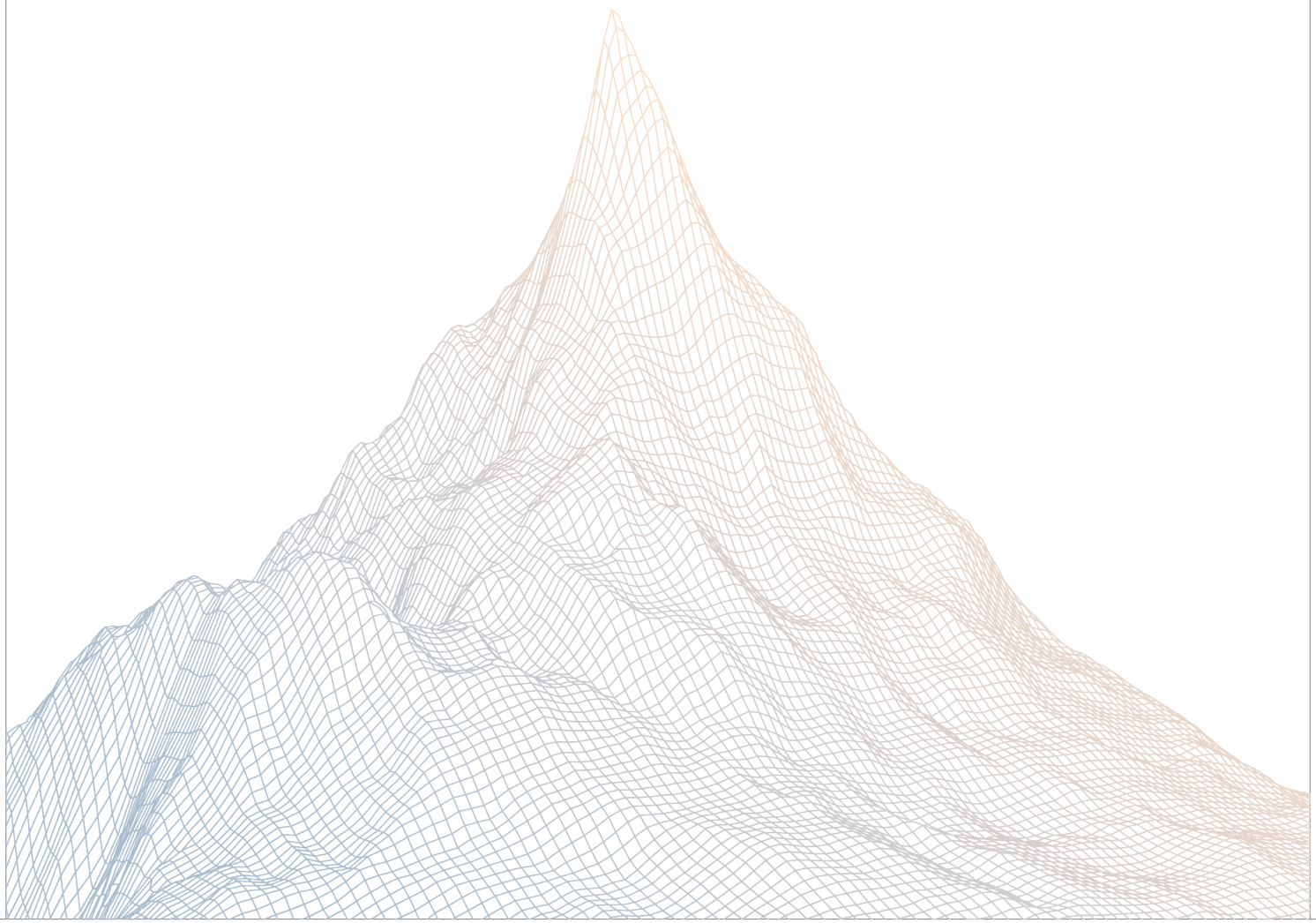
Zenith

Paxos Labs

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 14th to November 18th, 2025

AUDITED BY: peakbolt
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 Paxos Labs

Boring Vaults are flexible vault contracts that allow for intricate strategies, secured by both onchain and offchain mechanisms.

2.2 Scope

The engagement involved a review of the following targets:

Target	CommunityCodeDepositor
Repository	https://github.com/Ion-Protocol/nucleus-boring-vault/pull/188
Commit Hash	`0x7032978ef1fc44271145bb593be45be1c7a0b6c5`
Files	`Diff from 83138181c5d3017bd44b0225ff5ee5754b8adcfe src/helper/CommunityCodeDepositor/**/*.sol`

Target	OneToOneQueue
Repository	https://github.com/Ion-Protocol/nucleus-boring-vault/pull/187
Commit Hash	`0x5b615c65fcdd17f8dfaa693e5dcc754f92df504a`

Files	`OneToOneQueue.sol`
	`QueueAccessAuthority.sol`
	`SimpleFeeModule.sol`
	`access/AccessAuthority.sol`
	`access/Pausable.sol`
	`access/VerboseAuth.sol`

2.3 Audit Timeline

November 14, 2025	Audit start
November 18, 2025	Audit end
November 23, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	1
Medium Risk	2
Low Risk	6
Informational	5
Total Issues	**15**

3

Findings Summary

ID	Description	Status
C-1	lastProcessedOrder incorrectly set, causing processOrders to break	Resolved
H-1	forceRefund() fails to return the fee amount when feeAsset == wantAsset	Resolved
M-1	DoS on order processing/refunds when using tokens with blacklist feature	Resolved
M-2	Signature hash lacks domain separation, enabling cross-chain replay attacks	Resolved
L-1	CommunityCodeDepositor does not support share lock mechanism in teller	Acknowledged
L-2	DoS risk due to unhandled permit reverts	Resolved
L-3	Minimum order size check before fees	Acknowledged
L-4	processOrders() should perform external call after state updates	Resolved
L-5	setFeeModule() will apply new fee structure retroactively and affect refunds	Resolved
L-6	calculateOfferFees() should round up fee amount	Resolved
I-1	getOrderStatus returns misleading status for non-existent orders	Resolved
I-2	uint128 truncation without bounds check	Resolved
I-3	Lack of Order expiry can cause issue	Acknowledged
I-4	_verifyDepositor doesn't handle permit for feeAsset	Acknowledged
I-5	CommunityCodeDepositor do not support fee-on-transfer tokens	Acknowledged

4

Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] `lastProcessedOrder` incorrectly set, causing `processOrders` to break

`SEVERITY`: Critical	`IMPACT`: High
`STATUS`: Resolved	`LIKELIHOOD`: High

Target

- OneToOneQueue.sol#L467

Description:

In `processOrders`, after processing an order, the code increment `++lastProcessedOrder` instead of setting `lastProcessedOrder` to the actual `orderIndex` that was processed. This causes `lastProcessedOrder` to be set incorrectly, causing emitted `orderIndex` to be wrong and `getOrderStatus` could return wrong status and causing `processOrders` to break.

Scenario:

1. **Initial State:**- Orders 1, 2, 3 exist (all `DEFAULT`)
   - `lastProcessedOrder = 0`
2. **Order 1 force processed:**
   - `forceProcess(1)` called
   - Order 1 marked `PRE_FILLED`
   - `lastProcessedOrder` stays at 0
3. **Order 2 force processed:**
   - `forceProcess(2)` called
   - Order 2 marked `PRE_FILLED`
   - `lastProcessedOrder` stays at 0
4. **Order 3 processed via processOrders:**
   - `processOrders(3)` called
   - `startIndex = 0 + 1 = 1`, `endIndex = 0 + 3 = 3`

- Loop i=0: `orderIndex = 1`, PRE_FILLED, continue (`lastProcessedOrder` stays 0)
- Loop i=1: `orderIndex = 2`, PRE_FILLED, continue (`lastProcessedOrder` stays 0)
- Loop i=2: `orderIndex = 3`, `DEFAULT`:
  - NFT burned
  - Want assets transferred
  - `++lastProcessedOrder` → `lastProcessedOrder = 1`

