Pashov Audit Group

Theo Security Review

March 19th 2026 - March 20th 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Theo

Theo is a protocol that enables permissioned minting of ThUSD, a USD-pegged stablecoin backed by stablecoin collateral such as USDC and USDT. Mint and redeem operations are restricted to addresses holding the MINTER_ROLE, with exchange rates determined off-chain by a backend server.

5. Executive Summary

A time-boxed security review of the **theo-network/contracts-v2** repository was done by Pashov Audit Group, during which **montecristo, Klaus, 0xunforgiven** engaged to review **Theo**. A total of **7** issues were uncovered.

Protocol Summary	
Project Name	Theo
Protocol Type	USD stablecoin minting
Timeline	March 19th 2026 - March 20th 2026

Review commit hash:
- 340318a11380008e91580689e8b81b5e5b1ea0bf (theo-network/contracts-v2)

Fixes review commit hash:
- 6889ae088b30a4b80f76c36feed465e9df410fc0 (theo-network/contracts-v2)

Scope

`ThUSDMinter.sol`

6. Findings

Findings count

Severity	Amount
Low	7
Total findings	**7**

Summary of findings

ID	Title	Severity	Status
[L-01]	`MAX_FEE_BPS` hardcoded as constant limits fee adjustment during depeg events	Low	Acknowledged
[L-02]	`mintDestination` or `redeemDestination` enables ThUSD issuance without collateral	Low	**Resolved**
[L-03]	Redemption fee is untrackable on chain	Low	**Resolved**
[L-04]	Attacker can spam tiny mint redeem requests to cause gas waste from minters	Low	**Resolved**
[L-05]	Redemption fee rounded down in favor of the user	Low	**Resolved**
[L-06]	Exchange ratio fails to account for asset depegging or price change	Low	Acknowledged
[L-07]	Missing backend signature validation for exchange rate and expiry	Low	Acknowledged

Low findings

[L-01] `MAX_FEE_BPS` hardcoded as constant limits fee adjustment during depeg events

Description

`MAX_FEE_BPS` is declared as a `constant` at 10 (0.1%). USD-pegged collateral tokens have historically depegged far beyond 0.1%:

- USDC: dropped to $0.88–0.90 during the SVB collapse
- USDT: dropped to $0.96 during the Terra/LUNA crash

The ratio check in `_validateOrder()` enforces `thusdAmount <= collateralAmount` (mint) and `collateralAmount <= thusdAmount` (redeem) on a normalized amount basis, with no price oracle. When a collateral token depegs, an attacker buys the depegged token cheaply, passes the amount-based ratio check, mints ThUSD, and redeems for a non-depegged collateral token. For example, during a 10% depeg, an attacker deposits 1M USDC (worth $900K) and mints 1M ThUSD — the ratio check passes because 1M <= 1M by amount, despite the value gap. A subsequent redeem into USDT yields approximately 999K USDT ($999K) after the 0.1% fee, netting approximately $99K profit. The 0.1% fee is negligible against a 5-10% depeg spread, and the admin has no on-chain tool to raise the fee ceiling during such events.

Make `MAX_FEE_BPS` a state variable configurable by the admin, or raise it to a reasonable level.

Acknowledgement Comment

The fee is primarily intended as a processing fee for closing positions to obtain collateral tokens, not as a mechanism to handle depegging events. The contract is designed so that only addresses with the MINTER_ROLE can call mint and redeem, with all amounts determined by the Theo backend system. In cases where a collateral token depegs, the backend system will adjust the exchange ratios accordingly, and in cases of a significant depeg, the minter contract will be paused to prevent further operations.

[L-02] `mintDestination` or `redeemDestination` enables ThUSD issuance without collateral

Description

`_transferCollateralIn()` transfers collateral from `order.signer` to `mintDestination`, and `_transferCollateralOut()` transfers collateral from `redeemDestination` to `order.recipient`. When `mintDestination` or `redeemDestination` appears on both sides of these transfers, the collateral transfer becomes a self-transfer with no net balance change, while ThUSD is still minted or burned.

```
// ThUSDMinter.sol:216-218
function _transferCollateralIn(Order calldata order) internal {
    IERC20(order.collateral_asset).safeTransferFrom(order.signer, mintDestination,
order.collateral_amount);
}
```

```
// ThUSDMinter.sol:220-224
function _transferCollateralOut(Order calldata order) internal {
    uint256 fee = (order.collateral_amount * redeemFeeBps) / 10_000;
    uint256 amountAfterFee = order.collateral_amount - fee;
    IERC20(order.collateral_asset).safeTransferFrom(redeemDestination, order.recipient,
amountAfterFee);
}
```

Two self-transfer scenarios exist:

1. Mint with `signer == mintDestination`: collateral self-transfers within `mintDestination`, ThUSD is minted to the recipient. No new collateral enters the system.
2. Redeem with `recipient == redeemDestination`: collateral self-transfers within `redeemDestination`, ThUSD is burned from the signer. No collateral leaves the system.

