Security Audit Report for Puffer L2 Staking contracts

Date: July 29, 2024 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer L2 Staking contracts

Version History

Version	Date	Description
1.0	June 27, 2024	First release
2.0	July 29, 2024	Second 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 focus of this audit is on the the Puffer L2 Staking contracts of Puffer Finance [1]. Puffer L2 Staking contracts allows users to deposit ERC20 tokens (e.g., WETH) into the PufToken contract to mint a wrapped ERC20 token, which they can withdraw at any time. Additionally, users can stake their tokens in the PufLocker contract with a lock period and withdraw after the due date. Please note that only the contracts located within the mainnet-contracts/src folder in the repository are included in the scope of this audit. The files covered in this audit include:

```
mainnet-contracts/src/PufLocker.sol
mainnet-contracts/src/PufLockerStorage.sol
mainnet-contracts/src/PufToken.sol
mainnet-contracts/src/PufferL2Depositor.sol
mainnet-contracts/src/interface/IPufLocker.sol
mainnet-contracts/src/interface/IPufStakingPool.sol
mainnet-contracts/src/interface/IPufferL2Depositor.sol
```

Listing 1.1: Audit Scope for Commit Version 1

```
mainnet-contracts/src/PufLocker.sol
mainnet-contracts/src/PufferL2Depositor.sol
```

Listing 1.2: Audit Scope for Commit Version 3

Other files are not within the scope of the 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.

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 Version 1 and Version 3, as well as new code (Version 2 and Version 4) to fix issues in the audit report.

Project	Version	Commit Hash
Puffer L2 Staking contracts	Version 1	a466a7044229676afc3232475b67d7219e8535eb
	Version 2	6ea6fad4b4172fd59a4be3dedd5e1f40b31037dc
	Version 3	ffad259548be6d8aae54136e8f66dc1dc4eb67a0
	Version 4	5cdbb45863e8b845a9592bc3d45a4457b4542f39

[1] https://github.com/PufferFinance/puffer-contracts

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 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.

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.

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

Table 1.1: Vulnerability Severity Classification

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

- **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.

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **one** recommendation and **three** notes.

- High Risk: 1
- Low Risk: 3
- Recommendation: 1
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Potential immediate withdrawal due to uninitialized `lockPeriod` configurations	Software Security	Confirmed
2	Low	Potential precision loss due to normalization and denormalization	Software Security	Fixed
3	Low	Inconsistent check between `migrate` and `migrateWithSignature` functions	DeFi Security	Fixed
4	High	Incorrect `amount` parameter when invoking `migrate()`	DeFi Security	Fixed
5	-	Redundant invocation of function `permit()`	Recommendation	Fixed
6	-	Potential centralized risks	Note	-
7	-	Lack of support for non-standard ERC20 tokens	Note	-
8	-	pufTokens with different underlying tokens have different decimals	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential immediate withdrawal due to uninitialized `lockPeriod` configurations

Severity Low

Status Confirmed

Introduced by Version 1

Description The `minLockPeriod` and `maxLockPeriod` are not initialized in the function `initialize()` of contract `PufLocker`. In this case, users can deposit with `lockPeriod` equals `zero` and withdraw immediately, which may cause unexpected results.

```
function initialize(address accessManager) external initializer {
    require(accessManager != address(0));
    __AccessManaged_init(accessManager);
}
```

Listing 2.1: mainnet-contracts/src/PufLocker.sol

```
function deposit(address token, uint128 lockPeriod, Permit calldata permitData)
    external
    isAllowedToken(token)
    restricted
{
    if (permitData.amount == 0) {
        revert InvalidAmount();
    }
    PufLockerData storage $ = _getPufLockerStorage();

    if (lockPeriod < $.minLockPeriod || lockPeriod > $.maxLockPeriod) {
        revert InvalidLockPeriod();
    }

    // https://docs.openzeppelin.com/contracts/5.x/api/token/erc20#security_considerations
    try ERC20Permit(token).permit({
        owner: msg.sender,
        spender: address(this),
        value: permitData.amount,
        deadline: permitData.deadline,
        v: permitData.v,
        s: permitData.s,
        r: permitData.r
    }) { } catch { }

    IERC20(token).safeTransferFrom(msg.sender, address(this), permitData.amount);

    uint128 releaseTime = uint128(block.timestamp) + lockPeriod;

    $.deposits[msg.sender][token].push(Deposit(uint128(permitData.amount), releaseTime));

    emit Deposited(msg.sender, token, uint128(permitData.amount), releaseTime);
}
```

Listing 2.2: mainnet-contracts/src/PufLocker.sol

Impact Users can deposit and withdraw tokens immediately.

Suggestion Initialize `minLockPeriod` and `maxLockPeriod` in the function `initialize()`.

Feedback from the Project It has no impact from a smart contract perspective. It will be a normal deposit/withdrawal with 0 period. We are going to be calculating the rewards for the locking offchain so this is a case that can be handled and discarded off-chain.

