Security Audit Report for pufETH Contracts

Date: Apr 08, 2024

Version: 1.1

Contact: contact@blocksec.com

BlockSec Audit
Team

Digitally signed by BlockSec
Audit Team
Date: 2024.04.08 12:09:54
+08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts . . . 1

1.2 Disclaimer . . . 1

1.3 Procedure of Auditing . . . 2

  1.3.1 Software Security . . . 2

  1.3.2 DeFi Security . . . 2

  1.3.3 NFT Security . . . 3

  1.3.4 Additional Recommendation . . . 3

1.4 Security Model . . . 3

2 Findings **4**

2.1 Software Security . . . 4

  2.1.1 Potential `txHash` conflicts in the `Timelock` contract's pending queue . . . 4

2.2 Additional Recommendation . . . 5

  2.2.1 Remove duplicated code . . . 5

  2.2.2 Revise the compiler version . . . 5

  2.2.3 Add a sanity check on `newPauser` . . . 5

  2.2.4 Revise the inconsistent access controls on deposit logic . . . 6

2.3 Note . . . 6

  2.3.1 Potential risks of MEV attacks . . . 6

  2.3.2 Ensure the standard implementation of `accessManager` . . . 6

  2.3.3 Necessity to implement a fair `EigenLayer` airdrop distribution mechansim . . . 7

  2.3.4 Ensure no `stETH` tokens remain in the `PufferDepositor` contract . . . 7

Report Manifest

Item	Description
Client	Puffer Finance
Target	pufETH Contracts

Version History

Version	Date	Description
1.0	Jan 29, 2024	First Release
1.1	Apr 08, 2024	Fix the incorrect commit hash

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of pufETH Contracts[1] of Puffer Finance. The pufETH Contracts serve as a native liquid restaking token. Before the mainnet launch of Puffer, users could deposit `stETH` into the `PufferVault` and receive `pufETH` in return. Additionally, users can also utilize interfaces provided by `PufferDepositor` to swap tokens for `stETH` and deposit them into `PufferVault`. In the protocol, three different multisignature wallets control sensitive operations via the `Timelock` contract. These operations include modifying system configurations, suspending core contract functionality, depositing user-deposited `stETH` into `EigenLayer`, and initiating withdrawals from `EigenLayer` and `Lido`. Please note that the scope of this audit is limited to the following files:

- PufferDepositor.sol
- PufferVault.sol
- Timelock.sol
- DeployPuffETH.s.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
pufETH Contracts	`Version 1`	`9a2a470bd276b850daf66b15463d0a9ad9b38a0f`
	`Version 2`	`c46d4f1de6e22b2b8ff33111a7852225aef443e6`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always

---

[1] `https://github.com/PufferFinance/pufETH`

recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

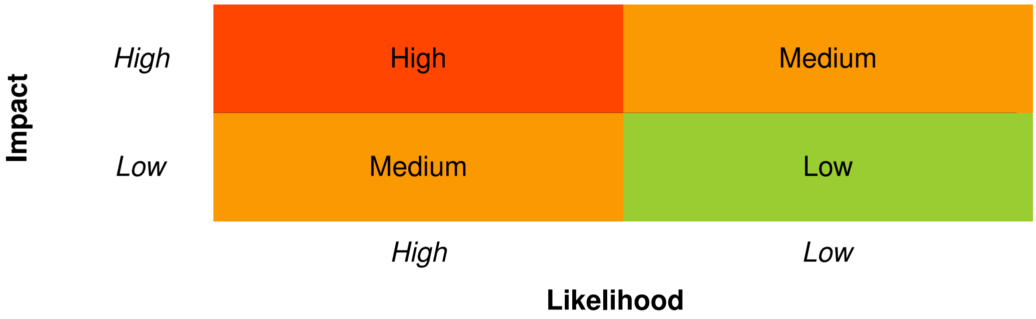
Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification



Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

---

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **one** potential issue. Besides, we also have **four** recommendations and **four** notes.