5. **Next processOrders Call Breaks:**
   - Next time `processOrders()` is called :
     - `startIndex = lastProcessedOrder + 1 = 1 + 1 = 2`
     - Loop i=0: `orderIndex = 2`, PRE_FILLED, continue (skipped)
     - Loop i=1: `orderIndex = 3`, DEFAULT (but NFT already burned!)
       - NFT is burned, so `ownerOf()` and `_burn` reverts
       - Entire `processOrders` call reverts

Recommendations:

Correctly setting `lastProcessedOrder` with `orderIndex`.

Paxos Labs: Resolved with PR-213.

Zenith: Verified. Resolved by incrementing `lastProcessedOrder` for all order types.

4.2 High Risk

A total of 1 high risk findings were identified.

[H-1] `forceRefund()` fails to return the fee amount when `feeAsset == wantAsset`

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- OneToOneQueue.sol#L564-L575

Description:

`forceRefund()` only transfers the `amountOffer` when refunding the users.

This is incorrect when `feeAsset == wantAsset` as the users had paid for the fees on top of `amountOffer`.

That means users would not be refunded the fee amount in that scenario.

```
function _forceRefund(uint256 orderIndex) internal {
    Order memory order = _getOrderEnsureDefault(orderIndex);

    // Mark as refunded
    queue[orderIndex].orderType = OrderType.REFUND;

    _checkBalanceQueue(order.offerAsset, order.amountOffer, orderIndex);
    _burn(orderIndex);
    order.offerAsset.safeTransfer(order.refundReceiver, order.amountOffer);

    emit OrderRefunded(orderIndex, queue[orderIndex]);
}
```

Recommendations:

Refund the fee amount when `feeAsset == wantAsset`.

Paxos Labs: Resolved with PR-208.

Zenith: Verified. Resolved by removing fee asset and only collect fee in offer asset.

4.3 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] DoS on order processing/refunds when using tokens with blacklist feature

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

Target

- OneToOneQueue.sol#L572
- OneToOneQueue.sol#L587
- OneToOneQueue.sol#L464

Description:

If an NFT owner and `order.refundReceiver` is owned by a address that reverts on ERC20 token transfers, for instance token with blacklist feature, the `processOrders`, `forceRefund` and `forceProcess` functions will revert when attempting to transfer tokens to that receiver. This can block the entire queue from processing, as all subsequent orders cannot be processed until the problematic order is handled.

```
function processOrders(uint256 ordersToProcess) public requiresAuthVerbose {
        // ...
        address receiver = ownerOf(orderIndex);
        _checkBalanceQueue(order.wantAsset, order.amountWant, orderIndex);

        _burn(orderIndex);
>>>         order.wantAsset.safeTransfer(receiver, order.amountWant);

        unchecked {
            orderIndex = ++lastProcessedOrder;
        }

        emit OrderProcessed(orderIndex, order, receiver, false);
    }
```

```
        emit OrdersProcessedInRange(startIndex, endIndex);
    }
```

The attacker can intentionally send the NFT and set `order.refundReceiver` to a blacklisted user, assuming both the want and offer assets have a blacklist feature.

Recommendations:

Consider using a pull-over-push design, with order settlement done via internal balance storage. Alternatively, handle cases where the transfer fails.

Paxos Labs: Resolved with PR-204.

Zenith: Verified. Failed transfer to the `receiver` no longer reverts, it now sets the order to `FAILED_TRANSFER` and sends the funds to `recoveryAddress`.

[M-2] Signature hash lacks domain separation, enabling cross-chain replay attacks

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

Target

- OneToOneQueue.sol#L484-L494

Description:

The signature hash used for EIP-2612 style order submissions does not include the contract address or chain Id. This allows signatures to be replayed across different chains if the same contract is deployed on multiple networks.

