



Security Review For Zoth



Collaborative Audit Prepared For:
Lead Security Expert(s):

Zoth
0x37

Date Audited:

0xTheBlackPanther
August 11 - August 25, 2026

Introduction

zVaults are Zoth's on-chain infrastructure for institutional-grade, yield-generating investment strategies. Each vault issues an ERC-20 compliant, interest-bearing zTOKEN, representing a proportional share of the vault's net asset value (NAV). zVaults enable capital deployment across RWA, DeFi, private credit, lending, and other mandate-specific strategies, with each vault operating under predefined parameters and risk controls.

The architecture combines institutional fund-management structures with on-chain transparency, composability, and settlement, while providing the underlying infrastructure for asset management and vault operations.

Documentation: <https://docs.zoth.io/zoth/zvaults/what-are-zvaults>

Scope

Repository: MoveIndustries/vault-movement

Audited Commit: 2fb43f00699853446761000c539bbc013249e19c

Final Commit: f4ee5e72bca220f3c8a9c77e21d1619d983fea75

Files:

- move/zvault_core/sources/access_control.move
- move/zvault_core/sources/compliance.move
- move/zvault_core/sources/deposit_vault.move
- move/zvault_core/sources/redemption_vault.move
- move/zvault_core/sources/vault_base.move
- move/zvault_core/sources/zopal_token.move
- move/zvault_oracle/sources/oracle_access_control.move
- move/zvault_oracle/sources/price_oracle.move

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
0	6	1

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Blacklisted or sanctioned spenders can use existing zOPAL allowances [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-zoth-move-vaults-aug-5th-2026/issues/46>

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

Summary

zOPAL blocks sanctioned and blacklisted token owners and recipients, but does not validate the spender in delegated `transferFrom()` calls.

Vulnerability Detail

A user can approve a spender while the spender is compliant. If that spender is later blacklisted or sanctioned, the spender can still consume the remaining allowance through `transferFrom()`.

The compliance hook checks only `from` and `to`, it never checks `msg.sender`, the account executing the delegated transfer.

Impact

A blocked address can continue moving zOPAL using previously granted allowances, bypassing the intended transfer restrictions. The issue is limited to allowances that were already approved by token owners.

Code Snippet

zOPAL.sol

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal override {
    if (from != address(0)) {
        _requireNotSanctioned(from);
        _onlyNotBlacklisted(from);
    }

    if (to != address(0)) {
        _requireNotSanctioned(to);
        _onlyNotBlacklisted(to);
    }
}
```

```
}  
}
```

OpenZeppelin's inherited `transferFrom()` uses `msg.sender` as the spender but does not pass it to this hook:

[ERC20Upgradeable.sol](#)

```
address spender = _msgSender();  
_spendAllowance(from, spender, amount);  
_transfer(from, to, amount);
```

Tool Used

Manual review

Recommendation

Override `transferFrom()` and validate `_msgSender()` against the sanctions list and blacklist before calling the inherited implementation:

```
function transferFrom(  
    address from,  
    address to,  
    uint256 amount  
) public override returns (bool) {  
    _requireNotSanctioned(_msgSender());  
    _onlyNotBlacklisted(_msgSender());  
  
    return super.transferFrom(from, to, amount);  
}
```

Discussion

aryaparas

The blacklist and the sanction list is not to determine who can do transactions which involve ZOPAL . It is only to make sure that ZOPAL is not sent from a blacklisted wallet address or sent to a blacklisted wallet address. In this case, the TO and FRO not being blacklisted/sanctioned suffice.

pantheraudits

Thanks for clarifying, since compliance intentionally applies only to `from` and `to`, this behavior is expected, so I withdraw the finding then.

Issue M-2: Queued approval settlement bypasses the vault pause [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-08-zoth-move-vaults-aug-5th-2026/issues/48>

Summary

Queued approval functions do not check the vault's global or per-function pause state. As a result, pending requests can still mint, burn, and settle funds after the vault has been paused.

Vulnerability Detail

User-facing request creation checks the pause state, but approval functions only check administrator authorization.

e.g Solidity `safeApproveRequest()` and `approveRequest()` lack a pause modifier:

DepositVault.sol:357

```
function safeApproveRequest(
    uint256 requestId,
    uint256 newOutRate
) external onlyVaultAdmin {
    _approveRequest(requestId, newOutRate, true, true);
}
```

The shared approval routine still mints zOPAL:

DepositVault.sol:589

