SHERLOCK

Security Review For Tori Finance

Collaborative Audit Prepared For: **Tori Finance**
Lead Security Expert(s): **defsec**
Drynooo
Date Audited: **January 15 - January 22, 2026**

Introduction

Tori Finance is a synthetic dollar protocol designed to democratize access to sustainable, institutional-grade yield. Through its synthetic dollar (trUSD) and yield-bearing (strUSD) tokens, Tori enables permissionless participation in delta-neutral strategies, previously reserved for accredited investors and large institutions. The protocol operates with a fully transparent, on-chain balance sheet, leveraging real-time proof of reserves to ensure verifiability at every layer of the stack.

Scope

Repository: sherlock-scoping/Tori-Finance__contracts

Audited Commit: 829967e94f648dfb4f797926078b71ffc50ac572

Final Commit: eb5074003fa2e344ad8e34f464c7080c4c715ed5

Files:

- contracts/core/StakedTrUSD.sol
- contracts/core/ToriMinting.sol
- contracts/core/TrUsdSilo.sol
- contracts/core/TrUSD.sol
- contracts/governance/ToriTimelock.sol
- contracts/interfaces/IOmnichainAdapter.sol
- contracts/interfaces/IOmnichainOFT.sol
- contracts/interfaces/IRateLimiter.sol
- contracts/interfaces/IStakedTestUSDCooldown.sol
- contracts/interfaces/IStakedTestUSD.sol
- contracts/interfaces/IStakedTrUSDCooldown.sol
- contracts/interfaces/IStakedTrUSD.sol
- contracts/interfaces/ITestSiloDefinitions.sol
- contracts/interfaces/ITrUSDDefinitions.sol
- contracts/interfaces/ITrUsdSiloDefinitions.sol
- contracts/libraries/OFTAdapterOwnable2Step.sol
- contracts/libraries/OFTOwnable2Step.sol
- contracts/libraries/RateLimiter.sol
- contracts/omnichain/StakedTrUSDAdapter.sol

- contracts/omnichain/StakedTrUSDOFT.sol
- contracts/omnichain/TrUSDAdapter.sol
- contracts/omnichain/TrUSDOFT.sol

Final Commit Hash

eb5074003fa2e344ad8e34f464c7080c4c715ed5

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	7	14

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: `transferToCustody()` function always reverts due to incorrect validation of ERC20 token address against EOA check [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/23

Summary

The `transferToCustody()` function contains a logic error where it validates the asset parameter (ERC20 token address) using `_validCustodyAddress()`, which checks if an address has no bytecode. Since all ERC20 tokens are deployed contracts with bytecode, this validation always fails, making the function completely non-functional.

Vulnerability Detail

The `_validCustodyAddress()` function checks if an address is an EOA (no bytecode) or the contract itself:

```
function _validCustodyAddress(address addr) internal view returns (bool) {
    return addr.code.length == 0 || addr == address(this);
}
```

However, this function is called with the asset parameter (the ERC20 token to transfer):

```
function transferToCustody(address asset, uint256 amount, Route calldata route)
    external
    nonReentrant
    onlyRole(COLLATERAL_MANAGER_ROLE)
{
    if (!_validCustodyAddress(asset)) revert InvalidAddress();
    // ...
    IERC20(asset).safeTransfer(route.addresses[i], amountToTransfer);
}
```

Logic Contradiction:

- For `_validCustodyAddress(asset)` to return true: `asset.code.length == 0` (no bytecode) or `asset == address(this)`
- For `IERC20(asset).safeTransfer()` to work: asset must be an ERC20 contract (has bytecode)

These requirements are mutually exclusive. Any valid ERC20 token (USDC, USDT, DAI, etc.) is a deployed contract with `code.length > 0`, so the validation always fails.

Impact

`COLLATERAL_MANAGER_ROLE` cannot transfer any collateral to custodians via this function.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/384b484f748cc0971d940aaacde6f6d2dd09d3bc/ToriMinting.sol#L518

Tool Used

Manual Review

Recommendation

Remove the incorrect validation or replace it with proper token validation.

Discussion

defsec

Fix is confirmed with deleting condition and adding a remainder control.

Issue M-1: Rounding errors in ratio calculations can leave unaccounted tokens in the contract [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/26

Summary

The `transferToCustody()` function in `ToriMinting.sol` calculates transfer amounts using integer division, which can result in rounding errors. Unlike `_transferCollateral()` which handles remainders by sending them to the last address, `transferToCustody()` does not account for rounding differences, potentially leaving dust amounts permanently stuck in the contract.

Vulnerability Detail

The `transferToCustody()` function distributes tokens to multiple custodians based on ratios:

```
for (uint256 i = 0; i < route.addresses.length;) {
    amountToTransfer = (amount * route.ratios[i]) / 10_000;
    if (amountToTransfer > 0) {
        IERC20(asset).safeTransfer(route.addresses[i], amountToTransfer);
    }
    unchecked { ++i; }
}
```

The Problem:

- Integer division `(amount * route.ratios[i]) / 10_000` rounds down
- When ratios don't perfectly divide the amount, the sum of all `amountToTransfer` values will be less than `amount`
- The remainder (dust) remains in the contract with no mechanism to recover it

Example:

- `amount = 1000`
- `ratios = [3333, 3333, 3334]` (sums to 10000)
- `amountToTransfer[0] = (1000 * 3333) / 10000 = 333` (rounding down from 333.3)
- `amountToTransfer[1] = (1000 * 3333) / 10000 = 333` (rounding down from 333.3)
- `amountToTransfer[2] = (1000 * 3334) / 10000 = 333` (rounding down from 333.4)
- **Total transferred = 999, Remaining = 1** (stuck in contract)

Comparison with `_transferCollateral()`: The `_transferCollateral()` function correctly handles this by sending the remainder to the last address:

```
uint128 remainingBalance = amount - totalTransferred;
if (remainingBalance > 0) {
    token.safeTransferFrom(from, addresses[addresses.length - 1], remainingBalance);
}
```

Impact

Small amounts of tokens can accumulate in the contract over time.

Code Snippet

File: `contracts/core/ToriMinting.sol:539-547`

```
for (uint256 i = 0; i < route.addresses.length;) {
    amountToTransfer = (amount * route.ratios[i]) / 10_000;
    if (amountToTransfer > 0) {
        IERC20(asset).safeTransfer(route.addresses[i], amountToTransfer);
    }
    unchecked { ++i; }
}
// Missing: remainder handling like _transferCollateral()
```

File: `contracts/core/ToriMinting.sol:732-735` - Correct implementation in `_transferCollateral()`:

```
uint128 remainingBalance = amount - totalTransferred;
if (remainingBalance > 0) {
    token.safeTransferFrom(from, addresses[addresses.length - 1], remainingBalance);
}
```

Tool Used

Manual Review

Recommendation

Add remainder handling similar to `_transferCollateral()`.

Discussion

defsec

Fixed with handling rounding case.

Issue M-2: `SOFT_RESTRICTED_STAKER_ROLE` can bypass restrictions and withdraw with yield [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/30

Summary

The `_withdraw()` and `_update()` functions in `StakedTrUSD.sol` only check for `FULL_RESTRICTED_STAKER_ROLE` but not `SOFT_RESTRICTED_STAKER_ROLE`. This allows soft-restricted users (those in jurisdictions where yield is not permitted) to receive strUSD via transfer and subsequently withdraw it for TrUSD, effectively earning yield in violation of their restrictions.

Vulnerability Detail

- `SOFT_RESTRICTED_STAKER_ROLE`: Users in jurisdictions where earning yield is not permitted. They should not be able to deposit OR withdraw (realize yield).
- `FULL_RESTRICTED_STAKER_ROLE`: Completely blocked users (e.g., OFAC sanctions). Cannot hold or transfer tokens.

Actual Behavior:

```
// _deposit() - CORRECTLY blocks SOFT_RESTRICTED
function _deposit(address caller, address receiver, uint256 assets, uint256 shares)
↪  internal override {
    if (hasRole(SOFT_RESTRICTED_STAKER_ROLE, caller) ||
    ↪  hasRole(SOFT_RESTRICTED_STAKER_ROLE, receiver)) {
        revert OperationNotAllowed();  //  Correctly blocked
    }
    // ...
}

// _withdraw() - does not block SOFT_RESTRICTED
function _withdraw(address caller, address receiver, address _owner, uint256
↪  assets, uint256 shares) internal override {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
        hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver) ||
        hasRole(FULL_RESTRICTED_STAKER_ROLE, _owner)) {
        revert OperationNotAllowed();
    }
    // SOFT_RESTRICTED_STAKER_ROLE is NOT checked - users can withdraw!
    // ...
}

// _update() - does not block SOFT_RESTRICTED for receiving
```

```
function _update(address from, address to, uint256 amount) internal virtual
↪  override {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, from) && to != address(0)) {
        revert OperationNotAllowed();
    }
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, to)) {
        revert OperationNotAllowed();
    }
    // SOFT_RESTRICTED_STAKER_ROLE is NOT checked - users can receive transfers!
    super._update(from, to, amount);
}
```

Impact

Users in jurisdictions where earning yield is prohibited can circumvent restrictions.

Code Snippet

File: contracts/core/StakedTrUSD.sol:346-359

```
function _withdraw(address caller, address receiver, address _owner, uint256
↪  assets, uint256 shares)
    internal
    override
    nonReentrant
    notZero(assets)
    notZero(shares)
{
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
    ↪  hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver) ||
    ↪  hasRole(FULL_RESTRICTED_STAKER_ROLE, _owner)) {
        revert OperationNotAllowed();
    }
    // Missing: SOFT_RESTRICTED_STAKER_ROLE check

    super._withdraw(caller, receiver, _owner, assets, shares);
    _checkMinShares();
}
```

File: contracts/core/StakedTrUSD.sol:361-369

```
function _update(address from, address to, uint256 amount) internal virtual
↪  override {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, from) && to != address(0)) {
        revert OperationNotAllowed();
    }
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, to)) {
```

```
        revert OperationNotAllowed();
    }
    // Missing: SOFT_RESTRICTED_STAKER_ROLE check for receiving
    super._update(from, to, amount);
}
```

Tool Used

Manual Review

Recommendation

Add `SOFT_RESTRICTED_STAKER_ROLE` checks to `_withdraw()`.

Discussion

defsec

Fixed with adding a condition.

Issue M-3: Blacklisted shares continue earning rewards during vesting period [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/35

Summary

When admin tries to burn blacklisted user's shares via `redistributeLockedAmount(from, address(0))`, the function reverts if there's an active vesting period. During this time (up to 8 hours), the blacklisted user's shares continue earning rewards.

Vulnerability Detail

The `redistributeLockedAmount` function allows admin to burn a blacklisted user's shares and redistribute the underlying assets to all stakers by calling with `to = address(0)`:

```
// StakedTrUSD.sol:172-189
function redistributeLockedAmount(address from, address to) external nonReentrant
↪  onlyRole(DEFAULT_ADMIN_ROLE) {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, from) &&
    ↪  !hasRole(FULL_RESTRICTED_STAKER_ROLE, to)) {
        uint256 amountToDistribute = balanceOf(from);
        uint256 assetsToVest = previewRedeem(amountToDistribute);

        _burn(from, amountToDistribute);

        if (to == address(0)) {
            _updateVestingAmount(assetsToVest);  // Can revert!
        } else {
            _mint(to, amountToDistribute);
        }

        emit LockedAmountRedistributed(from, to, amountToDistribute);
    } else {
        revert OperationNotAllowed();
    }
}
```

However, `_updateVestingAmount` reverts if there's an ongoing vesting period:

```
// StakedTrUSD.sol:319-325
function _updateVestingAmount(uint256 newAmount) internal {
    if (getUnvestedAmount() > 0) revert StillVesting();  // Blocks during vesting

    StakedTrUSDStorage storage $ = _getStakedTrUSDStorage();
    $.vestingAmount = newAmount;
```

```
    $.lastDistributionTimestamp = block.timestamp;
}
```

Impact

Legitimate stakers receive fewer rewards because blacklisted shares still count toward totalSupply.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/main/Tori-Finance__contracts/contracts/core/StakedTrUSD.sol#L172

Tool Used

Manual Review

Recommendation

Add a mechanism to immediately halt reward accrual for blacklisted users, or allow burning to a holding contract.

Discussion

defsec

Hi @ertnec , are we missing a fix for that one?

ertnec

Hi @defsec

https://github.com/Tori-Finance/contracts/pull/11

I had attached this PR to issue #35

defsec

Fix is confirmed.

Issue M-4: Missing FULL_RESTRICTED_STAKER_ROLE check in StakedTrUSD _deposit allows sanctioned entities to deposit funds [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/39

Summary

The `_deposit` function in `StakedTrUSD.sol` is designed to restrict blacklisted users from participating in the protocol, but it only checks for the `SOFT_RESTRICTED_STAKER_ROLE`, omitting the check for the `FULL_RESTRICTED_STAKER_ROLE`. This allows fully frozen entities to deposit funds into the protocol by specifying a third-party receiver, thereby bypassing restrictions.

Vulnerability Detail

In the `_deposit` function of `StakedTrUSD.sol`, the logic explicitly prohibits users with the `SOFT_RESTRICTED_STAKER_ROLE` from participating as either the `caller` (fund provider) or the `receiver` (share recipient).