Each contract instance maintains its own `usedSignatureHashes` mapping. When a user signs an order on chain A, the hash is marked as used on that chain. But attacker can take the signature and replay it on another chain with identical parameters.

```
function _verifyDepositor(SubmitOrderParams calldata params)
    internal returns (address depositor) {
// ...
      bytes32 hash = keccak256(
          abi.encode(
              params.amountOffer,
              params.offerAsset,
              params.wantAsset,
              params.receiver,
              params.refundReceiver,
              params.signatureParams.deadline,
              params.signatureParams.approvalMethod,
              params.signatureParams.nonce
              // Missing: block.chainid and address(this)
          )
      );
      if (usedSignatureHashes[hash]) revert SignatureHashAlreadyUsed(hash);
      usedSignatureHashes[hash] = true;

      depositor = ECDSA.recover(hash,
    params.signatureParams.eip2612Signature);
```

```
        // ..
}
```

Recommendations:

Implement eip-2612 properly, include contract address and chain Id in the signature hash to ensure each chain and contract instance produces unique hashes.

Paxos Labs: Resolved with PR-209.

Zenith: Verified. Resolved by adding chainid and contract address to signature hash.

4.4 Low Risk

A total of 6 low risk findings were identified.

[L-1] `CommunityCodeDepositor` does not support share lock mechanism in teller

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

Target

- CommunityCodeDepositor.sol#L181-L182

Description:

The `TellerWithMultiAssetSupport` contract has a share lock mechanism that prevents share of transfer for a specified locked period.

However, `CommunityCodeDepositor` does not support the share lock mechanism and will fail if the feature is enabled. This will cause all deposits via `CommunityCodeDepositor` to fail.

```
function _deposit(
    ERC20 depositAsset,
    uint256 depositAmount,
    uint256 minimumMint,
    address to,
    bytes calldata communityCode
)
  ....
    shares = teller.deposit(depositAsset, depositAmount, minimumMint);
    ERC20(boringVault).safeTransfer(to, shares);
```

Recommendations:

If the `CommunityCodeDepositor` is intended to support the share lock mechanism, consider tracking the deposited shares and allow claim of them after the shares are unlocked.

Paxos Labs: Acknowleged. When we use CommunityCodeDepositor, we are not using the

sharelock mechanism and have no intention of ever doing so. It is okay for the CommunityCodeDepositor to fail if the sharelock mechanism is turned on.

[L-2] DoS risk due to unhandled `permit` reverts

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

Target

- OneToOneQueue.sol#L513-L522
- CommunityCodeDepositor.sol#L151

Description:

The `permit` call can revert if the signature has already been used. Since there's no error handling or try/catch, any operation that relies on `permit` becomes vulnerable to a DoS attack. An attacker can front-run by calling `permit` first, causing the subsequent operations that try to call `permit` to revert.

```
function _verifyDepositor(SubmitOrderParams calldata params)
    internal returns (address depositor) {
      // Do nothing if using standard ERC20 approve
      if (params.signatureParams.approvalMethod ==
    ApprovalMethod.EIP2612_PERMIT) {
          IERC20Permit(address(params.offerAsset))
              .permit(
                  depositor,
                  address(this),
                  params.amountOffer,
                  params.signatureParams.deadline,
                  params.signatureParams.approvalV,
                  params.signatureParams.approvalR,
                  params.signatureParams.approvalS
              );
      }
      // ..
}
```

Recommendations:

Consider wrap the `permit` inside try/catch.

Paxos Labs: Resolved with PR-206 and `@ec20b70e50 ...`.

Zenith: Verified. Resolved by adding `try/catch` for `permit()`.

[L-3] Minimum order size check before fees

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

Target

- OneToOneQueue.sol#L374-L382

Description:

The minimum order size check is performed on the gross `amountOffer` before fees are deducted. After fee calculation, the net amount going to the receiver (`newAmountForReceiver`) could be below the intended minimum order size.

```
   function submitOrder(SubmitOrderParams calldata params)
    public requiresAuthVerbose returns (uint256 orderIndex) {
      // ...
      uint256 minimumOrderSize
    = minimumOrderSizePerAsset[address(params.offerAsset)];
>>>   if (params.amountOffer < minimumOrderSize)
    revert AmountBelowMinimum(params.amountOffer, minimumOrderSize);
      if (params.receiver == address(0)) revert ZeroAddress();
      if (params.refundReceiver == address(0)) revert ZeroAddress();
      }

      address depositor = _verifyDepositor(params);
>>>   (uint256 newAmountForReceiver, IERC20 feeAsset, uint256 feeAmount) =
          feeModule.calculateOfferFees(params.amountOffer,
    params.offerAsset, params.wantAsset, params.receiver);
      // ...
}
```

Recommendations:

Move the `minimumOrderSize` check after fee calculation.

Paxos Labs: Acknowledged. However there is not fix as the intention is for the minimumOrderSize to be applied before the fees are taken.

[L-4] `processOrders()` should perform external call after state updates

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

Target

- OneToOneQueue.sol#L464

Description:

`processOrders()` will perform an external call to `order.wantAsset.safeTransfer()` before updating the `orderIndex`.

This violates the CEI pattern and could cause reentrancy issues if the `wantAsset` supports ERC777.

```
function processOrders(uint256 ordersToProcess) public requiresAuthVerbose {
        ...
        _burn(orderIndex);
        order.wantAsset.safeTransfer(receiver, order.amountWant);

        unchecked {
            orderIndex = ++lastProcessedOrder;
        }
```

Recommendations:

Consider performing the external call to `order.wantAsset.safeTransfer()` after updating the `orderIndex`.

Paxos Labs: Resolved with PR-210 and `@a7df4775bf ...`.

Zenith: Verified. Resolved by removing `orderIndex` update in `processOrder()` to ensure external call is done after state changes. Also in `submitOrder()` the external calls are shifted to be after state changes (e.g. update of `queue[orderIndex]`).

[L-5] `setFeeModule()` will apply new fee structure retroactively and affect refunds

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

Target

- OneToOneQueue.sol#L177-L187

Description:

`setFeeModule()` allows the authorized caller to change the fee module and update the fee structure.

However, this will cause the new fee structure to be applied on existing orders in the queue. If a refund is made for these orders, it could refund the incorrect fee amount to the users.

```
function setFeeModule(IFeeModule _feeModule) external requiresAuthVerbose {
    if (address(_feeModule) == address(0)) revert ZeroAddress();

    IFeeModule oldFeeModule = feeModule;
    feeModule = _feeModule;
    emit FeeModuleUpdated(oldFeeModule, _feeModule);
}
```

Recommendations:

Consider removing this ability if it's not required. Otherwise store the fee amount in the order, so that refunds would be applied correctly when new fee module is updated.

Paxos Labs: Resolved with PR-208.

Zenith: Verified. Resolved by removing fee asset and only collect fee in offer asset.

[L-6] `calculateOfferFees()` should round up fee amount

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

Target

SimpleFeeModule.sol#L31-L44

Description:

`calculateOfferFees()` will round down the fee amount and this can cause the `feeAmount` to round down to zero for small `amount` orders.

```
function calculateOfferFees(
    uint256 amount,
    IERC20 offerAsset,
    IERC20 wantAsset,
    address receiver
)
    external
    view
    returns (uint256 newAmountForReceiver, IERC20 feeAsset,
    uint256 feeAmount)
{
    feeAmount = (amount * offerFeePercentage) / ONE_HUNDRED_PERCENT;
    newAmountForReceiver = amount - feeAmount;
    feeAsset = IERC20(offerAsset);
}
```

Recommendations:

Consider rounding up the `feeAmount` to prevent rounding to zero.

Paxos Labs: Resolved with PR-211.

Zenith: Verified. Resolved by rounding up `feeAmount`.

4.5 Informational

A total of 5 informational findings were identified.