```
zToken.mint(request.sender, amountZToken);
```

The same pattern exists in Move also, approval functions authenticate the administrator but do not call `vault_base::assert_fn_not_paused()` before settlement. The approval path can therefore continue minting or burning while paused.

Impact

During an emergency pause, normal administrator processing can still finalize previously queued operations. Queued deposits may mint zOPAL, while queued redemptions may burn escrowed zOPAL and pay USDC.

This weakens the emergency containment mechanism and allows pending queues to continue changing supply and reserves during an incident. Because the requests are

already funded, this is an emergency-control failure rather than an unbacked-mint issue.

Code Snippet

Move deposit approval lacks a pause check:

[deposit_vault.move:315](#)

Its settlement routine mints while no pause assertion is performed:

[deposit_vault.move:734](#)

Tool Used

Manual Review

Recommendation

Add the appropriate pause check to every queued approval and bulk-approval entrypoint, or enforce it inside the shared settlement routine:

```
whenFnNotPaused(_DEPOSIT_REQUEST_SELECTOR)
```

For Move, call

before minting, burning, paying, or finalizing the request. The same check should cover both deposit & redemption approval paths.

Issue M-3: Move queued deposit approval mints zOPAL to recipients who become non-compliant [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-zoth-move-vaults-aug-5th-2026/issues/49>

Summary

Move queued deposits check compliance only when the request is created. Approval later mints zOPAL to the stored recipient without checking the recipient's current compliance status.

Vulnerability Detail

`do_deposit_request()` validates the sender and custom recipient at request creation:

`deposit_vault.move:607`

However, `approve_request_internal()` only checks request status, tolerance, and supply cap before minting:

`deposit_vault.move:714`

The stored recipient is therefore not revalidated at the actual mint boundary.

Impact

A recipient who becomes blacklisted or sanctioned after creating a queued deposit can still receive newly minted zOPAL through normal administrator approval. This defeats the protocol's intended compliance restriction at the value-creation boundary.

Code Snippet

The mint occurs at `deposit_vault.move:740`.

Tool Used

Manual Review

Recommendation

Re-run the current compliance check against `request.sender` inside `approve_request_internal()` immediately before minting and finalizing the request.

Issue M-4: Rejected Deposit Requests Permanently Consume Deposit Allowance [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-08-zoth-move-vaults-aug-5th-2026/issues/53>

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

Summary

Rejecting a deposit request does not restore the token-specific on-chain deposit allowance consumed when the request was created. An attacker can use the off-chain refund process to repeatedly consume a finite global allowance and block subsequent deposits of that asset.

Vulnerability Detail

Each payment token has a protocol-managed internal allowance stored in `tokensConfig[token].allowance`. This is not the user's ERC-20 approval; it is a vault-level capacity limit configured by the vault admin.

When a user creates a deposit request, `_depositRequest()` calls `_calcAndValidateDeposit()`. That function consumes the token's configured allowance before the request is created:

```
_requireAndUpdateAllowance(tokenIn, amountToken);
```

For a finite allowance, `_requireAndUpdateAllowance()` deducts the requested amount:

```
require(prevAllowance >= amount, "MV: exceed allowance");
tokensConfig[token].allowance -= amount;
```

If the request is later rejected, `rejectRequest()` only marks it as cancelled:

```
mintRequests[requestId].status = RequestStatus.Canceled;
```

It does not restore the amount deducted from `tokensConfig[token].allowance`.

The protocol refunds rejected users through its existing off-chain process. However, this refund does not alter the vault's on-chain internal allowance. An attacker can submit requests that are subsequently rejected, recover their stablecoins through the off-chain refund process, and repeat the process until the finite allowance is exhausted. Unless an admin manually calls `changeTokenAllowance()`, each rejected request permanently consumes configured deposit capacity.

This issue applies only when the payment token has a finite configured allowance. Tokens configured with `MAX_UINT` are not decremented by `_requireAndUpdateAllowance()`.

Impact

An attacker can exhaust a payment token's finite on-chain deposit allowance while recovering the deposited assets through the off-chain refund process. Once exhausted, all subsequent deposits and deposit requests using that token revert with `MV: exceed allowance`.

This creates a protocol-wide denial of service for deposits of the affected asset and requires manual administrator intervention to restore capacity. It is particularly relevant where the allowance is used as a daily, strategy, or asset-exposure limit.

Code Snippet

zVault/contracts/DepositVault.sol

```
function _depositRequest(
    address tokenIn,
    uint256 amountToken,
    address recipient
) private returns (uint256 requestId, CalcAndValidateDepositResult memory
    ↪ calcResult) {
    // ...
    calcResult = _calcAndValidateDeposit(user, tokenIn, amountToken, false);
    // ...
}
```

zVault/contracts/DepositVault.sol

```
function rejectRequest(uint256 requestId) external onlyVaultAdmin {
    Request memory request = mintRequests[requestId];

    require(request.sender != address(0), "DV: request not exist");
    require(request.status == RequestStatus.Pending, "DV: request not pending");

    mintRequests[requestId].status = RequestStatus.Canceled;

    emit RejectRequest(requestId, request.sender);
}
```

zVault/contracts/abstract/ManageableVault.sol

```
function _requireAndUpdateAllowance(
    address token,
    uint256 amount
) internal {
    uint256 prevAllowance = tokensConfig[token].allowance;
    if (prevAllowance == MAX_UINT) return;
```

```
require(prevAllowance >= amount, "MV: exceed allowance");
tokensConfig[token].allowance -= amount;
}
```

Tool Used

Manual Review

Recommendation

Store the exact internal allowance amount consumed by each deposit request. When the request is rejected, restore that amount before marking the request as cancelled.

For example, add an `allowanceConsumed` field to the request structure and implement:

```
tokensConfig[request.tokenIn].allowance += request.allowanceConsumed;
```

The restoration should be skipped for tokens configured with `MAX_UINT`. If the protocol intentionally wants rejected requests to consume capacity, this behavior should be explicitly documented and the allowance should not be described or operated as available deposit capacity.

Discussion

aryaparas

We acknowledge that the reported accounting behavior is valid if request-based deposits are enabled. However, request-based deposits are disabled in our current launch configuration, and only instant deposits are supported. Therefore, users cannot create deposit requests that could later be rejected and consume allowance.

We will keep the request-deposit flow disabled and will not enable it until allowance restoration on rejection is implemented and tested. Accordingly, this finding has no impact on the currently supported production flow but remains a valid latent issue for any future enablement of requested deposits.

Issue M-5: Approved Deposit Requests Can Mint Below the Minimum [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-08-zoth-move-vaults-aug-5th-2026/issues/54>

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

Summary

`depositRequest()` validates the minimum zOPAL mint amount using the request-time NAV, but approval recalculates the mint amount using a newer NAV without reapplying the minimum check. A request can therefore mint less zOPAL than the configured minimum.

Vulnerability Detail

At request creation, `_calcAndValidateDeposit()` calculates `mintAmount` and validates `minAmount` and, for a first deposit, `minZTokenAmountForFirstDeposit`.

At approval, `_approveRequest()` recalculates the final amount using `newOutRate`:

```
uint256 amountZToken =  
    (request.usdAmountWithoutFees * (10 ** 18)) / newOutRate;
```

It mints this amount directly without calling `_validateMinAmount()`. If zOPAL NAV increases while the request is pending, the final minted amount decreases and can fall below the minimum.

Example:

```
minAmount = 100 zOPAL  
Request-time NAV = $1.00; net deposit = $100 → passes with 100 zOPAL  
Approval-time NAV = $1.05 → mints 95.238 zOPAL
```

`safeApproveRequest()` does not prevent this: its variation-tolerance check can still allow a NAV increase large enough to place the final mint amount below the threshold. `approveRequest()` has no tolerance check.

Impact

The configured minimum mint amount, including the first-deposit minimum, can be bypassed. This allows deposits smaller than the protocol's intended participation or anti-dust threshold.

Code Snippet

zVault/contracts/DepositVault.sol

```
function _approveRequest(
    uint256 requestId,
    uint256 newOutRate,
    bool isSafe,
    bool revertAboveSupplyCap
) private returns (bool) {
    Request memory request = mintRequests[requestId];
    // ...

    uint256 amountZToken = (request.usdAmountWithoutFees * (10 ** 18)) /
        newOutRate;

    // Missing: _validateMinAmount(request.sender, amountZToken);
    zToken.mint(request.sender, amountZToken);
}
```

Tool Used

Manual Review

Recommendation

Validate the final mint amount in `_approveRequest()` before minting:

```
_validateMinAmount(request.sender, amountZToken);
```

The validation should use the same account whose first-deposit status is intended to govern the request.

Discussion

aryaparas

We acknowledge that this behavior is valid for deferred deposit requests: a NAV change between request creation and approval could cause the final minted amount to fall below the applicable minimum.

Request-based deposits are disabled in our current launch configuration, and only instant deposits are supported. Therefore, this path is not reachable in the currently supported production flow.

We will keep request deposits disabled until approval-time minimum validation, using the current NAV and the intended recipient's first-deposit status, is implemented and tested.

Issue M-6: Redemption Requests Use a Stale Output-Token Price [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-zoth-move-vaults-aug-5th-2026/issues/56>

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

Summary

Redemption requests use the current zOPAL NAV at approval but the output token price captured at request creation. Price changes while a request is pending cause users or the protocol to receive an incorrect settlement value.

Vulnerability Detail

When a user creates a redemption request, the vault records the current output token price in `request.tokenOutRate`. At approval, it calculates the payout with a new zOPAL NAV but this stale output-token price:

```
amountOut = request.amountZToken * newZTokenRate / request.tokenOutRate;
```

For a non-stable output token, this mixes two valuation times.

Request: zOPAL NAV = \$1.00, tokenOut = \$1.00, amount = 100 zOPAL

Approval: zOPAL NAV = \$1.00, tokenOut = \$0.90

Current code pays 100 tokenOut, worth \$90.

Current-price settlement should pay 111.11 tokenOut, worth \$100.

If tokenOut appreciates instead, the request pays too many tokens and the protocol or requestRedeemer bears the loss. `safeApproveRequest()` only constrains the zOPAL NAV supplied by the admin; it does not refresh or validate `tokenOutRate`.

Impact

Pending redemption requests can settle at an incorrect USD value whenever the selected output token changes price. A de-pegging token can underpay users, while an appreciating token can overpay users and deplete the redemption reserve.

Code Snippet

`zVault/contracts/RedemptionVault.sol`

```
// At request creation
TokenConfig storage config = tokensConfig[tokenOutCopy];
tokenOutRate = _getTokenRate(config.dataFeed, config.stable);

redeemRequests[requestId] = Request({
    sender: user,
    tokenOut: tokenOutCopy,
    status: RequestStatus.Pending,
    amountZToken: amountZTokenWithoutFee,
    zTokenRate: zTokenRate,
    tokenOutRate: tokenOutRate
});
```

```
// At approval
uint256 amountTokenOutWithoutFee = _truncate(
    (request.amountZToken * newZTokenRate) / request.tokenOutRate,
    tokenDecimals
);
```

Tool Used

Manual Review

Recommendation

At approval, read and validate the current output-token price together with the current zOPAL NAV, then calculate the payout using rates from the same point in time.

```
amountOut = request.amountZToken * currentZTokenRate / currentTokenOutRate;
```

Add a user-supplied minTokenOut and request deadline so a request cannot be settled at an output amount the user did not accept.

Discussion

pantheraudits

Good one, but this only matters for non-stable tokens. current USDC/USDCx is stable and fixed at \$1, so it does not affect the present deployment.

Issue L-1: Incorrect Unit in First-Deposit Minimum Comment [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-zoth-move-vaults-aug-5th-2026/issues/57>

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

Summary

The comment for `minZTokenAmountForFirstDeposit` describes the value as a USD amount, while the variable is a zOPAL amount in 18-decimal units.

Vulnerability Detail

The value is compared against the final zOPAL mint amount:

```
require(
    amountZTokenWithoutFee >= minZTokenAmountForFirstDeposit,
    "DV: mint amount < min"
);
```

Its effective USD threshold changes with zOPAL NAV; it is not a fixed USD amount.

Impact

The incorrect unit can mislead administrators when configuring the first-deposit threshold and developers integrating with the vault.

Code Snippet

zVault/contracts/DepositVault.sol

```
/**
 * @notice minimal USD amount for first user`s deposit
 */
uint256 public minZTokenAmountForFirstDeposit;
```

Tool Used

Manual Review

Recommendation

Update the comment:

```
/**  
 * @notice Minimum zToken amount that must be minted for a user's first deposit.  
 */
```

Discussion

aryaparas

We acknowledge this. This is a documentation-only issue and does not affect contract execution or accounting. The variable is denominated in zOPAL base units and is compared against the final zOPAL mint amount, not a fixed USD value.

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.