CANTINA

Bitcorn OFT 69d1ec

Security Review

Cantina Managed review by:
D-Nice, Lead Security Researcher
Sujith Somraaj, Security Researcher

December 12, 2024

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	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
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. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Corn is a layer 2 network focused on revolutionizing the capital efficiency of Bitcoin as a nascent asset class. Designed to provide scalable infrastructure that leverages Ethereum in a manner that allows for the secure management of billions of dollars in liquidity with low transactional costs secured by the Bitcoin L1.

From Dec 6th to Dec 10th the Cantina team conducted a review of bitcorn-oft on commit hash 69d1ec6c. The team identified a total of **13** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 3
- Gas Optimizations: 4
- Informational: 3

The scope of the review included all the smart contracts reviewed in a previous engagement as well as the newly added `SimpleSwapFacility`, in place of `SwapFacility`.

3 Findings

3.1 Medium Risk

3.1.1 `WrappedBitcornNativeOFTAdapter.send` doesn't refund excess sent for LayerZero fees

Severity: Medium Risk

Context: WrappedBitcornNativeOFTAdapter.sol#L150-L194, WrappedBitcornNativeOFTAdapter.sol#L170-L172

Description: The `WrappedBitcornNativeOFTAdapter` contract overrides the default behaviour of the LayerZero OApp dependencies, where the `msg.value` being sent must match the associated `fee.nativeFee` parameter that the user is pledging. In the override, it can match or exceed.

Under the standard behaviour, any excess sent that is not charged by LayerZero, would be refunded by the `endpoint` contract to the noted `refundAddress`. This case does occur for the `WrappedBitcornNativeOFTAdapter` contract when the user doesn't have enough wrapped tokens, and instead directly sends them natively. But, in the case their balance is sufficient, any excess native amount sends exceeding the fee, are not refunded by the adapter, and kept by it, even though it has a `refundAddress` parameter that is intended to handle such scenarios with a refund.

Proof of Concept:

```
function test_oftWrapperLzSend_oversend() public {
    uint256 tokensToSend = 1 ether;
    address user = utils.getNextUserAddress();

    vm.deal(user, 100 ether);
    _mintAndBridgeBitcornToCorn(user, tokensToSend, tokensToSend);

    bytes memory options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(200000, 0);
    SendParam memory sendParam =
        SendParam(ethEid, addressToBytes32(user), tokensToSend, tokensToSend, options, "", "");
    MessagingFee memory fee = wBTCN.quoteSend(sendParam, false);

    assertGe(wBTCN.balanceOf(user), tokensToSend, "User must have enough WBTCN to send");

    uint iBal = address(wBTCN).balance;
    vm.prank(user);
    wBTCN.send{value: fee.nativeFee + 5555555555}(sendParam, fee, payable(address(this)));

    console.log(address(wBTCN).balance - iBal);
    assertEq(
        address(wBTCN).balance - iBal, 0, "adapter balance should not have increased"  // fails here
    );
}
```

showcases excess of native fee being kept by an increased adapter balance being present, equivalent to the 5555555555 value oversend.

Recommendation: To be the most secure, retain the standard behaviour. This would require reverting the current flexibility of being able to partially use your wrapped balance and `msg.value` to initiate a send, needing users to always wrap their tokens first before sends.

Otherwise, consider reverting, as previous versions of this contract did, on the secondary branch when `msg.value` doesn't match the native fee. So changing

```
} else if (msg.value < feeWithExtraAmount.nativeFee
```

back to

```
} else if (msg.value != feeWithExtraAmount.nativeFee
```

A third solution which requires trusting the endpoint contract, although for the initial branch case the trust is already present, would be to always pre-set the native fee, to the current `msg.value`. That would involve changing the `feeWithExtraAmount` declaration to

```
MessagingFee memory feeWithExtraAmount = MessagingFee({nativeFee: msg.value, lzTokenFee: _fee.lzTokenFee});
```

which should also work with the insufficient wrapped branch, where the minted amount gets discounted from the fee being sent. The prior noted secondary branch check can be dropped with this pre-set, as it will always pass.

