Puffer contracts

Security Review

Cantina Managed review by:

Sujith Somraaj, Security Researcher
Ladboy233, Security Researcher

April 25, 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	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

Puffer is a decentralized native liquid restaking protocol (nLRP) built on Eigenlayer. It makes native restaking on Eigenlayer more accessible, allowing anyone to run an Ethereum Proof of Stake (PoS) validator while supercharging their rewards.

From Apr 16th to Apr 19th the Cantina team conducted a review of puffer-contracts on commit hash 01016568. The team identified a total of **3** issues:

Issues Found

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

3 Findings

3.1 Low Risk

3.1.1 Incorrect fee calculation in `maxWithdraw` function

Severity: Low Risk

Context: PufferVaultV5.sol#L461

Description: The `maxWithdraw()` function in `PufferVaultV5.sol` incorrectly applies exit fees to the vault's total liquidity rather than adding fees to individual withdrawal requests. This leads to an artificial reduction in the reported maximum withdrawal amount, potentially causing confusion and preventing users from withdrawing their full entitled assets.

Users will be shown a lower maximum withdrawal amount than what the protocol can support. This creates an artificial liquidity constraint that becomes more severe as the exit fee increases. For example, with a 1% exit fee, only around 99% of the actual liquidity would be available for withdrawal. The current implementation calculates available liquidity as:

```
uint256 availableLiquidity = vaultLiquidity - _feeOnRaw(vaultLiquidity, getExitFeeBasisPoints());
```

However, the protocol's withdrawal mechanism adds fees to individual withdrawal amounts rather than reducing the total liquidity pool. This is evident in the `previewWithdraw()` function, which adds fees to the requested withdrawal amount:

```
function previewWithdraw(uint256 assets) public view virtual override returns (uint256) {
    uint256 fee = _feeOnRaw(assets, getExitFeeBasisPoints());
    return super.previewWithdraw(assets + fee);
}
```

Proof of Concept: Place the following test inside `test` folder:

```
import {Test} from "forge-std/Test.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "src/PufferVaultV5.sol";

import "forge-std/console.sol";

interface Upgrade {
    function upgradeToAndCall(address newImpl, bytes memory data) external payable;
}

contract AuditTest is Test {
    PufferVaultV5 public vaultImpl;
    PufferVaultV5 public vault;

    function setUp() external {
        vm.createSelectFork("https://mainnet.infura.io/v3/<YOUR INFURA KEY>");
        vault = PufferVaultV5(payable(0xD9A442856C234a39a81a089C06451EBAa4306a72));
        vaultImpl = new PufferVaultV5(IStETH(0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84),
        ↪   ILidoWithdrawalQueue(address(0)), IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2),
        ↪   IPufferOracleV2(0x0BE2aE0edbeBb517541DF217EF0074FC9a9e994f),
        ↪   IPufferRevenueDepositor(0x21660F4681aD5B6039007f7006b5ab0EF9dE7882));

        vm.startPrank(0x3C28B7c7Ba1A1f55c9Ce66b263B33B204f2126eA);
        Upgrade(0xD9A442856C234a39a81a089C06451EBAa4306a72).upgradeToAndCall(address(vaultImpl), "");
    }

    function test_maxRedeem() external {
        address user = address(420);
        deal(vault.asset(), user, 100000e18);

        vm.startPrank(user);
        ERC20(vault.asset()).approve(address(vault), 100000e18);
        vault.deposit(100000e18, user);

        vm.startPrank(address(vault));
        ERC20(vault.asset()).transfer(address(0), 100000e18);

        console.log("user balance:", vault.balanceOf(user));
        uint256 maxRedeem = vault.maxRedeem(user);
        uint256 maxWithdraw = vault.maxWithdraw(user);
        console.log("max withdraw:", maxWithdraw);
        console.log("max redeem:", maxRedeem);
        console.log("preview redeem:", vault.previewRedeem(vault.balanceOf(user)));

        vm.startPrank(user);
        uint256 assets = vault.withdraw(maxWithdraw, user, user);
        console.log("assets:", assets);

        console.log(ERC20(vault.asset()).balanceOf(address(vault)));
        console.log(address(vault).balance);
    }
}
```

Using the original code, the user can withdraw `2723.01E` from the vault, even though there is more than `2750E` available in liquidity. This discrepancy arises because the calculation in the maxWithdraw function is flawed. In the same circumstances, if the user uses `maxRedeem()`, they can redeem the full `2750E`.

Recommendation:

1. The `maxWithdraw()` function should not subtract fees from the total liquidity. Instead, it should simply compare the user's assets with the vault's total liquidity:

   ```
   function maxWithdraw(address owner) public view virtual override returns (uint256 maxAssets) {
       uint256 maxUserAssets = previewRedeem(balanceOf(owner));
       uint256 vaultLiquidity = (_WETH.balanceOf(address(this)) + (address(this).balance));

       // Return the minimum of user's assets and available liquidity
       return Math.min(maxUserAssets, vaultLiquidity);
   }
   ```

   This correction ensures that the maximum withdrawal amount is properly calculated, allowing users to withdraw their full entitled assets up to the available liquidity.

2. Consider adding fuzz tests to ensure all cases are properly covered for the function.

Puffer: Fixed in PR 112.

Cantina Managed: Fix verified.

3.2 Informational

3.2.1 `PufferModuleManager#customExternalCall` does not check if calldata is validated in `avsRegistryCoordinator`

Severity: Informational

Context: PufferModuleManager.sol#L268-L277

Description: In `RestakingOperatorController.sol`, the code that triggers external calls in `restakingOperator` validates if the custom call is allowed via `checkCustomCallData`:

```
/**
 * @notice Custom external call to the restaking operator
 * @dev This function can be called by Operator owners
 * @param restakingOperator The address of the restaking operator
 * @param data The data to call the restaking operator with
 */
function customExternalCall(address restakingOperator, bytes calldata data) external payable override {
    require(_operatorOwners[restakingOperator] == msg.sender, NotOperatorOwner(restakingOperator, msg.sender));
    bytes4 selector = bytes4(data[:4]);
    require(_allowedSelectors[selector], NotAllowedSelector(selector));
    if (selector == CUSTOM_CALL_SELECTOR) {
        _checkCustomCallData(data);
    }
    (bool success,) = restakingOperator.call{ value: msg.value }(data);
    require(success, CustomCallFailed());
    emit CustomExternalCall(restakingOperator, data, msg.value);
}
```

and

```
function _checkCustomCallData(bytes calldata data) private view {
    (address avsRegistryCoordinator, bytes memory customCalldata) = abi.decode(data[4:], (address, bytes));
    require(
        _avsContractsRegistry.isAllowedRegistryCoordinator(avsRegistryCoordinator, customCalldata),
        ↪  Unauthorized()
    );
}
```

But in `PufferModuleManager#customExternalCall`, the code that triggers external calls in `restakingOperator` has no equivalent validation.

The calldata and target that is not allowed in `avsRegistryCoordinator` can still be executed via `PufferModuleManager#customExternalCall`.

Recommendation: Consider adding `_checkCustomCallData` in `PufferModuleManager#customExternalCall` as well.

Puffer: Acknowledged. This is intentional because our Multisig has the capability to execute `PufferModuleManager#customExternalCall`. Our goal is to limit restaking operator administrators with `AvsRegistryCoordinator`.

Cantina Managed: Acknowledged.

3.2.2 Incorrect function signature in `CUSTOM_CALL_SELECTOR` comment

Severity: Informational

Context: RestakingOperatorController.sol#L20

Description: The comment for the `CUSTOM_CALL_SELECTOR` constant in the `RestakingOperatorController.sol` contract contains an incorrect function signature description that does not match the actual selector value. Current code:

```
bytes4 private constant CUSTOM_CALL_SELECTOR = 0x58fa420c; //
↪  bytes4(keccak256("customCalldataCall(address,bytes)(address,bytes,uint256)"))
```

The selector value `0x58fa420c` corresponds to:

```
bytes4(keccak256("customCalldataCall(address,bytes)"))
```

The comment incorrectly includes return types and uses an invalid function signature format with separate parentheses for inputs and outputs.

Recommendation: Update the comment to correctly document the function signature:

```
bytes4 private constant CUSTOM_CALL_SELECTOR = 0x58fa420c; //
↪  bytes4(keccak256("customCalldataCall(address,bytes)"))
```

Puffer: Fixed in PR 113.

Cantina Managed: Fix verified.