[I-1] `getOrderStatus` returns misleading status for non-existent orders

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

Target

- OneToOneQueue.sol#L357-L359

Description:

The `getOrderStatus` function does not validate that the `orderIndex` exists before returning a status. If called with an `orderIndex > latestOrder` (a non-existent order), it will return `OrderStatus.PENDING` instead of reverting or returning an error, which is misleading.

```
function getOrderStatus(uint256 orderIndex)
    external view returns (OrderStatus) {
    Order memory order = queue[orderIndex];

    if (order.orderType == OrderType.PRE_FILLED) {
        return OrderStatus.COMPLETE_PRE_FILLED;
    }

    if (order.orderType == OrderType.REFUND) {
        return OrderStatus.COMPLETE_REFUNDED;
    }

    if (orderIndex > lastProcessedOrder) {
>>>     return OrderStatus.PENDING;  // Returns PENDING for non-existent
    orders
    } else {
        return OrderStatus.COMPLETE;
    }
}
```

Recommendations:

Add validation to check if the order exists.

Paxos Labs: Resolved with PR-207.

Zenith: Verified. Resolved by handling non-existent order index.

[I-2] `uint128` truncation without bounds check

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

Target

- OneToOneQueue.sol#L407-L410

Description:

The `amountOffer` parameter is `uint256` but is unsafe-casted to `uint128` when storing in the `Order` struct without any bounds check. If a user submits an order with `amountOffer > type(uint128).max`, the value will silently truncate.

```
    function submitOrder(SubmitOrderParams calldata params)
     public requiresAuthVerbose returns (uint256 orderIndex) {
// ...
    Order memory order = Order({
>>>     amountOffer: uint128(params.amountOffer),  // Silent truncation if >
    uint128.max
>>>     amountWant: _getWantAmountInWantDecimals(
            uint128(newAmountForReceiver),  // Also truncated here
            params.offerAsset,
            params.wantAsset
        ),
        // ...
    });
// ...
}
```

Recommendations:

Add bounds check before casting.

Paxos Labs: Resolved with `@6bfd942b43 ...`. To view changes it may be best to compare against the following commit: `@4ee169cc6a ...`.

Zenith: Verified. SafeCast is now used.

[I-3] Lack of `Order` expiry can cause issue

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

Target

- OneToOneQueue.sol#L368-L424

Description:

Orders in the queue do not have a deadline for processing, and the exchange rate is fixed at submission time (1:1 adjusted for decimals). If a stablecoin or other asset depegs after order submission but before processing, users will receive depegged tokens worth significantly less than what they originally offered, with no mechanism to cancel or update the order. A real-world example is the recent USDX depeg incident.

Recommendations:

Add order deadline and expiration mechanism.

Paxos Labs: Acknowledged. This is intentional and we may always refund orders in an absolute emergency.

[I-4] `_verifyDepositor` doesn't handle permit for `feeAsset`

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

Target

- OneToOneQueue.sol#L397

Description:

The `_verifyDepositor` function only handles permit approval for `params.offerAsset`, but the fee module can return a different `feeAsset`. When `feeAsset ≠ offerAsset`, the code attempts to transfer `feeAsset` from the depositor without having permit approval for it, causing the transfer to fail with insufficient allowance.

Recommendations:

Extend permit to handle both assets, or document the behavior.

Paxos Labs: Acknowledged. We have removed the fee asset in the H1 fix so this is no longer applicable.

[I-5] `CommunityCodeDepositor` do not support fee-on-transfer tokens

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

Target

- CommunityCodeDepositor.sol

Description:

The `CommunityCodeDepositor` contract does not support fee-on-transfer tokens. When a user deposits tokens, the contract transfers `depositAmount` from the user to itself, but then uses the original `depositAmount` value instead of checking the actual balance received when depositing into the teller. For fee-on-transfer tokens, the contract receives less than `depositAmount` due to the transfer fee, causing the subsequent transfer to the vault to fail.

Recommendations:

Check the balance before and after the transfer to use the actual amount received, or document that the contracts do not support fee-on-transfer tokens.

Paxos Labs: Acknowledged. We do not intend to ever support fee on transfer tokens with this contract.