Do note, as with the initial insufficient wrapped balance branch, that this requires trusting the endpoint to handle the refund, and the various entities involved in the security stack for their LayerZero configuration to not frontrun the pricing mechanisms.

Bitcorn: Reverted to original behavior in commit 667acdc3.

Cantina Managed: Verified fix.

3.1.2 Inconsistent pause enforcement in `WrappedBitcornNativeOFTAdapter`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_credit` function in the `WrappedBitcornNativeOFTAdapter.sol` contract lacks the `whenNotPaused` modifier, allowing token credits even when the contract is paused.

This enables users to receive tokens through cross-chain transfers during a `pause`, undermining the effectiveness of the emergency pause mechanism. It also creates an inconsistent state, with some operations blocked while others continue after an emergency pause.

Recommendation: Add the `whenNotPaused` modifier to the `_credit` function to ensure consistent pause behavior across all token operations:

```
  function _credit(address _to, uint256 _amountLD, uint32 _srcEid)
      internal
      virtual
      override
+     whenNotPaused
      returns (uint256 amountReceivedLD)
  {
      (bool success,) = _to.call{value: _amountLD}("");
      if (!success) {
          revert WithdrawalFailed();
      }
      return _amountLD;
  }
```

Bitcorn: Fixed in 4adafa00

Cantina Managed: Verified fix.

3.1.3 Minting users may not be able to swap back to their original collateral token, having to rely on alternative collateral tokens

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The architecture revolves around multiple `SwapFacility` type contracts being able to mint BitcornOFT. These different swap facilities will accept a variety of wrapped BTC tokens.

This allows users that may have swapped their hypothetical wrapped aBTC tokens for BitcornOFT, to swap their debt for the entirety of the bBTC token Vault. This means that all users that may have swapped their bBTC for BitcornOFT, would not be able to swap back to bBTC, and would have to swap to aBTC. This includes the fee recipient of the bBTC Swap Facility, there would not be any bBTC collateral to back their fees, and instead it would be the aBTC and any other Swap Facility contracts.

Proof of Concept: This scenario is depicted in the following proof of concept:

```
function test_clear_vault_collateral_with_other_swapfacility() public {
    // Set fees to 10%
    uint256 swapInFeeRate = 1000;
    vm.prank(admin);
    simpleSwapFacility.setSwapInFeeRate(swapInFeeRate);

    // Setup initial swap amounts
    uint256 collateralIn = 10e8; // 10 WBTC

    address user = utils.getNextUserAddress();

    wBTC.mint(user, collateralIn);
    vm.startPrank(user);
    wBTC.approve(address(simpleSwapFacility), collateralIn);
    simpleSwapFacility.swapExactCollateralForDebt(
        collateralIn,
        1,
        user,
        block.timestamp
    );
    vm.stopPrank();

    address user2 = utils.getNextUserAddress();

    uint vaultBal = wBTC.balanceOf(address(simpleSwapFacilityVault));
    uint neededToEmpty = vaultBal * 1e10 * 10_000 / 10_000;
    vm.prank(admin);
    bitcornProxyEth.mintTo(user2, neededToEmpty); // simulate mint debt on another swapfacility
    vm.prank(user2);
    bitcornProxyEth.approve(address(simpleSwapFacility), type(uint256).max);
    vm.startPrank(user2);
    simpleSwapFacility.swapExactDebtForCollateral(
        neededToEmpty,
        1,
        user2,
        block.timestamp
    );
    vm.stopPrank();

    vaultBal = wBTC.balanceOf(address(simpleSwapFacilityVault));
    assertEq(vaultBal, 0, "vault should be emptied");
    uint256 collectableFees = simpleSwapFacility.getCollectableFees();
    assertGt(collectableFees, vaultBal);
    assertEq(simpleSwapFacility.debtMinted(), 0);
    // no vault balance backing the debt it had minted for user1, it now is backed only by other collateral
    // by another user's actions from a different vault
}
```

At face value, this seems like an inconvenience at most, that would force users to have to use another swap facility, and not get their originally wanted collateral token back, meaning they may incur additional costs from outside protocols to swap back to their original collateral.

The real issue is that if aBTC is exploited or loses its value/peg, the original aBTC depositors could swap out safely with other user's bBTC collateral, leaving the bBTC depositors with valueless aBTC. Hence the value of the BitcornOFT is only as strong as its weakest collateral link, among all debt owners for the average owner.

Recommendation: Adding metadata to each minted token that denotes the `SwapFacility` it can be redeemed from would directly address this issue, however, it would hurt the fungibility of BitcornOFT, and would make this metadata attribution complex at the omnichain level, although possible with a system that would simply distribute the tokens in the order of the swap facility's with the most unpaused backing collateral.

The current system achieves flexibility, and keeps the token completely fungible across chains, however, the equivalent value floor of BitcornOFT is equivalent to its lowest valued backing collateral. This can be somewhat addressed by active operators of the contract, in that if an exploited or depegging collateral token exists, all swap facilities should be paused, or at the very least the swap in for the exploited token disabled, and swap outs for the other collateral disabled. In this case, there'd be a socialized loss of collateral among all BitcornOFT owners, rather than a run on the bank/collateral, where a few, potentially bad collateral depositors save their bad collateral, or even worse exploiters are putting in bad collateral and draining the good collateral from other users.

If going with the latter, consider having a communicated action plan for users under such a scenario, if there will be an attempt to pause timely, and have BitcornOFT retain at least the new average value of collateral among all users, or if a run on the bank is what users should expect in this scenario.

The operators should be highly vigilant to only choose the highest quality of collateral, and for both operators and users to understand that more types of collateral, with this architecture is not more safe, but actually more risk prone, as the exploit/depeg risk is not isolated, but each collateral's risk profile cumulatively affects the system. Consider forcing an equal-weight threshold system among backing collateral to lower the effects of this risk.

Bitcorn: Agree this is a fundamental risk of this swap facility design. We anticipate BTCN to be backed in it's entirety by the weakest of the collateral assets (with relative caps, fees, and pausing as limiting factors as managed by risk council). The users have no guarantee that they will get their original backing asset out.

> The operators should be highly vigilant to only choose the highest quality of collateral, and for both operators and users to understand that more types of collateral, with this architecture is not more safe, but actually more risk prone, as the exploit/depeg risk is not isolated, but each collateral's risk profile cumulatively affects the system.

Agree, the intent is to keep the collateral backing list tightly controlled in this paradigm.

> Consider forcing an equal-weight threshold system among backing collateral to lower the effects of this risk.

In future iterations, we are exploring ideas of a "redemption priority queue", returning basket of assets, etc...

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 `Vault.initialize` lacks some sanity checks for its parameters

Severity: Low Risk

Context: SimpleSwapFacility.sol#L72-L79, Vault.sol#L22-L31

Description: The parameters `swapFacility` and `token` are explicit lacking zero-address validation. `token` is implicitly checked, in that if it is not a valid or at least masquerading ERC20 contract, the function will revert.

The `swapFacility` though could be set to `address(0)` which would leave the `Vault` in an an invalid state, and would result in a loss of any collateral swapped via associated `SwapFacility` contract, until at least least an upgrade to correct it.

Recommendation: Explicitly do zero-address validation for `swapFacility`. Additionally consider an OpenZeppelin utilities `isContract` check for `initialAuthority` or any other parameters sent that are expected to be pre-deployed contracts, thereby it may be applicable for `swapFacility`. Likewise, it would be good practice on the `swapFacility` side to actually check that it has a sufficient threshold of transfer approval from the Vault it is setting in its own initializer.

Bitcorn: Acknowledged. Will retain existing behavior and leverage script-based confirmation of correct parameters prior to deployment.

Cantina Managed: Acknowledged.

3.2.2 Missing fee-free exit path for `feeRecipient`

Severity: Low Risk

Context: SimpleSwapFacility.sol#L181

Description: In `SimpleSwapFacility.sol`, the `feeRecipient` collects fees in `debtToken` whenever users exchange their collateral for debt, or the other way around. These fees accumulate over time and can be collected through the `collectFees()` function, where the `feeRecipient` receives debt tokens.

To convert these debt tokens back to collateral tokens, they must use the `swapExactDebtForCollateral()` function, which incurs a swap-out fee rate. As a result, a portion of the fees earned will be recycled back into the fee accumulator. This creates a cycle of fee-on-fee charges whenever they attempt to convert any part of their tokens.

For instance, if both swap fees are set at 100 basis points (1%), the fee recipient would incur a loss of 1% of their earned fees when trying to convert them to collateral. These deducted fees would then be added back to the total accumulated fees, and ultimately credited to the fee recipient again, creating an infinite loop.

Recommendation: Consider creating a privileged swap function for the fee recipient to swap without swap-out fees.

Bitcorn: Acknowledging and keeping this in mind as a future upgrade option. Original behavior will be retained.

If there are fees and the fee recipient wants to redeem the underlying, the following approach will approximate the behavior described.

Given that the fee recipient is the same entity as the administrative role holder, you can temporarily allow redemption without fees by sandwiching the collectFees() call (which is permissioned) between calls that set fees to zero and then set them to the original value after the call.

This can be atomically executed by a multisig or other smart contract wallet, or a zap could be made to execute it in a way that practically has the same result of the proposed solution at the expense of more gas usage.

Cantina Managed: Acknowledged.

3.2.3 Users sending small amounts (~Satoshi or less) to `swapExactDebtForCollateral` may get no collateral

Severity: Low Risk

Context: SimpleSwapFacility.sol#L181-L202

Description: Currently the function doesn't have any checks ensuring it is actually sending out any collateral, other than ensuring it is above the user supplied `collateralOutMin`. If that parameter is set to 0, ~satoshi amounts with certain fee levels, or any sub-satoshi amounts will be taken by the swap facility in their entirety, and a transfer of 0 collateral occurs.

Recommendation: Users are provided with a method to enforce they receive something with `collateralOutMin` and should ensure it is set to non-zero to avoid this bug.

The contract authors may also wish to enforce a requirement that the final `collateralOut` value must be non-zero, or revert, to avoid potential 0 value spam token transfers being initiated from the vault on behalf of user.

Bitcorn: Mitigated in commit 67cac72f.

Cantina Managed: Verified fix.

3.3 Gas Optimization

3.3.1 Redundant external call to `decimals()` in `SimpleSwapFacility` constructor

Severity: Gas Optimization

Context: SimpleSwapFacility.sol#L87

Description: The constructor makes a redundant external call to `collateralToken.decimals()`. The value is called twice when it could be cached and reused, resulting in unnecessary gas costs during contract deployment.

```
constructor(address _collateralToken, address _debtToken, address _vault) {
    // ...
    uint8 decimals = collateralToken.decimals();  // First call
    if (decimals != 8 && decimals != 18) revert InputTokenInvalidDecimals(decimals);
    to18ConversionFactor = 10 ** uint256(18 - collateralToken.decimals());  // Second call
}
```

Saves one external call to `decimals()` during contract deployment.

Recommendation: Consider using the cached decimal value in the variable `decimals` to prevent redundant external calls.

Bitcorn: Fixed in commit 7588af00

Cantina Managed: Verified fix.

3.3.2 Implement multiple setter function for significant gas saving potential

Severity: Gas Optimization

Context: SimpleSwapFacility.sol#L54-L59, SimpleSwapFacility.sol#L302-L314

Description: With the `SimpleSwapFacility` contract now having multiple of its admin configurable state variables such as `feeRecipient`, `swapInEnabled`, and `swapOutFeeRate`, sharing a single storage slot, it is inefficient to call each individual setter separately. Implementing a multiple setter function, would allow for significant gas savings by condensing storage writes and reads into a single opcode, saving at least ~20,000 gas cost for when setting 2 at a time, or up to ~120,000 in gas savings for when configuring all 5.

This also improves configurations, in that the contract does not require a pause to safely configure multiple parameters at once for it.

Recommendation:

```
function multiSet(uint256 _feeIn,
    uint _feeOut,
    bool _swapIn,
    bool _swapOut,
    address _feeGetter) external requiresAuth {
        if (_feeIn != swapInFeeRate)
            _setSwapInFeeRate(_feeIn);
        if (_feeOut != swapOutFeeRate)
            _setSwapOutFeeRate(_feeOut);
        if (_swapIn != swapInEnabled)
            _setSwapInEnabled(_swapIn);
        if (_swapOut != swapOutEnabled)
            _setSwapOutEnabled(_swapOut);
        if (_feeGetter != feeRecipient)
            _setFeeRecipient(_feeGetter);
    }
```

is an example implementation which depends on refactoring the current setter logic to have external/internal logic patterns:

```
function setSwapOutEnabled(bool _swapOutEnabled) external requiresAuth {
    _setSwapOutEnabled(_swapOutEnabled);
}
```

```
function _setSwapOutEnabled(bool _swapOutEnabled) internal {
    swapOutEnabled = _swapOutEnabled;
    emit SwapOutEnabledSet(_swapOutEnabled);
}
```

similar is implementable across the other `SwapFacility` variant contracts, where the variables are appropriately packed alongside `feeRecipient`.

Bitcorn: Acknowledged. This would be particularly useful for setting both fees at the same time (which would commonly be the case). However, the gas cost savings are not a concern in context for infrequent administrative operations.

Cantina Managed: Acknowledged.

3.3.3 `SimpleSwapFacility.getMintableDebtAmount()` should cache its accessed state variables

Severity: Gas Optimization

Context: SimpleSwapFacility.sol#L248-L250

Description: The noted function accesses 2 state variables, under the unlikely branch once, and under the likeliest branch twice.

Recommendation: Since they are accessed twice in the likelier branch, optimize by caching their access. This will result in a slight gas overhead on the unlikely branch where the mint cap is met, for gas savings on the more frequent branch.

Bitcorn: Mitigated in commit 13a3d11e.

Cantina Managed: Verified fix.

3.3.4 Set `accumulatedFee` directly to 0 instead of subtracting by itself

Severity: Gas Optimization

Context: SimpleSwapFacility.sol#L262-L270

Description: In `SimpleSwapFacility.collectFees()`, the `accumulatedVariable` is subtracted by itself, which is equivalent to setting it to 0, while introducing additional unnecessary opcodes to achieve the same thing.

Recommendation: Be clearer, and directly set it to 0 for efficiency.

For the greatest efficiency, consider actually resetting the amount to 1, and to consider 1 as the zero/initial value for that variable, to always keep the slot nonzero for the same consistent and cheaper `SSTORE` costs on it (`accumulatedFee - 1`) would be sent out instead. It will introduce some additional complexity, but will avoid certain users having to pay more gas for swaps, due to a prior collection of fees by admins.

Bitcorn: Acknowledged optimization potential. Retaining existing behavior for keeper operation, will save for future version.

Cantina Managed: Acknowledged.

3.4 Informational

3.4.1 Remove unused file imports

Severity: Informational

Context: SwapFacilitySwapAndBridgeZap.sol#L11, SwapFacilitySwapAndBridgeZap.sol#L15

Description: The `SwapFacilitySwapAndBridgeZap.sol` contains two import statements not used anywhere in the contract, affecting code quality.

Recommendation: Consider removing unused file imports.

Bitcorn: Fixed in commit 9fc271e3.

Cantina Managed: Verified fix.

3.4.2 Minor code quality issues: typos

Severity: Informational

Context: SwapFacilitySwapAndBridgeZap.sol#L66

Description: The codebase contains minor typos that should be addressed for better code quality and maintainability.

- `SwapFacilitySwapAndBridgeZap.sol`:

```diff
- /// @dev sendParam.amountLD will be overriden to the actual amount out of the swap.
+ /// @dev sendParam.amountLD will be overridden to the actual amount out of the swap.
```

Recommendation: Consider fixing the typos.

Bitcorn: Fixed in commit d0220a02

Cantina Managed: Verified fix.

3.4.3 Utilize underscores as thousands separator to help legibility with larger values

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Underscores may be utilized in numerical values within Solidity contracts to aid in their readability, specifically pertaining to larger values, as a thousands separator.

Recommendation: Use underscores as a separator for larger values.

```
uint256 public constant MAX_BPS = 10_000;
uint256 public constant MAX_FEE_RATE = 1_000;
```

Bitcorn: Acknowledged as more clear.

Cantina Managed: Acknowledged.