Zellic

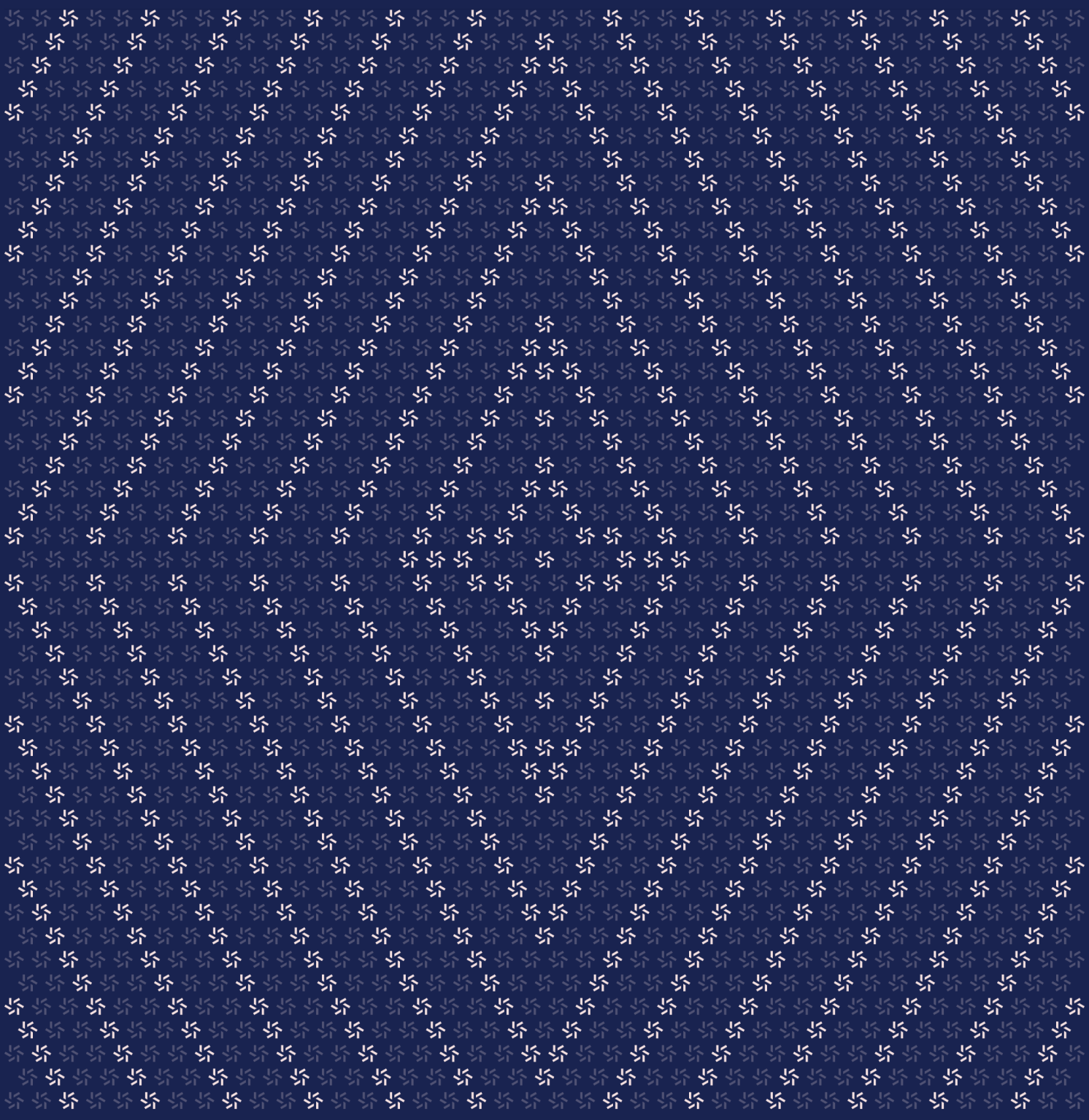
Prepared for
Bryce Morrow
Adam Hamden
Charlie Pyle
HourGlass

Prepared by
Aaron Esau
Quentin Lemauf
Zellic

October 3, 2025

Stable Predeposit

Smart Contract Security Assessment



Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for HourGlass from September 26th to October 3rd, 2025. During this engagement, Zellic reviewed Stable Predeposit'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:

- What kind of mistakes made by contract admins during management could lead to Recovery mode being activated to rescue user funds?
- Are the assumptions made about interactions with SparkLend and USDT0/LayerZero correct?

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

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 Stable Predeposit contracts, we discovered one finding, which was of medium impact.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Stable Predeposit

HourGlass contributed the following description of Stable Predeposit:

> The Hourglass Stable Pre-Deposit Vaults are smart contracts deployed on Ethereum that accept stablecoin deposits (USDT, USDC, frxUSD) before the launch of Stable, a new Layer 1 blockchain. During the pre-deposit phase, users deposit stablecoins at a fixed 1:1 ratio and receive vault shares, with some vaults generating yield through protocols like SparkLend while others implement KYC-segregated treasury management. Once Stable launches, depositors can redeem their shares and bridge the underlying to Stable through the LayerZero integration.

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:

Stable Predeposit Contracts

Type	Solidity
Platform	EVM-compatible
Target	stable-pre-deposit
Repository	https://github.com/hourglass-foundation/stable-pre-deposit/ ↗
Version	`4b07ff620f27dd5e4796937fcdfd0437a3dc7d15`

Programs	`HourglassStableVaultKYC.sol`
	`HourglassStableVaultFrax.sol`
	`HourglassStableVault.sol`
	`HourglassStableVaultBridge.sol`
	`base/HourglassStableVaultBridgeManager.sol`
	`base/HourglassStableVaultDepositCapManager.sol`
	`base/HourglassStableVaultDepositWindowManager.sol`
	`base/HourglassStableVaultKYCManager.sol`
	`base/HourglassStableVaultTreasuryManager.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.7 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau
Engineer
aaron@zellic.io

Quentin Lemauf
Engineer
quentin@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 26, 2025	Kick-off call
September 26, 2025	Start of primary review period
October 3, 2025	End of primary review period

3. Detailed Findings

3.1. User funds stay locked when the bridge is not KYC-approved

Target	HourglassStableVaultKYC		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The vault only updates KYC status through `_setKycStatusWithAccounting` in the KYC phase, which rejects accounts that hold no shares:

```
// src/HourglassStableVaultKYC.sol:883-885
uint256 userShares = balanceOf(user);
if (userShares == 0) revert NoSharesFound();
```

The LayerZero bridge contract is deployed with a zero share balance. During the KYC phase, `batchSetKycStatus([bridge], true)` therefore reverts, so the bridge cannot be whitelisted unless operators mint a share to it while the Deposit phase is still open. Later, when Withdraw mode starts, the bridge redemption flow (`HourglassStableVaultBridge._redeemSharesAndBridgeToStable`) pulls user shares and calls `redeemBridge`, which delegates to `redeemKyc`:

```
// src/HourglassStableVaultKYC.sol:660-670
function redeemKyc(...) public ...
    onlyCallerIsBridge onlyMode(OperationalMode.Withdraw)
    onlyKycApproved(owner) {
    (uint256 usdtOut,,) = previewRedeem(owner, shares);
    ...
}
```

Because the bridge was never KYC-approved, `onlyKycApproved(owner)` always fails and every Withdraw-phase redemption reverts.

Impact

Withdraw-mode bridge withdrawals are blocked — user funds stay locked until Recovery mode is available.

The requirement to premint a bridge share is not documented, and the development team was not aware of it at the time this issue was noted, so missing this step is very likely in practice.

Recommendations

Allow trusted infrastructure (e.g., the bridge) to be KYC-approved even with a zero share balance, or seed the bridge with a share during deployment so `_setKycStatusWithAccounting` can succeed while in the KYC phase.

Nevertheless, we recommend simplifying this workflow as much as possible to reduce the chance of human error.

Remediation

This issue has been acknowledged by HourGlass, and a fix was implemented in commit `614981fb` ↗.

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. Code-quality improvements

These following notes on code quality are not directly security related.

Remove unused `receive()` handlers and companion `recoverEth`

The vault implementations (`HourglassStableVault`, `HourglassStableVaultFrax`, `HourglassStableVaultKYC`) expose an empty `receive()` hook:

```
receive() external payable { }
```

Each contract also wires an admin-only `recoverEth` helper purely to drain whatever slips in via the passive handler. Because the vaults do not intentionally accept native coin deposits and the ecosystem already exposes targeted recovery flows on the bridge contract, this pattern just introduces maintenance surface without delivering functionality. We recommend dropping both the `receive()` function and the paired `recoverEth` entry points so unexpected ETH transfers revert outright.

Force full Spark withdrawal during recovery

Note that `HourglassStableVault.withdrawFromSparkRecovery` forwards the caller-provided `amount` to `_withdrawFromSpark`:

```
function withdrawFromSparkRecovery(uint256 amount)
    external onlyMode(OperationalMode.Recovery) nonReentrant {
    _withdrawFromSpark(amount);
}
```

Allowing arbitrary partial withdrawals makes the recovery path dependent on user-provided inputs when the goal is to unwind the entire Spark position. Because `_withdrawFromSpark(0)` already translates to "withdraw everything", consider hardcoding `0` (or otherwise ignoring the argument) so recovery mode always brings all liquidity back into the vault for subsequent share redemptions.

Drop redundant `unchecked` increment in KYC batch loop

In `HourglassStableVaultKYC.batchSetKycStatus`, the loop increments `i` inside an `unchecked` block:

```
for (uint256 i = 0; i < length;) {
    address user = users[i];
    if (user == address(0)) revert ZeroAddress();
    _setKycStatusWithAccounting(user, status);
    unchecked {
        ++i;
    }
}
```

Since the project targets Solidity ^0.8.29, the compiler already elides redundant overflow checks for simple `++i` increments (per the 0.8.22 release). Removing the manual `unchecked` block makes the loop clearer without changing gas costs.

Remove unreachable `DepositCapNotSet` branch

Note that `HourglassStableVaultDepositCapManager._getMaxDepositAgainstCap` guards against an unset cap:

```
if (_depositCap == 0) revert DepositCapNotSet();
```

However, `_depositCap` is always initialized through `_setDepositCap`, which rejects zero values:

```
if (newCap == 0 || newCap < currentDeposits) revert InvalidDepositCap();
```

Because the zero check can never fire, the `DepositCapNotSet` revert is effectively dead code. We suggest removing this code.

Use `forceApprove` for USDT allowances

`HourglassStableVault`'s constructor currently performs a vanilla `approve` call against USDT:

```
USDT.approve(address(SPARK_LEND_USDT_POOL), type(uint256).max);
```

Because USDT mandates zeroing allowances before reapproval, switch to OZ's `SafeERC20.forceApprove`, which wraps that reset logic. Update the constructor to:

```
USDT.forceApprove(address(SPARK_LEND_USDT_POOL), type(uint256).max);
```

Consider using a merkle tree for KYC status

The current KYC status management in `HourglassStableVaultKYC` relies on a mapping and batch updates, which will be gas-intensive for large user bases. Consider implementing a merkle tree-based approach ↗ for KYC status verification. However, note that this would require relatively large code changes to implement.

4.2. Lack of bridging gas refunds

Note that `_redeemSharesAndBridgeToStable` does not appear to refund the excess fees from LayerZero message sending. It would be best practice to include this; however, we do not consider the lack of a fee refund to be an issue, because the code includes recovery functions. Additionally, a user would have to manually interact with the contract — and mistakenly send too much — to cause this to be an issue.

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: HourglassStableVaultBridge.sol

Function: `bridgeToStable(uint256 shares, address recipient, bytes lzOpts, uint16 slippageBps)`

This function unwraps vault shares and bridges underlying assets to Stable, over LayerZero.

Inputs

- `shares`
  - **Control**: Full.
  - **Constraints**: The amount cannot be zero. Due to the `transferFrom` call, the user must have at least this many shares and have approved the bridge contract to spend them.
  - **Impact**: The number of shares to redeem and bridge.
- `recipient`
  - **Control**: Full.
  - **Constraints**: The address cannot be the zero address.
  - **Impact**: The address on Stable that will receive the bridged assets.
- `lzOpts`
  - **Control**: Full.
  - **Constraints**: Must be a valid LayerZero options byte array (enforced by the messaging library in LayerZero).
  - **Impact**: Options for the LayerZero message, such as extra gas.
- `slippageBps`
  - **Control**: Full.
  - **Constraints**: Must be between 0 and `MAX_SLIPPAGE_BPS` (500).
  - **Impact**: The slippage tolerance in basis points for the bridge.

Branches and code coverage

Intended branches

- Successfully able to call the function with intended behavior.
  - ☑ Test coverage

Negative behavior

- `shares` cannot be zero.
  - ☑ Negative test
- `recipient` cannot be the zero address.
  - ☑ Negative test
- Slippage is too high.
  - ☑ Negative test
- Insufficient fee is paid.
  - ☑ Negative test
- Caller has insufficient balance or allowance of vault shares.
  - ☑ Negative test

Function call analysis

- `this.VAULT.transferFrom(msg.sender, address(this), shares)`
  - **What is controllable?** `shares`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The transaction reverts.
- `this._redeemSharesAndBridgeToStable(shares, msg.sender, this._addressToBytes32(recipient), lzOpts, slippageBps) -> this.VAULT.redeemBridge(shares, address(this), address(this))`
  - **What is controllable?** All of the arguments to some extent, besides `msg.sender`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The transaction reverts.

Function: `bridgeToStableWithPermit(uint256 shares, address recipient, bytes lzOpts, uint16 slippageBps, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

This function is very similar to `bridgeToStable` as it calls the same internal function, but it uses an ERC-20 permit to approve the bridge contract to spend the user's vault shares, instead of requiring a prior approval transaction. Please read our notes in the signoff above for the `bridgeToStable` function, as they apply here as well.

Function: `recoverErc20(address token, uint256 amount)`

This function allows the owner to recover any ERC-20 tokens stuck in the contract, at any time.

Inputs

- `token`
  - **Control**: Full.
  - **Constraints**: None, besides allowing an external call to `safeTransfer(address,uint256)`.
  - **Impact**: The address of the token to recover.
- `amount`
  - **Control**: Full.
  - **Constraints**: The amount cannot be zero and cannot exceed the contract's balance of the token.
  - **Impact**: The amount of tokens to recover.

Branches and code coverage

Intended branches

- Successfully able to call the function with intended behavior.
  - ☑ Test coverage

Negative behavior

- Caller must be the owner.
  - ☑ Negative test
- The contract has sufficient balance of the token.
  - ☐ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(token), recipient, amount)`
  - **What is controllable?** All arguments are controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The transaction reverts.

Function: `recoverEth()`

This function allows the owner to recover any ETH stuck in the contract, at any time.

Branches and code coverage

Intended branches

- Successfully able to call the function with intended behavior.
  - ☑ Test coverage

Negative behavior

- Caller must be the owner.
  - ☑ Negative test
- The ETH transfer must succeed.
  - ☑ Negative test

Function call analysis

- None.

5.2. Module: HourglassStableVaultFrax.sol

Function: `deposit(uint256 assets, address receiver)`

This function is the primary entry point for users to supply FRXUSDT during the deposit window. It mints shares 1:1 after transferring the tokens into the vault.

Inputs

- `assets`

- **Control**: N/A.
- **Constraints**: Must be greater than zero and within the remaining cap (`maxDeposit`).
- **Impact**: Determines the number of shares minted (1:1) and increases the total supply tracked by the vault.

- `receiver`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives freshly minted shares.

Branches and code coverage

Intended branches

- Shares are minted 1:1 with the deposited USDT.
  - ☑ Test coverage

Negative behavior

- Calls outside Deposit mode and window revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(this.asset()), caller, address(this), assets)`
  - **What is controllable?** The caller controls the number of FRXUSDT to deposit (`assets`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `mint(uint256 shares, address receiver)`

This function is an alternative deposit path that targets a desired share amount during the deposit window.

Inputs

- `shares`

    - **Control**: N/A.
    - **Constraints**: Must be greater than zero and within the remaining cap (`maxDeposit`).
    - **Impact**: Determines how much FRXUSDT will be transferred.
- `receiver`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero.
    - **Impact**: Receives the requested share amount.

Branches and code coverage

Intended branches

- Shares are minted 1:1 with the deposited FRXUSDT.
    - ☑ Test coverage

Negative behavior

- Calls outside Deposit mode and window revert.
    - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(this.asset()), caller, address(this), assets)`
    - **What is controllable?** The caller controls the number of shares to mint (`shares`) and the receiver (`receiver`).
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `modifyPendingDepositWindow(uint64 newStart, uint64 newEnd)`

This function allows administrators to reschedule a deposit window that has not yet begun — only callable by `ADMIN_ROLE` holders.

Inputs

- `newStart`

- **Control**: N/A.
- **Constraints**: Must be in the future, nonzero, and strictly before `newEnd`.
- **Impact**: Updates `_depositStart`, shifting when users can begin contributing capital.

- `newEnd`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and after `newStart`, and the window must still be pending.
  - **Impact**: Updates `_depositEnd`, altering the deposit-window duration.

Branches and code coverage

Intended branches

- Pending deposit window is modified with the new start and end timestamps.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the deposit window is already live, preventing midstream changes.
  - ☑ Negative test
- Revert if the timestamps are invalid (zero values, start >= end, or start <= `block.timestamp`).
  - ☑ Negative test

Function call analysis

- None.

Function: `modifyStartedDepositWindow(uint64 newEnd)`

This function lets administrators extend (but not reopen) an already started deposit window by updating its end timestamp — only callable by `ADMIN_ROLE` holders.

Inputs

- `newEnd`

- **Control**: N/A.
- **Constraints**: Must be strictly greater than the current `block.timestamp`.
- **Impact**: Adjusts `_depositEnd` while preserving the existing start time, which has already passed.

Branches and code coverage

Intended branches

- Update the `_depositEnd` with the new end timestamp.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the window has not started yet.
  - ☑ Negative test
- Revert if the new end timestamp is in the past.
  - ☐ Negative test

Function call analysis

- None.

Function: `recoverErc20(address token, address to)`

This function sweeps excess ERC-20 balances — only callable by `ADMIN_ROLE` holders.

Inputs

- `token`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Transfer to `to` to rescue the funds from the contract.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the recoverable ERC-20 amount.

Branches and code coverage

Intended branches

- If FRXUSD, only the surplus (balance minus protected amount) is eligible for transfer.
  - ☑ Test coverage
- Revert if the balance is less than the protected amount.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☑ Negative test

Function call analysis

- `IERC20(token).balanceOf(address(this))`
  - **What is controllable?** The token contract is chosen by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20(token).safeTransfer(to, recoverable)`
  - **What is controllable?** Admin controls `to` and the specific `token` address — `recoverable` is derived from on-chain balances and protected thresholds.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `recoverEth(address to)`

This function enables admins to sweep accidental ETH deposits — only callable by `ADMIN_ROLE` holders.

Inputs

- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.

- **Impact**: Receives the full ETH balance from the vault in a single transfer.

Branches and code coverage

Intended branches

- Revert if the ETH balance is zero.
  - ☑ Test coverage
- Forward all ETH to the recipient.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☑ Negative test
- Revert if the recipient is the zero address.
  - ☑ Negative test

Function call analysis

- `payable(to).call{ value: recoverable }("")`
  - **What is controllable?** Admin specifies `to`, while `recoverable` equals the contract's entire ETH balance at call time.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** `success` is checked, and if false, the transaction reverts.

Function: `redeemBridge(uint256 shares, address receiver, address owner)`

This is an alternative function only callable by the bridge contract to redeem shares from users withdrawing funds from the bridge itself.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the owner's redeemable balance (`maxRedeem`).
  - **Impact**: Delegates to `redeem` to process the redemption.

- `receiver`
    - **Control**: N/A.
    - **Constraints**: N/A.
    - **Impact**: Receives the underlying after redemption.
- `owner`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero.
    - **Impact**: Shares burned and USDT paid to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
    - ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.
    - ☑ Negative test

Function call analysis

- `this.redeem(shares, receiver, owner)`
    - **What is controllable?** The bridge controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeemRecovery(uint256 shares, address receiver, address owner)`

This function is a permissionless redemption path available in the Recovery phase so users can exit without the bridge once the timelock expires (180 days).

Inputs

- `shares`

- **Control**: N/A.
- **Constraints**: Must be greater than zero and limited by the owner's balance/allowance (`maxRedeem`).
- **Impact**: Determines the USDT withdrawn from the vault.

- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the redeemed USDT.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Experiences a share burn equal to the redemption.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside Recovery mode revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** The caller controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeem(uint256 shares, address receiver, address owner)`

This function is a bridge-driven share burn that withdraws the proportional USDT amount for a given share quantity — only in the Withdraw phase.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the owner's redeemable balance (`maxRedeem`).
  - **Impact**: Determines the share amount burned.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the redeemed USDT.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Has shares burned in exchange for USDT delivered to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside the Withdraw mode revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** The bridge controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setBridgeContract(address newBridgeContract)`

This function registers or updates the bridge contract permitted to initiate withdrawals and redemptions on behalf of users — only callable by `ADMIN_ROLE` holders.

Inputs

- `newBridgeContract`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Updates the `_bridgeContract` state variable.

Branches and code coverage

Intended branches

- Update the `_bridgeContract` with the new bridge contract address.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Withdraw mode revert.
  - ☑ Negative test
- Revert if the new bridge contract address is the zero address.
  - ☑ Negative test

Function call analysis

- None.

Function: `setDepositCap(uint256 newCap)`

This function updates the maximum total principal the vault will accept during the Deposit phase — only callable by `ADMIN_ROLE` holders.

Inputs

- `newCap`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and at least `totalSupply()`.

- **Impact**: Defines how much additional liquidity (if any) can be accepted before the cap is hit.

Branches and code coverage

Intended branches

- Update the `_depositCap` with the new cap.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the new cap is zero or less than the current supply.
  - ☑ Negative test

Function call analysis

- None.

Function: `transitionToRecoveryMode()`

This function is a permissionless emergency lever that moves the vault into its terminal Recovery phase after the configured timelock (180 days).

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Recovery`.
  - ☑ Test coverage

Negative behavior

- Revert if the current timestamp is before the recovery timestamp.
  - ☑ Negative test

Function call analysis

- None.

Function: `transitionToWithdrawMode()`

This function is an admin-controlled state change that closes the Yield phase and enables the Withdraw phase.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Withdraw`.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Yield mode revert.
  - ☑ Negative test

Function call analysis

- None.

Function: `transitionToYieldMode()`

This function is an admin-controlled state change that closes the Deposit phase and enables the Yield phase. It requires the configured deposit window to have ended.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Yield`.
  - ☑ Test coverage

Negative behavior

- Non-admin callers, or calls outside Deposit mode or after the deposit window ends, revert.
  - ☑ Negative test

Function call analysis

- None.

Function: `withdraw(uint256 assets, address receiver, address owner)`

This is a bridge-only function that burns shares from `owner` and transfers FRXUSDT to `receiver` — only in the Withdraw phase.

Inputs

- `assets`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero and cannot exceed the owner's withdrawable balance (`maxWithdraw`).
  - **Impact**: Determines how many shares are burned and how much FRXUSDT is transferred.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the withdrawn principal.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Loses the corresponding shares while FRXUSDT is delivered to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and FRXUSDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** The bridge controls the number of FRXUSDT to withdraw (`assets`) and the receiver (`receiver`).

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.3. Module: HourglassStableVaultKYC.sol

Function: `batchSetKycStatus(address[] users, bool status)`

This function marks up to 100 users as KYC-approved (or revokes approval) in a single transaction, adjusting pool accounting as each user flips status. It is only callable by `ADMIN_ROLE` holders in the KYC phase.

Inputs

- `users`
  - **Control**: N/A.
  - **Constraints**: Array must be nonempty, length <= `MAX_KYC_BATCH_SIZE`, and should contain no zero addresses.
  - **Impact**: Each user's shares move between the non-KYC and KYC accounting pools via `_setKycStatusWithAccounting`.
- `status`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Determines the new status of the batch of users.

Branches and code coverage

Intended branches

- Update the `_kycStatus` with the new status.
  - ☑ Test coverage
- Reverts on empty batches or if the size exceeds `MAX_KYC_BATCH_SIZE`.
  - ☑ Test coverage
- Zero addresses inside the batch revert with `ZeroAddress`.
  - ☐ Test coverage

Negative behavior

- Non-admin callers revert.

☑ Negative test

Function call analysis

- None.

Function: `deposit(uint256 assets, address receiver)`

This function is the primary entry point for users to supply USDC during the deposit window. It mints non-KYC shares 1:1 and increments the non-KYC accounting pool so users can later opt into KYC or redeeming without approval.

Inputs

- `assets`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the remaining cap (`maxDeposit`).
  - **Impact**: Increases `sharesNonKyc`, mints the same number of shares, and transfers USDC from the caller.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: The receiver gains newly minted non-KYC shares.

Branches and code coverage

Intended branches

- Shares are minted 1:1 with the deposited USDC, and `sharesNonKyc` is incremented by the deposited amount.

  ☑ Test coverage
- Cap enforcement via `maxDeposit` ensures deposits cannot exceed the configured limit.

  ☑ Test coverage

Negative behavior

- Calls outside Deposit phase and window revert.

  ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(HourglassStableVaultKYC.USDC, msg.sender, address(this), assets)`
    - **What is controllable?** The caller controls the number of USDC to deposit (`assets`) and the receiver (`receiver`).
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `modifyPendingDepositWindow(uint64 newStart, uint64 newEnd)`

This function allows administrators to reschedule a deposit window that has not yet begun — only callable by `ADMIN_ROLE` holders.

Inputs

- `newStart`
    - **Control**: N/A.
    - **Constraints**: Must be in the future, nonzero, and strictly before `newEnd`.
    - **Impact**: Updates `_depositStart`, shifting when users can begin contributing capital.
- `newEnd`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero and after `newStart`, and the window must still be pending.
    - **Impact**: Updates `_depositEnd`, altering the deposit-window duration.

Branches and code coverage

Intended branches

- Pending deposit window is modified with the new start and end timestamps.
    - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
    - ☑ Negative test
- Revert if the deposit window is already live, preventing midstream changes.

☑ Negative test

- Revert if the timestamps are invalid (zero values, start >= end, or start <= `block.timestamp`).

☑ Negative test

Function call analysis

- None.

Function: `modifyStartedDepositWindow(uint64 newEnd)`

This function lets administrators extend (but not reopen) an already started deposit window by updating its end timestamp — only callable by `ADMIN_ROLE` holders.

Inputs

- `newEnd`
  - **Control**: N/A.
  - **Constraints**: Must be strictly greater than the current `block.timestamp`.
  - **Impact**: Adjusts `_depositEnd` while preserving the existing start time, which has already passed.

Branches and code coverage

Intended branches

- Update the `_depositEnd` with the new end timestamp.

☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.

☑ Negative test

- Revert if the window has not started yet.

☑ Negative test

- Revert if the new end timestamp is in the past.

☐ Negative test

Function call analysis

- None.

Function: `previewRedeem(address user, uint256 shares)`

This function is a view helper returning the amounts a user would receive for burning `shares`; non-KYC users receive USDC 1:1, while KYC users receive pro-rata USDT and undeployed USDC based on the KYC pool balances.

Inputs

- `user`
  - **Control**: N/A.
  - **Constraints**: Reverts if `shares` exceed the user's balance.
  - **Impact**: Determines whether the KYC or non-KYC branch is used.
- `shares`
  - **Control**: N/A.
  - **Constraints**: Must not exceed the user's balance.
  - **Impact**: Directly drives the computed USDC/USDT amounts.

Branches and code coverage

Intended branches

- If the user is KYC-approved, returns `(pro-rata USDT, pro-rata undeployed USDC, true)`.
  - ☑ Test coverage
- Otherwise returns `(0, shares, false)` for the non-KYC refund path.
  - ☐ Test coverage

Negative behavior

- Reverts when the user lacks enough shares.
  - ☐ Negative test

Function call analysis

- None.

Function: `recoverErc20(address token, address to)`

This function sweeps excess ERC-20 balances — only callable by `ADMIN_ROLE` holders.

Inputs

- `token`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Transfer to `to` to rescue the funds from the contract.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the recoverable ERC-20 amount.

Branches and code coverage

Intended branches

- If USDC or USDT, only the surplus (balance minus protected amount) is eligible for transfer.
  - ☑ Test coverage
- Revert if the balance is less than the protected amount.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☑ Negative test

Function call analysis

- `IERC20(token).balanceOf(address(this))`
  - **What is controllable?** The token contract is chosen by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20(token).safeTransfer(to, recoverable)`
  - **What is controllable?** Admin controls `to` and the specific `token` address —

`recoverable` is derived from on-chain balances and protected thresholds.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `recoverEth(address to)`

This function enables admins to sweep accidental ETH deposits — only callable by `ADMIN_ROLE` holders.

Inputs

- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the full ETH balance from the vault in a single transfer.

Branches and code coverage

Intended branches

- Revert if the ETH balance is zero.
  - ☑ Test coverage
- Forward all ETH to the recipient.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☑ Negative test
- Revert if the recipient is the zero address.
  - ☑ Negative test

Function call analysis

- `payable(to).call{ value: recoverable }("")`
  - **What is controllable?** Admin specifies `to`, while `recoverable` equals the contract's entire ETH balance at call time.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** `success` is checked, and if false, the transaction reverts.

Function: `redeemBridge(uint256 shares, address receiver, address owner)`

This is an alternative function only callable by the bridge contract to redeem shares from users withdrawing funds from the bridge itself.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the owner's redeemable balance (`maxRedeem`).
  - **Impact**: Delegates to `redeem` to process the redemption.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the underlying after redemption.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Shares are burned, and USDT is paid to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.
  - ☑ Negative test

Function call analysis

- `this.redeem(shares, receiver, owner)`
  - **What is controllable?** The bridge controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeemKyc(uint256 shares, address receiver, address owner)`

This function is a bridge-only redemption path during the Withdraw phase that burns KYC-approved shares and sends pro-rata USDT to the receiver based on the vault's available USDT liquidity. It is only callable by the bridge contract during the Withdraw phase.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero, owned/approved by `owner`, and only available in the Withdraw phase.
  - **Impact**: Reduces `sharesKyc`, burns the shares, and releases USDT equal to the user's pro-rata portion.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the computed USDT amount.
- `owner`
  - **Control**: KYC-approved account whose shares are burned.
  - **Constraints**: `onlyKycApproved` enforces status and `onlyCallerIsBridge` ensures the bridge triggers redemption.
  - **Impact**: Loses shares while their claim in USDT diminishes proportionally.

Branches and code coverage

Intended branches

- Burn shares and transfer USDT to the receiver (bridge).
  - ☑ Test coverage

- The caller is the bridge contract and calls inside the Withdraw phase.
  - ☐ Test coverage

Negative behavior

- The caller is not the bridge contract, and calls outside the Withdraw phase revert.
  - ☐ Negative test

Function call analysis

- `this.previewRedeem(owner, shares)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20.safeTransfer(HourglassStableVaultKYC.USDT, receiver, usdtOut)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeemNonKyc(uint256 shares, address receiver, address owner)`

This function lets non-KYC shareholders redeem at a constant 1:1 USDC rate regardless of vault phase, guaranteeing principal preservation for users who never pass KYC.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and owned or approved by `owner`.
  - **Impact**: Burns the shares, decrements `sharesNonKyc`, and transfers equal USDC.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives USDC equal to the redeemed shares.

- `owner`
    - **Control**: N/A.
    - **Constraints**: Must be flagged as non-KYC via `onlyNonKyc`.
    - **Impact**: Share balance decreases by `shares`.

Branches and code coverage

Intended branches

- Callable in every mode because non-KYC funds must always be liquid.
    - ☑ Test coverage
- Spends allowance when `msg.sender != owner`.
    - ☑ Test coverage
- Updates accounting before transferring USDC.
    - ☑ Test coverage

Function call analysis

- `SafeERC20.safeTransfer(HourglassStableVaultKYC.USDC, receiver, shares)`
    - **What is controllable?** The caller controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeemRecoveryKyc(uint256 shares, address receiver, address owner)`

This function is emergency redemption for KYC users during the Recovery phase — burns shares and returns both pro-rata USDT and any remaining undeployed USDC.

Inputs

- `shares`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero and owned/approved.

    - **Impact**: Burns the shares, reduces `sharesKyc`, and may decrease `usdcKycDeployable` if undeployed USDC is returned.
- `receiver`
    - **Control**: N/A.
    - **Constraints**: N/A.
    - **Impact**: Obtains both asset types calculated by `previewRedeem`.
- `owner`
    - **Control**: N/A.
    - **Constraints**: Must be marked KYC via `onlyKycApproved`.
    - **Impact**: Share balance drops.

Branches and code coverage

Intended branches

- Burn shares and return both USDT and undeployed USDC.
    - ☑ Test coverage
- The caller is not the owner, and calls outside the Recovery phase revert.
    - ☐ Test coverage

Negative behavior

- The caller is the owner and calls inside the Recovery phase.
    - ☐ Negative test

Function call analysis

- `this.previewRedeem(owner, shares)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20.safeTransfer(HourglassStableVaultKYC.USDT, receiver, usdtOut)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `SafeERC20.safeTransfer(HourglassStableVaultKYC.USDC, receiver, usdcOut)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setBridgeContract(address newBridgeContract)`

This function registers or updates the bridge contract permitted to initiate withdrawals and redemptions on behalf of users — only callable by `ADMIN_ROLE` holders.

Inputs

- `newBridgeContract`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Updates the `_bridgeContract` state variable.

Branches and code coverage

Intended branches

- Update the `_bridgeContract` with the new bridge contract address.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Withdraw mode revert.
  - ☑ Negative test
- Revert if the new bridge contract address is the zero address.
  - ☑ Negative test

Function call analysis

- None.

Function: `setDepositCap(uint256 newCap)`

This function updates the maximum total principal the vault will accept during the Deposit phase — only callable by `ADMIN_ROLE` holders.

Inputs

- `newCap`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and at least `totalSupply()`.
  - **Impact**: Defines how much additional liquidity (if any) can be accepted before the cap is hit.

Branches and code coverage

Intended branches

- Update the `_depositCap` with the new cap.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the new cap is zero or less than the current supply.
  - ☑ Negative test

Function call analysis

- None.

Function: `setTreasuryAddress(address newTreasury)`

This function updates the destination address for treasury operations — only callable by `ADMIN_ROLE` holders.

Inputs

- `newTreasury`
  - **Control**: N/A.

- **Constraints**: Must be nonzero.
- **Impact**: Updates the `_treasuryAddress` state variable.

Branches and code coverage

Intended branches

- Update the `_treasuryAddress` with the new treasury address.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☐ Negative test

Function call analysis

- None.

Function: `transferToTreasury(uint256 amount)`

This function is a treasury role hook used during Yield mode to deploy KYC-approved USDC to the designated treasury address, decreasing the deployable accounting bucket. It is only callable by `TREASURY_ROLE` holders within the Yield phase.

Inputs

- `amount`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero, <= `usdcKycDeployable`, and <= on-contract USDC balance.
  - **Impact**: Reduces `usdcKycDeployable` and transfers the same amount of USDC to `_treasuryAddress`.

Branches and code coverage

Intended branches

- Revert if the amount is greater than `usdcKycDeployable`.
  - ☑ Test coverage
- Revert if the amount is greater than the contract's USDC balance.

☑ Test coverage

- Transfer the amount of USDC to `_treasuryAddress`.

☑ Test coverage

Negative behavior

- The caller is not the `TREASURY_ROLE` holder, and calls outside the Yield phase revert.

☑ Negative test

Function call analysis

- `HourglassStableVaultKYC.USDC.balanceOf(address(this))`
  - **What is controllable?** USDC address is constant.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20.safeTransfer(HourglassStableVaultKYC.USDC, _treasuryAddress, amount)`
  - **What is controllable?** USDC address is constant, and `_treasuryAddress` is set by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `transitionToKycMode()`

This function is an admin-controlled state change that closes the Deposit phase and enables the KYC phase.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Kyc`.

☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Unwind mode revert.

☑ Negative test

Function call analysis

- None.

Function: `transitionToRecoveryMode()`

This function is a permissionless emergency lever that moves the vault into its terminal Recovery phase after the configured timelock (180 days).

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Recovery`.

☑ Test coverage

Negative behavior

- Revert if the current timestamp is before the recovery timestamp.

☑ Negative test

Function call analysis

- None.

Function: `transitionToWithdrawMode()`

This function is an admin-controlled state change that closes the Yield phase and enables the Withdraw phase. It requires that at least the deposited amount has been redeemed from SparkLend.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Withdraw`.

☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Yield mode revert.

  ☑ Negative test

Function call analysis

- None.

Function: `transitionToYieldMode()`

This function is an admin-controlled state change that closes the KYC phase and enables the Yield phase.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Yield`.

  ☑ Test coverage

Negative behavior

- Non-admin callers, or calls outside the KYC phase or after the deposit window ends, revert.

  ☑ Negative test

Function call analysis

- None.

5.4. Module: HourglassStableVault.sol

Function: `deployToSpark(uint256 deployAmt)`

This function moves undeployed USDT owned by the vault into the SparkLend market during the Yield phase. The admin can deploy a specific amount or pass `0` to sweep the entire idle balance, after which the pool pulls the funds using the preset allowance.

Inputs

- `deployAmt`

- **Control**: N/A.
- **Constraints**: Must be 0 or less than the outstanding deployable balance (`totalSupply - usdtDeployed`).
- **Impact**: Amount of USDT to deploy to SparkLend.

Branches and code coverage

Intended branches

- Reverts if all shares are already accounted for in `usdtDeployed`.
  - ☑ Test coverage
- Passing `deployAmt == 0` sweeps the entire deployable amount into Spark.
  - ☑ Test coverage
- Reverts if providing an explicit amount larger than `deployable`.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside the Yield phase revert.
  - ☑ Negative test
- Reentrancy is blocked by `nonReentrant`.
  - ☐ Negative test

Function call analysis

- `SPARK_LEND_USDT_POOL.supply(asset: address(USDT), amount: deployAmt, onBehalfOf: address(this), referralCode: SPARK_REFERRAL_CODE)`
  - **What is controllable?** The admin influences `deployAmt` — asset, receiver, and referral code are fixed constants determined at deployment.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The pool could theoretically call back via ERC-20 hooks, but `ReentrancyGuard` blocks reentry into vault functions.

Function: `deposit(uint256 assets, address receiver)`

This function is the primary entry point for users to supply USDT during the deposit window. It mints shares 1:1 after transferring the tokens into the vault.

Inputs

- `assets`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero and within the remaining cap (`maxDeposit`).
  - **Impact**: Determines the number of shares minted (1:1) and increases the total supply tracked by the vault.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives freshly minted shares.

Branches and code coverage

Intended branches

- Shares are minted 1:1 with the deposited USDT.
  - ☑ Test coverage

Negative behavior

- Calls outside the Deposit phase and window revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(this.asset()), caller, address(this), assets)`
  - **What is controllable?** The caller controls the number of USDT to deposit (`assets`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `mint(uint256 shares, address receiver)`

This function is an alternative deposit path that targets a desired share amount during the deposit window.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero and within the remaining cap (`maxDeposit`).
  - **Impact**: Determines how much USDT will be transferred.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the requested share amount.

Branches and code coverage

Intended branches

- Shares are minted 1:1 with the deposited USDT.
  - ☑ Test coverage

Negative behavior

- Calls outside the Deposit phase and window revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(this.asset()), caller, address(this), assets)`
  - **What is controllable?** The caller controls the number of shares to mint (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `modifyPendingDepositWindow(uint64 newStart, uint64 newEnd)`

This function allows administrators to reschedule a deposit window that has not yet begun — only callable by `ADMIN_ROLE` holders.

Inputs

- `newStart`
  - **Control**: N/A.
  - **Constraints**: Must be in the future, nonzero, and strictly before `newEnd`.
  - **Impact**: Updates `_depositStart`, shifting when users can begin contributing capital.
- `newEnd`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and after `newStart`, and the window must still be pending.
  - **Impact**: Updates `_depositEnd`, altering the deposit-window duration.

Branches and code coverage

Intended branches

- Pending deposit window is modified with the new start and end timestamps.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the deposit window is already live, preventing midstream changes.
  - ☑ Negative test
- Revert if the timestamps are invalid (zero values, start >= end, or start <= `block.timestamp`).
  - ☑ Negative test

Function call analysis

- None.

Function: `modifyStartedDepositWindow(uint64 newEnd)`

This function lets administrators extend (but not reopen) an already started deposit window by updating its end timestamp — only callable by `ADMIN_ROLE` holders.

Inputs

- `newEnd`
  - **Control**: N/A.
  - **Constraints**: Must be strictly greater than the current `block.timestamp`.
  - **Impact**: Adjusts `_depositEnd` while preserving the existing start time, which has already passed.

Branches and code coverage

Intended branches

- Update the `_depositEnd` with the new end timestamp.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.
  - ☑ Negative test
- Revert if the window has not started yet.
  - ☑ Negative test
- Revert if the new end timestamp is in the past.
  - ☐ Negative test

Function call analysis

- None.

Function: `previewRedeemBridge(uint256 shares)`

This function exposes the same preview logic used by `redeem`, so bridge integrations can quote redemption outcomes.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Returns the expected USDT output if the redemption were executed now.

Branches and code coverage

Intended branches

- Return the expected USDT output if the redemption were executed now.
  - ☑ Test coverage

Function call analysis

- None.

Function: `recoverErc20(address token, address to)`

This function sweeps excess ERC-20 balances — only callable by `ADMIN_ROLE` holders.

Inputs

- `token`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Transfer to `to` to rescue the funds from the contract.
- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the recoverable ERC-20 amount.

Branches and code coverage

Intended branches

- If USDT or SPUSDT, only the surplus (balance minus protected amount) is eligible for transfer.
  - ☑ Test coverage
- Revert if the balance is less than the protected amount.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.
  - ☑ Negative test

Function call analysis

- `IERC20(token).balanceOf(address(this))`
  - **What is controllable?** Token contract is chosen by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20(token).safeTransfer(to, recoverable)`
  - **What is controllable?** Admin controls `to` and the specific `token` address — `recoverable` is derived from on-chain balances and protected thresholds.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `recoverEth(address to)`

This function enables admins to sweep accidental ETH deposits — only callable by `ADMIN_ROLE` holders.

Inputs

- `to`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives the full ETH balance from the vault in a single transfer.

Branches and code coverage

Intended branches

- Revert if the ETH balance is zero.
  - ☑ Test coverage
- Forward all ETH to the recipient.
  - ☑ Test coverage

Negative behavior

- Non-admin callers revert.

- ☑ Negative test
- Revert if the recipient is the zero address.
  - ☑ Negative test

Function call analysis

- `payable(to).call{ value: recoverable }("")`
  - **What is controllable?** Admin specifies `to`, while `recoverable` equals the contract's entire ETH balance at call time.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** `success` is checked, and if false, the transaction reverts.

Function: `redeemBridge(uint256 shares, address receiver, address owner)`

This is an alternative function only callable by the bridge contract to redeem shares from users withdrawing funds from the bridge itself.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the owner's redeemable balance (`maxRedeem`).
  - **Impact**: Delegates to `redeem` to process the redemption.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the underlying after redemption.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Shares are burned, and USDT is paid to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.

  ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.

  ☑ Negative test

Function call analysis

- `this.redeem(shares, receiver, owner)`
  - **What is controllable?** The bridge controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeemRecovery(uint256 shares, address receiver, address owner)`

This function is a permissionless redemption path available in the Recovery phase so users can exit without the bridge once the timelock expires (180 days).

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero and limited by the owner's balance/allowance (`maxRedeem`).
  - **Impact**: Determines the USDT withdrawn from the vault.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the redeemed USDT.
- `owner`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Experiences a share burn equal to the redemption.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside Recovery mode revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** The caller controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `redeem(uint256 shares, address receiver, address owner)`

This function is a bridge-driven share burn that withdraws the proportional USDT amount for a given share quantity — only in the Withdraw phase.

Inputs

- `shares`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and within the owner's redeemable balance (`maxRedeem`).
  - **Impact**: Determines the share amount burned.
- `receiver`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: Receives the redeemed USDT.
- `owner`
  - **Control**: N/A.

- **Constraints**: Must be nonzero.
- **Impact**: Has shares burned in exchange for USDT delivered to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
  - ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
  - **What is controllable?** The bridge controls the number of shares to redeem (`shares`) and the receiver (`receiver`).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setBridgeContract(address newBridgeContract)`

This function registers or updates the bridge contract permitted to initiate withdrawals and redemptions on behalf of users — only callable by `ADMIN_ROLE` holders.

Inputs

- `newBridgeContract`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero.
  - **Impact**: Updates the `_bridgeContract` state variable.

Branches and code coverage

Intended branches

- Update the `_bridgeContract` with the new bridge contract address.

  ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Withdraw mode revert.

  ☑ Negative test
- Revert if the new bridge contract address is the zero address.

  ☑ Negative test

Function call analysis

- None.

Function: `setDepositCap(uint256 newCap)`

This function updates the maximum total principal the vault will accept during the Deposit phase — only callable by `ADMIN_ROLE` holders.

Inputs

- `newCap`
  - **Control**: N/A.
  - **Constraints**: Must be nonzero and at least `totalSupply()`.
  - **Impact**: Defines how much additional liquidity (if any) can be accepted before the cap is hit.

Branches and code coverage

Intended branches

- Update the `_depositCap` with the new cap.

  ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Deposit mode revert.

  ☑ Negative test
- Revert if the new cap is zero or less than the current supply.

  ☑ Negative test

Function call analysis

- None.

Function: `totalAssets()`

This function extends the ERC-4626 view by counting deployed USDT while the vault sits in Yield or Unwind phases, giving depositors visibility into accrued yield before withdrawals reopen.

Branches and code coverage

Intended branches

- Always include `super.totalAssets()`, which captures on-contract USDT.
  - ☑ Test coverage
- When `_mode` is Yield or Unwind, add `SPUSDT.balanceOf(address(this))` to represent deployed USDT.
  - ☑ Test coverage

Function call analysis

- `IERC20(address(SPUSDT)).balanceOf(address(this))`
  - **What is controllable?** SPUSDT is fixed during construction — the balance reflects Spark's accounting for supplied USDT.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `transitionToRecoveryMode()`

This function is a permissionless emergency lever that moves the vault into its terminal Recovery phase after the configured timelock (180 days).

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Recovery`.
  - ☑ Test coverage

Negative behavior

- Revert if the current timestamp is before the recovery timestamp.
  - ☑ Negative test

Function call analysis

- None.

Function: `transitionToUnwindMode()`

This function is an admin-controlled state change that closes the Yield phase and enables the Unwind phase.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Unwind`.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside the Yield phase revert.
  - ☑ Negative test

Function call analysis

- None.

Function: `transitionToWithdrawMode()`

This function is an admin-controlled state change that closes the Unwind phase and enables the Withdraw phase. It requires that at least the deposited amount has been redeemed from SparkLend.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Withdraw`.

☑ Test coverage

Negative behavior

- Non-admin callers or calls outside the Unwind phase revert.
  ☑ Negative test
- Revert if the amount redeemed is less than the amount deployed.
  ☑ Negative test

Function call analysis

- None.

Function: `transitionToYieldMode()`

This function is an admin-controlled state change that closes the Deposit phase and enables the Yield phase. It requires the configured deposit window to have ended.

Inputs

None.

Branches and code coverage

Intended branches

- Update the `_mode` to `OperationalMode.Yield`.
  ☑ Test coverage

Negative behavior

- Non-admin callers, or calls outside the Deposit phase or after the deposit window ends, revert.
  ☑ Negative test

Function call analysis

- None.

Function: `withdrawFromSparkRecovery(uint256 amount)`

This function is a permissionless variant of `withdrawFromSpark` used after Recovery mode activates.

Inputs

- `amount`
  - **Control**: N/A.
  - **Constraints**: `0` sweeps all; otherwise, the value must not exceed the contract's aToken balance.
  - **Impact**: Updates `usdtRedeemed` and instructs the Spark pool to return liquidity to the vault contract.

Branches and code coverage

Intended branches

- Delegate to `_withdrawFromSpark` for the actual withdrawal logic.
  - ☑ Test coverage

Negative behavior

- Reentrancy is blocked by `nonReentrant`.
  - ☐ Negative test
- Calls outside Recovery mode revert.
  - ☑ Negative test

Function call analysis

- None.

Function: `withdrawFromSpark(uint256 amount)`

This function is an admin-only exit path used in Unwind mode to redeem USDT from SparkLend. The caller can withdraw a specific amount or all the available balance if `amount == 0`.

Inputs

- `amount`

- **Control**: N/A.
- **Constraints**: Accepts `0` for a full sweep or any value up to the on-contract aToken balance.
- **Impact**: Updates `usdtRedeemed` and instructs the Spark pool to return liquidity to the vault contract.

Branches and code coverage

Intended branches

- Reverts if the contract holds no spUSDT.
  - ☑ Test coverage
- Reverts if the amount is greater than the contract's aToken balance (if not 0).
  - ☑ Test coverage
- Increments `usdtRedeemed` before withdrawing from SparkLend.
  - ☑ Test coverage

Negative behavior

- Non-admin callers or calls outside Unwind mode revert.
  - ☑ Negative test
- Reentrancy is blocked by `nonReentrant`.
  - ☐ Negative test

Function call analysis

- None.

Function: `withdraw(uint256 assets, address receiver, address owner)`

This is a bridge-only function that burns shares from `owner` and transfers USDT to `receiver` — only in the Withdraw phase.

Inputs

- `assets`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero and cannot exceed the owner's withdrawable balance (`maxWithdraw`).

    - **Impact**: Determines how many shares are burned and how much USDT is transferred.
- `receiver`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero.
    - **Impact**: Receives the withdrawn principal.
- `owner`
    - **Control**: N/A.
    - **Constraints**: Must be nonzero.
    - **Impact**: Loses the corresponding shares while USDT is delivered to `receiver`.

Branches and code coverage

Intended branches

- Shares are burned, and USDT is transferred.
    - ☑ Test coverage

Negative behavior

- Calls outside Withdraw mode revert.
    - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(this.asset()), receiver, assets)`
    - **What is controllable?** The bridge controls the number of USDT to withdraw (`assets`) and the receiver (`receiver`).
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

6. Assessment Results

During our assessment on the scoped Stable Predeposit contracts, we discovered one finding, which was of medium impact.

6.1. Disclaimer

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

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

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