However, the logic fails to check if the `caller` holds the stricter `FULL_RESTRICTED_STAKER_ROLE`. Although the `_update` function in the contract prevents a `FULL_RESTRICTED` user from receiving minted shares (as the `to` address), it does not prevent them from initiating the transaction as the `caller`.

This means an address marked as `FULL_RESTRICTED` can call `deposit(amount, innocentUser)`, successfully transferring restricted assets into the Vault and minting shares for another user. This violates the compliance hierarchy design, where the permissions of a fully restricted role should be stricter than or equal to those of a soft restricted role.

Impact

This breaks the protocol's AML (Anti-Money Laundering) and compliance invariants. "Dirty funds" from fully sanctioned or frozen entities can enter the protocol, and the most restricted users (FULL) can bypass deposit operations that are blocked for soft restricted users (SOFT).

Code Snippet

```
function _deposit(address caller, address receiver, uint256 assets, uint256 shares)
    internal
    override
    nonReentrant
```

```
    notZero(assets)
    notZero(shares)
{
    // @audit-issue Only checks SOFT_RESTRICTED, missing FULL_RESTRICTED check
    if (hasRole(SOFT_RESTRICTED_STAKER_ROLE, caller) ||
    ↪  hasRole(SOFT_RESTRICTED_STAKER_ROLE, receiver)) {
        revert OperationNotAllowed();
    }
    super._deposit(caller, receiver, assets, shares);
    _checkMinShares();
}
```

Tool Used

Manual Review

Recommendation

Modify the `_deposit` function to explicitly prohibit the `FULL_RESTRICTED_STAKER_ROLE` from participating in deposits as either the `caller` or the `receiver`.

Issue M-5: StakedTrUSD.reportLoss is blocked during vesting, preventing handling of bad debt and allowing users to exit fully, forcing remaining stakers to bear the loss [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/40

Summary

The `StakedTrUSD.reportLoss` function includes a check that reverts the transaction if there are unvested rewards (`getUnvestedAmount() > 0`). This prevents the protocol from handling bad debt during the 8-hour vesting window. Informed users can exploit this delay to exit via the cooldown mechanism at an inflated valuation (that does not reflect the loss), forcing remaining stakers to absorb the entire loss.

Vulnerability Detail

The `StakedTrUSD` contract manages staking and reward distribution. When bad debt occurs (e.g., collateral default), an admin with the `REWARDER_ROLE` calls `reportLoss` to burn assets, thereby lowering the share price (exchange rate) and distributing the loss among all holders.

However, the `reportLoss` function enforces the following check:

```
if (getUnvestedAmount() > 0) revert StillVesting();
```

This means if rewards are currently vesting linearly (the vesting period is 8 hours), the admin is blocked from reporting the loss. The attack scenario is as follows:

1. An off-chain loss or collateral default occurs while rewards are vesting in the contract.
2. The admin attempts to call `reportLoss` to update the exchange rate, but the transaction reverts due to the `StillVesting` error.
3. Informed users call `cooldownShares` or `cooldownAssets`. Since the loss has not been recognized, the system calculates shares based on the current inflated exchange rate and transfers the corresponding underlying assets to the `TrUsdSilo`.
4. Once inside the `TrUsdSilo`, these assets are effectively treated as withdrawn by the users and are no longer subject to exchange rate fluctuations of `StakedTrUSD`.
5. After the vesting period ends, the admin can finally call `reportLoss`, but it can only burn the assets remaining in the vault.

6. As a result, users who exit first escape with zero loss, while remaining stakers suffer a magnified loss.

Impact

The risk-sharing mechanism of the protocol is broken. In emergency situations, the protocol fails to update the Net Asset Value (NAV) in a timely manner. This creates an incentive for a "bank run," where users who act first can preserve their capital fully, while remaining passive stakers bear 100% of the bad debt, leading to fund loss and unfairness.

Code Snippet

```
function reportLoss(uint256 amount) external nonReentrant onlyRole(REWARDER_ROLE)
↪  notZero(amount) {
    if (getUnvestedAmount() > 0) revert StillVesting(); // @audit-issue Blocks
    ↪  critical risk management operations

    uint256 available = totalAssets();
```

Tool Used

Manual Review

Recommendation

Remove the `getUnvestedAmount()` check in the `reportLoss` function.

When reporting a loss, asset burning should be allowed to execute immediately. If the loss amount is significant, logic could be implemented to proportionally deduct from both vested assets and unvested rewards, or simply allow the admin to intervene and handle bad debt at any time without being locked out by the vesting period.

Discussion

drynooo

@ertnec **Token burning is skipped when the loss is covered by unvested rewards.**

The formula for `totalAssets()` is `Balance - UnvestedAmount`. When reporting a loss, you correctly reduce the `UnvestedAmount` (the deduction term), but if you fail to `burn` the actual tokens (the Balance term), the `totalAssets()` will erroneously increase.

Example:

1. **Initial State**: Balance = 150, Unvested = 50.

- `totalAssets` = 150 - 50 = **100**.

2. **Action**: `reportLoss(20)`.
3. **Execution**: The code offsets `vestingAmount` by 20, making Unvested = 30. Since `remainingLoss` becomes 0, the `burn` execution is skipped.
4. **Bug Result**: Balance remains 150.
    - New `totalAssets` = 150 - 30 = **120**.

Reporting a loss of 20 resulted in a profit of 20 for users.

Regardless of the `remainingLoss`, the full `amount` should be burned.

ertnec

Hi @drynooo , thanks for your findings.

i merged new pr to fix this issue https://github.com/Tori-Finance/contracts/pull/17.

drynooo

Hi @ertnec . Sorry, I don't really understand the new fix logic, let me elaborate on some of the problems I see. If my understanding is wrong, please correct me.

1. Users lose more money than the reported loss Directly subtracting `amount` from `\$.vestingAmount` assumes a 1:1 reduction in the `unvested` value, ignoring the time-decay factor (t_{rem}/T_{total}).

- **Example**: `unvested` is 100 (50% time remaining). You report a loss of 10.
- The code burns `2 * 10 = 20` tokens.
- However, `vestingAmount` reduction only lowers the actual `unvested` value by $10 \times 0.5 = 5$.
- **Result**: The protocol burns 20 tokens to reduce liability by 5. Users suffer a net loss of 15 assets for a reported loss of 10.

2. Permanent Denial of Service (DoS) Risk `totalAssets()` is calculated as `Balance - Unvested`. The function burns physical tokens (`Balance`) significantly faster than it reduces the `Unvested` liability.

- **Scenario**: Vesting just started (90% remaining). You report a loss.
- The code burns `2 * Loss` from Balance, but Unvested only drops by `0.9 * Loss`.
- If this happens repeatedly, or with a large amount, `Balance` will drop below `Unvested`.
- **Result**: `totalAssets()` tries to calculate `Small_Balance - Large_Unvested`, triggering an underflow revert. **All deposits and withdrawals will be permanently locked.**

drynooo

Hi @ertnec , I think the variable update part is correct, but there are still some issues with the burn part.

1. **Double Burn:** The line `uint256 totalBurn = amount + unvestedCancelled;` creates a logic error where the protocol physically destroys assets for both the actual loss AND the accounting adjustment. This effectively penalizes stakers twice.
2. **Incorrect Availability Calculation:** The line `uint256 available = totalAssets() + unvestedCancelled;` is mathematically flawed. Since `vestingAmount` is updated *before* this line, `totalAssets()` subtracts the *new* unvested amount. This formula fails to reconstruct the actual contract balance, leading to potential reverts (DoS) or leaving "zombie assets" in the contract.

Example:

- **State:** Vault has **1,000 USDC**. **100 USDC** is Unvested.
- **Action:** `reportLoss(10)` is called.
- **Execution:**
    1. `newUnvested` correctly becomes **90**.
    2. **Double Burn:** `totalBurn` = 10 (loss) + 10 (cancelled) = **20**. The protocol burns 20 USDC to cover a 10 USDC loss.
    3. **Bad Math:** `available` calculates to `(1000 - 90) + 10 = 920`, while the true balance is 1,000.

Fix: Discard the complex `available` calculation and use the actual token balance. Burn only the reported loss amount.

Issue M-6: Lack of on-chain oracle circuit breaker creates arbitrage risk during collateral de-peg [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/42

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `verifyOrder` function relies solely on off-chain signatures and fixed 1:1 exchange logic, lacking an on-chain oracle (e.g., Chainlink) as a safety circuit breaker. If a collateral asset de-pegs, attackers can exploit valid signatures to arbitrage, harming the protocol's solvency.

Vulnerability Detail

In the `ToriMinting.sol` contract, `mint` and `redeem` operations validate requests via the `verifyOrder` function. This validation logic primarily relies on:

1. An EIP-712 signature generated off-chain.
2. The `verifyStablesLimit` function, which only checks if the input collateral amount and the generated TrUSD amount are approximately 1:1 (allowing for a small delta).

The contract assumes that supported collateral (e.g., USDC) is always worth 1 USD. However, the contract lacks the ability to fetch real-time market prices of the collateral (e.g., it does not integrate a Chainlink oracle).

If a supported collateral asset (e.g., USDC) severely de-pegs (e.g., drops to $0.90) within the signature's validity period:

1. An attacker can use a valid, obtained signature to call `mint`.
2. The contract passes the `verifyStablesLimit` check, mistakenly assuming 1 USDC is still worth 1 USD.
3. The attacker mints TrUSD at full face value using devalued USDC.
4. The attacker can then redeem TrUSD for other healthy assets (e.g., USDT) or sell it on the market.

Even if the off-chain signer is trusted and stops service after a de-peg, unexpired signatures issued during the de-peg event or during the backend's reaction latency can still be exploited.

Impact

When collateral de-pegs, the protocol receives devalued assets while minting stablecoins at face value. This leads to bad debt for the protocol and severely impacts TrUSD's solvency and peg stability.

Code Snippet

```
function verifyOrder(Order calldata order, Signature calldata signature) public
↪  view returns (bytes32) {
    // ... (Signature verification omitted)

    // Only checks token type and internal ratio limits, no external price check
    if (tokenConfig[order.collateral_asset].tokenType != TokenType.STABLE) {
        revert UnsupportedAsset();
    }
    // verifyStablesLimit only calculates ratio based on amounts, assuming 1:1 value
    if (!verifyStablesLimit(order.collateral_amount, order.testusd_amount,
    ↪  order.collateral_asset, order.order_type)) {
        revert InvalidStablePrice();
    }

    // ...
}
```

Tool Used

Manual Review

Recommendation

It is recommended to introduce an on-chain oracle (e.g., Chainlink) as a circuit breaker. Add a check in the `verifyOrder` or `mint` logic to ensure the collateral's market price is within a safe range (e.g., > $0.98). If the price falls below the threshold, the transaction should revert to protect the protocol, even if the signature is valid.

Discussion

ertnec

Acknowledged - The protocol intentionally uses off-chain pricing by trusted backend services rather than on-chain oracles. Depeg handling: Backend monitors real-time prices and adjusts order pricing accordingly. If USDC depegs to $0.90, backend signs orders at actual market value (1000 USDC → 900 TrUSD), not face value. Why no oracle: We prefer not to introduce oracle dependency and associated risks (manipulation,

downtime, staleness). Our monitoring infrastructure already handles price verification off-chain with short signature expiry windows limiting exposure during backend reaction time.

Issue M-7: Bypassing TrUSD Rate Limits via StakedTrUSD Bridging [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/44

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`TrUSD` and `StakedTrUSD` currently enforce mutually independent cross-chain Rate Limits. An attacker can bypass the `TrUSD` flow control by staking `TrUSD` into `StakedTrUSD`, bridging the staked tokens, and redeeming them on the destination chain.

Vulnerability Detail

In the current architecture, both `TrUSDAdapter` and `StakedTrUSDAdapter` inherit from the `RateLimiter` contract. This means they each maintain a separate `rateLimits` mapping state that is not shared between them.

`StakedTrUSD` is backed by `TrUSD`, and their values are tightly coupled. The rate limit is designed to restrict the outflow speed of `TrUSD` assets across chains during hacks or extreme market volatility, protecting protocol solvency.

However, since the rate limit buckets are separated, an attacker can bypass the restrictions using the following flow:

1. When the `TrUSD` cross-chain limit is exhausted or the attacker wishes to send more than the limit allows.
2. The attacker deposits `TrUSD` into the `StakedTrUSD` contract to obtain `strUSD`.
3. The attacker bridges `strUSD` (consuming the `StakedTrUSDAdapter`'s rate limit, not the `TrUSDAdapter`'s).
4. On the destination chain, the attacker redeems the received `strUSD` for `TrUSD`.

This mechanism renders the flow control for the underlying `TrUSD` asset ineffective.

Impact

Disruption of the protocol's Defense-in-Depth mechanism. Rate limits are a critical line of defense preventing rapid fund draining or infinite minting during cross-chain bridge security incidents. If attackers can bypass this limit, the protocol cannot effectively contain rapid asset outflows during an attack, potentially leading to greater economic loss.

Code Snippet

`TrUSDAdapter` and `StakedTrUSDAdapter` inherit `RateLimiter` separately, leading to state isolation:

```
// TrUSDAdapter.sol
contract TrUSDAdapter is OFTAdapterOwnable2Step, Pausable, RateLimiter {
    // ...
    // Uses independent rateLimits storage
}
```

```
// StakedTrUSDAdapter.sol
contract StakedTrUSDAdapter is TrUSDAdapter {
    // Inherits TrUSDAdapter; if deployed as a separate contract, it possesses its
    ↪  own storage space and independent rateLimits
}
```

```
abstract contract RateLimiter is IRateLimiter {
    // Each inheriting contract has its own independent mapping
    mapping(uint32 dstEid => RateLimit limit) public rateLimits;

    function _checkAndUpdateRateLimit(uint32 _dstEid, uint256 _amount) internal {
        // ...
    }
}
```

Tool Used

Manual Review

Recommendation

It is recommended that `TrUSDAdapter` and `StakedTrUSDAdapter` share the same Rate Limiter state, or be configured as a whole when setting limits. For example, the `RateLimiter` logic could be extracted into a separate external contract (SharedRateLimiter), where both Adapters call this external contract to check and update limits before bridging.

Discussion

ertnec

Acknowledge - This is a valid architectural consideration.

Issue L-1: `transferCollateral` does not account for fee-on-transfer tokens leading to under-collateralized minting [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/24

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `_transferCollateral()` function in `ToriMinting` does not verify the actual amount received after token transfers. If a fee-on-transfer token is added to the allowed token list, the protocol would receive less collateral than expected while minting the full `TrUSD` amount, resulting in under-collateralization.

Vulnerability Detail

The _transferCollateral function transfers collateral from users during minting without checking the actual received amount:

```
function _transferCollateral(
    uint128 amount,
    address asset,
    address from,
    address[] calldata addresses,
    uint128[] calldata ratios
) internal {
    if (!tokenConfig[asset].isActive) revert UnsupportedAsset();

    IERC20 token = IERC20(asset);
    uint128 totalTransferred = 0;

    for (uint128 i = 0; i < addresses.length;) {
        uint128 amountToTransfer = (amount * ratios[i]) / 10_000;
        token.safeTransferFrom(from, addresses[i], amountToTransfer);
        totalTransferred += amountToTransfer;
        // @audit No verification of actual received amount
        unchecked {
            ++i;
        }
    }

    uint128 remainingBalance = amount - totalTransferred;
    if (remainingBalance > 0) {
        token.safeTransferFrom(from, addresses[addresses.length - 1],
        ↪  remainingBalance);
```

```
    }
}
```

Scenario with Fee-on-Transfer Token:

1. Admin adds a fee-on-transfer token (e.g., USDT with fees enabled, deflationary token) to the allowed list.
2. User signs mint order: 1000 tokens collateral → 1000 TrUSD.
3. Transfer executes: 1000 tokens sent, but only 990 received by custodians (1% fee).
4. Protocol mints full 1000 TrUSD against only 990 tokens of actual collateral.
5. Protocol becomes under-collateralized by 10 TrUSD per 1000 minted.

Impact

TrUSD minted against less collateral than the signed order specifies.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/384b484f748cc0971d940aaacde6f6d2dd09d3bc/ToriMinting.sol#L280

Tool Used

Manual Review

Recommendation

Consider checking actual balance change.

Discussion

ertnec

Acknowledged - Fee-on-transfer token compatibility is handled at the backend/quote level. Token onboarding requires timelock approval ensuring proper vetting. The protocol currently only supports standard stablecoins (USDC, USDT) which do not charge transfer fees.

Issue L-2: Misleading error message in blacklist access control [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/25

Summary

Error message incorrectly suggests only blackLister can update blacklist when owner is also authorized.

Vulnerability Detail

The `updateBlackList()` function in `StakedTrUSDOFT.sol` allows both `blackLister` and `owner()` to call it, but the error message `OnlyBlackLister()` incorrectly suggests only the blackLister is authorized.

```
function updateBlackList(address _user, bool _isBlackListed) external {
    if (msg.sender != blackLister && msg.sender != owner()) revert
    ↪  OnlyBlackLister();
    blackList[_user] = _isBlackListed;
    emit BlackListUpdated(_user, _isBlackListed);
}
```

Impact

Misleading error message.

Tool Used

Manual Review

Recommendation

Rename the error to accurately reflect that both roles are authorized.

Discussion

defsec

Fix is confirmed.

Issue L-3: Route validation does not check for duplicate addresses [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/27

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `verifyRoute()` function in `ToriMinting.sol` does not check for duplicate addresses in `route.addresses`. This allows the same custodian address to appear multiple times, receiving multiple transfers that sum to more than their intended ratio.

Vulnerability Detail

The `verifyRoute()` function validates routes but does not check for duplicate addresses:

```
function verifyRoute(Route calldata route) public view returns (bool) {
    uint128 totalRatio = 0;

    // ... length checks ...

    // Validate each address is approved custodian and sum ratios
    for (uint128 i = 0; i < route.addresses.length;) {
        if (!_custodianAddresses.contains(route.addresses[i]) ||
            route.addresses[i] == address(0) ||
            route.ratios[i] == 0) {
            return false;
        }
        totalRatio += route.ratios[i];  // No check for duplicates
        unchecked { ++i; }
    }

    // Total ratio must equal 10000 (100%)
    return (totalRatio == 10_000);
}
```

The Problem:

- If the same address appears multiple times in `route.addresses`, it will receive multiple transfers.
- Each occurrence transfers `(amount * ratios[i]) / 10_000`.
- The total received by the duplicate address will be the sum of all its ratios.
- This bypasses the intended single-address-per-route.

Impact

Single custodian can receive more than intended by appearing multiple times.

Tool Used

Manual Review

Recommendation

Add duplicate address detection in `verifyRoute()`.

Discussion

ertnec

Acknowledged - The totalRatio must equal 10,000 (enforced), so duplicate addresses cannot cause more than 100% of amount to be transferred. The custodian receives exactly what the ratios dictate - duplicates only waste gas. Additionally, only trusted COLLATERAL_MANAGER_ROLE can call this function.

Issue L-4: Missing validation allows rate limit window to be set to zero [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/28

Summary

The `RateLimiter` contract's `_setRateLimits()` function does not validate that the `window` parameter is greater than zero. When `window` is set to 0, the rate limiting mechanism is effectively disabled, allowing unlimited cross-chain transfers.

Vulnerability Detail

The `_setRateLimits()` function in `RateLimiter.sol` allows setting rate limit configurations without validating that `window > 0`:

```
function _setRateLimits(RateLimitConfig[] memory _rateLimitConfigs) internal {
    unchecked {
        for (uint256 i = 0; i < _rateLimitConfigs.length; i++) {
            RateLimit storage rl = rateLimits[_rateLimitConfigs[i].dstEid];

            // Update limit and window, preserve amountInFlight/lastUpdated
            rl.limit = _rateLimitConfigs[i].limit;
            rl.window = _rateLimitConfigs[i].window;  // No validation that window
            ↪  > 0
        }
    }
    emit RateLimitsChanged(_rateLimitConfigs);
}
```

In `_amountCanBeSent()`, when `window == 0`, the condition `timeSinceLastDeposit >= _window` is always true (since `timeSinceLastDeposit` is always >= 0):

```
function _amountCanBeSent(
    uint256 _amountInFlight,
    uint256 _lastUpdated,
    uint256 _limit,
    uint256 _window
) internal view returns (uint256 currentAmountInFlight, uint256 amountCanBeSent) {
    uint256 timeSinceLastDeposit = block.timestamp - _lastUpdated;

    if (timeSinceLastDeposit >= _window) {  // Always true when window == 0
        // Window fully elapsed, reset to zero
        currentAmountInFlight = 0;
        amountCanBeSent = _limit;  // Always returns full limit
    } else {
```

```
        // Linear decay calculation
        uint256 decay = (_limit * timeSinceLastDeposit) / _window;
        // ...
    }
}
```

Impact

If an admin accidentally sets `window` to 0, rate limiting is completely bypassed.

Code Snippet

```
function _setRateLimits(RateLimitConfig[] memory _rateLimitConfigs) internal {
    unchecked {
        for (uint256 i = 0; i < _rateLimitConfigs.length; i++) {
            RateLimit storage rl = rateLimits[_rateLimitConfigs[i].dstEid];

            // Update limit and window, preserve amountInFlight/lastUpdated
            rl.limit = _rateLimitConfigs[i].limit;
            rl.window = _rateLimitConfigs[i].window;  // No validation that window
            ↪  > 0
        }
    }
    emit RateLimitsChanged(_rateLimitConfigs);
}
```

Tool Used

Manual Review

Recommendation

Add validation in `_setRateLimits()` to ensure `window > 0`.

Discussion

defsec

Fix is confirmed.

Issue L-5: Owner can be blacklisted in StakedTrUSDOFT and causing cross-chain transfers to fail [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/29

Summary

Unlike StakedTrUSD which has the notOwner(target) modifier preventing the owner from being blacklisted, StakedTrUSDOFT has no such protection. If the owner gets blacklisted, cross-chain transfers will revert because the fallback redirect to owner will fail.

Vulnerability Detail

In StakedTrUSD.sol, blacklisting the owner is explicitly prevented:

```
modifier notOwner(address target) {
    if (target == owner()) revert CantBlacklistOwner();
    _;
}

function addToBlacklist(address target, bool isFullBlacklisting)
    external
    onlyRole(BLACKLIST_MANAGER_ROLE)
    notOwner(target)  // Prevents blacklisting owner
{
    // ...
}
```

However, in StakedTrUSDOFT.sol, there is no such protection:

```
function updateBlackList(address _user, bool _isBlackListed) external {
    if (msg.sender != blackLister && msg.sender != owner()) revert
    ↪  OnlyBlackListerOrOwner();
    blackList[_user] = _isBlackListed;  // No check if _user is owner
    emit BlackListUpdated(_user, _isBlackListed);
}
```

Impact

If owner is blacklisted, no funds can be received cross-chain for any blacklisted recipient.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/69407f43457189fda9e3a7ce2e1da95d1185de20/StakedTrUSDOFT.sol#L48

Tool Used

Manual Review

Recommendation

Add owner protection matching `StakedTrUSD`:

```
error CantBlacklistOwner();

function updateBlackList(address _user, bool _isBlackListed) external {
    if (msg.sender != blackLister && msg.sender != owner()) revert
    ↪  OnlyBlackListerOrOwner();
    if (_user == owner()) revert CantBlacklistOwner();  // Add this check
    blackList[_user] = _isBlackListed;
    emit BlackListUpdated(_user, _isBlackListed);
}
```

Discussion

defsec

Fixed with adding a missing check.

Issue L-6: Event parameter order inconsistency [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/31

Summary

Event definitions use inconsistent parameter ordering , some use `(oldValue, newValue)` while others use `(newValue, oldValue)`.

Vulnerability Detail

The codebase has inconsistent conventions for event parameter ordering:

Events using (old, new) convention:

```
// ToriMinting.sol:128
event MaxMintPerBlockChanged(uint128 oldMaxMintPerBlock, uint128
↪  newMaxMintPerBlock, address indexed asset);

// ToriMinting.sol:129
event MaxRedeemPerBlockChanged(uint128 oldMaxRedeemPerBlock, uint128
↪  newMaxRedeemPerBlock, address indexed asset);

// IStakedTrUSDCooldown.sol:14
event CooldownDurationUpdated(uint24 previousDuration, uint24 newDuration);
```

Events using (new, old) convention:

```
// TrUSD.sol:28
event MinterUpdated(address indexed newMinter, address indexed oldMinter);

// TrUSD.sol:29, ToriMinting.sol:153, StakedTrUSD.sol
event TimelockAdminUpdated(address indexed newTimelockAdmin, address indexed
↪  oldTimelockAdmin);
```

Impact

General Health

Tool Used

Manual Review

Recommendation

Standardize all events to use `(oldValue, newValue)` convention.

Discussion

defsec

Fix is confirmed.

Issue L-7: `RewardsReceived` event emits identical values [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/32

Summary

The `RewardsReceived` event emits the same value for both parameters, making the second parameter redundant.

Vulnerability Detail

In `StakedTrUSD.sol`, the `transferInRewards` function emits:

```
// StakedTrUSD.sol:117-122
function transferInRewards(uint256 amount) external nonReentrant
↪   onlyRole(REWARDER_ROLE) notZero(amount) {
    _updateVestingAmount(amount);
    IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);

    emit RewardsReceived(amount, amount);  // Both parameters are identical
}
```

The event is defined as:

```
// IStakedTrUSD.sol:7
event RewardsReceived(uint256 indexed amount, uint256 newVestingTrUSDAmount);
```

Impact

Redundant event parameter wastes gas.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/main/Tori-Finance__contracts/contracts/core/StakedTrUSD.sol#L121

Tool Used

Manual Review

Recommendation

Remove the redundant parameter.

Discussion

defsec

Fix is confirmed.

Issue L-8: Cooldown duration changes create unfair waiting periods [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/33

Summary

When admin reduces `cooldownDuration`, users who already started their cooldown must still wait for their original (longer) duration, while new users get the shorter duration.

Vulnerability Detail

When a user calls `cooldownAssets()` or `cooldownShares()`, their `cooldownEnd` is set based on the **current** `cooldownDuration`:

```
// StakedTrUSD.sol:225-237
function cooldownAssets(uint256 assets) external ensureCooldownOn returns (uint256)
↪  {
    if (assets > maxWithdraw(msg.sender)) revert ExcessiveWithdrawAmount();

    uint256 shares = previewWithdraw(assets);

    StakedTrUSDStorage storage $ = _getStakedTrUSDStorage();
    $.cooldowns[msg.sender].cooldownEnd = uint104(block.timestamp) +
    ↪  $.cooldownDuration;  // Set at call time
    $.cooldowns[msg.sender].underlyingAmount += assets;

    _withdraw(msg.sender, address($.silo), msg.sender, assets, shares);

    return shares;
}
```

The `unstake()` function checks the user's stored `cooldownEnd`:

```
// StakedTrUSD.sol:211-223
function unstake(address receiver) external {
    StakedTrUSDStorage storage $ = _getStakedTrUSDStorage();
    UserCooldown storage userCooldown = $.cooldowns[msg.sender];
    uint256 assets = userCooldown.underlyingAmount;

    if (block.timestamp >= userCooldown.cooldownEnd || $.cooldownDuration == 0) {
        userCooldown.cooldownEnd = 0;
        userCooldown.underlyingAmount = 0;
        $.silo.withdraw(receiver, assets);
    } else {
        revert InvalidCooldown();
```

```
    }
}
```

Scenario

1. `cooldownDuration = 90 days`
2. **Day 0**: Alice calls `cooldownAssets(1000)` → `cooldownEnd = Day 0 + 90 days = Day 90`
3. **Day 5**: Admin reduces `cooldownDuration` to `7 days`
4. **Day 5**: Bob calls `cooldownAssets(1000)` → `cooldownEnd = Day 5 + 7 days = Day 12`
5. **Day 12**: Bob can `unstake()` => OK
6. **Day 12**: Alice cannot `unstake()` - must wait until Day 90 X

Impact

Early stakers penalized compared to later stakers.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/47acdd95010012d2df09400b6313f6c44e1e6b4d/Tori-Finance__contracts/contracts/core/StakedTrUSD.sol#L260-L261

Tool Used

Manual Review

Recommendation

Allow users to ”re-cooldown” to get the new duration or use `min(userCooldown.cooldownEnd, block.timestamp + cooldownDuration)` in unstake check.

Discussion

defsec

Hi @ertnec , are we missing a fix for that one?

ertnec

Hi @defsec ,

https://github.com/Tori-Finance/contracts/pull/16

I had attached this PR to both #33 and #36. Maybe it caused an error somehow in dashboard.

Issue L-9: Blacklist transactions can be frontrun [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/34

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Malicious actors can evade blacklisting by monitoring the public mempool and front-running `addToBlacklist` transactions by transferring their tokens to a new address.

Vulnerability Detail

The `addToBlacklist()` function is a standard on-chain transaction visible in the public mempool:

```
// StakedTrUSD.sol:135-142
function addToBlacklist(address target, bool isFullBlacklisting)
    external
    onlyRole(BLACKLIST_MANAGER_ROLE)
    notOwner(target)
{
    bytes32 role = isFullBlacklisting ? FULL_RESTRICTED_STAKER_ROLE :
    ↪  SOFT_RESTRICTED_STAKER_ROLE;
    _grantRole(role, target);
}
```

Impact

1. Admin identifies a malicious user and decides to blacklist their address.
2. Admin sends transaction: `addToBlacklist(maliciousUser, true)`.
3. Malicious user monitors Ethereum mempool and sees the pending transaction.
4. Malicious user front-runs with higher gas: `transfer(newAddress, balanceOf(maliciousUser))`.
5. Front-run tx lands first - tokens moved to `newAddress`.
6. Admin tx lands `maliciousUser` is blacklisted but has 0 balance.
7. Malicious user controls tokens at `newAddress` which is not blacklisted.

Tool Used

Manual Review

Recommendation

All admin transactions (especially blacklisting) should be submitted through private transaction services.

Discussion

ertnec

Acknowledged - Will use Flashbots/private mempool for admin transactions.

Issue L-10: No way to cancel or reduce cooldown requests [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/36

Summary

Once a user initiates a cooldown via `cooldownAssets()` or `cooldownShares()`, there is no way to cancel the request, reduce the amount, or re-stake the assets. Users are locked into waiting for the full cooldown period.

Vulnerability Detail

The cooldown functions only allow increasing the `underlyingAmount`:

```
// StakedTrUSD.sol:225-237
function cooldownAssets(uint256 assets) external ensureCooldownOn returns (uint256)
 ↪  {
    if (assets > maxWithdraw(msg.sender)) revert ExcessiveWithdrawAmount();

    uint256 shares = previewWithdraw(assets);

    StakedTrUSDStorage storage $ = _getStakedTrUSDStorage();
    $.cooldowns[msg.sender].cooldownEnd = uint104(block.timestamp) +
     ↪  $.cooldownDuration;
    $.cooldowns[msg.sender].underlyingAmount += assets;  // Only increases!

    _withdraw(msg.sender, address($.silo), msg.sender, assets, shares);  // Assets
     ↪  sent to silo

    return shares;
}

// StakedTrUSD.sol:239-251
function cooldownShares(uint256 shares) external ensureCooldownOn returns (uint256)
 ↪  {
    if (shares > maxRedeem(msg.sender)) revert ExcessiveRedeemAmount();

    uint256 assets = previewRedeem(shares);

    StakedTrUSDStorage storage $ = _getStakedTrUSDStorage();
    $.cooldowns[msg.sender].cooldownEnd = uint104(block.timestamp) +
     ↪  $.cooldownDuration;
    $.cooldowns[msg.sender].underlyingAmount += assets;  // Only increases!

    _withdraw(msg.sender, address($.silo), msg.sender, assets, shares);  // Assets
     ↪  sent to silo
```

```
    return assets;
}
```

The `TrUsdSilo` contract only has a `withdraw` function callable by the staking vault:

```
// TrUsdSilo.sol:29-31
function withdraw(address to, uint256 amount) external onlyStakingVault {
    trUSD.safeTransfer(to, amount);
}
// No user-facing cancel function
```

Step	Action	Result
1	User calls `cooldownAssets(10000)`	10,000 TrUSD sent to silo, shares burned
2	Market conditions change, user wants to re-stake	**No option available**
3	User must wait 7 days (default cooldown)	Opportunity cost, no yield during cooldo
4	After cooldown, user calls `unstake()`	Finally receives TrUSD
5	User deposits again	Gets new shares at potentially worse rat

Impact

Users cannot change their mind after initiating cooldown.

Code Snippet

https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/blob/main/Tori-Finance__contracts/contracts/core/StakedTrUSD.sol#L225

Tool Used

Manual Review

Recommendation

Consider adding a `cancelCooldown()` function.

Issue L-11: Blacklist manager can remove themselves from blacklist [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/37

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

A `BLACKLIST_MANAGER_ROLE` holder who gets blacklisted with `FULL_RESTRICTED_STAKER_ROLE` can call `removeFromBlacklist()` to remove themselves, defeating the freeze mechanism's intent.