Add checks in `_validateOrder()` that `mintDestination` and `redeemDestination` cannot be the signer or recipient.

[L-03] Redemption fee is untrackable on chain

Description

`_transferCollateralOut()` computes a fee and subtracts it from the collateral transferred to the recipient. The fee amount is not stored in any state variable and is not emitted in any event. The `Redeem` event logs only `order.collateral_amount` (the pre-fee amount).

```
// ThUSDMinter.sol:220-223
function _transferCollateralOut(Order calldata order) internal {
    uint256 fee = (order.collateral_amount * redeemFeeBps) / 10_000;
    uint256 amountAfterFee = order.collateral_amount - fee;
```

```
    IERC20(order.collateral_asset).safeTransferFrom(redeemDestination, order.recipient,
amountAfterFee);
}
```

```
// ThUSDMinter.sol:99
emit Redeem(order.signer, order.recipient, order.collateral_asset, order.collateral_amount,
order.thusd_amount);
```

The fee silently remains in `redeemDestination` with no record distinguishing it from the general collateral pool. It is impossible to distinguish between protocol fee revenue and the reserve amount of `redeemDestination`.

Track fees on-chain and emit fee information in the `Redeem` event.

[L-04] Attacker can spam tiny mint redeem requests to cause gas waste from minters

Description

There is no minimum mint/redeem amount. As such, an attacker can spam lots of 1 wei mint/redeem requests to cause gas waste for minters.

[L-05] Redemption fee rounded down in favor of the user

Description

In the `_transferCollateralOut` function, the protocol fee is calculated using standard integer division:

```
uint256 fee = (order.collateral_amount * redeemFeeBps) / 10_000;
```

In Solidity, integer division always rounds down. This means the calculated fee will be slightly less than the intended mathematical value, and in cases where `collateral_amount * redeemFeeBps` is less than 10,000, the fee will be zero. While the "dust" lost per transaction is small, it consistently favors the user over the protocol and can accumulate over a high volume of transactions. To fix this issue, modify the fee calculation to round up in favor of the protocol.

[L-06] Exchange ratio fails to account for asset depegging or price change

Description

The `_validateOrder` function enforces <=1 price ratio (normalized for decimals) between collateral assets (like USDC/USDT) and ThUSD.

```
if (order.order_type == OrderType.MINT) {
    if (thusdAmountNormalized > collateralAmountNormalized) revert InvalidAssetRatio();
}
```

This logic assumes that the collateral asset will always maintain a value of exactly $1. If a supported stablecoin (e.g., USDT) depegs and its market value drops to $0.90, the contract will still allow users to mint 1 ThUSD (valued at $1) for 1 USDT (valued at $0.90), leading to immediate protocol undercollateralization and potential drainage of the `redeemDestination` pool by arbitrageurs.

Acknowledgement Comment

Similar to L-01, in cases where the depeg of a collateral token is too significant, the minter contract will be paused to stop operations. The MINTER_ROLE is also responsible for verifying the correct USD value of the collateral tokens being used, providing an additional layer of protection against depeg-related risks.

[L-07] Missing backend signature validation for exchange rate and expiry

Description

The documentation states that a "backend server determines exchange rates off-chain" and "constructs an Order." However, the `_validateOrder` function only verifies that the signature matches the `order.signer` (the user). Since the user provides the Order struct and the signature, a malicious user can bypass the backend entirely. They can craft an order with a manipulated `thusd_amount` or `collateral_amount` (an extremely favorable exchange rate) and a very long expiry, sign it themselves, and submit it. The contract will accept it as long as the user is whitelisted, because there is no on-chain check to verify that the backend authorized those specific parameters.

```
        // check order hash and signature
        bytes32 orderHash = hashOrder(order);
        if (usedOrderHashes[orderHash]) revert OrderAlreadyUsed();

        address recovered = ECDSA.recover(orderHash, signature);
        if (recovered != order.signer) revert InvalidSignature();
```

The backend should sign the orderHash too, and the contract should verify this second signature against a known authorized `backend_signer` address before executing the mint or redeem.

Acknowledgement Comment

The backend system verifies that the order exchange rates are correct before forwarding the transaction. A malicious user would need to have the MINTER_ROLE assigned to them and be whitelisted in order for this attack to succeed. Since the backend system holds the

MINTER_ROLE and validates the order exchange rate before calling the minter contract, an order with an invalid exchange rate would never be submitted on-chain. Additionally, the backend verifies that the original order from step one is exactly the same as the signed order with the final ThUSD and collateral amounts provided in the final submission.