- Low Risk: 1
- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	Low	Potential `txHash` conflicts in the `Timelock` contract's pending queue	Software Security	Fixed
2	-	Remove duplicated code	Recommendation	Fixed
3	-	Revise the compiler version	Recommendation	Fixed
4	-	Add a sanity check on `newPauser`	Recommendation	Fixed
5	-	Revise the inconsistent access controls on deposit logic	Recommendation	Fixed
6	-	Potential risks of MEV attacks	Note	-
7	-	Ensure the standard implementation of `accessManager`	Note	-
8	-	Necessity to implement a fair `EigenLayer` airdrop distribution mechansim	Note	-
9	-	Ensure no `stETH` tokens remain in the `PufferDepositor` contract	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential `txHash` conflicts in the `Timelock` contract's pending queue

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Timelock` contract, the `queueTransaction` function keeps track of pending operations using the `queue` mapping, with the `txHash` serving as the key. This `txHash` is the `keccak256` hash resulting from the encoding of `target` and `callData`, as shown on line 123. Operations are eligible for execution only after the `lockedUntil` time has passed. However, there is a risk that a new operation with the same `target` and `callData` could overwrite an existing one in the queue.

```
118    function queueTransaction(address target, bytes memory callData) public returns (bytes32) {
119        if (msg.sender != OPERATIONS_MULTISIG) {
120            revert Unauthorized();
121        }
122
123        bytes32 txHash = keccak256(abi.encode(target, callData));
124        uint256 lockedUntil = block.timestamp + delay;
125        queue[txHash] = lockedUntil;
126
```

```
        emit TransactionQueued(txHash, target, callData, lockedUntil);

        return txHash;
    }
```

Listing 2.1: Timelock.sol

Impact The previous conflicting operation cannot be executed due to the overwriting.

Suggestion Add a unique `operation_id` for each operation.

2.2 Additional Recommendation

2.2.1 Remove duplicated code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `initiateStETHWithdrawalFromEigenLayer` function of the `PufferVault` contract, the assignment on line 171 is duplicated and can be removed.

```
170    uint256[] memory strategyIndexes = new uint256[](1);
171    strategyIndexes[0] = 0;
```

Listing 2.2: PufferVault.sol

Impact N/A

Suggestion Remove the duplicated code.

2.2.2 Revise the compiler version

Status Fixed in `Version 2`

Introduced by `Version 1`

Description To enable naming the mapping parameters, the `Timelock` contract should specify that the compiler version is equal to or greater than `0.8.18`.

```
mapping(bytes32 transactionHash => uint256 lockedUntil) public queue;
```

Listing 2.3: Timelock.sol

Impact N/A

Suggestion Revise the compiler version.

2.2.3 Add a sanity check on `newPauser`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `_setPauser` function in the `Timelock` contract does not verify whether the `newPauser` is non-zero.

```
function setPauser(address newPauser) public {
    if (msg.sender != address(this)) {
        revert Unauthorized();
    }
    _setPauser(newPauser);
}
```

Listing 2.4: Timelock.sol

Impact N/A

Suggestion Add the corresponding sanity check.

2.2.4 Revise the inconsistent access controls on deposit logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description An inconsistency exists in the access controls between the `PufferDepositor` and `PufferVault` contracts regarding the deposit functionality.

Specifically, the `PufferDepositor` contract declares the depositing functions (`swapAndDeposit1Inch`, `swapAndDepositWithPermit1Inch`, `swapAndDeposit`, `swapAndDepositWithPermit`, and `depositWstETH`) with a `restricted` modifier for access control. However, this access control is not applied to the `deposit` and `mint` functions in the `PufferVault` contract. This means that, even when the `PufferDepositor` contract is paused, the `deposit` and `mint` functions can still be invoked without limitations. Consequently, users can still exchange tokens for `stETH` directly on a third-party DEX and deposit them into the `PufferVault` without any restrictions.

Impact N/A

Suggestion Apply consistent access control logic for depositing functions.

2.3 Note

2.3.1 Potential risks of MEV attacks

Description The `swapAndDeposit1Inch` function in the `PufferDepositor` contract allows users to specify swap parameters for swapping tokens for `stETH` via the `_1INCH_ROUTER`. However, the function does not verify whether slippage protection is set in the `callData`, potentially exposing users to sandwich attacks.

Feedback from the Project We acknowledge this as a risk and will have warnings on the frontend to set their slippage accordingly when interfacing with 1inch / sushi.

2.3.2 Ensure the standard implementation of `accessManager`

Description The `accessManager` contract code is not included in the provided repository or dependencies and is therefore outside the scope of this audit. Given that the `accessManager` manages critical access controls, it is assumed for the purposes of this audit that its implementation follows the standardized OpenZeppelin `AccessManager`. Furthermore, it is recommended that the `accessManager` be governed by a multisignature wallet to mitigate potential risks of centralization.

```
function initialize(address accessManager) external initializer {
    __AccessManaged_init(accessManager);
    __ERC20Permit_init("pufETH");
    __ERC4626_init(_ST_ETH);
    __ERC20_init("pufETH", "pufETH");
}
```

Listing 2.5: PufferDepositor.sol

Feedback from the Project We deploy OZ's `AccessManager` in the deployPufETH.s.sol, the ownership of `AccessManager` is transferred to `TimeLock` after deployment.

2.3.3 Necessity to implement a fair `EigenLayer` airdrop distribution mechansim

Description The `PufferVault` deposits `stETH` into `EigenLayer` on behalf of users to farm points for `EigenLayer` airdrops. However, in the current `PufferVault` contract implementation, there is no mechanism to distribute airdrops to depositors. The project should ensure that there is a fair mechanism for distributing airdrops from EigenLayer. If the distribution relies solely on user shares in the vault, it may introduce the potential for front-running arbitrage.

Feedback from the Project This would be done in mainnet implementation of the contract. Will likely transfer Eigen tokens to a distributor contract.

2.3.4 Ensure no `stETH` tokens remain in the `PufferDepositor` contract

Description The `PufferDepositor` contract allows users to swap any tokens on third-party DEXes (e.g., `1inch`, `SushiSwap`) for `stETH` and deposits the acquired `stETH` into the `PufferVault` contract. However, any remaining `stETH` in the `PufferDepositor` contract could potentially be claimed by anyone. For instance, the `swapAndDeposit1Inch` function does not validate the `swapData` passed to `1inch`. Exploiting this, users could manipulate the `amountOut` returned and claim extra `stETH`.

```
function swapAndDeposit1Inch(address tokenIn, uint256 amountIn, bytes calldata callData)
    public
    virtual
    restricted
    returns (uint256 pufETHAmount)
{
    SafeERC20.safeTransferFrom(IERC20(tokenIn), msg.sender, address(this), amountIn);
    SafeERC20.safeIncreaseAllowance(IERC20(tokenIn), address(_1INCH_ROUTER), amountIn);

    // PUFFER_VAULT.deposit will revert if we get no stETH from this contract
    (bool success, bytes memory returnData) = _1INCH_ROUTER.call(callData);
    if (!success) {
        revert SwapFailed(address(tokenIn), amountIn);
    }

    uint256 amountOut = abi.decode(returnData, (uint256));

    if (amountOut == 0) {
        revert SwapFailed(address(tokenIn), amountIn);
```

```
        }

        return PUFFER_VAULT.deposit(amountOut, msg.sender);
    }
```

Listing 2.6: PufferDepositor.sol

According to the protocol design, the `PufferDepositor` contract is not intended to hold assets, rendering the aforementioned attack vector unfeasible. The only exception applies to scenarios involving accidental token transfers to this contract.

Feedback from the Project The `PufferDepositor` does not custody funds. We are relying on the 1inch backend to supply a correct route and then the stETH is deposited to the PufferVault.