2.1.2 Potential precision loss due to normalization and denormalization

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, the `pufToken` contract serves as a `wrapped ERC20` contract, automatically converting the decimals of `TOKEN` and `STANDARD_DECIMALS` when users deposit or withdraw. However, both the `_normalizeAmount` and `_denormalizeAmount` functions can result in precision loss, leading to some dust tokens being locked in the contract.

```
function _deposit(address depositor, address account, uint256 amount) internal {
    TOKEN.safeTransferFrom(msg.sender, address(this), amount);

    uint256 normalizedAmount = _normalizeAmount(amount);

    if (totalSupply() + normalizedAmount > totalDepositCap) {
        revert TotalDepositCapReached();
    }

    // Mint puffToken to the account
    _mint(account, normalizedAmount);

    // If the user is deposiing using the factory, we emit the ‘depositor‘ from the parameters
    if (msg.sender == address(PUFFER_FACTORY)) {
        emit Deposited(depositor, account, amount);
    } else {
        // If it is a direct deposit not coming from the depositor, we use msg.sender
        emit Deposited(msg.sender, account, amount);
    }
}
```

Listing 2.3: mainnet-contracts/src/PufToken.sol

```
function _normalizeAmount(uint256 amount) internal view returns (uint256 normalizedAmount) {
    if (_TOKEN_DECIMALS > _STANDARD_TOKEN_DECIMALS) {
        return amount / (10 ** (_TOKEN_DECIMALS - _STANDARD_TOKEN_DECIMALS));
    } else if (_TOKEN_DECIMALS < _STANDARD_TOKEN_DECIMALS) {
        return amount * (10 ** (_STANDARD_TOKEN_DECIMALS - _TOKEN_DECIMALS));
    }
    return amount;
}

function _denormalizeAmount(uint256 amount) internal view returns (uint256 denormalizedAmount)
    {
    if (_TOKEN_DECIMALS > _STANDARD_TOKEN_DECIMALS) {
        return amount * (10 ** (_TOKEN_DECIMALS - _STANDARD_TOKEN_DECIMALS));
    } else if (_TOKEN_DECIMALS < _STANDARD_TOKEN_DECIMALS) {
        return amount / (10 ** (_STANDARD_TOKEN_DECIMALS - _TOKEN_DECIMALS));
    }
    return amount;
}
```

Listing 2.4: mainnet-contracts/src/PufToken.sol

Impact The precision loss could lock some dust `pufToken` tokens within the contact.

Suggestion Revise the code logic to mitigate precision loss.

2.2 DeFi Security

2.2.1 Inconsistent check between `migrate` and `migrateWithSignature` functions

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `pufToken` contract, the `migrateWithSignature()` function allows users to migrate their tokens using a migration signature. However, this function lacks the `validateAddressAndAmount` modifier compared to the `migrate` function. This inconsistency can potentially be exploited to bypass the validation, leading to unexpected behaviors.

```
function migrate(uint256 amount, address migratorContract, address destination)
    external
    onlyAllowedMigratorContract(migratorContract)
    validateAddressAndAmount(destination, amount)
    whenNotPaused
{
    _migrate({ depositor: msg.sender, amount: amount, destination: destination, migratorContract: migratorContract });
}

/**
 * @inheritdoc IPufStakingPool
 */
function migrateWithSignature(
    address depositor,
    address migratorContract,
    address destination,
    uint256 amount,
    uint256 signatureExpiry,
    bytes memory stakerSignature
) external onlyAllowedMigratorContract(migratorContract) whenNotPaused {
    if (block.timestamp >= signatureExpiry) {
        revert ExpiredSignature();
    }

    bytes32 structHash = keccak256(
        abi.encode(
            _MIGRATE_TYPEHASH,
            depositor,
            migratorContract,
            destination,
            address(TOKEN),
            amount,
            signatureExpiry,
            _useNonce(depositor)
        )
    );

    if (!SignatureChecker.isValidSignatureNow(depositor, _hashTypedDataV4(structHash), stakerSignature)) {
        revert InvalidSignature();
```

```
    }

    _migrate({ depositor: depositor, amount: amount, destination: destination, migratorContract
        : migratorContract });
}
```

Listing 2.5: mainnet-contracts/src/PufToken.sol

Impact The validation in the `validateAddressAndAmount` modifier can be bypassed with the `migrateWithSignature()` function.

Suggestion Add the `validateAddressAndAmount` modifier for the `migrateWithSignature()` function.

