Security Audit Report for Puffer Withdrawal Smart Contracts

Date: September 24, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer Withdrawal Smart Contracts

Version History

Version	Date	Description
1.0	September 24, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-25 08:10:26+0000

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 Puffer Withdrawal Smart Contracts [1] of Puffer Finance. The Puffer Withdrawal Smart Contracts feature enables withdrawals on the `PufferVault` contract and introduces a "2-step withdrawal" mechanism, allowing users to exchange pufETH for WETH without any fees. The contracts covered in this audit include:

```
mainnet-contracts/src/PufferWithdrawalManager.sol
mainnet-contracts/script/DeployPufferWithdrawalManager.s.sol
```

Listing 1.1: Audit Scope for this 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 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 initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Puffer Withdrawal Smart Contracts	Version 1	238822edf145f781827ccb199e26fda83fe18c15
	Version 2	916af117ab84f98889c8a87fcb81de3c00daa047

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-contracts

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.

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:

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

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **two** recommendations and **two** notes.

- Medium Risk: 1
- Low Risk: 2
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Medium	Lack of overflow check on `batchFinalizationExchangeRate`	Software Security	Fixed
2	Low	Lack of sandwich attack defense in function `finalizeWithdrawals()`	DeFi Security	Confirmed
3	Low	Potential leftover assets due to inconsistencies in withdrawal calculations	DeFi Security	Fixed
4	-	Ensure `recipient != 0` in function `_processWithdrawalRequest()`	Recommendation	Fixed
5	-	Ensure `newMaxWithdrawalAmount > MIN_WITHDRAWAL_AMOUNT` in function `changeMaxWithdrawalAmount()`	Recommendation	Fixed
6	-	`DeployPufferWithdrawalManager.s.sol` only sets the `AccessManager` for `holesky` network	Note	-
7	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Lack of overflow check on `batchFinalizationExchangeRate`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `pufETHToETHExchangeRate` field in the `WithdrawalBatch` struct is of type `uint64`. According to the function `finalizeWithdrawals()`, its value is derived from the `pufETH` vault, representing the exchange rate at the finalization time. However, if this exchange rate exceeds `type(uint64).max` (i.e., `pufETH:ETH > 18`), the value will be truncated during the explicit conversion, resulting in an incorrect rate.

```
struct WithdrawalBatch {
    uint64 pufETHToETHExchangeRate; // packed slot 0
    uint96 toBurn; // packed slot 0
    uint96 toTransfer; // packed slot 0
}
```

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

```
for (uint256 i = finalizedWithdrawalBatch + 1; i <= withdrawalBatchIndex; ++i) {
    uint256 batchFinalizationExchangeRate = PUFFER_VAULT.convertToAssets(1 ether);

    WithdrawalBatch storage batch = $.withdrawalBatches[i];

    uint256 expectedETHAmount = batch.toTransfer;
    uint256 pufETHBurnAmount = batch.toBurn;

    uint256 ethAmount = (pufETHBurnAmount * batchFinalizationExchangeRate) / 1 ether;
    uint256 transferAmount = Math.min(expectedETHAmount, ethAmount);

    PUFFER_VAULT.transferETH(address(this), transferAmount);
    PUFFER_VAULT.burn(pufETHBurnAmount);

    batch.pufETHToETHExchangeRate = uint64(batchFinalizationExchangeRate);

    emit BatchFinalized({
        batchIdx: i,
        expectedETHAmount: expectedETHAmount,
        actualEthAmount: transferAmount,
        pufETHBurnAmount: pufETHBurnAmount
    });
}
```

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

Impact An incorrect rate is stored in `batch.pufETHToETHExchangeRate`, potentially disrupting the withdrawal completion process.

Suggestion Use `SafeCast.toUint64()` method to apply overflow checks against the `batchFinalizationExchangeRate` value, or use the `uint256` type instead.

2.2 DeFi Security

2.2.1 Lack of sandwich attack defense in function `finalizeWithdrawals()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `finalizeWithdrawals()` will redeem `pufETH` for `Ether` at the rate of `min(batchExchangeRate, batchFinalizationExchangeRate)`, which will suddenly increase the pufETH price if `batchExchangeRate` is less than the `batchFinalizationExchangeRate`. Malicious users can leverage this to conduct sandwich attacks.

Specifically, the attacker can deposit `Ethers` to contract `PufferVaultV3` before the finalization and withdraw the `Ethers` later to gain the profit.

```
function finalizeWithdrawals(uint256 withdrawalBatchIndex) external restricted {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();

    // Check if all the batches that we want to finalize are full
    require(withdrawalBatchIndex < $.withdrawals.length / BATCH_SIZE, BatchesAreNotFull());

    uint256 finalizedWithdrawalBatch = $.finalizedWithdrawalBatch;

    require(withdrawalBatchIndex > finalizedWithdrawalBatch, BatchAlreadyFinalized(
        withdrawalBatchIndex));

    // Start from the finalized batch + 1 and go up to the given batch index
    for (uint256 i = finalizedWithdrawalBatch + 1; i <= withdrawalBatchIndex; ++i) {
        uint256 batchFinalizationExchangeRate = PUFFER_VAULT.convertToAssets(1 ether);

        WithdrawalBatch storage batch = $.withdrawalBatches[i];

        uint256 expectedETHAmount = batch.toTransfer;
        uint256 pufETHBurnAmount = batch.toBurn;

        uint256 ethAmount = (pufETHBurnAmount * batchFinalizationExchangeRate) / 1 ether;
        uint256 transferAmount = Math.min(expectedETHAmount, ethAmount);

        PUFFER_VAULT.transferETH(address(this), transferAmount);
        PUFFER_VAULT.burn(pufETHBurnAmount);

        batch.pufETHToETHExchangeRate = uint64(batchFinalizationExchangeRate);

        emit BatchFinalized({
            batchIdx: i,
            expectedETHAmount: expectedETHAmount,
            actualEthAmount: transferAmount,
            pufETHBurnAmount: pufETHBurnAmount
        });
    }

    $.finalizedWithdrawalBatch = withdrawalBatchIndex;
}
```

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

Impact The pufETH holders will suffer from a loss of rewards.

Suggestion Revise the withdrawal logic. For example, separate the relative pufETH and Ethers from the price calculation of the vault in function requestWithdrawal().

Feedback from the project Since the attacker would need a huge amount of money to carry out this attack, and the potential profit is insignificant compared to the cost, it's very unlikely they would even attempt it.

2.2.2 Potential leftover assets due to inconsistencies in withdrawal calculations

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In the current implementation, after a batch of withdrawals is finalized, withdrawals can be completed to claim the underlying assets. The function `finalizeWithdrawals()` retrieves `Ether` from the `PufferVaultV3`, calculating the amount based on the formula: $S_1 = \min(\sum(X_i \times R_i), \sum X_i \times R_f)$, where X_i represents the `pufETHAmount`, R_i is the `pufETHtoETHExchangeRate` for each individual withdrawal, and R_f is the `batchFinalizationExchangeRate` used for the entire batch at finalization.

Subsequently, in the function `completeQueuedWithdrawal()`, the `Ethers` each withdrawal can claim is determined by the minimum of the `withdrawal.pufETHtoETHExchangeRate` (i.e., R_i) and the `batchFinalizationExchangeRate` (i.e., R_f). The total claimable `Ethers` for the batch is expressed as $S_2 = \sum(X_i \times \min(R_i, R_f))$.

If R_f is greater than all R_i, S_1 and S_2 will be equal. However, if a black swan event occurs causing R_f to drop below any R_i, S_1 can be less than S_2. In this case, excess `Ethers` may be left in the contract, unable to be claimed.

```
function finalizeWithdrawals(uint256 withdrawalBatchIndex) external restricted {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();

    // Check if all the batches that we want to finalize are full
    require(withdrawalBatchIndex < $.withdrawals.length / BATCH_SIZE, BatchesAreNotFull());

    uint256 finalizedWithdrawalBatch = $.finalizedWithdrawalBatch;

    require(withdrawalBatchIndex > finalizedWithdrawalBatch, BatchAlreadyFinalized(
        withdrawalBatchIndex));

    // Start from the finalized batch + 1 and go up to the given batch index
    for (uint256 i = finalizedWithdrawalBatch + 1; i <= withdrawalBatchIndex; ++i) {
        uint256 batchFinalizationExchangeRate = PUFFER_VAULT.convertToAssets(1 ether);

        WithdrawalBatch storage batch = $.withdrawalBatches[i];

        uint256 expectedETHAmount = batch.toTransfer;
        uint256 pufETHBurnAmount = batch.toBurn;

        uint256 ethAmount = (pufETHBurnAmount * batchFinalizationExchangeRate) / 1 ether;
        uint256 transferAmount = Math.min(expectedETHAmount, ethAmount);

        PUFFER_VAULT.transferETH(address(this), transferAmount);
        PUFFER_VAULT.burn(pufETHBurnAmount);

        batch.pufETHToETHExchangeRate = uint64(batchFinalizationExchangeRate);

        emit BatchFinalized({
            batchIdx: i,
            expectedETHAmount: expectedETHAmount,
            actualEthAmount: transferAmount,
            pufETHBurnAmount: pufETHBurnAmount
        });
```

```
    }

    $.finalizedWithdrawalBatch = withdrawalBatchIndex;
}
```

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

```
function completeQueuedWithdrawal(uint256 withdrawalIdx) external restricted {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();

    uint256 batchIndex = withdrawalIdx / BATCH_SIZE;
    require(batchIndex <= $.finalizedWithdrawalBatch, NotFinalized());

    Withdrawal storage withdrawal = $.withdrawals[withdrawalIdx];

    // Check if the withdrawal has already been completed
    require(withdrawal.recipient != address(0), WithdrawalAlreadyCompleted());

    uint256 batchSettlementExchangeRate = $.withdrawalBatches[batchIndex].
        pufETHToETHExchangeRate;

    uint256 payoutExchangeRate = Math.min(withdrawal.pufETHToETHExchangeRate,
        batchSettlementExchangeRate);
    uint256 payoutAmount = (uint256(withdrawal.pufETHAmount) * payoutExchangeRate) / 1 ether;

    address recipient = withdrawal.recipient;

    // remove data for some gas savings
    delete $.withdrawals[withdrawalIdx];

    // Wrap ETH to WETH
    WETH.deposit{ value: payoutAmount }();

    WETH.transfer(recipient, payoutAmount);

    emit WithdrawalCompleted({
        withdrawalIdx: withdrawalIdx,
        ethPayoutAmount: payoutAmount,
        payoutExchangeRate: payoutExchangeRate,
        recipient: recipient
    });
}
```

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

Impact The inconsistent calculations could lock some `Ethers` within the contract.

Suggestion Revise the code logic to mitigate inconsistent calculations or implement methods to handle unexpected leftover `Ethers`.

2.3 Additional Recommendation

2.3.1 Ensure `recipient != 0` in function `_processWithdrawalRequest()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, there is no check on the recipient in function `_processWithdrawalRequest()`, which may lead to DoS because `completeQueuedWithdrawal()` will revert if the recipient is zero.

```
function _processWithdrawalRequest(uint128 pufETHAmount, address recipient) internal
     oneWithdrawalRequestAllowed {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();

    require(pufETHAmount >= MIN_WITHDRAWAL_AMOUNT, WithdrawalAmountTooLow());
    require(pufETHAmount <= $.maxWithdrawalAmount, WithdrawalAmountTooHigh());

    // Always transfer from the msg.sender
    PUFFER_VAULT.transferFrom(msg.sender, address(this), pufETHAmount);

    uint256 withdrawalIndex = $.withdrawals.length;

    uint256 batchIndex = withdrawalIndex / BATCH_SIZE;

    if (batchIndex == $.withdrawalBatches.length) {
        // Push empty batch when the previous batch is full
        $.withdrawalBatches.push(WithdrawalBatch({ toBurn: 0, toTransfer: 0,
            pufETHToETHExchangeRate: 0 }));
    }

    uint256 pufETHToETHExchangeRate = PUFFER_VAULT.convertToAssets(1 ether);
    uint256 expectedETHAmount = pufETHAmount * pufETHToETHExchangeRate / 1 ether;

    WithdrawalBatch storage batch = $.withdrawalBatches[batchIndex];
    batch.toBurn += uint96(pufETHAmount);
    batch.toTransfer += uint96(expectedETHAmount);

    $.withdrawals.push(
        Withdrawal({
            pufETHAmount: pufETHAmount,
            recipient: recipient,
            pufETHToETHExchangeRate: pufETHToETHExchangeRate.toUint128()
        })
    );

    emit WithdrawalRequested({
        withdrawalIdx: withdrawalIndex,
        batchIdx: batchIndex,
        pufETHAmount: pufETHAmount,
        recipient: recipient
    });
}
```

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

```
function completeQueuedWithdrawal(uint256 withdrawalIdx) external restricted {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();

    uint256 batchIndex = withdrawalIdx / BATCH_SIZE;
    require(batchIndex <= $.finalizedWithdrawalBatch, NotFinalized());

    Withdrawal storage withdrawal = $.withdrawals[withdrawalIdx];

    // Check if the withdrawal has already been completed
    require(withdrawal.recipient != address(0), WithdrawalAlreadyCompleted());

    uint256 batchSettlementExchangeRate = $.withdrawalBatches[batchIndex].
        pufETHToETHExchangeRate;

    uint256 payoutExchangeRate = Math.min(withdrawal.pufETHToETHExchangeRate,
        batchSettlementExchangeRate);
    uint256 payoutAmount = (uint256(withdrawal.pufETHAmount) * payoutExchangeRate) / 1 ether;

    address recipient = withdrawal.recipient;

    // remove data for some gas savings
    delete $.withdrawals[withdrawalIdx];

    // Wrap ETH to WETH
    WETH.deposit{ value: payoutAmount }();

    WETH.transfer(recipient, payoutAmount);

    emit WithdrawalCompleted({
        withdrawalIdx: withdrawalIdx,
        ethPayoutAmount: payoutAmount,
        payoutExchangeRate: payoutExchangeRate,
        recipient: recipient
    });
}
```

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

Impact Invalid requests cannot be completed.

Suggestion Add a check to ensure the `recipient` is not zero in function `_processWithdrawalRequest()`.

2.3.2 Ensure `newMaxWithdrawalAmount > MIN_WITHDRAWAL_AMOUNT` in function `changeMaxWithdrawalAmount()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `changeMaxWithdrawalAmount()` lacks a check on the `newMaxWithdrawalAmount` parameter to ensure it is larger than `MIN_WITHDRAWAL_AMOUNT`. This oversight allows a smaller `maxWithdrawalAmount` configured and could lead to DoS on the withdrawal request.

```
function changeMaxWithdrawalAmount(uint256 newMaxWithdrawalAmount) external restricted {
    WithdrawalManagerStorage storage $ = _getWithdrawalManagerStorage();
    emit MaxWithdrawalAmountChanged($.maxWithdrawalAmount, newMaxWithdrawalAmount);
    $.maxWithdrawalAmount = newMaxWithdrawalAmount;
}
```

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

Impact N/A

Suggestion Apply checks on the `newMaxWithdrawalAmount` parameter.

2.4 Note

2.4.1 `DeployPufferWithdrawalManager.s.sol` only sets the `AccessManager` for `holesky` network

Introduced by Version 1

Description The deployment script `PufferWithdrawalManager.s.sol` lacks the logic to broadcast the transactions to networks other than the `holesky` testnet.

```
function run() public {
    Generate2StepWithdrawalsCalldata calldataGenerator = new Generate2StepWithdrawalsCalldata()
        ;

    vm.startBroadcast();

    PufferWithdrawalManager withdrawalManagerImpl =
        ((new PufferWithdrawalManager(BATCH_SIZE, PufferVaultV3(payable(_getPufferVault())),
            IWETH(_getWETH()))));

    withdrawalManager = PufferWithdrawalManager(
        (
            payable(
                new ERC1967Proxy{ salt: bytes32("PufferWithdrawalManager") }(
                    address(withdrawalManagerImpl),
                    abi.encodeCall(PufferWithdrawalManager.initialize, address(_getAccessManager
                        ()))
                )
            )
        )
    );

    vm.label(address(withdrawalManager), "PufferWithdrawalManagerProxy");
    vm.label(address(withdrawalManagerImpl), "PufferWithdrawalManagerImplementation");

    encodedCalldata = calldataGenerator.run({
        pufferVaultProxy: _getPufferVault(),
        withdrawalManagerProxy: address(withdrawalManager),
        paymaster: _getPaymaster(),
        withdrawalFinalizer: _getOPSMultisig(),
```

```
        pufferProtocolProxy: _getPufferProtocol()
    });

    console.log("Queue from Timelock -> AccessManager", _getAccessManager());
    console.logBytes(encodedCalldata);

    if (block.chainid == holesky) {
        (bool success,) = address(_getAccessManager()).call(encodedCalldata);
        require(success, "AccessManager.call failed");
    }

    vm.stopBroadcast();
}
```

Listing 2.9: mainnet-contracts/script/DeployPufferWithdrawalManager.s.sol

Feedback from the Project On Holesky, the deployer can execute it right away. On Mainnet, we need to do a multisig tx.

2.4.2 Potential centralization risks

Introduced by `Version 1`

Description The protocol includes several privileged functions to update critical configurations, such as the withdrawal configurations. 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.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS