Puffer Finance Initial Audit Report

Version 0.1

0xLuckyLuke

January 15, 2024

Puffer Finance Audit Report

0xLuckyLuke

Jan 15, 2024

Puffer Finance Audit Report

Prepared by: 0xLuckyLuke

Table of Contents

See table

    - [L-1] Missing Input Validation.
    - [L-2] Code Architecture and Unresolved Issues.
    - [L-3] USDT allowance on Ethereum mainnet has a special requirement.
    - [L-4] If the recipient is added to the USDC blacklist, then these functions will not work.
    - [L-5] Incomplete tests.

- Informational

    - [I-1] Using public visibility instead of external can increase gas.
    - [I-2] Incomplete README file.
    - [I-3] Use of hard-coded addresses in PufferDepositor::[24-30] PufferVault::[37-40] and may cause errors.
    - [I-4] Repeated code in PufferDepositor contract.
    - [I-5] Unlocked Pragma.
    - [I-6] Privileged Roles.
    - [I-7] Uinitialized contract.
    - [I-8] Greedy contract.
    - [I-9] Functions Can Fail Due to High Gas Usage.
    - [I-10] ERC4626 does not work with fee-on-transfer tokens.
    - [I-11] SUSHI_ROUTER.processRoute is pausable function.
    - [I-12] unrestricted for loop processes an unlimited array.
    - [I-13] Event is missing indexed fields.
    - [I-14] Handle Storage for upgradable contract.
    - [I-15] authorizeUpgrade() may allow unauthorized upgrades, and calls to unimplemented functions.

Disclaimer

Our assessment report offers insights based on a defined scope and timeframe, using provided materials. While we strive to identify vulnerabilities, our report may not cover all potential issues and should not be seen as exhaustive. We present findings without warranties, and we do not endorse specific projects or assure complete security. We cannot guarantee the absence of subsequent issues or offer investment advice. Smart contract security reviews have limitations, and subsequent reviews, bug bounty programs, and monitoring are recommended. Our report does not guarantee the discovery of all security issues, and we advise independent audits and bug bounty programs for comprehensive security assurance. The scope of our assessment is limited to specified code sections, excluding underlying language, compiling toolchain, and infrastructure security.

Risk Classification

	Impact-High	Impact-Medium	Impact-Low
Likelihood-High	H	H/M	M
Likelihood-Medium	H/M	M	M/L
Likelihood-Low	M	M/L	L

High: Funds are directly at risk, or a severe disruption of the protocol's core functionality.

Medium: Funds are indirectly at risk, or some disruption of the protocol's functionality.

Low: Funds are not at risk.

Audit Details

The findings described in this document correspond the following:

Repository URL:

```
https://github.com/PufferFinance/pufETH
```

commit hash:

```
5c0d51fc6700530dedc3d5a6f6009918fc8ea398
```

Scope

```
src/
--- NoImplementation.sol
--- PufferDepositor.sol
--- PufferDepositorStorage.sol
--- PufferOracle.sol
--- PufferVault.sol
--- PufferVaultMainnet.sol
--- PufferVaultStorage.sol
--- Timelock.sol
```

Protocol 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.

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	0
Low	5
Info	15
Gas Optimizations	0
Total	0

Findings

High

[H-1] The absence of guardianSignatures checking allows any address to set the variables of this function.

Description: The PufferOracle::proofOfReserve function is incomplete.

Impact: any address to set the variables of this function.

Proof of Concept:

```
    function proofOfReserve(
        uint256 newEthAmountValue,
        uint256 newLockedEthValue,
        uint256 pufETHTotalSupplyValue, // @todo what to do with this?
        uint256 blockNumber,
        uint256 numberOfActiveValidators,
        bytes[] calldata guardianSignatures
    ) external {
        // Check the signatures (reverts if invalid)
        // GUARDIAN_MODULE.validateProofOfReserve({
        //     ethAmount: ethAmount,
        //     lockedETH: lockedETH,
        //     pufETHTotalSupply: pufETHTotalSupply,
        //     blockNumber: blockNumber,
        //     numberOfActiveValidators: numberOfActiveValidators,
        //     guardianSignatures: guardianSignatures
        // });

        // if ((block.number - lastUpdate) < _UPDATE_INTERVAL) {
        //     revert OutsideUpdateWindow();
        // }

        ethAmount = newEthAmountValue;
        lockedETH = newLockedEthValue;
        pufETHTotalSupply = pufETHTotalSupply;
        lastUpdate = blockNumber;

        emit BackingUpdated(newEthAmountValue, newLockedEthValue,
           pufETHTotalSupplyValue, blockNumber);
    }
```

Recommended Mitigation: Add access control (and preventing front running too but it seems front running in this function is not very important)

Low

[L-1] Missing Input Validation.

Description: Functions in the PufferDepositor and PufferVault contracts lacking proper input validation:

PufferDepositor::constructor(): Missing validation on pufferVault. PufferDepositor::initialize(): Missing validation on accessManager. PufferDepositor::_allowToken(): Missing validation on token. PufferDepositor::_disallowToken(): Missing validation on token. PufferVault::initialize(): Missing validation on accessManager. PufferDepositor::initiateETHWithdrawalsFromLido(): Missing validation on amounts. PufferDepositor::swapAndDeposit. PufferDepositor::swapAndDepositWithPermit.

Impact: The absence of input validation poses security risks, including input manipulation and unauthorized access, compromising contract integrity and fund security.

Recommended Mitigation: Implement robust input validation in affected functions to prevent vulnerabilities. Validate addresses, tokens, and withdrawal amounts to enhance contract security.

[L-2] Code Architecture and Unresolved Issues.

Description: Code architecture, questions and issues should be resolved before deployment. There are 4 instances of this issue: File: src/PufferOracle.sol #1

```
55: uint256 pufETHTotalSupplyValue, // @todo what to do with this?
```

File: src/PufferVault.sol #2

```
75: //@todo can probably be optimized
```

File: src/PufferVaultMainnet.sol #3,4

```
37: + address(this).balance; //@todo when you add oracle pufferOracle.
    getLockedEthAmount()
40: //@todo weth wrapping and unwrapping logic, native ETH deposit
    method
```

Impact: Leaving these questions and issues unresolved could lead to confusion, inefficient code, or even unexpected behavior in the deployed contracts. It's crucial to address these concerns to ensure the reliability and efficiency of the codebase.

Recommended Mitigation: Before deploying the contracts, it's essential to resolve these questions and issues by providing clear explanations, optimizing where possible, and implementing necessary logic for efficient operation. This involves reviewing the code architecture, addressing outstanding questions, and refining functionalities for smooth deployment and operation.

[L-3] USDT allowance on Ethereum mainnet has a special requirement.

Description: USDT has a race condition protection mechanism on ethereum chain that it does not allow users to change the allowance without first changing the allowance to 0. Here is the approve function in USDT on Ethereum: https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code#L205

```
    function approve(address _spender, uint _value) public
       onlyPayloadSize(2 * 32) {

          // To change the approve amount you first have to reduce the
             addresses`
          //  allowance to zero by calling `approve(_spender, 0)` if it
             is not
          //  already 0 to mitigate the race condition described here:
```

```
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment
            -263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)
            ));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

The detaliled description mentioned here: https://solodit.xyz/issues/m-07-trading-will-not-work-on-ethereum-if-usdt-is-used-code4rena-tigris-trade-tigris-trade-contest-git

Impact: Regarding the PufferDepositor::swapAndDeposit function that needs the client side approve as mentioned in the tests, the approve function is called with type(uint256).max amount but the swap amount is less than this. As a result, the transaction will revert when calling approve for the second time.

```
File: test/integration/PufferTest.integration.t.sol

// USDT doesn't have a permit, so the user needs to approve it to our
    contract
SafeERC20.safeIncreaseAllowance(IERC20(USDT), address(pufferDepositor),
     type(uint256).max);
pufferDepositor.swapAndDeposit({ amountIn: tokenInAmount, tokenIn: USDT
    , amountOutMin: 0, routeCode: routeCode });
```

Recommended Mitigation: To address the USDT allowance race condition issue:

1. Before calling swapAndDeposit, manage USDT allowance properly.
2. Follow USDT's race condition protection: If allowance isn't zero, set it to zero first. Then, set the new allowance value. Here's the updated code for swapAndDeposit:

```
function swapAndDeposit(uint256 amountIn, address tokenIn) external {
    uint256 currentAllowance = IERC20(tokenIn).allowance(msg.sender,
        address(this));
    if (currentAllowance != 0) {
        require(IERC20(tokenIn).approve(address(this), 0), "Failed to
            reset allowance");
    }

    require(IERC20(tokenIn).approve(address(this), type(uint256).max),
        "Failed to approve allowance");

    // Continue with the swap and deposit operation
    // [Insert remaining logic for swap and deposit]
}
```

Test the updated function thoroughly to ensure compliance and prevent transaction reversion due to allowance race conditions. Clearly document the changes for reference.

[L-4] If the recipient is added to the USDC blacklist, then these functions will not work.

Description: There's a problem that might affect functions—PufferDepositorswapAndDeposit, Puffer-Depositor::swapAndDepositWitermit, and PufferDepositor::depositWstETH—if the person receiving the tokens is put on a "blacklist" list for USDC. This could let a tricky person stop USDC transactions, making it tough for others to use these functions.

Impact: This issue makes deposit-related functions risky and could even cause a kind of "service denial" if the person's address is put on the blacklist list for USDC.

Recommended Mitigation: 1. Check the List: Make sure to check the list before doing anything in the affected functions to make sure the person's address isn't on the blacklist for USDC.

```
if ($.allowedTokens[IERC20(tokenIn)] == false || isBlacklisted(msg.
    sender, USDC)) {
     revert TokenNotAllowed(tokenIn);
```

2. Hold Off on Sending Tokens: Instead of sending tokens right away, think about keeping track of how many there are. Let the person get them later to avoid the risk of being on a blacklist list.

```
     SafeERC20.safeTransferFrom(IERC20(tokenIn), msg.sender, address(
         this), amountIn);
     return _PUFFER_VAULT.deposit(stETHAmountOut, msg.sender);
```

[L-5] Incomplete tests.

Description: The overview of the test coverage can bee seen by forge coverage command. The detailed info about uncovered lines of code can bee seen by the following command:

```
forge coverage --report debug
```

Informational

[I-1] Using public visibility instead of external can increase gas.

Description: To improve gas PufferDepositor::swapAndDeposit and PufferDepositor::swapAndDepositWithPermit functions can be defined as external.

[I-2] Incomplete README file.

Description: Doesn’t mention installing dependencies how to build and run the tests.

[I-3] Use of hard-coded addresses in PufferDepositor::[24-30] PufferVault::[37-40] and may cause errors.

Recommended Mitigation: Set addresses when contracts are created rather than using hard-coded values. This practice will facilitate testing.

[I-4] Repeated code in PufferDepositor contract.

Recommended Mitigation: Remove follwing line:

```
    import { ERC4626Upgradeable } from "@openzeppelin-contracts-
       upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol";
    - import { ERC4626Upgradeable } from "@openzeppelin-contracts-
       upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol";
```

[I-5] Unlocked Pragma.

Description: Each Solidity file begins with a header that declares its version using the format solidity ˆ0.8.*. The presence of the caret symbol (ˆ) preceding the version number indicates an “unlocked pragma.” This suggests that the compiler is configured to utilize the designated version or any newer version, thus explaining why it is referred to as “unlocked.”

Recommended Mitigation: To ensure uniformity and avoid unforeseen issues in the future, it is advisable to omit the caret symbol. This action “locks” the file to a particular version of Solidity, thereby stabilizing its behavior.

[I-6] Privileged Roles.

Description: InPufferOracle ::proofOfReserve The guard can set the values of this function intentionally or accidentally wrong, while most of the variables of this function can be obtained through calculation or asking external oracles.

[I-7] Uinitialized contract.

Description: The PufferDepositor contract is inherited from UUPSUpgradeable contract but never initialized. Although the __UUPSUpgradeable_init function has no implementation, the contract is upgradable and might change in the future.

[I-8] Greedy contract.

Description: PufferVault is a greedy contract that can receive ether which can never be redeemed. The receive() function in this contract allows the contract to receive Ether but has no mechanism for withdrawing these Ethers.

[I-9] Functions Can Fail Due to High Gas Usage.

Description: The claimWithdrawalsFromLido() and initiateETHWithdrawalsFromLido() functions in PufferVault contract include loops with no limit on the size of the arrays being iterated, which may cause the failure of operation due to high gas consumption.

[I-10] ERC4626 does not work with fee-on-transfer tokens.

Description: https://github.com/code-423n4/2022-02-tribe-turbo-findings/issues/26 https://solodit.xyz/issues/m-26-erc4626-does-not-work-with-fee-on-transfer-tokens-sherlock-astaria-astaria-git

usdc & usdt are fee-on-transfer tokens

[I-11] SUSHI_ROUTER.processRoute is pausable function.

Description: if it paused all deposit function of this platform will pause too.

Recommended Mitigation: Consider updating the address or use multiple routing options for these situations.

[I-12] unrestricted for loop processes an unlimited array.

Description: This finding is informational and highlights a concern in the initiateETHWithdrawals-FromLido function, where an unrestricted for loop processes an unlimited array. The code employs a for loop without imposing a limit on the array size, potentially leading to gas inefficiencies. While not posing an immediate Denial-of-Service (DOS) attack risk, this practice could result in higher gas costs, especially if an excessively large array is processed.

Impact: The vulnerability is more of a gas efficiency concern rather than a direct risk of DOS attack. Processing an unlimited array might lead to higher gas consumption, impacting transaction costs.

Recommended Mitigation: Be mindful of the potential gas costs associated with processing large arrays. While not an immediate security risk, this practice might lead to increased transaction expenses. PufferVault::claimWithdrawalsFromLido initiateETHWithdrawalsFromLido

[I-13] Event is missing indexed fields.

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

```
    Found in src/PufferOracle.sol: Line: 17
    Found in src/interface/IPufferDepositor.sol: Line: 20
    Found in src/interface/IPufferDepositor.sol: Line: 24
    Found in src/interface/IPufferVault.sol: Line: 13
    Found in src/interface/IPufferVault.sol: Line: 17
    Found in src/interface/Other/IWETH.sol: Line: 7
    Found in src/interface/Other/IWETH.sol: Line: 8
```

[I-14] Handle Storage for upgradable contract.

Description: Do we need to add a gap in the variables sections to allow defining new variables or it's handled with other strategies?

[I-15] authorizeUpgrade() may allow unauthorized upgrades, and calls to unimplemented functions.

Description: Security risks in authorizeUpgrade() may allow unauthorized upgrades, and calls to unimplemented functions can lead to unexpected issues. PufferDepositor::_authorizeUpgrade PufferVault::_authorizeUpgrade PufferVaultMainnet::_authorizeUpgrade

Impact: Unauthorized upgrades pose security risks, and unimplemented function calls may lead to unexpected vulnerabilities.

Recommended Mitigation:

1. Access Control for Upgrade: Implement proper access controls in authorizeUpgrade() to limit usage to authorized entities.
2. Review Function Calls: Check and remove calls to unimplemented functions, ensuring smoother operation.