Vulnerability Detail

```
// StakedTrUSD.sol:144-151
function removeFromBlacklist(address target, bool isFullBlacklisting)
    external
    onlyRole(BLACKLIST_MANAGER_ROLE)  // Only checks for manager role
    notOwner(target)                   // Only protects owner
{
    bytes32 role = isFullBlacklisting ? FULL_RESTRICTED_STAKER_ROLE :
    ↪  SOFT_RESTRICTED_STAKER_ROLE;
    _revokeRole(role, target);  // Can pass self as target!
}
```

Scenario

Step	Action	Result
1	Alice has BLACKLIST_MANAGER_ROLE	Can blacklist others
2	Admin blacklists Alice with FULL_RESTRICTED_STAKER_ROLE	Alice should be frozen
3	Alice calls `removeFromBlacklist(alice, true)`	Succeeds! Alice removes hers
4	Alice transfers her tokens	Escapes restriction

Impact

Blacklist managers cannot be effectively frozen.

Tool Used

Manual Review

Recommendation

Prevent blacklisted users from calling blacklist functions.

Discussion

ertnec

Acknowledged - The BLACKLIST_MANAGER_ROLE will be assigned exclusively to trusted, protocol-controlled addresses.

Issue L-12: Compilation Failure Due to ReentrancyGuardUpgradeable Removal in OpenZeppelin v5.5 [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/38

Summary

The project utilizes the OpenZeppelin Contracts Upgradeable library. Due to OpenZeppelin removing `ReentrancyGuardUpgradeable` (or modifying its path/implementation) in version v5.5.0, and Foundry potentially pulling this latest incompatible version, `StakedTrUSD.sol` fails to compile.

Vulnerability Detail

In `StakedTrUSD.sol`, the contract imports `ReentrancyGuardUpgradeable`:

```
import "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
```

According to the OpenZeppelin v5.5.0 release notes, changes in the library structure have made the above import path invalid or the file unavailable.

Even if `package.json` defines a version range, Foundry's dependency management (via git submodules or remappings) may fetch the latest v5.5.0 version if not strictly locked. This results in build failures during compilation due to missing dependencies or incompatibility.

Impact

The project cannot compile, making it impossible to run tests or deploy the contracts.

Code Snippet

```
import "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
```

Tool Used

Manual Review

Recommendation

It is recommended to take one of the following actions:

1. **Lock Dependency Version**: Strictly lock `@openzeppelin/contracts-upgradeable` to a compatible version (e.g., v5.4.0) in both `package.json` and Foundry configuration to ensure it does not auto-upgrade to v5.5.0.
2. **Adapt to New Version**: If v5.5.0 is required, update the code according to the OpenZeppelin migration guide to use the new import paths or storage patterns.

Issue L-13: rescueOrphanFunds fails to reset vesting-Amount causing temporary DoS [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/41

Summary

The `rescueOrphanFunds` function in `StakedTrUSD` extracts idle funds but fails to reset `vestingAmount`. This causes the `totalAssets()` function to revert due to integer underflow. Consequently, users cannot deposit funds during the remaining vesting period.

Vulnerability Detail

The `StakedTrUSD` contract uses `rescueOrphanFunds` to allow the admin to withdraw all underlying assets when `totalSupply() == 0`. However, this function only transfers the asset balance but fails to reset the `vestingAmount` variable (which tracks unvested rewards) to zero.

The logic for calculating total assets in `StakedTrUSD` is as follows:

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

If `rescueOrphanFunds` is called while there are still unvested rewards (`getUnvestedAmount() > 0`):

1. `IERC20(asset()).balanceOf(address(this))` becomes `0`.
2. `getUnvestedAmount()` remains a positive number.
3. `totalAssets()` executes `0 - positive_number`, causing an EVM arithmetic underflow and revert.

Since the ERC4626 `deposit` and `mint` flows rely on `totalAssets()` to calculate exchange rates, this causes a Denial of Service (DoS) where the contract cannot process any new deposits until the vesting period ends (up to 8 hours).

Impact

After the admin executes the rescue, the contract becomes non-functional for the remainder of the vesting period (`VESTING_PERIOD`, default 8 hours). During this time, no users can deposit funds.

Code Snippet

```
function rescueOrphanFunds(address to) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (totalSupply() != 0) revert OperationNotAllowed();
    if (to == address(0)) revert InvalidZeroAddress();
    uint256 balance = IERC20(asset()).balanceOf(address(this));
    if (balance == 0) revert NoOrphanFunds();
    IERC20(asset()).safeTransfer(to, balance); // @audit-issue Asset removed but
    ↪  vestingAmount not reset
    emit OrphanFundsRescued(to, balance);
}
```

Tool Used

Manual Review

Recommendation

In the `rescueOrphanFunds` function, both `vestingAmount` and `lastDistributionTimestamp` should be reset to ensure consistency between the state and the asset balance.

Issue L-14: `removeAllowedToken` inadvertently blocks users from redeeming collateral, causing fund freezing [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-tori-finance-jan-15th/issues/43

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `removeAllowedToken` function in `ToriMinting` is used to remove supported tokens. However, the function uses the `delete` keyword to clear the token configuration, resetting `isActive` to `false`. Since the `redeem` function requires `isActive` to be `true` during order verification, this action prevents all users holding the asset from redeeming, effectively freezing user funds.

Vulnerability Detail

In `ToriMinting.sol`, the `removeAllowedToken` function is intended to remove a token from the allowed list. It uses `delete tokenConfig[_token]`, which resets all fields in the struct to their default values, including setting `isActive` to `false`.

```
function removeAllowedToken(address _token) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (_token == address(0)) revert ZeroAddress();
    if (!tokenConfig[_token].isActive) revert TokenNotAllowed(_token);
    delete tokenConfig[_token]; // @audit-issue The delete operation resets
    ↪  isActive to false
    emit TokenRemoved(_token);
}
```

When users call the `redeem` function, `verifyOrder` is invoked for checks. `verifyOrder` explicitly requires the token to be active:

```
function verifyOrder(Order calldata order, Signature calldata signature) public
↪  view returns (bytes32) {
    // ...
    if (!tokenConfig[order.collateral_asset].isActive) revert
    ↪  TokenNotAllowed(order.collateral_asset);
    // ...
}
```

Therefore, once an admin calls `removeAllowedToken` to stop new business for a token, the redemption channel for that token is also immediately cut off.

Impact

When the admin removes support for a token, users who have deposited that token cannot retrieve their collateral via the `redeem` function. Although the admin can restore access by re-adding the token, user funds are effectively frozen in this state, breaking the protocol's exit mechanism.

Code Snippet

Tool Used

Manual Review

Recommendation

It is recommended to modify the logic of `removeAllowedToken` to only disable `mint` operations while retaining `redeem` permissions, allowing users to withdraw funds. Alternatively, introduce a new state (e.g., `isRedeemable`) to separate minting and redemption permissions.

Discussion

ertnec

Acknowledged - Users can redeem with any other supported collateral token. The protocol supports multiple collateral types (USDC, USDT ...), allowing users to exit even if one token is removed.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.