(_sharesOut)` amount, and if `isolatedBackingEnabled` is true, then `msg.sender` must be `router` or `individualCustodian`.
  - **Impact**: The target number of branded stablecoin shares to mint — drives the required asset input via `previewMint` (ceil rounding) and updates `brandedCoinMinted`.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)`).
  - **Impact**: The address that receives the newly minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `mint` should compute `assetsIn = previewMint(_sharesOut)`, `SafeERC20.safeTransferFrom` `custodianTkn`, and `brandedStablecoin.mint` `_sharesOut` shares to `_receiver`, emitting `Deposit`.

- ☑ The `mint` function should return `assetsIn` equal to `previewMint(_sharesOut)` and increase `brandedCoinMinted` by `_sharesOut`.
- ☐ When `isolatedBackingEnabled` is enabled, `mint` should succeed when called by `router` or `individualCustodian`.

Negative behavior

- ☑ Revert with `ERC4626ExceededMaxMint` when `_sharesOut > maxMint(_receiver)`.
- ☑ Revert with `NotIsolatedBacking` when `isolatedBackingEnabled` is enabled and `msg.sender` is not `router` or `individualCustodian`.
- ☐ Revert when `_receiver == address(0)`.

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

This function redeems branded stablecoin shares for underlying `custodianTkn` (local frxUSD) and transfers the resulting assets to `_receiver`. The `_owner` parameter is retained for ERC-4626 ABI compatibility but must equal `msg.sender`.

Inputs

- `_sharesIn`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxRedeem(_owner)` (bounded by both the owner's share balance and the contract's available `custodianTkn` liquidity), and if `isolatedBackingEnabled` is true, then `msg.sender` must be `router` or `individualCustodian` due to the `checkIsolatedBacking` gate in the internal `_withdraw` path.
  - **Impact**: The number of branded stablecoin shares to burn from the caller — determines the `custodianTkn` assets returned (after applying `redeemFee` via `previewRedeem`).
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 transfers revert on `address(0)` in standard implementations).
  - **Impact**: The address that receives the redeemed `custodianTkn` assets.
- `_owner`
  - **Control**: Full control.
  - **Constraints**: Must equal `msg.sender` (`TokenOwnerShouldBeSender`).
  - **Impact**: The owner of the shares being redeemed (kept for ERC-4626 ABI compatibility but restricted to `msg.sender`).

Branches and code coverage

Intended branches

- ☑ Calling `redeem` should compute `assetsOut = previewRedeem(_sharesIn)`, call `brandedStablecoin.burnFrom`, then call `SafeERC20.safeTransfer` to transfer `assetsOut` `custodianTkn` to `_receiver`, emitting `Withdraw`.
- ☑ The `redeem` function should return `assetsOut` equal to `previewRedeem(_sharesIn)` and decrease `brandedCoinMinted` by `_sharesIn`.
- ☐ When `isolatedBackingEnabled` is enabled, `redeem` should succeed when called by `router` or `individualCustodian` with `_owner == msg.sender`.

Negative behavior

- ☑ Revert with `TokenOwnerShouldBeSender` when `_owner != msg.sender`.
- ☑ Revert with `ERC4626ExceededMaxRedeem` when `_sharesIn > maxRedeem(_owner)`.
- ☑ Revert with `ERC4626ExceededMaxRedeem` when the contract has insufficient `custodianTkn` liquidity (e.g., after `recoverERC20` drains assets).
- ☐ Revert with `NotIsolatedBacking` when `isolatedBackingEnabled` is enabled and `msg.sender` is not `router` or `individualCustodian`.

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

This function withdraws an exact amount of underlying `custodianTkn` (local frxUSD) to `_receiver` by burning the required branded stablecoin shares from `msg.sender`, including fee-adjusted share calculation and liquidity checks. The `_owner` parameter is retained for ERC-4626 ABI compatibility but must equal `msg.sender`.

Inputs

- `_assetsOut`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxWithdraw(_owner)` (bounded by both the owner's shares and the contract's `custodianTkn` balance), and if `redeemFee` is nonzero, the required shares are computed via `previewWithdraw` with ceil rounding.
  - **Impact**: The amount of frxUSD the caller wants to receive — determines how many shares will be burned and how much `custodianTkn` is transferred out.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 transfers revert on `address(0)` in standard implementations).

- **Impact**: The address that receives the withdrawn `custodianTkn` assets.

- `_owner`
  - **Control**: Full control.
  - **Constraints**: Must equal `msg.sender` (`TokenOwnerShouldBeSender`), and if `isolatedBackingEnabled` is true, then `msg.sender` must also be `router` or `individualCustodian` due to the `checkIsolatedBacking` gate in the internal `_withdraw` path.
  - **Impact**: The owner of the shares being burned (kept for ERC-4626 ABI compatibility but restricted to `msg.sender`).

Branches and code coverage

Intended branches

- ☑ Calling `withdraw` should compute `sharesIn = previewWithdraw(_assetsOut)`, call `brandedStablecoin.burnFrom`, then call `SafeERC20.safeTransfer` to transfer `_assetsOut` `custodianTkn` to `_receiver`, emitting `Withdraw`.
- ☑ The `withdraw` function should return `sharesIn` equal to `previewWithdraw(_assetsOut)` and decrease `brandedCoinMinted` accordingly.
- ☐ When `isolatedBackingEnabled` is enabled, `withdraw` should succeed when called by `router` or `individualCustodian` with `_owner == msg.sender`.

Negative behavior

- ☑ Revert with `TokenOwnerShouldBeSender` when `_owner != msg.sender`.
- ☑ Revert with `ERC4626ExceededMaxWithdraw` when `_assetsOut > maxWithdraw(_owner)`.
- ☑ Revert with `ERC4626ExceededMaxWithdraw` when the contract has insufficient `custodianTkn` liquidity (e.g., after `recoverERC20` drains assets).
- ☐ Revert with `NotIsolatedBacking` when `isolatedBackingEnabled` is enabled and `msg.sender` is not `router` or `individualCustodian`.

5.2. Module: Component #2: Branded Stablecoin / BrandedFactory

Function: `_deployBrandedTokenChained(string name, string symbol, bytes32 salt)`

This internal function deterministically deploys the branded stablecoin system using `CREATE2`. It deploys a branded stablecoin proxy (`coin`), a BrandedCustodian beacon proxy (`custodian`), and a CrossChainRouter beacon proxy (`router`); initializes them; and grants minting rights to the custodian. When `bytes4(salt) == 0xFFFFFFFF`, it also configures isolated backing, including deploying/registering a siloed custodian on mainnet.

Inputs

- `name`
  - **Control**: Full control.
  - **Constraints**: Must be ABI decodable from the upstream call path and compatible with the branded stablecoin initializer (`initialize(string,string,address,address)`).
  - **Impact**: The ERC-20 token name passed to the branded stablecoin initialization call.
- `symbol`
  - **Control**: Full control.
  - **Constraints**: Must be ABI decodable from the upstream call path and compatible with the branded stablecoin initializer (`initialize(string,string,address,address)`).
  - **Impact**: The ERC-20 token symbol passed to the branded stablecoin initialization call.
- `salt`
  - **Control**: Full control.
  - **Constraints**: Used as the `CREATE2` salt for the deployed proxies — reusing a salt for the same init code will cause a deployment failure (`Create2Failed`). If `bytes4(salt) == 0xFFFFFFFF`, the salt also signals isolated-backing mode, and on mainnet, the encoded backing address must map to an approved siloed custodian implementation in `custodianRegistry`.
  - **Impact**: Determines the deterministic addresses for `coin`, `custodian`, and `router`, and toggles/configures isolated backing behavior (including backing asset selection and siloed custodian deployment).

Branches and code coverage

Intended branches

- ☑ Deployment should create code at the deterministic `computeProxyAddress(name, symbol, salt)` address via `CREATE2`.
- ☑ Deployment should initialize the token via `IProxyAdmin(proxyAdmin).upgradeAndCall` and grant `MINTER_ROLE` to the custodian via `IGDollar(coin).grantRole`.
- ☑ When `bytes4(salt) == 0xFFFFFFFF`, deployment should configure isolated backing via `BrandedCustodian.setUpIsolatedBacking` and initialize the router with `isolatedBackingEnabled = true`.

Negative behavior

- ☑ Revert with `Create2Failed` when attempting to deploy twice with the same `salt`.
- ☐ Revert with `InvalidIsolatedBacking` when isolated backing is requested but the encoded backing asset is not registered in `custodianRegistry`.

- ☐ Revert when the downstream initialization calls for `custodian` or `router` fail.

Function: `lzCompose(address _oft, bytes32 _guid, bytes _message, address _executor, bytes _executorData)`

This function is the LayerZero compose entry point for deploying a branded stablecoin system on the destination chain. It authenticates that the call originates from the configured LZ endpoint and Fraxtal factory, deduplicates messages, decodes `(name, symbol, salt)` from the compose payload, and calls `_deployBrandedTokenChained(name, symbol, salt)`.

Inputs

- `_oft`
  - **Control**: Full control.
  - **Constraints**: Must equal `frxUSDLocal` and `msg.sender` must be `endpoint`; otherwise, the call reverts (`InvalidOFT` / `NotEndpoint`).
  - **Impact**: Identifies the OFT/OApp that delivered the compose message — used in message authentication and `messageHash` derivation.
- `_guid`
  - **Control**: Full control.
  - **Constraints**: Not used by this implementation (present for ABI compatibility with the LZ compose interface).
  - **Impact**: Placeholder compose GUID — has no effect on execution or state.
- `_message`
  - **Control**: Full control.
  - **Constraints**: Must decode via `OFTComposeMsgCodec` such that `srcEid == 30255`, `composeFrom == fraxtalFactory`, and the decoded compose payload matches `(string name, string symbol, bytes32 salt)`. The derived `messageHash` must not have been processed before; otherwise, the function returns early.
  - **Impact**: Encoded LZ compose message that carries the deployment parameters and drives the downstream `_deployBrandedTokenChained` call.
- `_executor`
  - **Control**: Full control.
  - **Constraints**: Not used by this implementation (present for ABI compatibility with the LZ compose interface).
  - **Impact**: Placeholder LZ executor address — has no effect on execution or state.
- `_executorData`
  - **Control**: Full control.

- **Constraints**: Not used by this implementation (present for ABI compatibility with the LZ compose interface).
- **Impact**: Placeholder executor metadata — has no effect on execution or state.

Branches and code coverage

Intended branches

- ☑ A valid compose message should pass authentication checks and call `_deployBrandedTokenChained(name, symbol, salt)`, deploying a token at `computeProxyAddress(name, symbol, salt)`.
- ☑ When `messageProcessed[messageHash]` is already true, `lzCompose` should return early without calling `_deployBrandedTokenChained`.
- ☑ Pre-marking a message as processed via `setMessageProcessed` should cause `lzCompose` to skip deployment for that message hash.

Negative behavior

- ☑ Revert with `NotEndpoint` when `msg.sender != endpoint`.
- ☑ Revert with `InvalidOFT` when `_oft != frxUSDLocal`.
- ☑ Revert with `InvalidSrcEid` when the decoded `srcEid` is not the expected Fraxtal EID.
- ☑ Revert with `NotFromFactory` when the decoded `composeFrom` does not match `fraxtalFactory`.
- ☐ Revert when the compose payload cannot be decoded into `(string name, string symbol, bytes32 salt)`.

5.3. Module: Component #2: Branded Stablecoin / BrandedSiloedCustodian

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

This function deposits the isolated backing asset (`custodianTkn`) into the inherited `FrxUSDCustodian` to mint frxUSD then deposits that frxUSD into the `localCustodian` to mint branded stablecoin shares to `_receiver`.

Inputs

- `_assetsIn`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient `custodianTkn` balance and allowance for `SafeERC20.safeTransferFrom` (in the inherited deposit flow), and the amount must be within the inherited `maxDeposit` limit (mint cap). The downstream `localCustodian.deposit` must also accept calls from this contract (typically as `individualCustodian` in isolated mode).

- **Impact**: The amount of backing assets to convert into frxUSD (shares) and then into branded stablecoin shares (after applying any mint fees and conversion logic in both custodians).

- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)` in standard implementations).
  - **Impact**: The address that receives the minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `deposit` should mint frxUSD via `FrxUSDCustodian.deposit(_assetsIn, address(this))` and then mint branded stablecoins via `localCustodian.deposit(frxUsdShares, _receiver)`.
- ☑ A successful deposit should result in the receiver holding newly minted branded stablecoin shares.
- ☑ And `deposit(0, receiver)` should not change state if it does not revert (checked via a low-level call).

Negative behavior

- ☑ Revert when the caller has not approved enough backing `custodianTkn` allowance for the inherited deposit flow.
- ☑ Revert when `_receiver == address(0)`.

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

This function mints an exact amount of branded stablecoin shares by first minting the same amount of frxUSD shares via the inherited `FrxUSDCustodian.mint` and then calling `localCustodian.mint` to mint branded stablecoins to `_receiver`.

Inputs

- `_sharesOut`
  - **Control**: Full control.
  - **Constraints**: Must be within the inherited `maxMint` limit (mint cap), and the caller must have sufficient backing `custodianTkn` balance and allowance for the computed asset input returned by the inherited mint flow. The downstream `localCustodian.mint` must also accept calls from this contract (typically as `individualCustodian` in isolated mode).

- **Impact**: The number of frxUSD shares (and branded stablecoin shares) to mint to the receiver, determining the required backing asset amount via `previewMint`.

- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)` in standard implementations).
  - **Impact**: The address that receives the minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `mint` should mint frxUSD via `FrxUSDCustodian.mint(_sharesOut, address(this))` and then mint branded stablecoins via `localCustodian.mint(_sharesOut, _receiver)`.
- ☑ A successful mint should result in the receiver holding `_sharesOut` branded stablecoin shares and return the backing asset amount consumed.
- ☐ Minting to a receiver different from `msg.sender` should mint branded stablecoins to `_receiver`.

Negative behavior

- ☑ Revert when the caller has not approved enough backing `custodianTkn` allowance for the inherited mint flow.
- ☐ Revert when `_receiver == address(0)`.

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

This function redeems branded stablecoin shares from `msg.sender` into the backing asset by redeeming the branded stablecoin into frxUSD via `localCustodian.redeem` then redeeming frxUSD into backing assets via the inherited `FrxUSDCustodian.redeem` flow to `_receiver`. The `_owner` parameter is kept for ERC-4626 ABI compatibility but is unused.

Inputs

- `_sharesIn`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient branded stablecoin balance and allowance to this contract for `transferFrom`, and the downstream `localCustodian.redeem` and inherited redemption flow must succeed.
  - **Impact**: The number of branded stablecoin shares to burn/redeem — determines how much frxUSD is obtained and then how many backing assets

are returned to `_receiver`.

- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must be able to receive the backing asset token transfer from the inherited custodian.
  - **Impact**: The address that receives the redeemed backing assets.
- `_owner`
  - **Control**: Full control.
  - **Constraints**: This parameter is unused by the function body and does not affect execution.
  - **Impact**: Kept for ERC-4626 ABI compatibility — has no functional effect.

Branches and code coverage

Intended branches

- ☑ Calling `redeem` should pull branded stablecoins via `transferFrom`, redeem frxUSD via `localCustodian.redeem`, and redeem backing assets via `redeemInheritedExternal`, transferring assets to `_receiver`.
- ☑ The `redeem` function should transfer backing assets to a receiver different from `msg.sender` when `_receiver` is specified.
- ☑ After a full redemption, the siloed custodian should not retain dust balances of branded tokens, frxUSD, or backing assets.
- ☑ The `_owner` parameter should not affect execution (a bogus `_owner` should still succeed).
- ☑ And `redeem(0, receiver, owner)` should not change state if it does not revert (checked via a low-level call).

Negative behavior

- ☑ Revert when the caller has not approved enough branded stablecoin allowance for `transferFrom`.
- ☑ Revert when the requested redeem amount exceeds the caller's branded stablecoin balance.
- ☐ Revert when `_receiver == address(0)`.

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

This function withdraws backing assets by pulling branded stablecoins from `msg.sender`, redeeming them into frxUSD via `localCustodian.redeem`, and withdrawing backing assets via the inherited `FrxUSDCustodian.withdraw` flow to `_receiver`. The `_owner` parameter is kept for ERC-4626 ABI compatibility but is unused.

Inputs

- `_assetsOut`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient branded stablecoin balance and allowance to this contract for `transferFrom`, and the downstream `localCustodian.redeem` and inherited withdrawal flow must succeed. The internal flow computes an frxUSD amount via `previewWithdraw(_assetsOut)`, so unit consistency between `_assetsOut` and the inherited withdrawal call is required for successful execution.
  - **Impact**: The nominal amount of backing assets the caller wants to withdraw — used to compute the frxUSD shares needed and drives the size of redemption/withdrawal operations.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must be able to receive the backing asset token transfer from the inherited custodian.
  - **Impact**: The address that receives the withdrawn backing assets.
- `_owner`
  - **Control**: Full control.
  - **Constraints**: This parameter is unused by the function body and does not affect execution.
  - **Impact**: Kept for ERC-4626 ABI compatibility — has no functional effect.

Branches and code coverage

Intended branches

- ☑ Calling `withdraw` should pull branded stablecoins via `transferFrom`, redeem frxUSD via `localCustodian.redeem`, and withdraw backing assets via `withdrawInheritedExternal`, transferring assets to `_receiver`.
- ☑ The `withdraw` function should transfer backing assets to a receiver different from `msg.sender` when `_receiver` is specified.
- ☑ After a full withdrawal, the siloed custodian should not retain dust balances of branded tokens, frxUSD, or backing assets.
- ☑ The `_owner` parameter should not affect execution (a bogus `_owner` should still succeed).
- ☑ And `withdraw(0, receiver, owner)` should not change state if it does not revert (checked via a low-level call).

Negative behavior

- ☑ Revert when the caller has not approved enough branded stablecoin allowance for `transferFrom`.

- ☑ Revert when the requested withdraw amount exceeds the caller's branded stablecoin balance.
- ☑ Revert when `_receiver == address(0)`.

5.4. Module: Component #2: Branded Stablecoin / BrandedSiloedCustodianWithOracle

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

This function deposits the isolated backing asset (`custodianTkn`) into the inherited `FrxUSDCustodianWithOracle` to mint frxUSD using oracle-based conversions then deposits that frxUSD into the `localCustodian` to mint branded stablecoin shares to `_receiver`.

Inputs

- `_assetsIn`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient `custodianTkn` balance and allowance for `SafeERC20.safeTransferFrom`, and the amount must be within the inherited `maxDeposit` limit (mint cap). Oracle reads must be valid (non-stale, positive price); otherwise, the call can revert with `OracleError`. The downstream `localCustodian.deposit` must accept calls from this contract (typically as `individualCustodian` in isolated mode).
  - **Impact**: The amount of backing assets to convert into frxUSD and then into branded stablecoin shares, with the frxUSD conversion rate influenced by the configured oracle price.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)` in standard implementations).
  - **Impact**: The address that receives the minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `deposit` should run the inherited `updateOracle` modifier and then execute `FrxUSDCustodianWithOracle.deposit` followed by `localCustodian.deposit`.
- ☑ A successful deposit should mint branded stablecoin shares to `_receiver` using oracle-priced conversions.
- ☐ And `deposit(0, receiver)` should not change state if it does not revert.

Negative behavior

- ☑ Revert when the custodian oracle call fails (or returns invalid/stale data), causing the inherited conversion path to revert.
- ☐ Revert when the caller has not approved enough backing `custodianTkn` allowance for the inherited deposit flow.
- ☐ Revert when `_receiver == address(0)`.

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

This function mints an exact amount of branded stablecoin shares by first minting the same amount of frxUSD shares via the inherited `FrxUSDCustodianWithOracle.mint` using oracle-based conversions then calling `localCustodian.mint` to mint branded stablecoins to `_receiver`.

Inputs

- `_sharesOut`
  - **Control**: Full control.
  - **Constraints**: Must be within the inherited `maxMint` limit (mint cap), and the caller must have sufficient backing `custodianTkn` balance and allowance for the computed asset input returned by the inherited mint flow. Oracle reads must be valid (non-stale, positive price); otherwise, the call can revert with `OracleError`. The downstream `localCustodian.mint` must also accept calls from this contract.
  - **Impact**: The number of frxUSD shares (and branded stablecoin shares) to mint to the receiver, determining the required backing asset amount via `previewMint` under the current oracle price.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address (ERC-20 minting reverts on `address(0)` in standard implementations).
  - **Impact**: The address that receives the minted branded stablecoin shares.

Branches and code coverage

Intended branches

- ☑ Calling `mint` should run the inherited `updateOracle` modifier and then execute `FrxUSDCustodianWithOracle.mint(_sharesOut, address(this))` followed by `localCustodian.mint(_sharesOut, _receiver)`.
- ☑ A successful mint should mint branded stablecoin shares to `_receiver` and return a backing asset amount consistent with the oracle price.

- ☐ Minting to a receiver different from `msg.sender` should mint branded stablecoins to `_receiver`.

Negative behavior

- ☐ Revert when the custodian oracle returns invalid/stale data, causing the inherited conversion path to revert.
- ☐ Revert when the caller has not approved enough backing `custodianTkn` allowance for the inherited mint flow.
- ☐ Revert when `_receiver == address(0)`.

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

This function redeems branded stablecoin shares from `msg.sender` into the backing asset by redeeming the branded stablecoin into frxUSD via `localCustodian.redeem` then redeeming frxUSD into backing assets via the inherited `FrxUSDCustodianWithOracle.redeem` flow to `_receiver`. The `_owner` parameter is kept for ERC-4626 ABI compatibility but is unused.

Inputs

- `_sharesIn`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient branded stablecoin balance and allowance to this contract for `transferFrom`, and the downstream `localCustodian.redeem` and inherited redemption flow must succeed. Oracle reads must be valid (non-stale, positive price); otherwise, the inherited conversion functions can revert with `OracleError`.
  - **Impact**: The number of branded stablecoin shares to redeem — determines how much frxUSD is obtained and then how many backing assets are returned to `_receiver` under the current oracle price.
- `_receiver`
  - **Control**: Full control.
  - **Constraints**: Must be able to receive the backing asset token transfer from the inherited custodian.
  - **Impact**: The address that receives the redeemed backing assets.
- `_owner`
  - **Control**: Full control.
  - **Constraints**: This parameter is unused by the function body and does not affect execution.
  - **Impact**: Kept for ERC-4626 ABI compatibility — has no functional effect.

Branches and code coverage

Intended branches

- ☑ Calling `redeem` should pull branded stablecoins via `transferFrom`, call `localCustodian.redeem`, and then call `FrxUSDCustodianWithOracle.redeem` to transfer backing assets to `_receiver`.
- ☑ A successful redemption should return backing assets consistent with the oracle price (validated via fuzz tests).
- ☐ Redeeming to a receiver different from `msg.sender` should transfer backing assets to `_receiver` when the receiver is compliant/allowlisted.
- ☐ The `_owner` parameter should not affect execution (a bogus `_owner` should still succeed).

Negative behavior

- ☐ Revert when the caller has not approved enough branded stablecoin allowance for `transferFrom`.
- ☐ Revert when the backing token enforces recipient restrictions and `_receiver` is not compliant/allowlisted.
- ☐ Revert when the custodian oracle returns invalid/stale data, causing the inherited conversion path to revert.
- ☐ Revert when `_receiver == address(0)`.

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

This function withdraws backing assets by pulling branded stablecoins from `msg.sender`, withdrawing the corresponding frxUSD from the `localCustodian`, and withdrawing backing assets via the inherited `FrxUSDCustodianWithOracle.withdraw` flow to `_receiver`. The `_owner` parameter is kept for ERC-4626 ABI compatibility but is unused.

Inputs

- `_assetsOut`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient branded stablecoin balance and allowance to this contract for `transferFrom`, and the downstream `localCustodian.withdraw` and inherited withdrawal flow must succeed. Oracle reads must be valid (non-stale, positive price); otherwise, the inherited conversion functions can revert with `OracleError`.
  - **Impact**: The amount of backing assets the caller wants to withdraw — drives the branded stablecoin burn amount (computed via `localCustodian.previewWithdraw`) and the inherited withdrawal size.
- `_receiver`

- **Control**: Full control.
- **Constraints**: Must be able to receive the backing asset token transfer from the inherited custodian.
- **Impact**: The address that receives the withdrawn backing assets.

- `_owner`
  - **Control**: Full control.
  - **Constraints**: This parameter is unused by the function body and does not affect execution.
  - **Impact**: Kept for ERC-4626 ABI compatibility — has no functional effect.

Branches and code coverage

Intended branches

- ☑ Calling `withdraw` should pull branded stablecoins via `transferFrom`, call `localCustodian.withdraw`, and then call `FrxUSDCustodianWithOracle.withdraw` to transfer backing assets to `_receiver`.
- ☑ A successful withdrawal should transfer `_assetsOut` backing assets to `_receiver` and burn an oracle-priced amount of branded stablecoin shares (validated via fuzz tests).
- ☑ With a compliant/allowlisted receiver, withdrawing to a different `_receiver` should succeed.
- ☐ The `_owner` parameter should not affect execution (a bogus `_owner` should still succeed).

Negative behavior

- ☑ Revert when withdrawing to a noncompliant/nonallowlisted `_receiver` and the backing token enforces recipient restrictions.
- ☐ Revert when the caller has not approved enough branded stablecoin allowance for `transferFrom`.
- ☐ Revert when `_receiver == address(0)`.
- ☐ Revert when the custodian oracle returns invalid/stale data, causing the inherited conversion path to revert.

5.5. Module: Component #2: Branded Stablecoin / CrossChainRouter

Function: `mintAndSend(uint32 dstEid, address recipient, uint256 amount)`

This function bridges frxUSD cross-chain to mint branded stablecoins to `recipient` on the destination chain. It pulls local frxUSD from the caller, sends it via `remoteHopV2.sendOFT` with a compose payload, and relies on the destination router's `hopCompose` to deposit into the custodian and mint the branded stablecoin.

Inputs

- `dstEid`
  - **Control**: Full control.
  - **Constraints**: Must correspond to a destination chain supported by `remoteHopV2` and may have a nondefault router address configured via `eidToNonDeterministicAddress[dstEid]`.
  - **Impact**: The LayerZero endpoint ID of the destination chain for the frxUSD OFT transfer.
- `recipient`
  - **Control**: Full control.
  - **Constraints**: Must be a valid EVM address expected by the destination compose handler (encoded as `abi.encode(recipient)`).
  - **Impact**: The ultimate recipient address on the destination chain that will receive minted branded stablecoins during `hopCompose`.
- `amount`
  - **Control**: Full control.
  - **Constraints**: Reverts if `isolatedBackingEnabled` is true. The amount is rounded down to a multiple of 1e12 before transfers, and the caller must have sufficient frxUSD balance and allowance for `frxusd.transferFrom`. The downstream `remoteHopV2.sendOFT` call must succeed.
  - **Impact**: The frxUSD amount to bridge (prerounding) — determines how much frxUSD is sent cross-chain and consequently how many branded stablecoin shares can be minted on the destination.

Branches and code coverage

Intended branches

- ☑ The `mintAndSend` function should call `frxusd.transferFrom` to transfer frxUSD from the caller to `address(this)`, approve `remoteHopV2`, and call `remoteHopV2.sendOFT` with `composeMsg = abi.encode(recipient)`.
- ☑ Dust removal should round `amount` down to a multiple of 1e12 before transferring and before calling `remoteHopV2.sendOFT`.
- ☐ If `eidToNonDeterministicAddress[dstEid]` is set, `mintAndSend` should route to that address instead of `bytes32(uint160(address(this)))`.
- ☐ Excess ETH provided in `msg.value` should be refunded via `_refundExcess`.

Negative behavior

- ☑ Revert with `IsolatedBackingEnabled` when `isolatedBackingEnabled` is true.
- ☐ Revert when the caller has insufficient frxUSD allowance/balance for `frxusd.transferFrom`.

- ☐ Revert when `remoteHopV2.sendOFT` rejects the fee or destination parameters.

Function: `send(uint32 dstEid, address recipient, uint256 amount)`

This function coordinates a cross-chain transfer of branded stablecoins by pulling `coin` from the caller, redeeming it into local frxUSD via `custodian.redeem`, and calling `remoteHopV2.sendOFT` to send frxUSD cross-chain with a compose payload that mints branded stablecoins to `recipient` on the destination.

Inputs

- `dstEid`
  - **Control**: Full control.
  - **Constraints**: Must correspond to a destination chain supported by `remoteHopV2` and may have a nondefault router address configured via `eidToNonDeterministicAddress[dstEid]`.
  - **Impact**: The LayerZero endpoint ID of the destination chain for the frxUSD OFT transfer.
- `recipient`
  - **Control**: Full control.
  - **Constraints**: Must be a valid EVM address expected by the destination compose handler (encoded as `abi.encode(recipient)`) — the destination router may revert or mint to `address(0)` depending on its token implementation.
  - **Impact**: The ultimate recipient address on the destination chain that will receive minted branded stablecoins during `hopCompose`.
- `amount`
  - **Control**: Full control.
  - **Constraints**: Rounded down to a multiple of 1e12 before transfers (`amount = (amount / 1e12) * 1e12`), and the caller must have sufficient `coin` balance and allowance for `coin.transferFrom`. The downstream `custodian.redeem` and `remoteHopV2.sendOFT` calls must succeed.
  - **Impact**: The branded stablecoin amount to bridge (prerounding) — drives the size of the `coin` transfer/burn and the frxUSD amount (`out`) sent cross-chain.

Branches and code coverage

Intended branches

- ☑ The `send` function should call `coin.transferFrom` to transfer `coin` from the caller to `address(this)`, call `custodian.redeem`, approve `remoteHopV2` to spend frxUSD, call

`remoteHopV2.sendOFT`, and then refund excess ETH via `_refundExcess`.

- ☑ Dust removal should round `amount` down to a multiple of 1e12 before transferring/burning and before calling `remoteHopV2.sendOFT`.
- ☑ If dust removal rounds `amount` down to 0, `send` can still forward fees (`msg.value`) and perform a `sendOFT` call with a zero token amount.
- ☐ If `eidToNonDeterministicAddress[dstEid]` is set, `send` should route to that address instead of `bytes32(uint160(address(this)))`.

Negative behavior

- ☐ Revert when the caller has insufficient `coin` allowance/balance for `coin.transferFrom`.
- ☐ Revert when `custodian.redeem` cannot honor the redemption (e.g., exceeds `maxRedeem` or insufficient liquidity).
- ☐ Revert when `remoteHopV2.sendOFT` rejects the fee or destination parameters.

5.6. Module: Component #2: Branded Stablecoin / FraxNetDepositV3

Function: `processDeposit(uint256 amount)`

This function processes a prefunded USDC balance held by the FraxNet deposit account. It deposits USDC into the configured `frxUSDCustodian` to mint frxUSD then either transfers/sends frxUSD to the configured destination or mints a branded stablecoin locally or via its cross-chain router. Any ETH held by this contract after execution is refunded to the caller via `_refundExcess()`.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The factory must not be paused, `amount` must be less than or equal to `1_000_000e6`, and the contract must already hold at least `amount` USDC (`USDC.balanceOf(address(this)) >= amount`). The downstream `frxUSDCustodian.deposit(amount, address(this))` and subsequent transfer/bridge calls must succeed.
  - **Impact**: The amount of USDC to convert into frxUSD (and optionally into a branded stablecoin) — determines the size of the custodian deposit and the amount transferred or bridged out.

Branches and code coverage

Intended branches

- ☑ The `processDeposit` function should approve USDC and call `frxUSDCustodian.deposit(amount, address(this))` to mint frxUSD.

- ☑ When `_asset != address(0)` and `targetEid == 30101`, `processDeposit` should mint the branded stablecoin to `targetAddress` via the branded custodian returned by `brandFactory.brandedTokenToUtils`.
- ☑ After `processDeposit`, USDC and frxUSD allowances should be consumed by downstream `transferFrom` calls.
- ☑ Also, `_refundExcess` should transfer `address(this).balance` to `msg.sender` (demonstrated via `processDeposit(0)`).
- ☐ When `_asset == address(0)` and `targetEid != 30101`, `processDeposit` should call `remoteHop.sendOFT` to bridge frxUSD cross-chain.
- ☐ When `_asset != address(0)` and `targetEid != 30101`, `processDeposit` should call the branded router `mintAndSend` to bridge and mint cross-chain.

Negative behavior

- ☐ Revert when the factory is paused.
- ☐ Revert when `amount > 1_000_000e6`.
- ☐ Revert when the contract has insufficient USDC balance for `amount`.
- ☐ Revert when the downstream custodian/router/bridge calls revert.

Function: `processRedemption(uint256 amount)`

This function redeems frxUSD held by the FraxNet deposit account into USDC and delivers it to the configured destination. It may aggregate redemption liquidity across the USDC custodian and an RWA redeemer. If the account is configured for a branded asset, it first redeems that branded token into frxUSD. Delivery occurs via direct transfer, Scroll L1 bridge, or CCTP depending on `targetEid`.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The factory must not be paused, and the contract must hold at least `amount` frxUSD after any branded-asset redemption. The requested amount must be serviceable by the combined redeemable liquidity (`maxRWARedeemable + maxUsdcRedeemable`), and destination-specific configs must be set (e.g., CCTP domain/messenger for non-mainnet/non-Scroll paths).
  - **Impact**: The amount of frxUSD to convert into USDC — drives the size of the custodian redemption(s) and the USDC delivery amount.

Branches and code coverage

Intended branches

- ☐ When `_asset != address(0)`, `processRedemption` should approve and call `IBrandedCustodian(custodian).redeem` via `_redeemAsset` before redeeming frxUSD.
- ☐ When `maxUsdcRedeemable >= amount`, `processRedemption` should redeem entirely via `usdcCustodian.redeem(amount, address(this), address(this))`.
- ☐ When `0 < maxUsdcRedeemable < amount`, `processRedemption` should redeem partially via `usdcCustodian.redeem(maxUsdcRedeemable, ...)` and redeem the remainder via `rwaRedeemer.redeemRWA`.
- ☐ When `maxUsdcRedeemable == 0`, `processRedemption` should redeem entirely via `rwaRedeemer.redeemRWA(amount)`.
- ☐ For `targetEid == 30101`, `processRedemption` should `USDC.transfer` to `address(uint160(uint256(targetAddress)))`.
- ☐ For `targetEid == 30214`, `processRedemption` should call `IL1USDCGateway.depositERC20` and then `_refundExcess`.
- ☐ For other EIDs, `processRedemption` should call `ITokenMessenger.depositForBurn` and then `_refundExcess`, using `targetUsdcAtaAddress` when `targetEid == 30168`.

Negative behavior

- ☐ Revert with `Paused` when the factory is paused.
- ☐ Revert with `InsufficientBalance` when `frxUSD.balanceOf(address(this)) < amount`.
- ☐ Revert with `SizeTooLarge` when `amount > maxRWARedeemable + maxUsdcRedeemable`.
- ☐ Revert with `CCTPNotConfiguredForThisChain` when CCTP configuration for `targetEid` is missing.
- ☐ Revert with `AtaAccountNotInitialized` when `targetEid == 30168` and the SOL USDC ATA recipient is unset.
- ☐ Revert with `RefundFailed` when `_refundExcess` cannot send ETH to `msg.sender`.

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

This function allows the `factory` contract to recover arbitrary ERC-20 tokens that are held by a FraxNet deposit account instance and transfer them to `recipient`. It is intended as an administrative escape hatch for accidentally sent tokens.

Inputs

- `tokenAddress`
  - **Control**: Full control.
  - **Constraints**: Only callable when `msg.sender == address(factory)` — `tokenAddress` must be a contract implementing ERC-20 `transfer`; otherwise, the call may revert.
  - **Impact**: The ERC-20 token address to recover from this contract.

- `tokenAmount`
  - **Control**: Full control.
  - **Constraints**: The contract must have at least `tokenAmount` balance of `tokenAddress`; otherwise, the ERC-20 transfer is expected to revert.
  - **Impact**: The amount of tokens to transfer out of the deposit account.
- `recipient`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address for standard ERC-20 implementations.
  - **Impact**: The recipient address that receives the recovered tokens.

Branches and code coverage

Intended branches

- ☐ When called by `factory`, `recoverERC20` should call `IERC20(tokenAddress).transfer(recipient, tokenAmount)` and move tokens out of the deposit account.
- ☐ The `recoverERC20` function should allow recovering accidentally sent ERC-20 tokens (including USDC/frxUSD).

Negative behavior

- ☐ Revert with `Forbidden` when called by any address other than `factory`.
- ☐ Revert when `IERC20(tokenAddress).transfer` fails (e.g., insufficient balance or nonstandard token behavior).

5.7. Module: Component #3: Odos Fraxlend Leverage Router PR #19 / FraxlendLeverOdos

Function: `recoverERC20(address token, uint256 amount)`

This admin-only function recovers ERC-20 tokens held by this contract by calling `token.transfer(msg.sender, amount)` via a low-level call.

Inputs

- `token`
  - **Control**: Full control.
  - **Constraints**: Caller must have `DEFAULT_ADMIN_ROLE`. For recovery to succeed, `token` should be an ERC20 contract implementing `transfer(address,uint256)` and this contract must hold at least `amount`

tokens. The low-level call return value is unchecked, so failures may not revert.

- **Impact**: The ERC20 token contract address to transfer out of `FraxlendLeverOdos`.

- `amount`
  - **Control**: Full control.
  - **Constraints**: Should be less than or equal to this contract's balance of `token` for a successful transfer. If `token.transfer` reverts or returns `false`, the failure is not surfaced by this function.
  - **Impact**: The number of `token` units to attempt to transfer to `msg.sender`.

Branches and code coverage

Intended branches

- ☑ Recovering an ERC-20 balance transfers `amount` to the admin (`test_recover`).
- ☐ Recovery should work for tokens that do not return a boolean value (nonstandard ERC-20s).

Negative behavior

- ☐ Revert when called by an account without `DEFAULT_ADMIN_ROLE`.
- ☐ Revert (or otherwise signal failure) when `token.transfer` fails or returns `false`.
- ☐ Revert when `token` is not a contract / does not implement `transfer(address,uint256)`.

Function: `swapExactTokensForTokens(uint256 amountIn, uint256 amountOutMin, address[] path, address to, uint256 deadline)`

This function implements the Fraxlend pair swapper interface by pulling `amountIn` of the sell token (`path[0]`) from the caller, decoding an Odos router calldata payload embedded in `path`, and forwarding that payload to `odosRouter` via a low-level call. The `amountOutMin`, `to`, and `deadline` parameters are accepted only for interface compatibility and are unused.

Inputs

- `amountIn`
  - **Control**: Full control.
  - **Constraints**: The `msg.sender` must have at least `amountIn` balance of `path[0]` and have approved this contract for at least `amountIn`. If `needsExternalBorrowCheck[msg.sender]` is `true` and `path[0]` equals `IFraxlend(msg.sender).asset()`, the function enforces `borrowLimit()` and reverts with `BorrowLimit()` when exceeded.

- **Impact**: The amount of input tokens pulled from `msg.sender` and made available to `odosRouter` for the swap.

- `amountOutMin`
  - **Control**: Full control.
  - **Constraints**: Unchecked by this contract — any slippage protection must be enforced by the Odos payload executed by `odosRouter`.
  - **Impact**: Legacy minimum-output parameter that callers must not rely on being enforced here.
- `path`
  - **Control**: Full control.
  - **Constraints**: Must be an Odos payload encoded as an `address[]` (typically produced by `encode()`). Also, `path.length` must be at least three, `path[1]` encodes the payload length, and `path[2..path.length-2]` contain 20-byte chunks of the payload. Otherwise, `decode(path)` reverts with `EmptyPath()` or produces malformed calldata.
  - **Impact**: Encoded transport for the Odos router calldata and the source of the sell token address (`path[0]`).
- `to`
  - **Control**: Full control.
  - **Constraints**: Unused.
  - **Impact**: Legacy recipient parameter retained for interface compatibility.
- `deadline`
  - **Control**: Full control.
  - **Constraints**: Unused — any deadline enforcement must be part of the Odos payload.
  - **Impact**: Legacy deadline parameter retained for interface compatibility.

Branches and code coverage

Intended branches

- ☑ Swap succeeds when called via a Fraxlend pair with a valid Odos payload encoded in `path` (`test_LeverageWithOdosRouterCall`).
- ☑ Decoding an encoded Odos payload used for swaps does not revert (`test_decodeGas`).
- ☐ Swap should return a populated `amounts` array consistent with the expected router interface.

Negative behavior

- ☐ Revert with `ShutOff()` when `shutoff` is enabled.
- ☐ Revert with `EmptyPath()` when `path.length < 3`.
- ☐ Revert with `SwapFailed()` when the `odosRouter` call fails.

- ☐ Revert with `BorrowLimit()` when the external borrow-limit check is enabled and the limit would be exceeded.
- ☐ Revert when `IERC20(path[0]).transferFrom` fails due to missing allowance or balance.

5.8. Module: Component #1: HopV2/HopV2

Function: `sendOFT(address _oft, uint32 _dstEid, bytes32 _recipient, uint256 _amountLD, uint128 _dstGas, bytes _data)`

This function initiates an OFT transfer through HopV2 with an optional compose payload. It constructs a `HopMessage`, trims LayerZero dust from `_amountLD`, and pulls the underlying asset from the caller. It then either (a) performs a local transfer with an optional `IHopComposer.hopCompose` call when `_dstEid == localEid`, or (b) routes the transfer through LayerZero by calling `_sendToDestination` when `_dstEid != localEid`. Finally, it validates `msg.value` against the computed fee and refunds any excess ETH to the caller.

Inputs

- `_oft`
  - **Control**: Full control.
  - **Constraints**: Must be approved in `approvedOft[_oft]` or the call reverts with `InvalidOFT()`. Must implement the expected OFT interfaces (`IOFT` / `IOFT2`) because Hop reads `token()` and `decimalConversionRate()` and calls `quoteSend()` / `send()`.
  - **Impact**: Selects which underlying ERC-20 asset is pulled and bridged and which `decimalConversionRate` is used for dust trimming and fee calculations.
- `_dstEid`
  - **Control**: Full control.
  - **Constraints**: If `_dstEid == localEid`, the transfer executes locally and no LayerZero send occurs. If `_dstEid != localEid`, the call executes the remote-send path and requires sufficient `msg.value` for fees. When called on `FraxtalHopV2`, `_dstEid` must be supported (`_dstEid == FRAXTAL_EID` or `remoteHop(_dstEid) != bytes32(0)`) or the call reverts with `InvalidDestinationChain()`.
  - **Impact**: The intended LayerZero Endpoint ID encoded into `HopMessage.dstEid`, which determines whether Hop executes locally or routes the transfer through LayerZero (and, for `RemoteHopV2`, whether Fraxtal will forward to a non-Fraxtal destination).
- `_recipient`
  - **Control**: Full control.

  - **Constraints**: For local sends, the decoded `address(uint160(uint256(_recipient)))` must be able to receive the underlying token, and if `_data` is nonempty it must implement `IHopComposer.hopCompose` or the call may revert. For remote sends, the meaning of `_recipient` depends on the derived routing (`_generateSendParam`) and whether a compose payload is included.
  - **Impact**: Identifies the recipient (as `bytes32`) that receives bridged tokens and, when `_data` is nonempty, the composer contract that receives a `hopCompose` call with `_data`.
- `_amountLD`
  - **Control**: Full control.
  - **Constraints**: The effective amount is dust-trimmed via `removeDust(_oft, _amountLD)` before any transfer occurs. The caller must have sufficient balance and allowance of `IOFT(_oft).token()` for the cleaned amount or `safeTransferFrom` will revert. If the cleaned amount is zero, no ERC-20 transfer is performed.
  - **Impact**: The requested transfer amount in local decimals — the cleaned amount is what is actually pulled from the caller and credited/transferred on the destination path.
- `_dstGas`
  - **Control**: Full control.
  - **Constraints**: Only impacts compose execution when `_data` is nonempty and a remote-send path is used. Remote implementations may clamp or override this value when constructing LayerZero executor options (for example, RemoteHopV2 enforces a minimum compose gas and uses a Fraxtal compose gas cap).
  - **Impact**: Gas budget used by the LayerZero executor for the destination compose call, affecting liveness (execution success) and fee estimation.
- `_data`
  - **Control**: Full control.
  - **Constraints**: Arbitrary bytes — when nonempty, it triggers compose behavior and increases executor fee estimates (for example, `quoteHop` scales with message length). On the local path, this data is passed directly into `IHopComposer.hopCompose` and any revert bubbles up.
  - **Impact**: Application payload carried inside `HopMessage.data` to be executed by the recipient composer and a primary driver of remote compose gas/fee sizing.

Branches and code coverage

Intended branches

☑ Local transfer path (`_dstEid == localEid`) should transfer tokens via `_sendLocal` when

`_data` is empty.

- ☑ Local compose path should call `IHopComposer.hopCompose` via `_sendLocal` when `_data` is nonempty.
- ☑ Remote transfer path (`_dstEid != localEid`) should call `_sendToDestination` and succeed when the quoted fee is provided.
- ☑ Dust trimming should apply `removeDust` and deliver only the cleaned amount to the recipient.
- ☑ Excess fee handling should refund `msg.value - sendFee` via `_handleMsgValue` when `msg.value > sendFee`.

Negative behavior

- ☑ Revert with `HopPaused()` when the hop is paused.
- ☑ Revert with `InvalidOFT()` when `_oft` is not approved.
- ☑ Revert with `InsufficientFee()` when `msg.value < sendFee`.
- ☑ Revert with `InvalidDestinationChain()` on Fraxtal when `_dstEid` is unsupported.
- ☐ Revert with `RefundFailed()` when refunding excess ETH fails.

5.9. Module: Component #7: Frax OFT Minter Array / MinterModule

Function: `_addMinter(address minter_address)`

This function grants an address minter privileges by setting `minters[minter_address] = true` and appending it to `minters_array`, enabling the address to pass `onlyMinters` access control for `minter_mint()` and `minter_burn_from()`.

Inputs

- `minter_address`
  - **Control**: Full control.
  - **Constraints**: The address must not already be an active minter or the call reverts with `AlreadyExists`. Even though `_removeMinter` uses `address(0)` as a tombstone value in `minters_array`, `address(0)` is not rejected.
  - **Impact**: The address to be authorized as a minter and appended to the on-chain minter enumeration array.

Branches and code coverage

Intended branches

- ☑ Adding a new minter makes `minters(minter_address)` return `true`.
- ☑ The added minter is appended to `minters_array`.
- ☑ The added minter can call `minter_mint()` and `minter_burn_from()` successfully.

Negative behavior

- ☑ Revert if the caller is not the owner (via the external `addMinter` wrapper).
- ☐ Revert if the minter already exists.
- ☐ Revert if `minter_address` is `address(0)`.

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

This function mints `m_amount` tokens to `m_address` via `_mint`, increments `totalMinted`, and emits `TokenMinterMinted`. It is intended to be called through an external entry point protected by the `onlyMinters` modifier.

Inputs

- `m_address`
  - **Control**: Full control.
  - **Constraints**: Must not be `address(0)` (enforced by the underlying ERC-20 `_mint`). Transfers may be blocked by pause/freeze checks in `_beforeTokenTransfer` when the caller is not the owner.
  - **Impact**: The account that receives the newly minted tokens and whose balance increases.
- `m_amount`
  - **Control**: Full control.
  - **Constraints**: Must not cause arithmetic overflow in `totalMinted` or total supply (Solidity 0.8+ checked arithmetic). Minting may be blocked by pause/freeze checks in `_beforeTokenTransfer`.
  - **Impact**: The amount of tokens to mint, increasing total supply, the recipient balance, and the `totalMinted` counter.

Branches and code coverage

Intended branches

- ☑ An authorized minter can mint tokens, and the recipient balance increases by the minted amount.
- ☑ The `totalMinted()` view reflects the minted amount.

Negative behavior

- ☑ Revert if the caller is not an approved minter.
- ☐ Revert if minting to `address(0)`.
- ☐ Revert if minting is blocked by pause/freeze checks.

Function: `_removeMinter(address minter_address)`

This function revokes an address's minter privileges by clearing the `minters` mapping entry and replacing the first matching `minters_array` element with `address(0)`, preserving array indexes for external enumerators.

Inputs

- `minter_address`
  - **Control**: Full control.
  - **Constraints**: The address must be an active minter (`minters[minter_address] == true`) or the call reverts with `AddressNonexistant`. Removal uses a linear scan over `minters_array` and sets `address(0)` as a tombstone to keep indexes stable.
  - **Impact**: The minter address to deactivate, preventing it from passing `onlyMinters` and marking its `minters_array` slot as removed.

Branches and code coverage

Intended branches

- ☑ Removing an existing minter clears the `minters` mapping entry.
- ☑ The removed minter's array slot is replaced with `address(0)`, and the array length is unchanged.

Negative behavior

- ☑ Revert if the caller is not the owner (via the external `removeMinter` wrapper).
- ☐ Revert if the minter does not exist.

5.10. Module: Component #1: HopV2/RemoteHopV2

Function: `lzCompose(address _oft, bytes32 _guid, bytes _message, address _executor, bytes _executorData)`

This function is the LayerZero entry point for delivering an OFT compose message into RemoteHopV2. It validates message integrity via `_validateComposeMessage`, decodes the `HopMessage` and credited amount from the payload, and overwrites `hopMessage.srcEid` and `hopMessage.sender` for untrusted composers. Finally, it executes `_sendLocal` to transfer tokens to the recipient and optionally call `IHopComposer.hopCompose`.

Inputs

- `_oft`
  - **Control**: Full control.
  - **Constraints**: Must be approved in `approvedOft[_oft]` or the call reverts with `InvalidOFT()`. The Hop contract must already hold the underlying balance credited by the OFT delivery so `_sendLocal` can transfer `amountLD` out.
  - **Impact**: The address of the originating OApp/token.
- `_guid`
  - **Control**: Full control.
  - **Constraints**: No contract-level constraints — this parameter is ignored by the implementation.
  - **Impact**: The globally unique identifier of the message.
- `_message`
  - **Control**: Full control.
  - **Constraints**: Must be a valid `OFTComposeMsgCodec`-encoded message — otherwise decoding (`srcEid`, `composeFrom`, `nonce`, `amountLD`, and `composeMsg`) or the ABI decode of the embedded `HopMessage` may revert. Messages are considered duplicates based on `keccak256(abi.encode(_oft, srcEid, nonce, composeFrom))` and will be ignored on subsequent deliveries.
  - **Impact**: The encoded message content in the format of the `OFTComposeMsgCodec`.
- `_executor`
  - **Control**: Full control.
  - **Constraints**: No contract-level constraints — this parameter is ignored by the implementation.
  - **Impact**: The executor address placeholder, unused by this implementation.
- `_executorData`
  - **Control**: Full control.
  - **Constraints**: No contract-level constraints — this parameter is ignored by the implementation.
  - **Impact**: Additional data for checking for a specific executor.

Branches and code coverage

Intended branches

- ☑ A trusted message should call `_sendLocal` and transfer `amountLD` to the decoded recipient when `hopMessage.data` is empty.
- ☑ A trusted message should call `IHopComposer.hopCompose` via `_sendLocal` when

`hopMessage.data` is nonempty.

- ☑ An untrusted composer should have `hopMessage.srcEid` and `hopMessage.sender` overwritten before executing `_sendLocal`.
- ☑ A duplicate message should be detected by `_validateComposeMessage` and ignored (early return) without double-spending tokens.
- ☐ And `Hop` event emission should be asserted for correct parameters (`_oft`, recipient, and `amountLD`).

Negative behavior

- ☑ Revert with `NotEndpoint()` when called by any address other than the configured LayerZero endpoint.
- ☑ Revert with `InvalidOFT()` when `_oft` is not approved.
- ☐ Revert with `HopPaused()` when the hop is paused.

5.11. Module: Component #1: HopV2/RemoteVaultDeposit

Function: `deposit(uint256 _amount, address _to)`

This function initiates a cross-chain deposit into a remote ERC-4626 vault. It transfers `ASSET` from the caller into RemoteVaultHop, forwards `msg.value` as the fee, and calls `RemoteVaultHop.deposit`. After the call returns, it refunds any ETH held by this contract to the caller.

Inputs

- `_amount`
  - **Control**: Full control.
  - **Constraints**: The caller must have sufficient `ASSET` balance and must have approved RemoteVaultDeposit to spend at least `_amount` `ASSET` (so that `safeTransferFrom` can transfer the funds into `REMOTE_VAULT_HOP`). The forwarded `msg.value` must be sufficient for `RemoteVaultHop.deposit` to succeed or the call will revert. The effective bridged amount may be dust-trimmed downstream by RemoteVaultHop/HopV2.
  - **Impact**: The amount of underlying asset the caller attempts to deposit cross-chain — this amount is moved into RemoteVaultHop, and the eventual remote deposit may use a dust-trimmed amount while leaving a remainder in RemoteVaultHop.
- `_to`
  - **Control**: Full control.
  - **Constraints**: For the end-to-end flow to complete, `_to` must be a valid nonzero recipient address, as the return leg mints receipt tokens to `userAddress` and minting to the zero address reverts.

- **Impact**: The address that will be encoded as `userAddress` for the cross-chain flow and ultimately receive the minted RemoteVaultDeposit receipt tokens on a successful `DepositReturn`.

Branches and code coverage

Intended branches

- ☑ Calling `deposit` should call `safeTransferFrom` to transfer `_amount` `ASSET` from the caller to `REMOTE_VAULT_HOP`, then call `RemoteVaultHop.deposit`.
- ☑ A deposit with LZ dust should succeed and leave the dust remainder in RemoteVaultHop.
- ☑ An end-to-end deposit return should mint receipt tokens to `_to` when `hopCompose(DepositReturn)` is processed.
- ☐ A deposit should refund only this call's leftover ETH back to the caller (and not the entire contract balance).

Negative behavior

- ☐ Revert when the caller has insufficient `ASSET` allowance or balance for `safeTransferFrom`.
- ☑ Revert when `RemoteVaultHop.deposit` rejects the call due to insufficient fee (`InsufficientFee()`).
- ☐ Revert when refunding ETH back to the caller fails.

Function: `redeem(uint256 _amount, address _to)`

This function initiates a cross-chain redemption from a remote ERC-4626 vault. It burns `_amount` receipt tokens from the caller, forwards `msg.value` as the fee, and calls `RemoteVaultHop.redeem`. After the call returns, it refunds any ETH held by this contract to the caller.

Inputs

- `_amount`
  - **Control**: Full control.
  - **Constraints**: The caller must have at least `_amount` receipt tokens or `_burn` will revert. The forwarded `msg.value` must be sufficient for `RemoteVaultHop.redeem` to succeed or the call will revert. The eventual underlying amount returned may be dust-trimmed downstream.
  - **Impact**: The quantity of receipt tokens to burn and the amount communicated to RemoteVaultHop as the redeem amount, determining how many remote vault shares are redeemed and how much underlying is returned to `_to` on the return leg.

- `_to`
  - **Control**: Full control.
  - **Constraints**: For the end-to-end flow to complete, `_to` must be a valid nonzero recipient address, as the return leg transfers underlying tokens to `userAddress` and many ERC-20 implementations revert on transfers to the zero address.
  - **Impact**: The address that will be encoded as `userAddress` for the cross-chain flow and ultimately receive the redeemed underlying tokens on a successful `RedeemReturn`.

Branches and code coverage

Intended branches

- ☑ The redemption should `_burn` receipt tokens and call `RemoteVaultHop.redeem`.
- ☑ An end-to-end redemption return should transfer the underlying to `_to` when `hopCompose(RedeemReturn)` is processed.
- ☐ A redemption should refund only this call's leftover ETH back to the caller (and not the entire contract balance).

Negative behavior

- ☑ Demonstrate that `redeem(0)` can sweep any historical ETH balance held by this contract due to full-balance refund logic.
- ☐ Revert when the caller attempts to redeem more receipt tokens than their balance.
- ☑ Revert when `RemoteVaultHop.redeem` rejects the call due to insufficient fee (`InsufficientFee()`).

6. Assessment Results

During our assessment on the scoped Frax (USD Stablecoin) contracts, we discovered 10 findings. No critical issues were found. Two findings were of medium impact, four were of low impact, and the remaining findings were informational in nature.

6.1. Observations and recommendations

This assessment covered seven components across multiple repositories, focusing on differential changes. No critical issues were identified, and the majority of findings are of informational or low severity. The protocol team provided clear rationale for their design decisions throughout the engagement.

The issues identified are addressable without major architectural changes. Below are key observations and recommendations for strengthening the codebase.

Implicit decimal assumptions in cross-component arithmetic

Several of the issues identified relate to implicit assumptions about token decimals across the Branded Stablecoin system (see Findings 3.1. ↗ and 3.2. ↗). As the protocol is designed to support multiple collateral types, including tokens with fewer than 18 decimals, we recommend ensuring that all arithmetic involving token amounts derives decimal parameters dynamically rather than relying on hardcoded values.

Inconsistent token-transfer safety patterns

Across the assessed components, token-transfer handling varies between raw ERC-20 calls, low-level calls, and safe wrappers (see Findings 3.5. ↗ and 3.7. ↗). Standardizing on a single safe transfer pattern across all contracts would reduce the surface for silent failures, particularly when interacting with nonstandard ERC-20 tokens.

Cross-chain operations rely on optimistic assumptions

The cross-chain deployment and messaging flows treat a successful `send` on the source chain as confirmation of execution on the destination chain (see Discussion point 4.4. ↗). While this simplifies the implementation and the protocol team has provided reasonable justification for this approach, it is worth noting that destination-side failures require manual administrator intervention to recover.

6.2. 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.