Coinbase
Security Review

Cantina Managed review by:
Zach Obront, Security Researcher
Lucas Goiriz, Junior Security Researcher

August 6, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Low severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or high. Critical findings should be directly vulnerable and have a high likelihood of being exploited. High findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

From July 24th to July 28th the Cantina team conducted a review of Coinbase - Optimism Bedrock shared revenue contracts. Code in scope was privately shared in a zip folder, therefore permalink references to a github repository are unavailable. Additional to the review, PR ethereum-optimism/optimism/pull/5886 was shared containing the FeeVault related revenue share code changes to the Optimism Monorepo.

During this period, a total of **11** issues in the following risk categories were identified:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 2
- Low Risk: 4
- Gas Optimizations: 2
- Informational: 2

3 Findings

3.1 High Risk

3.1.1 Vault's `totalProcessed` count can be inaccurately increased by any user

Severity: High Risk

Context: FeeVault.sol#L105-L106, SafeCall.sol#L12-L30

Description: Each of the 3 fee vaults inherits the `withdraw()` function from `FeeVault.sol`:

```
function withdraw() external {
    require(
        address(this).balance >= MIN_WITHDRAWAL_AMOUNT,
        "FeeVault: withdrawal amount must be greater than minimum withdrawal amount"
    );

    uint256 value = address(this).balance;
    totalProcessed += value;

    emit Withdrawal(value, RECIPIENT, msg.sender);
    emit Withdrawal(value, RECIPIENT, msg.sender, WITHDRAWAL_NETWORK);

    if (WITHDRAWAL_NETWORK == WithdrawalNetwork.L2) {
        SafeCall.send(RECIPIENT, gasleft(), value);
    } else {
        L2StandardBridge(payable(Predeploys.L2_STANDARD_BRIDGE)).bridgeETHTo{ value: value }(
            RECIPIENT,
            WITHDRAWAL_MIN_GAS,
            bytes("")
        );
    }
}
```

In the event that this function does not revert, `totalProcessed` is incremented by the total balance of the contract, on the assumption that these funds are processed and send out to `RECIPIENT`.

However, this is not necessarily the case.

In the event that `WITHDRAWAL_NETWORK == L2`, we send the funds using `SafeCall.send()`. As we can see from the SafeCall library, this function does not revert in the case that the transaction does not succeed. Instead, it simply returns `false`.

```
function send(
    address _target,
    uint256 _gas,
    uint256 _value
) internal returns (bool) {
    bool _success;
    assembly {
        _success := call(
            _gas, // gas
            _target, // recipient
            _value, // ether value
            0, // inloc
            0, // inlen
            0, // outloc
            0 // outlen
        )
    }
    return _success;
}
```

Since this return value is not checked, the `withdraw()` function will not revert if the `send()` runs out of gas.

This is a problem on a permissionless function, because it gives a malicious user the ability to send a precise amount of gas to ensure it will run out mid call. The result will be that the `totalProcessed` variable is incremented and the `Withdrawal` event is emitted, but no funds are sent.

Proof of Concept

The following test deploys a mock Receiver (which implements the `receive()` function from

`FeeDisburser.sol`, to correctly estimate gas), as well as a `BaseFeeVault`. What we can see is that, when the amount of gas passed is set to a specific value, the total processed increases by the full balance of the contract, while the funds are not transferred:

```
contract FeeVaultTest is Test {
    BaseFeeVault feeVault;
    Receiver recipient;

    function setUp() public {
        recipient = new Receiver();
        feeVault = new BaseFeeVault(address(recipient), 2 ether, FeeVault.WithdrawalNetwork.L2);
        recipient.setVaults(address(1), address(feeVault), address(2));
    }

    function testZach__FeeVaultOutOfGasAttack() public {
        vm.deal(address(feeVault), 10 ether);
        console.log("Starting Total Processed: ", feeVault.totalProcessed());
        console.log("Starting Vault Balance: ", address(feeVault).balance);
        console.log("Starting Recipient Balance: ", address(recipient).balance);
        feeVault.withdraw{gas: 55_000}();
        console.log("********");
        console.log("Ending Total Processed: ", feeVault.totalProcessed());
        console.log("Ending Vault Balance: ", address(feeVault).balance);
        console.log("Ending Recipient Balance: ", address(recipient).balance);
    }
}
```

```
Logs:
  Starting Total Processed:  0
  Starting Vault Balance:  10000000000000000000
  Starting Recipient Balance:  0
  ********
  Ending Total Processed:  10000000000000000000
  Ending Vault Balance:  10000000000000000000
  Ending Recipient Balance:  0
```

Recommendation: Consider applying the following change

```
 if (WITHDRAWAL_NETWORK == WithdrawalNetwork.L2) {
-    SafeCall.send(RECIPIENT, gasleft(), value);
+    bool success = SafeCall.send(RECIPIENT, gasleft(), value);
+    require(success);
 } else {
     ...
 }
```

3.2 Medium Risk

3.2.1 If Optimism Wallet has expensive `receive()` function, their payments can be bypassed

Severity: Medium Risk

Context: FeeDisburser.sol#L57

Description: When `disburseFees()` is called, the amount of fees to send to Optimism is calculated. Then `SafeCall` is used to send those funds to Optimism. Finally, the remaining balance of the contract is bridged to L1.

Similar to the issue raised in *"Vault's totalProcessed count can be inaccurately increased by any user"* , the `SafeCall` used to send these funds does not check the return value. This means that, if the call fails, the function will continue executing and will send its full remaining balance to Base's contract on L1.

```
SafeCall.send(OPTIMISM_WALLET, gasleft(), optimismRevenueShare);

// Send remaining funds to L1 wallet on L1
L2StandardBridge(payable(Predeploys.L2_STANDARD_BRIDGE)).bridgeETHTo{ value: address(this).balance }(
    L1_WALLET,
    WITHDRAWAL_MIN_GAS,
    bytes("")
);
```

While skipping Optimism's payments would be a big problem, I am considering this issue only a Medium Severity because it is quite unlikely. There are only two ways I can see the `SafeCall.send()` reverting:

1) Optimism deploys a contract to this address that reverts in the `receive()` function.
2) Optimism deploys a contract to this address where the `receive()` function uses so much gas that it can run out of gas, while the remaining 1/64th of the previous gas is sufficient to perform the bridge transaction. That would imply that the `receive()` function uses 63x as much gas as the bridge call, which is fairly unlikely.

Recommendation: Unfortunately, it is not safe to simply `require(success);`, because this opens the door to a DOS if Optimism deploys a contract to the `OPTIMISM_WALLET` address that reverts on transfers. This could happen maliciously, or accidentally, and would lead to all funds being locked in the `FeeDisburser` contract.

Instead, the best solution seems to be to check this return `success` value and, if it returns `false` set aside the fees for Optimism. This can be done by saving the value in a storage variable and, if the value is greater than zero, only sending `address(this).balance - optimismHoldings` on withdrawals.

Optimism would need a separate function to withdraw this `optimismHoldings` value directly, outside the regular fee disbursement process.

3.2.2 Optimism can be overpaid by up to 70% if `disburseFees()` is called regularly

Severity: Medium Risk

Context: FeeDisburser.sol#L125-L166

Description: The agreement between Optimism and Coinbase is that Optimism will earn a portion of the fees equal to:

```
max(netRevenue * 0.15, grossRevenue * 0.025)
```

It is not clear from the documentation what time period these calculations should be performed over, but presumably we are looking to reflect this formula over the long term.

Allowing `disburseFees()` to be called daily perturbs the formula in such a way that increases Optimism's payout.

To understand why, let's look at the two possible scenarios when calculating the long term payout:

1) `netRevenue * 0.15 > grossRevenue * 0.025`
2) `netRevenue * 0.15 < grossRevenue * 0.025`

In the first case, all of the `netRevenue` will be paid out to the Optimism wallet as its withdrawn from the vaults. However, any time that `disburseFees()` is called when `L1_FEE_VAULT` has a balance over `2 ether` and the other vaults have a balance under `2 ether`, the gross revenue payout will be higher *on that day*, and therefore an additional payout will happen of 2.5% of these L1 vault fees.

In the second case, at least 2.5% of all the revenue will be paid out. However, any days where `L1_FEE_VAULT` was not withdrawn and another vault was, we will be paying out the balance as `net revenue`, and will be paying 15% instead.

In both cases, Optimism earns a surplus of fees compared to the same calculation performed on an annual basis.

Proof of Concept

Starting with Scenario 1, imagine each of the three vaults earns 10 ether per month. This would result in annual calculations of:

- `netRevenue = 240 ether`
- `grossRevenue = 360 ether`
- optimism payout = `240 ether * 0.15 = 36 ether`

However, the total 360 ether could be distributed in 180 separate distributions, assuming the vaults reach 2 ether on different days. In this case, we would have:

- 120 days where 2 ether was withdrawn from a net revenue vault, resulting in `120 * 2 * 0.15 = 36 ether`
- 60 days where 2 ether was withdrawn from the L1 fee vault, resulting in `60 * 2 * 0.025 = 3 ether`

This results in an extra `3 ether` (+8.3%) of fees being sent to Optimism.

Looking at Scenario 2. Imagine that the L1 vault earns 30 ether per month, and the others earn 2.5 ether each. This would result in annual calculations of:

- `netRevenue = 60 ether`
- `grossRevenue = 360 + 60 = 420 ether`
- optimism payout = `420 ether * 0.025 = 10.5 ether`

However, we can again assume that these could be distributed over 210 separate withdrawals:

- 180 days where 2 ether was withdrawn from the L1 fee vault, resulting in `180 * 2 * 0.025 = 9 ether`
- 30 days where 2 ether was withdrawn from the other vaults, resulting in `30 * 2 * 0.15 = 9 ether`

This result in an extra `7.5 ether` (+71.4%) of fees being sent to Optimism.

Recommendation: There are a few ways to handle this inaccurate calculation:

1) Increase the `FEE_DISBURSEMENT_INTERVAL`. The longer this interval gets, the less likely it is that only one of the wallets has reached the required minimum withdrawal amount.
2) Store historical payout values and calculate Optimism's payouts using those the summed up value instead of the individual withdrawal values.
3) Ignore it. This can only benefit Optimism, not Coinbase, so if the Coinbase team is comfortable with the additional fees being sent to Optimism, it can be left as is.

3.3 Low Risk

3.3.1 Refactor `disburseFees()` to save gas and reduce DOS risk

Severity: Low Risk

Context: FeeDisburser.sol#L126-L149

Description: When `disburseFees()` is called, the function begins with the following actions (in this order):

- Check that `FEE_DISBURSEMENT_INTERVAL` has passed since the last call.
- Update the `lastDisbursementTime` so it can't be called again.
- Withdraw from both `netRevenue` fee vaults.
- Calculate the Optimism `netRevenue` share.
- Withdraw from the L1 fee vault.
- Check the balance of the contract.
- Return early if the balance of the contract is zero.
- Continue on with the rest of the function logic.

Some of these actions should be moved after the early return, for two main reasons:

1) Gas Savings: We set `lastDisbursementTime = block.timestamp` before the early return. In the case of an early return, these will cost `2900` gas. This can be updated after the early return to make the function more gas efficient. The same is true of calculating `optimismNetRevenueShare` and zeroing out the `netFeeRevenue`.
2) DOS Prevention: There is a minor DOS risk where a user can call `disburseFees()` just before a vault reaches `2 ether`. In this case, the `lastDisbursementTime` will still be updated, and the proper disbursement will not be possible for 24 hours. While this isn't a huge deal if the time is 24 hours, if that time is increased (as recommended in *Optimism can be overpaid by up to 70% if `disburseFees()` is called regularly*), it could become a risk.

Recommendation: Refactor the function to move these pieces of logic until after the early return:

```
 function disburseFees() external virtual {
    require(
        block.timestamp >= lastDisbursementTime + FEE_DISBURSEMENT_INTERVAL,
        "FeeDisburser: Disbursement interval not reached"
    );
-   lastDisbursementTime = block.timestamp;

    // Sequencer and base FeeVaults will withdraw fees to the FeeDisburser contract mutating netFeeRevenue
    feeVaultWithdrawal(payable(Predeploys.SEQUENCER_FEE_WALLET));
    feeVaultWithdrawal(payable(Predeploys.BASE_FEE_VAULT));

-   // Net revenue is the sum of sequencer fees and base fees
-   uint256 optimismNetRevenueShare = netFeeRevenue * OPTIMISM_NET_REVENUE_SHARE_BASIS_POINTS /
↪   BASIS_POINT_SCALE;
-   netFeeRevenue = 0;

    feeVaultWithdrawal(payable(Predeploys.L1_FEE_VAULT));

    // Gross revenue is the sum of all fees
    uint256 feeBalance = address(this).balance;

    // Stop execution if no fees were collected
    if (feeBalance == 0) {
        emit NoFeesCollected();
        return;
    }

+   lastDisbursementTime = block.timestamp;

+   // Net revenue is the sum of sequencer fees and base fees
+   uint256 optimismNetRevenueShare = netFeeRevenue * OPTIMISM_NET_REVENUE_SHARE_BASIS_POINTS /
↪   BASIS_POINT_SCALE;
+   netFeeRevenue = 0;

    ...
 }
```

3.3.2 Optimism will lose fee vault L1 withdrawal test coverage from PR 5886

Severity: Low Risk

Context: setup.go#L111-113

Description: In PR 5886, Coinbase introduced the possibility for Bedrock chains to set L2 addresses for withdrawals instead of Optimism's default option of withdrawing to L1.

Part of this PR was to edit the OP Stack's end to end tests to use L2 withdrawals:

```
SequencerFeeVaultRecipient:               common.Address{19: 1},
BaseFeeVaultRecipient:                    common.Address{19: 2},
L1FeeVaultRecipient:                      common.Address{19: 3},
BaseFeeVaultMinimumWithdrawalAmount:      uint642big(1000_000_000), // 1 gwei
L1FeeVaultMinimumWithdrawalAmount:        uint642big(1000_000_000), // 1 gwei
SequencerFeeVaultMinimumWithdrawalAmount: uint642big(1000_000_000), // 1 gwei
BaseFeeVaultWithdrawalNetwork:            uint8(1),                 // L2 withdrawal network
L1FeeVaultWithdrawalNetwork:              uint8(1),                 // L2 withdrawal network
SequencerFeeVaultWithdrawalNetwork:       uint8(1),                 // L2 withdrawal network
```

This did not leave Optimism with any remaining e2e tests for their own withdrawal process, which is also more common among OP stack chains.

Recommendation: Adjust one of the e2e tests in the OP Stack back to `L1 Withdrawal Network` so that codepath is included in their test suite as well.

3.3.3 `processFees()` can be called before Balance Tracker is initialized to take all fees as profit

Severity: Low Risk

Context: BalanceTracker.sol#L129-L139

Description: When `initialize()` is called on the Balance Tracker contract, it sets the storage variables of `systemAddresses` and `targetBalances` to the passed arguments.

Later, when `processFees()` is called, it iterates over the system addresses, paying each until they reach their target balance. If there are any funds leftover, they are passed to the `PROFIT_WALLET`:

```
function processFees() external nonReentrant {
    // Refills balances of systems addresses up to their target balances
    for (uint256 i = 0; i < systemAddresses.length; i++) {
        refillBalanceIfNeeded(systemAddresses[i], targetBalances[i]);
    }

    // Send remaining profits to profit wallet
    uint256 valueToSend = address(this).balance;
    bool success = SafeCall.send(PROFIT_WALLET, gasleft(), valueToSend);
    emit SentProfit(PROFIT_WALLET, success, valueToSend);
}
```

However, there is no check in `processFees()` that the contract has been initialized. This means that, before the contract is initialized, the `processFees()` function can be called, the for loop will be skipped (because the array is empty) and the full balance will be sent to `PROFIT_WALLET`.

Note that, because these functions are permissionless, even if the Coinbase team does not intend to do this, an outside party can perform this action, as long as there is any gap between Balance Tracker deployment and initialization.

Recommendation: Add a check to `processFees()` to make sure that the system addresses have been set before fees are withdrawn:

```
 function processFees() external nonReentrant {
+   uint systemAddressesLength = systemAddresses.length;
+   require(systemAddressesLength > 0);
    // Refills balances of systems addresses up to their target balances
-   for (uint256 i = 0; i < systemAddresses.length; i++) {
+   for (uint256 i = 0; i < systemAddressesLength; i++) {
        refillBalanceIfNeeded(systemAddresses[i], targetBalances[i]);
    }

    // Send remaining profits to profit wallet
    uint256 valueToSend = address(this).balance;
    bool success = SafeCall.send(PROFIT_WALLET, gasleft(), valueToSend);
    emit SentProfit(PROFIT_WALLET, success, valueToSend);
 }
```

Note that this fix also addresses the loop optimization opportunity described in *"Loop optimization opportunities"*.

3.3.4 SequencerFeeVault `l1FeeWallet()` function misleading on Base

Severity: Low Risk

Context: SequencerFeeVault.sol#L31-L37

Description: The Sequencer Fee Vault implements a `l1FeeWallet()` function, which returns the `RECIPIENT` address:

```
function l1FeeWallet() public view returns (address) {
    return RECIPIENT;
}
```

This function name assumes that the fee wallet is on L1, which is misleading on Base or any chain that uses `L2` as their Withdrawal Network.

Recommendation: Consider applying the following change

```
function ifFeeWallet() public view returns (address) {
+    if (WITHDRAWAL_NETWORK == WithdrawalNetwork.L2) revert("Fees will not be withdrawn to L1");
     return RECIPIENT;
}
```

3.4 Gas Optimization

3.4.1 Balance caching incurs in gas savings

Severity: Gas Optimization

Context: BalanceTracker.sol#L158-L163, BalanceTracker.sol#L158-L163, FeeVault.sol#L94-L99

Description: Each time a balance is read, it requires a separate execution of the `BALANCE` operation, resulting in a cost of 2600 gas units if the address is cold (it is accessed for the first time in the current execution scope), or 100 gas units if the address is warm (it is not accessed for the first time in the current execution scope). If the balance is saved in a local variable instead, every subsequent read requires the execution of the `MLOAD` operation, resulting in a cost of 3 gas units.

Recommendation: Consider caching balances if they are going to be read several times in the current execution scope for gas savings. In particular:

- BalanceTracker.sol#L158-L163

```
+ uint256 systemAddressBalance = _systemAddress.balance;
- if (_systemAddress.balance >= _targetBalance) {
+ if (systemAddressBalance  >= _targetBalance) {
            emit ProcessedFunds(_systemAddress, false, 0, 0);
            return;
        }

- uint256 valueNeeded = _targetBalance - _systemAddress.balance;
+ uint256 valueNeeded = _targetBalance - systemAddressBalance;
```

- BalanceTracker.sol#L164

```
- uint256 valueToSend = valueNeeded > address(this).balance ? address(this).balance : valueNeeded;
+ uint256 selfBalance = address(this).balance;
+ uint256 valueToSend = valueNeeded > selfBalance ? selfBalance : valueNeeded;
```

- FeeVault.sol#L94-L99

```
+ uint256 value = address(this).balance;
  require(
- address(this).balance >= MIN_WITHDRAWAL_AMOUNT,
+ value >= MIN_WITHDRAWAL_AMOUNT,
   "FeeVault: withdrawal amount must be greater than minimum withdrawal amount"
    );
- uint256 value = address(this).balance;
```

3.4.2 Loop optimization opportunities

Severity: Gas Optimization

Context: BalanceTracker.sol#L114-L117, BalanceTracker.sol#L131-L133

Description: Under certain safe scenarios, `for` loops in Solidity can be optimized by taking advantage of the following properties:

- Un-initialized unsigned integers are set to `0` by default.
- Caching variables that are read multiple times reduces gas costs as excess `SLOAD`s are avoided this way.
- Overflow protection can be disabled if the variable to increment has a clearly defined upper bound.
- The prefix increment is the cheapest way to increament a counter.

So, for instance, one can optimize the loop at BalanceTracker.sol#L131-L133 this way

```
+ uint256 systemAddresesLength = systemAddresses.length;
- for (uint256 i = 0; i < systemAddresesLength; i++) {
+ for (uint256 i ; i < systemAddresesLength; ) {
    refillBalanceIfNeeded(systemAddresses[i], targetBalances[i])
+   unchecked { ++i; }
    }
```

Recommendation: Where possible, consider implementing these `for` loops optimisations.

3.5 Informational

3.5.1 Revenue share business logic comment doesn't accurately reflect calculation

Severity: Informational

Context: FeeDisburser.sol#L122

Description: The natspec comment above the `disburseFees()` function describes the revenue share deal between Base and Optimism:

```
* Net Revenue              = sequencer FeeVault fee revenue + base FeeVault fee revenue
* Gross Revenue            = Net Revenue + l1 FeeVault fee revenue
* Optimism Revenue Share   = Maximum of Net Revenue and Gross Revenue
* L1 Wallet Revenue Share = Gross Revenue - Optimism Revenue Share
```

This calculation does not take into account that `OptimismRevenueShare` should be based on the percentages earned on the two different forms of revenue, not the totals.

Recommendation: Consider applying the following change

```
 *  Net Revenue              = sequencer FeeVault fee revenue + base FeeVault fee revenue
 * Gross Revenue            = Net Revenue + l1 FeeVault fee revenue
-* Optimism Revenue Share   = Maximum of Net Revenue and Gross Revenue
+* Optimism Revenue Share   = Maximum of 15% of Net Revenue and 2.5% of Gross Revenue
 * L1 Wallet Revenue Share = Gross Revenue - Optimism Revenue Share
```

3.5.2 WITHDRAWAL_MIN_GAS should be renamed to clarify purpose

Severity: Informational

Context: FeeVault.sol#L42

Description: The `FeeVault.sol` contract uses the existing `WITHDRAWAL_MIN_GAS` constant from the Optimism contracts before the PR:

```
uint32 internal constant WITHDRAWAL_MIN_GAS = 35_000;
```

Since there are now two ways to withdraw, it would be beneficial to rename this variable to make clear that it is the gas limit provided for L1 withdrawals, not L2.

Recommendation: Rename the variable to `L1_WITHDRAWAL_MIN_GAS`.