2.2.2 Incorrect `amount` parameter when invoking `migrate()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_migrate()` is utilized to migrate users' underlying tokens through the `migratorContract`. However, it invokes the function `migrate()` by passing the `amount` parameter (i.e., the `pufToken` burned) instead of the `deNormalizedAmount` (i.e., the underlying token to be migrated), leading to an incorrect amount of underlying tokens being transferred.

```
function _migrate(address depositor, uint256 amount, address destination, address
     migratorContract) internal {
    _burn(depositor, amount);

    uint256 deNormalizedAmount = _denormalizeAmount(amount);

    emit Migrated({
        depositor: depositor,
        destination: destination,
        migratorContract: migratorContract,
        amount: amount
    });

    TOKEN.safeIncreaseAllowance(migratorContract, deNormalizedAmount);

    IMigrator(migratorContract).migrate({ depositor: depositor, destination: destination,
        amount: amount });
}
```

Listing 2.6: mainnet-contracts/src/PufToken.sol

Impact Incorrect amount of underlying tokens will be transferred.

Suggestion Use `deNormalizedAmount` as the `amount` parameter when invoking `migrate()`.

2.3 Additional Recommendation

2.3.1 Redundant invocation of function `permit()`

Status Fixed in `Version 4`

Introduced by `Version 3`

Description When function `deposit()` of contract `PufLocker` is called from the contract `PufferL2Depositor`, the invocation of function `permit()` is redundant and will always trigger an internal call revert.

```
function _deposit(
    address token,
    address depositor,
    address account,
    uint256 amount,
    uint256 referralCode,
    uint128 lockPeriod
) internal {
    PufToken pufToken = PufToken(tokens[token]);

    IERC20(token).safeIncreaseAllowance(address(pufToken), amount);

    // If the lockPeriod is greater than 0 we wrap and then deposit the wrapped tokens to the locker contract
    if (lockPeriod > 0) {
        pufToken.deposit(depositor, address(this), amount);
        IERC20(address(pufToken)).safeIncreaseAllowance(address(PUFFER_LOCKER), amount);
        Permit memory permitData;
        permitData.amount = amount;
        // Tokens are being deposited to the locker contract for the account
        PUFFER_LOCKER.deposit(address(pufToken), account, lockPeriod, permitData);
    } else {
        // The account will receive the ERC20 tokens
        pufToken.deposit(depositor, account, amount);
    }

    emit DepositedToken(token, msg.sender, account, amount, referralCode);
}
```

Listing 2.7: mainnet-contracts/src/PufferL2Depositor.sol

```
function deposit(address token, address recipient, uint128 lockPeriod, Permit calldata permitData)
    external
    isAllowedToken(token)
    restricted
{
    if (permitData.amount == 0) {
        revert InvalidAmount();
    }
    PufLockerData storage $ = _getPufLockerStorage();

    if (lockPeriod < $.minLockPeriod || lockPeriod > $.maxLockPeriod) {
        revert InvalidLockPeriod();
```

```
    }

    // https://docs.openzeppelin.com/contracts/5.x/api/token/erc20#security_considerations
    try ERC20Permit(token).permit({
        owner: msg.sender,
        spender: address(this),
        value: permitData.amount,
        deadline: permitData.deadline,
        v: permitData.v,
        s: permitData.s,
        r: permitData.r
    }) { } catch { }

    IERC20(token).safeTransferFrom(msg.sender, address(this), permitData.amount);

    uint128 releaseTime = uint128(block.timestamp) + lockPeriod;

    $.deposits[recipient][token].push(Deposit(uint128(permitData.amount), releaseTime));

    emit Deposited(recipient, token, uint128(permitData.amount), releaseTime);
}
```

Listing 2.8: mainnet-contracts/src/PufLocker.sol

Suggestion Only invoke the function `permit()` if `permitData.deadline` is larger than `block.timestamp`.

2.4 Note

2.4.1 Potential centralized risks

Introduced by `Version 1`

Description In Puffer L2 Staking contracts, there are some privileged functions to update critical configurations, such as creating new staking token contracts. These functions have restricted modifiers and are claimed to be controlled by a multi-signature wallet. If most of the private keys in this wallet are controlled by a single entity, or if the private keys are leaked. The protocol can be potentially incapacitated.

2.4.2 Lack of support for non-standard ERC20 tokens

Introduced by `Version 1`

Description The protocol can only support standard ERC20 tokens. Supporting non-standard ERC20 tokens (e.g., deflationary tokens) can introduce potential security risks. For instance, if a token is a deflationary token, the actual amount received by users may differ from what they expected. Additionally, it is recommended to perform a thorough compatibility check when adding a token into the `isAllowedToken` list.

2.4.3 pufTokens with different underlying tokens have different decimals

Introduced by `Version 2`

Description The contract `pufToken` allows users to deposit their underlying tokens to mint `pufToken` at a 1:1 ratio. It is important to note that the decimals of "wrapped" pufTokens are consistent with its underlying token. Any external contracts that integrate with `pufToken` should take this design into account to avoid unexpected results.

```
function decimals() public view override returns (uint8 _decimals) {
    return _TOKEN_DECIMALS;
}
```

Listing 2.9: mainnet-contracts/src/PufToken.sol

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS