Zellic

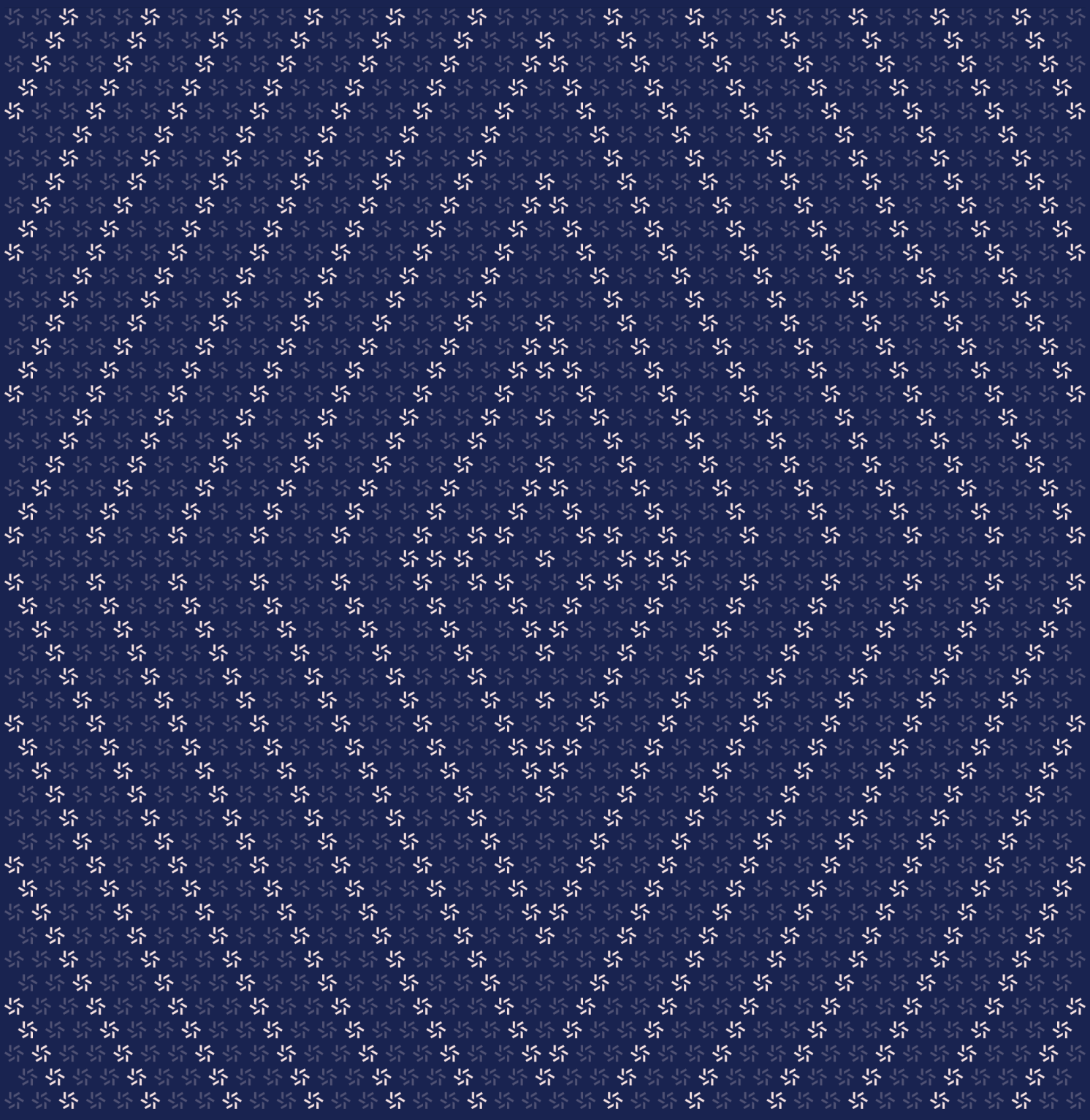
Prepared for
Ari Pingle
Theo

Prepared by
Filipe Alves
Jisub Kim
Zellic

April 16, 2026

ThUSD

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 Theo from April 14th to April 15th, 2026. During this engagement, Zellic reviewed ThUSD'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:

- Is the custom ERC-4626–based staking contract (sthUSD) secure?
- Are all critical functions protected by the appropriate roles and access controls?
- Is there any yield logic that could cause the vault to lose capital or distribute thUSD unfairly?
- Could an attacker withdraw another user's staked funds?
- Is the thUSD minting flow safe from manipulation or unauthorized minting?

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.

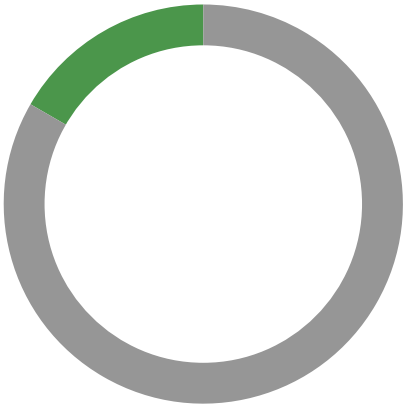
1.4. Results

During our assessment on the scoped ThUSD contracts, we discovered six findings. No critical issues were found. One finding was of low impact and the other findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	5



2. Introduction

2.1. About ThUSD

Theo contributed the following description of ThUSD:

> thUSD is an ERC-20 stablecoin. sthUSD is an asynchronous staking contract (based on ERC-4626) that allows users to stake thUSD and earn yield in thUSD. thUSDMinter is the contract that owns the minter role on thUSD. The general flow is: (1) a user calls the API to mint, approving USDC for thUSDMinter; (2) the backend processes the mint order and calls mint on thUSDMinter; (3) the user receives thUSD while USDC is sent to the backing wallet; (4) the user stakes thUSD into sthUSD; (5) the user can unstake sthUSD and claim thUSD after the unstaking period ends.

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:

ThUSD Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts-v2
Repository	https://github.com/theo-network/contracts-v2 ↗
Version	`ebce4e504538ce51629db7a078b9b06e4b0dcf7b`

Programs	`ThUSD.sol`
	`ThUSDMinter.sol`
	`SthUSD.sol`
	`SthUSDRewards.sol`

2.4. Project Overview

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

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 ↗

Jisub Kim
Engineer
jisub@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 14, 2026	Start of primary review period
April 15, 2026	End of primary review period

3. Detailed Findings

3.1. Repeated redemption requests reset lockup for all pending assets

Target	SthUSD		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `initiateRedeem()` function accumulates assets and shares into a single `RedeemRequestData` per address but overwrites `claimableTimestamp` on every call:

```
RedeemRequestData storage request = _redeemRequests[msg.sender];
request.assets += assets;
request.shares += shares_;
request.claimableTimestamp = block.timestamp + lockupPeriod;
```

If a user has a pending or already claimable request and calls `initiateRedeem()` again, all of their accumulated assets are locked again for the full lockup period, including amounts that were previously claimable.

This also affects third-party integrations. A wrapper contract calling `initiateRedeem()` from a single address on behalf of multiple users would reset the lockup for all pooled users every time any individual user initiates a new redemption.

Impact

Users who redeem more than once before claiming have all pending assets relocked. Wrapper contracts that pool multiple users behind a single address are especially affected, as any new redemption delays the entire pool.

Recommendations

Consider tracking redemption requests individually so that each request has its own lockup timer, or consider documenting this behavior for integrators.

Remediation

This issue has been acknowledged by Theo.

3.2. Signed order does not bind destination or redeem fee

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

Description

The `ORDER_TYPE` used for EIP-712 signing in ThUSDMinter does not include `mintDestination`, `redeemDestination`, or `redeemFeeBps`:

```
bytes32 private constant ORDER_TYPE =
    keccak256(
        "Order(uint8 order_type,uint256 expiry,uint256 nonce,address signer,"
        "address recipient,address collateral_asset,uint256
    collateral_amount,"
        "uint256 thusd_amount)"
    );
```

An admin can change these parameters between the time a user signs an order and the time it is executed, altering the collateral routing destination or the net payout on redemptions.

Impact

A user's signed order may be executed under different destination or fee conditions than those present at signing time. The practical risk is limited by the `MAX_FEE_BPS` cap (0.1% hard cap as of the time of writing) and by the trust assumption.

Recommendations

Consider including `mintDestination`, `redeemDestination`, and `redeemFeeBps` in the signed order type so that users can verify the exact execution parameters at signing time.

Remediation

This issue has been acknowledged by Theo.

3.3. No safeguard against fee-on-transfer collateral tokens

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

Description

During minting, the `_transferCollateralIn()` function transfers collateral from the signer to `mintDestination` using `safeTransferFrom` but does not verify the actual amount received:

```
function _transferCollateralIn(Order calldata order) internal {
    IERC20(order.collateral_asset).safeTransferFrom(
        order.signer, mintDestination, order.collateral_amount
    );
}
```

The contract then mints the full `thusd_amount` specified in the order regardless of how much collateral actually arrived. If a fee-on-transfer token were listed as a supported asset, the protocol would mint more thUSD than the collateral backing it.

Since supported collateral assets must be explicitly added by an admin via `addSupportedAsset()`, this is not a permissionless exploit path.

Impact

If an admin mistakenly lists a fee-on-transfer token as a supported collateral asset, the protocol would have no safeguard against undercollateralized minting.

Recommendations

Consider checking the balance of `mintDestination` before and after the transfer in `_transferCollateralIn()`, or consider documenting that fee-on-transfer tokens must not be added as supported assets.

Remediation

This issue has been acknowledged by Theo.

3.4. The `setYield()` function does not follow the checks-effects-interactions pattern

Target	SthUSD		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `setYield()` function performs an external token transfer before updating the contract's state variables:

```
function setYield(uint256 amount_) external whenNotPaused {
    // [...]

    // transfer yield to this contract
    IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount_);

    yieldAmount = amount_;
    lastYieldTimestamp = block.timestamp;

    emit YieldTransferredIn(amount_, vestingDuration);
}
```

The underlying asset is thUSD, a standard ERC-20 with no reentrant hooks, so this does not present an exploitable bug at the time of writing. However, ThUSD is an upgradable contract, and a future upgrade could introduce callback mechanisms that would allow reentering `setYield()` before `yieldAmount` and `lastYieldTimestamp` are updated.

Impact

There is no immediate impact given the current ThUSD implementation at the time of writing. If ThUSD were upgraded to include transfer hooks or callback mechanisms, the current control flow could allow reentrancy before the state is updated.

Recommendations

Consider moving the `safeTransferFrom` call after the state updates to follow the checks-effects-interactions pattern.

Remediation

This issue has been acknowledged by Theo.

3.5. Redundant zero-address check in `pause()`

Target	SthUSD		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `pause()` function in SthUSD includes a redundant check for `emergencyPauser == address(0)`:

```
function pause() external {
!     if (msg.sender != emergencyPauser || emergencyPauser == address(0))
          revert NotAuthorizedToPause();
      _pause();
}
```

Since `msg.sender` can never be `address(0)`, the condition `msg.sender != emergencyPauser` is already `true` whenever `emergencyPauser` is `address(0)`. The second disjunct is therefore unreachable as an independent branch.

Impact

There is no functional impact. The redundant check adds minor gas overhead and reduces code clarity.

Recommendations

Consider simplifying the guard.

Remediation

This issue has been acknowledged by Theo.

3.6. The `setRewardsBps()` function has no upper-bound check

Target	SthUSDRewards		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `setRewardsBps()` function accepts any `uint256` value with no maximum check:

```
function setRewardsBps(uint256 _rewardsBps) external onlyRole(BPS_SETTER_ROLE)
    {
    uint256 oldBps = rewardsBps;
    rewardsBps = _rewardsBps;
    emit RewardsBpsUpdated(oldBps, _rewardsBps);
}
```

A misconfigured or compromised `BPS_SETTER_ROLE` could set an arbitrarily high reward rate. The actual impact depends on how this value is consumed off chain.

Impact

Without an on-chain upper bound, there is no contract-level safeguard against accidental or malicious reward-rate misconfiguration.

Recommendations

Consider adding a reasonable maximum bound (e.g., `MAX_REWARDS_BPS`) and reverting if the provided value exceeds it.

Remediation

This issue has been acknowledged by Theo.

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. Block-based rate limits behave differently across chains

ThUSDMinter uses `block.number` to enforce `maxMintPerBlock` and `maxRedeemPerBlock`:

```
if (mintedPerBlock[block.number] + order.thusd_amount > maxMintPerBlock) {
    revert MaxMintPerBlockExceeded();
}
mintedPerBlock[block.number] += order.thusd_amount;
```

Block times vary significantly across chains — for example, between Ethereum (~12s), Base (~2s), and HyperEVM (~0.4s). The same `maxMintPerBlock` value results in vastly different effective throughput per unit of time depending on the deployment chain.

Since `maxMintPerBlock` and `maxRedeemPerBlock` are admin-configurable, this is an operational concern rather than a code defect. If thUSD is deployed on the same chains listed in the Theo deployment documentation, admins should tune these values per chain to achieve the intended rate limits.

4.2. Regarding an ERC-4626 inflation attack

SthUSD does not override `_decimalsOffset()` in the ERC-4626 vault, leaving it theoretically vulnerable to the classic ERC-4626 inflation attack as a griefing vector. In this attack, a first depositor can manipulate the share price by front-running deposits with a large direct transfer of the underlying asset, causing subsequent depositors to receive fewer shares than expected.

Theo has responded that the sthUSD vault is seeded at deployment with a nonnegligible amount of thUSD and that this initial deposit will never be redeemed.

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: SthUSD.sol (Staked thUSD Vault)

Function: `deposit(uint256 assets_, address receiver_)`

This function deposits thUSD into the vault and mints sthUSD shares to `receiver_` at the current exchange rate. It delegates to the standard ERC-4626 `deposit` implementation.

Inputs

- `assets_`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxDeposit(receiver_)`. The caller must have sufficient thUSD balance and allowance for `safeTransferFrom`.
  - **Impact**: The amount of thUSD transferred into the vault. Shares minted are determined by the current `totalAssets()` / `totalSupply()` ratio.
- `receiver_`
  - **Control**: Full control.
  - **Constraints**: Must not be `address(0)` (ERC-20 minting reverts).
  - **Impact**: The address that receives the newly minted sthUSD shares.

Branches and code coverage

Intended branches

- ☑ Should transfer `assets_` thUSD from `msg.sender` to the vault and mint corresponding shares to `receiver_`.
- ☑ First deposit should mint shares 1:1 with assets.
- ☑ The `totalAssets()` should increase by `assets_` after deposit.

Negative behavior

- ☑ Revert if the contract is paused.

Function: `initialize(IERC20 thUSD_, uint256 lockupPeriod_, uint256 vestingDuration_, address owner_)`

This function initializes the SthUSD upgradable ERC-4626 vault with the underlying thUSD asset, lockup period for async redeems, yield vesting duration, and owner. It can only be called once.

Inputs

- `thUSD_`
  - **Control**: Full control (by deployer).
  - **Constraints**: Must be a valid ERC-20 token address (OZ ERC-4626 does not validate this explicitly).
  - **Impact**: Sets the underlying asset of the ERC-4626 vault. It is immutable after initialization.
- `lockupPeriod_`
  - **Control**: Full control (by deployer).
  - **Constraints**: Must be less than or equal to `MAX_LOCKUP_PERIOD` (30 days).
  - **Impact**: Sets the initial lockup period for async redeem requests.
- `vestingDuration_`
  - **Control**: Full control (by deployer).
  - **Constraints**: Must be less than or equal to `MAX_VESTING_DURATION` (365 days).
  - **Impact**: Sets the initial duration over which yield is linearly vested.
- `owner_`
  - **Control**: Full control (by deployer).
  - **Constraints**: Must not be `address(0)` (OZ `__Ownable_init` reverts).
  - **Impact**: The address that becomes the initial owner with authority over admin functions and upgrades.

Branches and code coverage

Intended branches

- ☐ Should set token name to "Staked thUSD", symbol to "sthUSD", and decimals matching the underlying asset.
- ☐ Should set `lockupPeriod` and `vestingDuration` to the provided values.
- ☐ Should set owner to `owner_`.

Negative behavior

- ☐ Revert if called more than once.

- ☐ Revert with `MaxLockupPeriodExceeded` if `lockupPeriod_ > MAX_LOCKUP_PERIOD`.
- ☐ Revert with `MaxVestingDurationExceeded` if `vestingDuration_ > MAX_VESTING_DURATION`.
- ☐ Revert if `owner_` is `address(0)`.

Function: `initiateRedeem(uint256 shares_, address owner_)`

This function initiates an async redemption by burning sthUSD shares from `owner_` and creating a pending redeem request under `msg.sender` with a lockup period. The request accumulates into any existing pending request for `msg.sender`, resetting the lockup timer. Reverts if the contract is paused.

Inputs

- `shares_`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero, `convertToAssets(shares_)` must be greater than zero, and `owner_` must have sufficient sthUSD balance.
  - **Impact**: The number of sthUSD shares to burn. Converted to assets via `convertToAssets` at the current exchange rate and added to the pending request.
- `owner_`
  - **Control**: Full control.
  - **Constraints**: Must have sufficient sthUSD balance. If `msg.sender != owner_`, `msg.sender` must have sufficient sthUSD allowance from `owner_`.
  - **Impact**: The address whose shares are burned. The redeem request is stored under `msg.sender`, not `owner_`.

Branches and code coverage

Intended branches

- ☑ Should burn `shares_` from `owner_` (spending allowance if `msg.sender != owner_`).
- ☑ Should convert shares to assets via `convertToAssets` and add to `_redeemRequests[msg.sender]`.
- ☑ Should set `claimableTimestamp` to `block.timestamp + lockupPeriod`.
- ☑ When `totalSupply() == 0` after burning, should assign all remaining vault assets (including unvested yield) and reset yield state.
- ☑ The request is stored under `msg.sender`, not `owner_` — a third party with allowance receives the claim rights.

Negative behavior

- ☑ Revert with `ZeroAssetsOrShares` if `shares_ == 0` or converted assets equals zero.
- ☑ Revert if `msg.sender != owner_` and insufficient allowance.
- ☑ Revert if the contract is paused.

Function: `mint(uint256 shares_, address receiver_)`

This function mints an exact amount of sthUSD shares by pulling the required thUSD from the caller. It delegates to the standard ERC-4626 `mint` implementation.

Inputs

- `shares_`
  - **Control**: Full control.
  - **Constraints**: Must be less than or equal to `maxMint(receiver_)`. The caller must have sufficient thUSD balance and allowance for the computed asset amount.
  - **Impact**: The target number of sthUSD shares to mint. Required assets are determined by the current exchange rate with ceil rounding.
- `receiver_`
  - **Control**: Full control.
  - **Constraints**: Must not be `address(0)` (ERC-20 minting reverts).
  - **Impact**: The address that receives the newly minted sthUSD shares.

Branches and code coverage

Intended branches

- ☐ Should compute the required assets via `previewMint(shares_)`, transfer from `msg.sender`, and mint `shares_` sthUSD to `receiver_`.

Negative behavior

- ☑ Revert if the contract is paused.

Function: `pause()`

This function pauses the contract, disabling deposits, mints, withdrawals, redeems, yield distribution, and redeem initiation. It is only callable by the `emergencyPauser`.

Inputs

None.

Branches and code coverage

Intended branches

- ☑ Should block deposits when paused.
- ☑ Should block mints when paused.
- ☑ Should block `initiateRedeem` when paused.
- ☑ Should block withdrawals when paused.
- ☑ Should block redeems when paused.
- ☑ Should block `setYield` when paused.

Negative behavior

- ☑ Revert with `NotAuthorizedToPause` if `msg.sender` is not `emergencyPauser`.
- ☐ Revert if already paused.

Function: `redeem(uint256 shares_, address receiver_, address owner_)`

This function claims thUSD from a pending redeem request by specifying shares to consume, after the lockup period has elapsed. Only the request owner (`owner_ == msg.sender`) can claim. It transfers the proportional thUSD amount to `receiver_`.

Inputs

- `shares_`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero. Must be less than or equal to `request.shares`. Lockup must have elapsed.
  - **Impact**: The number of shares to consume from the pending request. If equal to `request.shares`, it is a full claim (all remaining assets transferred). Otherwise, assets are calculated proportionally with floor rounding.
- `receiver_`
  - **Control**: Full control.
  - **Constraints**: None explicitly (no zero-address check).
  - **Impact**: The address that receives the thUSD transfer.
- `owner_`
  - **Control**: Full control.
  - **Constraints**: Must have a pending redeem request with assets greater than

zero and elapsed lockup; `msg.sender` must equal `owner_`.
- **Impact**: The address whose redeem request is being claimed from.

Branches and code coverage

Intended branches

- ☑ Full claim (`shares_ == request.shares`) — should transfer all `request.assets` and zero-out the request.
- ☐ Partial claim — should transfer `floor(shares_ * request.assets / request.shares)` assets.
- ☑ After a full claim, the user should receive back the original deposit amount (1:1 without yield).

Negative behavior

- ☐ Revert with `ZeroAssetsOrShares` if `shares_ == 0`.
- ☐ Revert with `NoClaimableRequest` if `request.assets == 0`.
- ☐ Revert with `LockupNotElapsed` if `block.timestamp < request.claimableTimestamp`.
- ☐ Revert with `ExceedsClaimable` if `shares_ > request.shares`.
- ☑ Revert with `OwnerMustCompleteRedeem` if `msg.sender != owner_` (even with allowance).
- ☑ Revert if the contract is paused.

Function: `setEmergencyPauser(address newEmergencyPauser)`

This function updates the address authorized to pause the contract. It is only callable by the owner.

Inputs

- `newEmergencyPauser`
  - **Control**: Full control (by owner).
  - **Constraints**: None — `address(0)` is accepted, which would effectively disable emergency pausing.
  - **Impact**: Only this address can call `pause`. Note that the `pause` function also checks `emergencyPauser != address(0)` (which is redundant with `msg.sender != emergencyPauser` since `msg.sender` cannot be `address(0)`).

Branches and code coverage

Intended branches

- ☑ Should update `emergencyPauser` to the new address.

Negative behavior

- ☑ Revert if the caller is not the owner.

Function: `setLockupPeriod(uint256 lockupPeriod_)`

This function updates the lockup period for async redeem requests. It is only callable by the owner.

Inputs

- `lockupPeriod_`
  - **Control**: Full control (by owner).
  - **Constraints**: Must be less than or equal to `MAX_LOCKUP_PERIOD` (30 days). Can be set to zero.
  - **Impact**: All future `initiateRedeem` calls will use this lockup period. Does not affect existing pending requests.

Branches and code coverage

Intended branches

- ☐ Should update `lockupPeriod` to the new value.

Negative behavior

- ☑ Revert with `MaxLockupPeriodExceeded` if `lockupPeriod_ > MAX_LOCKUP_PERIOD`.
- ☐ Revert if the caller is not the owner.

Function: `setVestingDuration(uint256 vestingDuration_)`

This function updates the yield vesting duration. It is only callable by the owner when no yield is currently vesting. It resets `yieldAmount` and `lastYieldTimestamp` to zero.

Inputs

- `vestingDuration_`
  - **Control**: Full control (by owner).
  - **Constraints**: Must be less than or equal to `MAX_VESTING_DURATION` (365 days), and `getUnvestedAmount()` must be zero (no active vesting).
  - **Impact**: Sets the duration over which future yield distributions vest linearly. Resets yield state to prevent stale accounting.

Branches and code coverage

Intended branches

- ☑ Should reset `yieldAmount` and `lastYieldTimestamp` to zero.
- ☐ Should update `vestingDuration` to the new value.
- ☑ The `totalAssets()` should not change after updating vesting duration (no yield revival).

Negative behavior

- ☑ Revert with `MaxVestingDurationExceeded` if `vestingDuration_ > MAX_VESTING_DURATION`.
- ☑ Revert with `VestingInProgress` if `getUnvestedAmount() > 0`.
- ☐ Revert if the caller is not the owner.

Function: `setYield(uint256 amount_)`

This function distributes yield to the vault by transferring thUSD from the `yieldDistributor` and starting a new linear vesting period. It is only callable by the `yieldDistributor` when no yield is currently vesting.

Inputs

- `amount_`
  - **Control**: Full control (by yield distributor).
  - **Constraints**: Must be greater than zero, `vestingDuration` must be greater than zero, and `getUnvestedAmount()` must be zero. The caller must have sufficient thUSD balance and allowance for the transfer.
  - **Impact**: Transfers `amount_` thUSD into the vault. Sets `yieldAmount` to `amount_` and `lastYieldTimestamp` to `block.timestamp`. The yield vests linearly over `vestingDuration`, gradually increasing `totalAssets()`.

Branches and code coverage

Intended branches

- ☑ Should transfer `amount_` thUSD from `msg.sender` to the vault.
- ☑ Should set `yieldAmount` to `amount_` and `lastYieldTimestamp` to `block.timestamp`.
- ☑ After the call, `getUnvestedAmount()` should equal `amount_`.
- ☑ The `totalAssets()` should remain unchanged immediately after (yield is fully unvested).
- ☑ Yield should vest linearly over `vestingDuration`, increasing `totalAssets()` proportionally.

Negative behavior

- ☑ Revert with `InvalidYieldDistributor` if `msg.sender` is not `yieldDistributor`.
- ☑ Revert with `VestingDurationNotSet` if `vestingDuration == 0`.
- ☑ Revert with `VestingInProgress` if `getUnvestedAmount() > 0`.
- ☑ Revert with `ZeroAssetsOrShares` if `amount_ == 0`.
- ☑ Revert if the contract is paused.

Function: `setYieldDistributor(address newYieldDistributor_)`

This function updates the address authorized to call `setYield`. It is only callable by the owner.

Inputs

- `newYieldDistributor_`
  - **Control**: Full control (by owner).
  - **Constraints**: None — `address(0)` is accepted, which would effectively disable yield distribution.
  - **Impact**: Only this address can call `setYield` to distribute yield to the vault.

Branches and code coverage

Intended branches

- ☑ Should update `yieldDistributor` to the new address.

Negative behavior

- ☐ Revert if the caller is not the owner.

Function: `unpause()`

This function unpauses the contract. It is only callable by the owner.

Inputs

None.

Branches and code coverage

Intended branches

- ☑ Should set the contract to an unpaused state, resuming all operations.

Negative behavior

- ☑ Revert if the caller is not the owner.
- ☐ Revert if not currently paused.

Function: `withdraw(uint256 assets_, address receiver_, address owner_)`

This function claims thUSD assets from a pending redeem request after the lockup period has elapsed. Only the request owner (`owner_ == msg.sender`) can claim. It transfers thUSD to `receiver_`.

Inputs

- `assets_`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero. Must be less than or equal to `request.assets`. Lockup must have elapsed (`block.timestamp >= claimableTimestamp`).
  - **Impact**: The amount of thUSD to withdraw from the pending request. If equal to `request.assets`, it is a full claim (all remaining shares deducted). Otherwise, shares are deducted proportionally with ceil rounding.
- `receiver_`
  - **Control**: Full control.
  - **Constraints**: None explicitly (no zero-address check).
  - **Impact**: The address that receives the thUSD transfer.
- `owner_`
  - **Control**: Full control.
  - **Constraints**: Must have a pending redeem request with assets greater than zero and elapsed lockup; `msg.sender` must equal `owner_`.
  - **Impact**: The address whose redeem request is being claimed from.

Branches and code coverage

Intended branches

- ☑ Full claim (`assets_ == request.assets`) — should deduct all `request.shares` and transfer all `request.assets`.
- ☑ Partial claim — should deduct `ceil(assets_ * request.shares / request.assets)` shares and transfer `assets_`.
- ☐ If partial claim rounding consumes all shares, should include all remaining assets to avoid dust.

- ☑ After full claim, `maxWithdraw(owner_)` should return zero.

Negative behavior

- ☐ Revert with `ZeroAssetsOrShares` if `assets_ == 0`.
- ☐ Revert with `NoClaimableRequest` if `request.assets == 0`.
- ☑ Revert with `LockupNotElapsed` if `block.timestamp < request.claimableTimestamp`.
- ☑ Revert with `ExceedsClaimable` if `assets_ > request.assets`.
- ☑ Revert with `OwnerMustCompleteRedeem` if `msg.sender != owner_` (even with allowance).
- ☑ Revert if the contract is paused.

5.2. Module: SthUSDRewards.sol

Function: `setRewardsBps(uint256 _rewardsBps)`

This function updates the `rewardsBps` state variable. It is only callable by addresses with `BPS_SETTER_ROLE`.

Inputs

- `_rewardsBps`
  - **Control**: Full control (by `BPS_SETTER_ROLE` holder).
  - **Constraints**: None — no upper bound enforced. Can be set to any `uint256` value, including `0`.
  - **Impact**: Updates the stored rewards basis points value and emits `RewardsBpsUpdated`.

Branches and code coverage

Intended branches

- ☐ Should update `rewardsBps` to the new value.
- ☐ Should accept any `uint256` value (no upper bound enforced).

Negative behavior

- ☐ Revert if the caller does not have `BPS_SETTER_ROLE`.

5.3. Module: ThUSD.sol (ThUSD Token)

Function: `initialize(address _owner)`

This function initializes the ThUSD upgradable ERC-20 token, setting up the token name/symbol ("thUSD"), ERC20Burnable, ERC20Permit, Ownable2Step with the given owner, and UUPSUpgradeable. It can only be called once due to the `initializer` modifier.

Inputs

- `_owner`
  - **Control**: Full control (by deployer).
  - **Constraints**: Must not be the zero address (OZ `__Ownable_init` reverts on `address(0)`).
  - **Impact**: The address that becomes the initial owner, with authority to set the minter and authorize upgrades.

Branches and code coverage

Intended branches

- ☐ Should set the token name to "thUSD", symbol to "thUSD", and decimals to 6.
- ☐ Should set the owner to `_owner`.
- ☐ The ERC20Permit domain separator should be initialized with the name "thUSD".

Negative behavior

- ☐ Revert if called more than once (`Initializable: contract is already initialized`).
- ☐ Revert if `_owner` is `address(0)`.

Function: `mint(address to, uint256 amount)`

This function mints `amount` of thUSD tokens to the `to` address. It is only callable by the designated minter.

Inputs

- `to`
  - **Control**: Full control (by minter).
  - **Constraints**: Must not be the zero address (OZ ERC-20 `_mint` reverts on `address(0)`).
  - **Impact**: The address that receives the minted thUSD tokens.

- `amount`
    - **Control**: Full control (by minter).
    - **Constraints**: None explicitly — no supply cap enforced.
    - **Impact**: The amount of thUSD tokens minted. Increases `totalSupply` and `balanceOf(to)`.

Branches and code coverage

Intended branches

- ☑ Calling `mint` should increase `totalSupply` by `amount` and `balanceOf(to)` by `amount`.
- ☑ Should succeed when called by the `minter` address.

Negative behavior

- ☑ Revert with `OnlyMinter` if `msg.sender` is not the `minter`.

Function: `setMinter(address _minter)`

This function sets the address authorized to mint thUSD tokens. It is only callable by the owner.

Inputs

- `_minter`
    - **Control**: Full control (by owner).
    - **Constraints**: None — `address(0)` is accepted, which would effectively disable minting.
    - **Impact**: Sets the `minter` state variable. Only this address can call `mint`.

Branches and code coverage

Intended branches

- ☑ Should update the `minter` state variable.
- ☑ Setting the minter to a new address should allow that address to call `mint`.

Negative behavior

- ☑ Revert if the caller is not the owner.

5.4. Module: ThUSDMinter.sol

Function: `addSupportedAsset(address asset)`

This function adds a collateral asset to the set of supported assets for minting and redeeming. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `asset`
  - **Control**: Full control (by admin).
  - **Constraints**: Must not be `address(0)`, must not be the `thusd` address, and must not already be supported.
  - **Impact**: Enables the asset to be used as collateral in mint/redeem orders.

Branches and code coverage

Intended branches

- ☑ Should set `supportedAssets[asset]` to `true`.

Negative behavior

- ☑ Revert with `InvalidAssetAddress` if `asset` is `address(0)`.
- ☑ Revert with `InvalidAssetAddress` if `asset` is `thusd`.
- ☑ Revert with `InvalidAssetAddress` if `asset` is already supported.
- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `mint(Order calldata order, bytes calldata signature)`

This function processes a signed mint order; it validates the EIP-712 signature, transfers collateral from the signer to `mintDestination`, and mints thUSD to the recipient. It is subject to per-block rate limits, whitelist checks, and asset ratio validation.

Inputs

- `order`
  - **Control**: Constructed off chain by the signer and submitted by `MINTER_ROLE`.
  - **Constraints**: The `order.order_type` must be `MINT`; `order.collateral_amount` and `order.thusd_amount` must be greater than or equal to `MINIMUM_MINT_AMOUNT` (1e6); `order.expiry` must be greater than or equal to `block.timestamp`; `order.collateral_asset` must be a

supported asset; `order.signer` and `order.recipient` must be nonzero, whitelisted, and not blacklisted; and `order.signer` must not be `mintDestination` or `redeemDestination`. Normalized `thusd_amount` must not exceed normalized `collateral_amount` (1:1 ratio ceiling). The order hash must not have been used before.
    - **Impact**: Determines the collateral asset, amounts, and recipient of the mint; `thusd_amount` is added to `mintedPerBlock[block.number]`.
- `signature`
    - **Control**: Produced off chain by `order.signer`.
    - **Constraints**: Must be a valid ECDSA signature over the EIP-712 hash of the order, recovering to `order.signer`.
    - **Impact**: Authenticates the signer's consent to the order terms.

Branches and code coverage

Intended branches

- ☑ Should transfer `order.collateral_amount` of `order.collateral_asset` from `order.signer` to `mintDestination` via `safeTransferFrom`.
- ☑ Should mint `order.thusd_amount` of thUSD to `order.recipient`.
- ☑ Should increment `mintedPerBlock[block.number]` by `order.thusd_amount`.
- ☑ Should mark the order hash as used in `usedOrderHashes`.

Negative behavior

- ☑ Revert with `InvalidOrder` if `order.order_type` is not `MINT`.
- ☑ Revert with `MaxMintPerBlockExceeded` if `mintedPerBlock[block.number] + order.thusd_amount > maxMintPerBlock`.
- ☑ Revert with `InvalidAmount` if `collateral_amount` or `thusd_amount` is less than `MINIMUM_MINT_AMOUNT`.
- ☑ Revert with `OrderExpired` if `block.timestamp > order.expiry`.
- ☑ Revert with `UnsupportedAsset` if `order.collateral_asset` is not supported.
- ☑ Revert with `InvalidZeroAddress` if the signer or recipient is `address(0)`.
- ☑ Revert with `SignerIsDestination` if the signer is `mintDestination` or `redeemDestination`.
- ☑ Revert with `InvalidAssetRatio` if normalized `thusd_amount > collateral_amount`.
- ☑ Revert with `OrderAlreadyUsed` if the order hash was already used.
- ☑ Revert with `InvalidSignature` if the signature does not recover to `order.signer`.
- ☑ Revert if the signer or recipient is not whitelisted or is blacklisted.
- ☑ Revert if the caller does not have `MINTER_ROLE`.
- ☑ Revert if the contract is paused.

Function: `pause()`

This function pauses the contract, disabling `mint` and `redeem` operations. It is only callable by `EMERGENCY_ROLE`.

Inputs

None.

Branches and code coverage

Intended branches

- ☑ Should set the contract to a paused state, blocking mint and redeem.

Negative behavior

- ☑ Revert if the caller does not have `EMERGENCY_ROLE`.
- ☐ Revert if already paused.

Function: `redeem(Order calldata order, bytes calldata signature)`

This function processes a signed redeem order; it validates the EIP-712 signature, burns thUSD from the signer, and transfers collateral (minus fee) from `redeemDestination` to the recipient. It is subject to per-block rate limits and whitelist checks.

Inputs

- `order`
  - **Control**: Constructed off chain by the signer and submitted by `MINTER_ROLE`.
  - **Constraints**: The `order.order_type` must be `REDEEM`; `order.collateral_amount` and `order.thusd_amount` must be greater than or equal to `MINIMUM_MINT_AMOUNT` (1e6); `order.expiry` must be greater than or equal to `block.timestamp`; `order.collateral_asset` must be a supported asset; `order.signer` and `order.recipient` must be nonzero, whitelisted, and not blacklisted; and `order.signer` must not be `mintDestination` or `redeemDestination`. Normalized `collateral_amount` must not exceed normalized `thusd_amount`. The order hash must not have been used before. The `order.signer` must have approved ThUSDMinter for `thusd_amount` of thUSD.
  - **Impact**: Determines the collateral asset, amounts, and recipient of the redemption; `thusd_amount` is added to `redeemedPerBlock[block.number]`.
- `signature`

- **Control**: Produced off chain by `order.signer`.
- **Constraints**: Must be a valid ECDSA signature over the EIP-712 hash of the order, recovering to `order.signer`.
- **Impact**: Authenticates the signer's consent to the order terms.

Branches and code coverage

Intended branches

- ☑ Should burn `order.thusd_amount` of ThUSD from `order.signer` via `burnFrom`.
- ☑ Should transfer `order.collateral_amount - fee` of collateral from `redeemDestination` to `order.recipient`.
- ☑ The fee should be `Math.mulDiv(collateral_amount, redeemFeeBps, 10_000, Ceil)` — fee remains in `redeemDestination`.
- ☑ Should increment `redeemedPerBlock[block.number]` by `order.thusd_amount`.
- ☐ Should mark the order hash as used.

Negative behavior

- ☑ Revert with `InvalidOrder` if `order.order_type` is not `REDEEM`.
- ☑ Revert with `MaxRedeemPerBlockExceeded` if `redeemedPerBlock[block.number] + order.thusd_amount > maxRedeemPerBlock`.
- ☑ Revert with `InvalidAssetRatio` if normalized `collateral_amount > thusd_amount`.
- ☐ Revert with `OrderAlreadyUsed` if the order hash was already used.
- ☐ Revert with `InvalidSignature` if the signature does not recover to `order.signer`.
- ☐ Revert if the signer or recipient is not whitelisted or is blacklisted.
- ☑ Revert if the caller does not have `MINTER_ROLE`.
- ☐ Revert if the contract is paused.

Function: `removeSupportedAsset(address asset)`

This function removes a collateral asset from the set of supported assets. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `asset`
  - **Control**: Full control (by admin).
  - **Constraints**: Must currently be a supported asset.
  - **Impact**: Disables the asset from being used as collateral in future mint/redeem orders.

Branches and code coverage

Intended branches

- ☑ Should set `supportedAssets[asset]` to `false`.

Negative behavior

- ☑ Revert with `InvalidAssetAddress` if `asset` is not currently supported.
- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `setMaxMintPerBlock(uint256 newMax)`

This function updates the maximum amount of thUSD that can be minted per block. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `newMax`
  - **Control**: Full control (by admin).
  - **Constraints**: None — can be set to zero (effectively disabling minting) or any value up to `type(uint256).max`.
  - **Impact**: Sets the per-block minting cap. Setting to `0` prevents any minting.

Branches and code coverage

Intended branches

- ☑ Should update `maxMintPerBlock`.

Negative behavior

- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `setMaxRedeemPerBlock(uint256 newMax)`

This function updates the maximum amount of thUSD that can be redeemed per block. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `newMax`
  - **Control**: Full control (by admin).

- **Constraints**: None — can be set to zero (effectively disabling redemptions) or any value up to `type(uint256).max`.
- **Impact**: Sets the per-block redemption cap. Setting to `0` prevents any redemption.

Branches and code coverage

Intended branches

- ☑ Should update `maxRedeemPerBlock`.

Negative behavior

- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `setMintDestination(address newMintDestination)`

This function updates the address where collateral is sent during mint operations. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `newMintDestination`
  - **Control**: Full control (by admin).
  - **Constraints**: Must not be `address(0)`, `thusd`, or `address(this)` (enforced by `_checkBackingAddress`).
  - **Impact**: All future mint orders will transfer collateral to this address.

Branches and code coverage

Intended branches

- ☑ Should update `mintDestination`.

Negative behavior

- ☑ Revert with `InvalidBackingAddress` if `newMintDestination` is `address(0)`, `thusd`, or `address(this)`.
- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `setRedeemDestination(address newRedeemDestination)`

This function updates the address from which collateral is transferred during redeem operations. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `newRedeemDestination`
  - **Control**: Full control (by admin).
  - **Constraints**: Must not be `address(0)`, `thusd`, or `address(this)` (enforced by `_checkBackingAddress`).
  - **Impact**: All future redeem orders will pull collateral from this address.

Branches and code coverage

Intended branches

- ☑ Should update `redeemDestination`.

Negative behavior

- ☑ Revert with `InvalidBackingAddress` if `newRedeemDestination` is `address(0)`, `thusd`, or `address(this)`.
- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `setRedeemFeeBps(uint256 newFeeBps)`

This function updates the redemption fee in basis points. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

- `newFeeBps`
  - **Control**: Full control (by admin).
  - **Constraints**: Must be less than or equal to `MAX_FEE_BPS` (10, i.e., 0.1%).
  - **Impact**: Sets the fee deducted from collateral during redeem operations. Can be set to `0` to disable fees.

Branches and code coverage

Intended branches

- ☑ Should update `redeemFeeBps`.

Negative behavior

- ☑ Revert with `FeeTooHigh` if `newFeeBps > MAX_FEE_BPS`.
- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.

Function: `unpause()`

This function unpauses the contract, reenabling `mint` and `redeem` operations. It is only callable by `DEFAULT_ADMIN_ROLE`.

Inputs

None.

Branches and code coverage

Intended branches

- ☑ Should set the contract to an unpaused state, reenabling mint and redeem.

Negative behavior

- ☑ Revert if the caller does not have `DEFAULT_ADMIN_ROLE`.
- ☐ Revert if not currently paused.

6. Assessment Results

During our assessment on the scoped ThUSD contracts, we discovered six findings. No critical issues were found. One finding was of low impact and the other findings were informational in nature.

6.1. Disclaimer

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

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

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