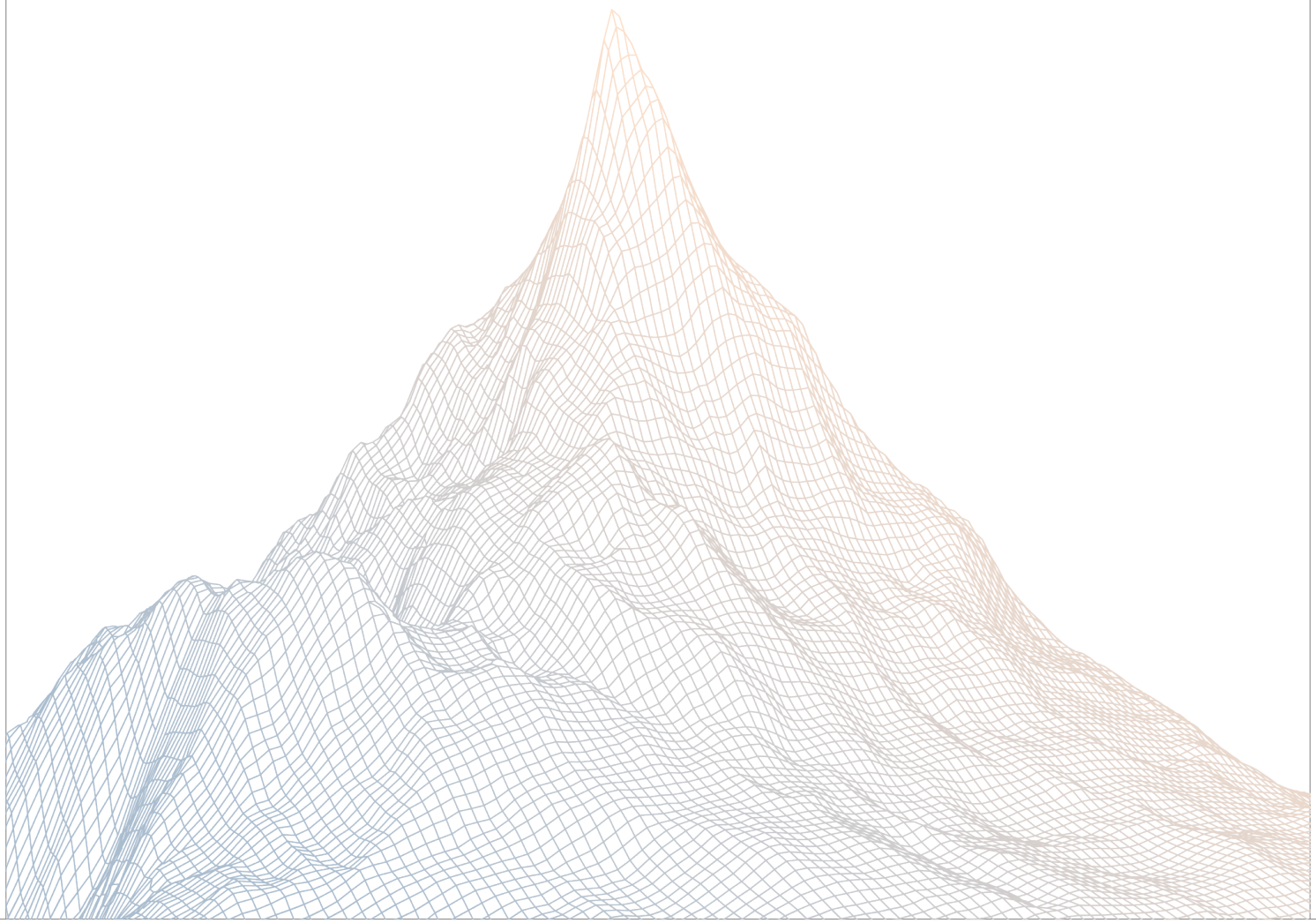
Zenith

Theo

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: April 3rd to April 6th, 2026

AUDITED BY: cccz
said

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Theo

Theo is building the infrastructure for a new kind of financial system: one that makes high-quality assets accessible to anyone, anywhere. In today's world, moving money globally is instant, but flexible access to global markets remains gated behind institutional barriers and legacy systems. We believe that must evolve.

Theo bridges this gap by bringing high-quality real-world assets onchain in a way that is truly compelling to retail and institutional

2.2 Scope

The engagement involved a review of the following targets:

Target	contracts-v2
Repository	https://github.com/theo-network/contracts-v2.git
Commit Hash	`92767d2badd65c85625429ad83d35ea572d7c3a2`
Files	`contracts/thusd/StUSD.sol`

Target	Mitigation Review
Repository	https://github.com/theo-network/contracts-v2.git
Commit Hash	`323641d8dd9fc85519dd145206a43f00e6459845`
Files	`contracts/thusd/StUSD.sol`

2.3 Audit Timeline

April 3, 2026	Audit start
April 6, 2026	Audit end
April 8, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	5
Informational	5
Total Issues	**10**

3

Findings Summary

ID	Description	Status
L-1	maxDeposit/maxMint/maxWithdraw/maxRedeem not pause-aware, violating ERC-4626 spec	Resolved
L-2	Rewards Vested During totalSupply == 0 Can Be Permanently Locked	Resolved
L-3	Missing _decimalsOffset() override enables first-depositor inflation attack	Acknowledged
L-4	Initialization Path Bypasses Setter Validation for Core Parameters	Resolved
L-5	pause() Does Not Freeze stUSD Transfers or Allowance Changes	Acknowledged
I-1	Delegated async redeem requires double allowance	Acknowledged
I-2	decimals() Override Bypasses ERC4626 Cached Underlying-Decimals Semantics	Resolved
I-3	preview and max Functions Describe Different Redemption States	Resolved
I-4	A Single stUSD Allowance Authorizes Both Live Shares and Queued Redemption Claims	Resolved
I-5	Theft of Dust Claimable Assets via withdraw() Rounding-to-Zero	Resolved

4

Findings

4.1 Low Risk

A total of 5 low risk findings were identified.

[L-1] `maxDeposit/maxMint/maxWithdraw/maxRedeem` not pause-aware, violating ERC-4626 spec

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L245-L259

Description:

StUSD overrides `deposit()` and `mint()` to add `whenNotPaused` and, but does not override `maxDeposit()` or `maxMint()` from the base `ERC4626Upgradeable`. The inherited implementations return `type(uint256).max` unconditionally.

Per ERC-4626 spec: *"MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0."*

Similarly, `withdraw()` and `redeem()` have `whenNotPaused`, but `maxWithdraw()` and `maxRedeem()` at do not check pause state.

When paused, all four `max*` view functions report non-zero values while the corresponding operations revert.

Recommendations:

Override all four functions to return 0 when paused.

Theo: Resolved with `@c54dfa507a ...`

Zenith: Verified. One minor note, `maxDeposit` and `maxMint` pass `address(this)` to super instead of forwarding the caller's parameter, but doesn't matter since OZ's default ignores the address and returns `type(uint256).max`.

[L-2] Rewards Vested During `totalSupply == 0` Can Be Permanently Locked

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L64-L159
- ERC4626Upgradeable.sol#L247-L255

Description:

`StUSD` excludes unvested rewards from `totalAssets()`, and `initiateRedeem()` snapshots the currently vested amount and burns the corresponding shares:

```
function totalAssets() public view override returns (uint256) {
    return IERC20(asset()).balanceOf(address(this)) - getUnvestedAmount()
    - _totalPendingRedeemAssets;
}
```

```
uint256 assets = previewRedeem(shares_);
_burn(owner_, shares_);

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

If the final holder exits during an active vesting period, the vault can enter a state where:

- `totalSupply == 0`
- some reward is still unvested

Any reward that vests during this zero-supply window is not attached to any outstanding share. Once vesting completes, those rewards sit in `totalAssets()` with `totalSupply == 0`.

At that point, the residual is locked because OpenZeppelin ERC4626 keeps a virtual share term even when real supply is zero:

```
function _convertToShares(uint256 assets, Math.Rounding rounding)
    internal view virtual returns (uint256) {
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
    totalAssets() + 1, rounding);
}

function _convertToAssets(uint256 shares, Math.Rounding rounding)
    internal view virtual returns (uint256) {
    return shares.mulDiv(totalAssets() + 1, totalSupply()
    + 10 ** _decimalsOffset(), rounding);
}
```

With `totalSupply == 0`, the vault still contains the virtual shares term `10 ** _decimalsOffset()`.

Minimal example:

```
1. User A owns all outstanding shares.
2. `setYield(100)` starts linear vesting.
3. Halfway through vesting, `initiateRedeem(totalSupply, userA)` snapshots
    only the currently vested amount.
4. All shares are burned, but part of the yield remains unvested.
5. Vesting continues with `totalSupply == 0`.
6. The rest vests into `totalAssets()` while noly virtual shares exist.
7. A later depositor cannot redeem that historical residual out of the vault.
```

Recommendations:

- Prevent rewards from vesting while `totalSupply == 0`.
- Alternatively, if the final share is burned during vesting, assign all later-vesting rewards to an explicit claimable request instead of letting them fall into unowned `totalAssets()`.

Theo: Resolved with `@b4c4f17965 ...`

Zenith: Verified. The fix sends unvested funds to the last redeemer when totalsupply == 0. One side effect is that the last redeemer may be vulnerable to frontrun attacks, but the team states that the last share will never be burned, so it's fine.

[L-3] Missing `_decimalsOffset()` override enables first-depositor inflation attack

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- StUSD.sol

Description:

StUSD does not override `_decimalsOffset()` from OZ ERC4626Upgradeable, which defaults to 0. The virtual share protection is `10^0 = 1`, negligible.

Attack scenario:

1. Attacker deposits 1 wei ThUSD → receives 1 share
2. Attacker donates 2,000,000 base units ($2.00) directly to StUSD via `transfer()`
3. Subsequent depositor with < $1.00 gets 0 shares:

```
shares = Floor(999,999 * 2 / 2,000,002) = 0
```

With 3+ sub-$1 victims, the attacker profits ~$0.50 on a $2.00 donation.

Recommendations:

Consider overriding `_decimalsOffset()` to increase the virtual share protection, making the attack more expensive.

Theo: Acknowledged. Upon deployment, this vault will be seeded with a non-negligible amount to prevent any initial inflation attack.

[L-4] Initialization Path Bypasses Setter Validation for Core Parameters

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L44-L97

Description:

`initialize()` writes `lockupPeriod` and `vestingDuration` directly during setup, but does not apply the same validation enforced by the corresponding owner setters:

```
function initialize(IERC20 thUSD_, uint256 lockupPeriod_,
    uint256 vestingDuration_, address owner_) external initializer {
    __StThUSD_init(thUSD_, lockupPeriod_, vestingDuration_, owner_);
}
```

```
function __StThUSD_init(IERC20 thUSD_, uint256 lockupPeriod_,
    uint256 vestingDuration_, address owner_) internal onlyInitializing {
    ...
    lockupPeriod = lockupPeriod_;
    vestingDuration = vestingDuration_;
}
```

By contrast, the post-deployment setters enforce explicit upper bounds:

```
function setLockupPeriod(uint256 lockupPeriod_) external onlyOwner {
    if (lockupPeriod_ > MAX_LOCKUP_PERIOD) revert MaxLockupPeriodExceeded();
    ...
}

function setVestingDuration(uint256 vestingDuration_) external onlyOwner {
    if (vestingDuration_ > MAX_VESTING_DURATION)
    revert MaxVestingDurationExceeded();
    ...
}
```

As a result, deployment-time initialization can install values that the contract would reject after deployment, such as a lockup period above `30 days` or a vesting duration above `365 days`. The same issue would recur in any future upgrade that introduces a `reinitializer` and directly assigns these fields without reusing the setter-side checks.

This does not give an external attacker a new entry point, but it weakens the upgrade and deployment safety model by making initialization less constrained than steady-state administration.

Recommendations:

Apply the same bounds checks inside the initialization path that are enforced by `setLockupPeriod()` and `setVestingDuration()`.

Theo: Resolved with `@9ccbb07699 ...`

Zenith: Verified.

[L-5] `pause()` Does Not Freeze `stUSD` Transfers or Allowance Changes

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- StUSD.sol#L19-L20
- StUSD.sol#L111-L118
- StUSD.sol#L123-L270

Description:

`StUSD` inherits `PausableUpgradeable`, and the pause gate is applied to core vault actions such as yield injection, redemption initiation, claim settlement, deposit, and mint:

```
function setYield(uint256 amount_) external whenNotPaused { ... }
function initiateRedeem(uint256 shares_, address owner_)
    external whenNotPaused { ... }
function withdraw(uint256 assets_, address receiver_, address owner_)
    public override whenNotPaused returns (uint256) { ... }
function redeem(uint256 shares_, address receiver_, address owner_)
    public override whenNotPaused returns (uint256) { ... }
function deposit(uint256 assets_, address receiver_)
    public override whenNotPaused returns (uint256) { ... }
function mint(uint256 shares_, address receiver_)
    public override whenNotPaused returns (uint256) { ... }
```

However, the contract does not pause ordinary ERC20 token mechanics. It does not inherit `ERC20Pausable`, and it does not override `transfer`, `transferFrom`, or the ERC20 state-update hook to apply pause checks to token movement.

As a result, during a paused state:

- users can still transfer live `stUSD`
- users can still use `transferFrom`
- users can still change allowances with `approve`
- users can still issue signed approvals through `permit`

This means the pause mechanism freezes vault entry, redemption, and claim paths, but it

does not freeze the secondary circulation of `stUSD`.

The practical side effect is that a pause event does not prevent live positions from being reallocated while the protocol is under emergency conditions. Users who understand the incident can still sell or transfer `stUSD` exposure to another party, and downstream integrators may continue to accept `stUSD` transfers or approvals unless they apply their own pause-aware controls. Although the recipient still cannot redeem while the vault remains paused, the economic risk of the frozen position can continue to move through the market.

This is not necessarily incorrect if the intended design is "pause vault operations only." It becomes a problem if stakeholders expect emergency pause to fully immobilize the `stUSD` asset during incidents.

Recommendations:

- Decide explicitly whether `pause()` is intended to freeze only vault operations or all `stUSD` movement.
- If full freeze semantics are intended, add pause checks to token transfers and allowance-consuming movement, for example by using an ERC20 pausable hook path.
- Document the current behavior clearly if it is intentional, so integrators and users do not assume that pause disables all `stUSD` circulation.

Theo: Acknowledged. This is the intended flow. Transfers are still allowed during a pause.

4.2 Informational

A total of 5 informational findings were identified.

[I-1] Delegated async redeem requires double allowance

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- StUSD.sol#L141-L168
- StUSD.sol#L185-L243

Description:

When a spender operates on behalf of a share owner through the async redeem flow, ERC-20 allowance is consumed twice for the same shares:

1. In `initiateRedeem()`, `_spendAllowance` deducts the allowance and the shares are burned.

(b) In `withdraw()` (or `redeem()`), `_spendAllowance` deducts allowance again, against shares that were already burned in step 1.

A spender approved for N shares needs 2N total allowance to complete the full initiate → claim cycle. The second check authorizes against shares that no longer exist.

Title

Recommendations:

Consider to store the authorized spender in the `RedeemRequestData` struct during `initiateRedeem` and skip the second allowance check for that spender.

Theo: Acknowledged. If the `initiateRedeem` uses an allowance, the `msg.sender` becomes the owner of the `RedeemRequestData`. In this case, the same `msg.sender` would not need to have a second allowance on the original owner since the redeem request is stored under themselves. Opting to not change the `RedeemRequestData` for simplicity.

[I-2] `decimals()` Override Bypasses ERC4626 Cached Underlying-Decimals Semantics

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L277-L279
- ERC4626Upgradeable.sol#L96-L128

Description:

`StUSD` overrides `decimals()` and directly queries the current asset's decimals on every call:

```
function decimals() public view override(ERC20Upgradeable,
    ERC4626Upgradeable) returns (uint8) {
    return ERC20(asset()).decimals();
}
```

This differs from OpenZeppelin's native `ERC4626Upgradeable` behavior, which caches the underlying asset decimals during initialization and later returns the cached value plus any configured decimals offset:

```
function __ERC4626_init_unchained(IERC20 asset_) internal onlyInitializing {
    ERC4626Storage storage $ = _getERC4626Storage();
    (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(asset_);
    $._underlyingDecimals = success ? assetDecimals : 18;
    $._asset = asset_;
}
```

```
function decimals() public view virtual override(IERC20Metadata,
    ERC20Upgradeable) returns (uint8) {
    ERC4626Storage storage $ = _getERC4626Storage();
    return $._underlyingDecimals + _decimalsOffset();
}
```

Under the current deployment model this does not immediately break `StUSD`, because the

underlying asset is `ThUSD`, which returns a fixed value of `6`:

```
function decimals() public pure override returns (uint8) {
    return 6;
}
```

However, the override removes two defensive properties provided by the standard ERC4626 implementation:

- it bypasses the cached fallback path used when the asset does not cleanly implement `decimals()`
- it ignores any future `_decimalsOffset()` customization introduced in a later upgrade

As a result, `StUSD.decimals()` becomes more tightly coupled to the runtime behavior of the underlying token than the inherited ERC4626 implementation expects. This is not a direct loss-of-funds issue in the present `StUSD` + `ThUSD` configuration, but it is an unnecessary deviation that weakens compatibility and upgrade safety.

Recommendations:

- Remove the custom `decimals()` override and inherit the default `ERC4626Upgradeable.decimals()` implementation.
- If a custom override is required, preserve the inherited semantics by returning the cached underlying decimals plus `_decimalsOffset()`, rather than re-querying the asset contract each time.

Theo: Resolved with `@7916f339df ...`

Zenith: Verified.

[I-3] `preview*` and `max*` Functions Describe Different Redemption States

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L141-L270

Description:

`StUSD` repurposes `withdraw()` and `redeem()` into claim-settlement functions for queued redemption requests, but it only overrides the execution paths and `maxWithdraw()` / `maxRedeem()`. The inherited `previewWithdraw()` and `previewRedeem()` functions still follow standard ERC4626 live-share conversion semantics.

Queued claims are created in `initiateRedeem()`:

```
uint256 assets = previewRedeem(shares_);
_burn(owner_, shares_);

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

Actual `withdraw()` and `redeem()` then operate on the queued request state rather than on live `stUSD` balances:

```
RedeemRequestData storage request = _redeemRequests[owner_];
if (request.assets == 0) revert NoClaimableRequest();
if (block.timestamp < request.claimableTimestamp) revert LockupNotElapsed();
```

At the same time, `maxWithdraw()` and `maxRedeem()` are explicitly tied to queued requests:

```
function maxWithdraw(address owner_) public view override returns (uint256)
    {
    RedeemRequestData storage request = _redeemRequests[owner_];
    if (block.timestamp >= request.claimableTimestamp && request.assets > 0)
```

```
    {
        return request.assets;
    }
    return 0;
}

function maxRedeem(address owner_) public view override returns (uint256) {
    RedeemRequestData storage request = _redeemRequests[owner_];
    if (block.timestamp >= request.claimableTimestamp && request.shares > 0)
    {
        return request.shares;
    }
    return 0;
}
```

This creates a semantic split:

- `previewDeposit()` / `previewMint()` still describe live ERC4626 deposit behavior
- `previewWithdraw()` / `previewRedeem()` still describe live-share conversions
- `maxWithdraw()` / `maxRedeem()` describe queued claim capacity
- `withdraw()` / `redeem()` settle queued claims rather than redeeming live shares directly

As a result, these functions no longer form a coherent ERC4626 interface surface. For example, an account may hold live `stUSD` and receive a positive value from `previewRedeem(balanceOf(user))`, while `maxRedeem(user)` returns `0` and `redeem( ... )` reverts because no queued claim exists. Integrators that assume the usual ERC4626 relationship between `preview*`, `max*`, and the executable methods can be misled into constructing invalid flows.

This is primarily an interface and integration risk rather than a direct loss-of-funds bug, but it makes `StUSD` materially different from a standard synchronous ERC4626 vault.

Recommendations:

- Override or clearly deprecate `previewWithdraw()` and `previewRedeem()` so they match the queued-claim semantics of `withdraw()` and `redeem()`.
- If preserving standard ERC4626 previews is desired, expose separate claim-preview methods for the async redemption flow instead of overloading the standard names.
- Document explicitly that `maxWithdraw()` and `maxRedeem()` refer to queued redeem requests, not live `stUSD` balances.

Theo: Resolved with `@e30ff9ce8e ...`

Zenith: Verified.

[I-4] A Single `stUSD` Allowance Authorizes Both Live Shares and Queued Redemption Claims

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L141-L241

Description:

`StUSD` uses the same ERC20 allowance namespace for two economically different states of the position.

When a third party initiates redemption on behalf of an account, the allowance authorizes control over live `stUSD` shares:

```
if (msg.sender ≠ owner_) {
    _spendAllowance(owner_, msg.sender, shares_);
}

uint256 assets = previewRedeem(shares_);
_burn(owner_, shares_);
```

After the shares are burned and converted into a queued redeem request, `withdraw()` and `redeem()` continue to use the same `stUSD` allowance to authorize claims on the queued position:

```
if (msg.sender ≠ owner_) {
    _spendAllowance(owner_, msg.sender, sharesToDeduct);
}
```

```
if (msg.sender ≠ owner_) {
    _spendAllowance(owner_, msg.sender, shares_);
}
```

As a result, a user granting `approve(spender, X)` is not only delegating control over up to `X` live vault shares, but also delegating control over the queued redemption claim that those

shares may later become. These are not the same economic object:

- live shares remain transferable and continue to track vault economics
- queued redemption claims are locked, non-transferable claim rights against reserved `ThUSD`

The current test suite explicitly accepts this behavior:

```
vm.prank(user1);
stUSD.approve(user2, DEPOSIT_AMOUNT);

vm.prank(user2);
stUSD.initiateRedeem(DEPOSIT_AMOUNT, user1);

assertEq(stUSD.pendingRedeemRequest(user2), DEPOSIT_AMOUNT);
```

and:

```
vm.prank(user1);
stUSD.approve(user2, DEPOSIT_AMOUNT);

vm.prank(user2);
stUSD.withdraw(DEPOSIT_AMOUNT, user2, user1);
```

This does not create a standalone theft vector if the intended model is "allowance conveys full economic control over the approved position." However, it materially broadens what an ordinary `stUSD` approval means and makes the authorization boundary less intuitive for users and integrators.

Recommendations:

- Explicitly document that `stUSD` allowance authorizes both live-share operations and queued redemption claims.
- If the intended model is narrower delegation, separate queued-claim authorization from ERC20 share allowance.
- Consider exposing dedicated approval or operator semantics for queued redemption claims so integrations can reason about the two asset states independently.
- Add interface-level documentation and tests that make the dual-use allowance model explicit.

Theo : Resolved with `@323641d8dd ...`

Zenith: Verified. `stUSD` allowance no longer authorizes queued redemption claims.

[I-5] Theft of Dust Claimable Assets via `withdraw()` Rounding-to-Zero

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StUSD.sol#L185-L241

Description:

`withdraw()` uses upward rounding when converting the requested asset amount into the number of queued shares to deduct:

```
if (assets_ == request.assets) {
    sharesToDeduct = request.shares;
} else {
    sharesToDeduct = assets_.mulDiv(request.shares, request.assets,
    Math.Rounding.Ceil);
}
```

This allows a partially claimed request to reach a state where `request.shares == 0` while `request.assets > 0`.

For example, if a matured request starts at `request.assets = 11` and `request.shares = 10`, then calling `withdraw(10, receiver, owner_)` evolves the request as follows:

```
sharesToDeduct = ceil(10 * 10 / 11) = 10;
request.assets = 11 - 10 = 1;
request.shares = 10 - 10 = 0;
```

Once this state is reached, the remaining dust assets can be stolen by any caller. `withdraw()` authorizes third-party callers with `_spendAllowance(owner_, msg.sender, sharesToDeduct)`, but when `sharesToDeduct == 0`, that check becomes ineffective:

```
if (msg.sender != owner_) {
    _spendAllowance(owner_, msg.sender, sharesToDeduct);
}
```

As a result, any external account can call `withdraw(remainingAssets, attacker, owner_)` and drain the leftover queued dust assets without approval.

`redeem()` does not have the same issue, because its full-claim branch clears both sides together and its partial branch reduces `shares` directly rather than deriving them with upward rounding.

Recommendations:

- Prevent `withdraw()` from leaving a request in the state `request.assets > 0 && request.shares == 0`.
- If `sharesToDeduct == request.shares`, treat the operation as a full claim and transfer the entire remaining `request.assets` instead of only `assets_`.

Theo: Resolved with `@5875e33e01 ...`.

Zenith: Verified.