Security Audit Report for Puffer Locker

Date: June 3, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer Locker

Version History

Version	Date	Description
1.0	June 3, 2025	First release

Signature

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 14 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 [1] of Puffer Locker of Puffer Finance. This contract is solely intended for protocol governance. Users can deposit a certain amount of PUFFER tokens and lock them for a specified period, during which the contract mints a corresponding amount of non-transferable vlPUFFER. Holding vlPUFFER grants users voting rights and participation in protocol governance. Once the lock period expires, users can burn their vlPUFFER to reclaim their original PUFFER tokens. Note this audit only focuses on the smart contracts in the following directories/files:

- src/vlPUFFER.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.

Other files are not within the scope of this audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

Project	Version	Commit Hash
Puffer Locker	Version 1	c64d833c0e3fe7878f6aee2b7ec0f53fdc0b9a4e
	Version 2	f9d66be7549d2c498dcd920f015e4f52ea18e8dd

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

[1] https://github.com/PufferFinance/puffer-locker/blob/dev/src/vlPUFFER.sol

not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always 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 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external calls and control flow
* Exception handling
* Data handling and flow
* Events operations
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanisms
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas efficiency
* Code quality and style

* Redundant logic and code
* Parameter validations
* Documentation and comments

1.3.3 Note

* Centralization risks
* Off-chain dependencies
* Threat modeling
* Protocol-specific assumptions

Note *The listed checkpoints cover the primary focus areas. Additional checks may be applied depending on the project's design. The audit emphasizes identifying security vulnerabilities rather than verifying standard functionality. When specifications are clear, we assume functional correctness and concentrate on uncovering potential security issues.*

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 five categories:

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

- **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.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we have **two** recommendations.

- Recommendation: 2

ID	Severity	Description	Category	Status
1	-	Incorrect comment on vlPUFFER minting logic	Recommendation	Fixed
2	-	Lack of check in function `kickUsers()`	Recommendation	Confirmed

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Incorrect comment on vlPUFFER minting logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users can invoke the `createLock()` function to deposit `PUFFER` tokens in exchange for `vlPUFFER`, calculated as `amount * multiplier`, where `multiplier` represents the number of months the tokens will be locked. However, a `comment` in the code suggests that locking 100 `PUFFER` for 2 years yields 24,000 `vlPUFFER`, which is incorrect based on the actual logic (100 `PUFFER` * 24 months = 2,400 `vlPUFFER`). This inconsistency between the comment and the implementation may confuse developers or integrators reviewing the code.

```
// If a user locks 100 PUFFER tokens for 2 years, they will get 24000 vlPUFFER
```

Listing 2.1: vlPUFFER.sol

```
function createLock(uint256 amount, uint256 multiplier) external {
   _createLock(amount, multiplier);
}
```

Listing 2.2: vlPUFFER.sol

```
function _createLock(uint256 amount, uint256 multiplier) internal onlyValidMultiplier(
    multiplier) whenNotPaused {
   require(amount >= _MIN_LOCK_AMOUNT, InvalidAmount());
   require(lockInfos[msg.sender].pufferAmount == 0, LockAlreadyExists());


   // Transfer PUFFER tokens to this contract using SafeERC20
   PUFFER.safeTransferFrom(msg.sender, address(this), amount);


   uint256 unlockTime = _calculateUnlockTime(multiplier);
   uint256 vlPUFFERAmount = amount * multiplier;


```

```
        uint256 supplyBefore = totalSupply();


        // Mint the vlPUFFER (non transferable)
        _mint(msg.sender, vlPUFFERAmount);


        // Update the lock information
        lockInfos[msg.sender] = LockInfo({ pufferAmount: amount, unlockTime: unlockTime });


        // delegate the voting power to themselves
        _delegate(msg.sender, msg.sender);


        emit Deposit({ user: msg.sender, pufferAmount: amount, unlockTime: unlockTime,
            vlPUFFERAmount: vlPUFFERAmount });
        emit Supply({ previousSupply: supplyBefore, currentSupply: totalSupply() });
    }
```

Listing 2.3: vlPUFFER.sol

Suggestion Revise the comment to ensure it accurately reflects the implemented logic and remains consistent with the code.

2.1.2 Lack of check in function `kickUsers()`

Status Confirmed

Introduced by `Version 1`

Description The `kickUsers()` function allows any caller to remove users who have failed to withdraw their `PUFFER` tokens after the grace period, transferring their tokens and awarding a 1% fee to the `kicker`. However, the function does not prevent users from including their own address in the kick list. Since self-kicking yields the same result as invoking `withdraw()`, this operation is redundant and serves no purpose.

```

    /**
     * @notice Kick multiple users and receive 1% of their PUFFER tokens as a reward
     * @param users Array of user addresses to kick
     */
    function kickUsers(address[] calldata users) external {
        uint256 totalKickerFee;


        for (uint256 i = 0; i < users.length; ++i) {
            address user = users[i];
            LockInfo memory lockInfo = lockInfos[user];


            if (lockInfo.pufferAmount == 0) {
                continue;
```

```
        }


        // The user has a grace period to withdraw their tokens
        require(lockInfo.unlockTime + _GRACE_PERIOD < block.timestamp, TokensMustBeUnlocked());


        uint256 vlPUFFERAmount = balanceOf(user);


        // 1% of the PUFFER tokens are sent to the kicker
        uint256 kickerFee = (lockInfo.pufferAmount * _KICKER_FEE_BPS) / _KICKER_FEE_DENOMINATOR
            ;
        totalKickerFee += kickerFee;


        // The rest of the PUFFER tokens are sent to the user
        uint256 pufferAmount = lockInfo.pufferAmount;


        delete lockInfos[user];


        _burn(user, vlPUFFERAmount);


        // Send the rest of the PUFFER tokens to the user
        PUFFER.safeTransfer(user, pufferAmount - kickerFee);


        emit UserKicked({ kicker: msg.sender, user: user, vlPUFFERAmount: vlPUFFERAmount,
            kickerFee: kickerFee });
    }


    // Send all kicker fees in a single transfer
    if (totalKickerFee > 0) {
        PUFFER.safeTransfer(msg.sender, totalKickerFee);
    }
}
```

Listing 2.4: vlPUFFER.sol

Suggestion Add a check to prevent users from kicking themselves.

Feedback from the project Users can invoke `kickUsers()` from a different address, therefore adding such a check would be meaningless.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS