BLOCKSEC

Security Audit
Report for Puffer Contracts

Date: July 25, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer Contracts

Version History

Version	Date	Description
1.0	July 25, 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 Contracts of Puffer Finance.

The Puffer Contracts enables instant withdrawals on the `PufferVaultV5` contract via a 1-step withdrawal mechanism with integrated fees. This upgrade introduces a treasury exit fee in addition to the existing exit fee, requiring users to pay both fees when withdrawing.

This audit focuses exclusively on changes made after commit a1ef1b5 [2] in the following contracts:

- mainnet-contracts/src/PufferVaultV5.sol
- mainnet-contracts/src/PufferVaultStorage.sol

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 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 Contracts	`Version 1`	`9bbccba476264d679cdcfa28605b3744ce5d724c`
	`Version 2`	`628898112b0b0f5bcd9a129d2723198823bb6357`

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

[2] https://github.com/PufferFinance/puffer-contracts/tree/a1ef1b5caefbdd4dd4b4c15162919055e95132d4

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 the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 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 [3] and Common Weakness Enumeration [4]. 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:

- **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.

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

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

Chapter 2 Findings

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

- Low Risk: 3
- Recommendation: 2

ID	Severity	Description	Category	Status
1	Low	Inconsistent fee calculation between the functions `withdraw()` and `redeem()`	Security Issue	Confirmed
2	Low	Potential DoS due to incorrect calculation	Security Issue	Fixed
3	Low	Incorrect calculation logic in the function `previewRedeem()`	Security Issue	Confirmed
4	-	Add non-zero checks for deposit and withdrawal operations	Recommendation	Confirmed
5	-	Implement a distinct error for the function `setTreasuryExitFeeBasisPoints()`	Recommendation	Confirmed

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Inconsistent fee calculation between the functions `withdraw()` and `redeem()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The functions `withdraw()` and `redeem()` allow users to retrieve underlying assets by burning shares and paying fees (i.e., `treasuryFee` and `exitFee`). The function `withdraw()` enables users to specify the asset amount, while `redeem()` enables users to specify the burned shares. However, their fee calculations are inconsistent.

Both functions compute `assetsWithFeeIncluded` to represent the total required assets, encompassing withdrawn assets plus fees, but then calculate fees differently. The function `withdraw()` calculates `treasuryFee` using the same method as `redeem()`, while `exitFee` is implicitly derived as `assetsWithFeeIncluded` minus `assets` and `treasuryFee`. This may lead to a lower `exitFee` than the configured rate. In contrast, the function `redeem()` calculates `treasuryFee` and `exitFee` explicitly using round-up approaches, and then calculates the `assets` amount by subtracting total fees from `assetsWithFeeIncluded`. This differing calculation logic leads to discrepancies in the `assets` and `exitFee` amounts when burning the same number of shares, depending on which function is invoked.

```
281   function withdraw(uint256 assets, address receiver, address owner)
282       public
283       virtual
284       override
285       revertIfDeposited
286       restricted
```

```
287         returns (uint256)
288     {
289         uint256 maxAssets = maxWithdraw(owner);
290         if (assets > maxAssets) {
291             revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
292         }
293         VaultStorage storage $ = _getPufferVaultStorage();
294
295         uint256 treasuryExitFeeBasisPoints = $.treasuryExitFeeBasisPoints;
296
297         uint256 assetsWithFeeIncluded = _assetsWithFee(assets, $.exitFeeBasisPoints +
                treasuryExitFeeBasisPoints);
298
299         uint256 treasuryFee = _feeOnRaw(assetsWithFeeIncluded, treasuryExitFeeBasisPoints);
300
301         uint256 shares = super.previewWithdraw(assetsWithFeeIncluded);
302
303         _wrapETH(assets + treasuryFee);
304
305         _withdraw({ caller: _msgSender(), receiver: receiver, owner: owner, assets: assets, shares:
                shares });
306
307         // Transfer fee to treasury if needed
308         if (treasuryFee > 0) {
309             SafeERC20.safeTransfer(_WETH, $.treasury, treasuryFee);
310         }
311
312         return shares;
```

Listing 2.1: mainnet-contracts/src/PufferVaultV5.sol

```
326     function redeem(uint256 shares, address receiver, address owner)
327         public
328         virtual
329         override
330         revertIfDeposited
331         restricted
332         returns (uint256)
333     {
334         uint256 maxShares = maxRedeem(owner);
335         if (shares > maxShares) {
336             revert ERC4626ExceededMaxRedeem(owner, shares, maxShares);
337         }
338
339         VaultStorage storage $ = _getPufferVaultStorage();
340
341         uint256 assetsWithFeeIncluded = super.previewRedeem(shares);
342
343         uint256 exitFee = _feeOnRaw(assetsWithFeeIncluded, $.exitFeeBasisPoints);
344         uint256 treasuryFee = _feeOnRaw(assetsWithFeeIncluded, $.treasuryExitFeeBasisPoints);
345
346         // nosemgrep basic-arithmetic-underflow
347         uint256 assets = assetsWithFeeIncluded - exitFee - treasuryFee;
```

```
348
349         _wrapETH(assets + treasuryFee);
350
351         _withdraw({ caller: _msgSender(), receiver: receiver, owner: owner, assets: assets, shares:
                shares });
352
353         // Transfer fee to treasury if needed
354         if (treasuryFee > 0) {
355             SafeERC20.safeTransfer(_WETH, $.treasury, treasuryFee);
356         }
357
358         return assets;
```

Listing 2.2: mainnet-contracts/src/PufferVaultV5.sol

Impact This could incentivize users to consistently select the withdrawal function with lower fees, resulting in a potential loss in protocol revenue.

Suggestion Revise the code logic accordingly.

Feedback from the project Such tiny amounts are never redeemed in practice. When splitting values between multiple parties, rounding effects are mathematically inevitable, meaning someone will always be off by a single wei in any non-even distribution. This is negligible and has no practical impact.

2.1.2 Potential DoS due to incorrect calculation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `maxRedeem()` determines the maximum redeemable share of a user by taking the minimum between the user's owned shares and the vault's liquidity corresponding shares. This calculation serves the function `redeem()` to verify the input `shares` parameter. However, the function incorrectly calculates the liquidity-based shares. Specifically, the function incorrectly passes `availableLiquidity` (i.e., the `Ether` and `Wrapped Ether` balance in the vault) to the overridden function `previewWithdraw()`. The function `previewWithdraw()` then adds additional fees to `availableLiquidity` by invoking the function `_assetsWithFee()`, and converts this inflated value to final shares. Consequently, the calculated share amount corresponds to an asset amount greater than the vault's actual available liquidity.

In extreme cases, this inconsistency can lead to a DoS (denial-of-service) vulnerability within the function `redeem()`. When the vault has lower available liquidity than user shares, the function `maxRedeem()` returns an inflated `maxSharesFromLiquidity`, allowing the user to unexpectedly pass the availability check but fail due to insufficient vault funds.

```
326   function redeem(uint256 shares, address receiver, address owner)
327         public
328         virtual
329         override
330         revertIfDeposited
```

```
331         restricted
332         returns (uint256)
333     {
334         uint256 maxShares = maxRedeem(owner);
```

Listing 2.3: mainnet-contracts/src/PufferVaultV5.sol

```
522     function maxRedeem(address owner) public view virtual override returns (uint256 maxShares) {
523         uint256 shares = balanceOf(owner);
524         // Calculate max shares based on available liquidity (WETH + ETH balance)
525         uint256 availableLiquidity = _WETH.balanceOf(address(this)) + (address(this).balance);
526         // Calculate how many shares can be redeemed from the available liquidity after fees
527         uint256 maxSharesFromLiquidity = previewWithdraw(availableLiquidity);
528         // Return the minimum of user's shares and shares from available liquidity
529         return Math.min(shares, maxSharesFromLiquidity);
```

Listing 2.4: mainnet-contracts/src/PufferVaultV5.sol

```
535     function previewWithdraw(uint256 assets) public view virtual override returns (uint256) {
536         return super.previewWithdraw(_assetsWithFee(assets, getTotalExitFeeBasisPoints()));
537     }
```

Listing 2.5: mainnet-contracts/src/PufferVaultV5.sol

```
641     function _assetsWithFee(uint256 assets, uint256 feeBasisPoints) internal pure virtual returns
            (uint256) {
642         return assets.mulDiv(_BASIS_POINT_SCALE, (_BASIS_POINT_SCALE - feeBasisPoints));
643     }
```

Listing 2.6: mainnet-contracts/src/PufferVaultV5.sol

Impact The vulnerability can lead to a DoS (denial-of-service) scenario where users pass the availability validation but fail to redeem.

Suggestion Revise the code logic accordingly.

2.1.3 Incorrect calculation logic in the function `previewRedeem()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `previewRedeem()` enables users to estimate the underlying assets they can receive after fees when redeeming shares. However, the fee calculation is inconsistent with the function `redeem()` that executes the actual redemption.

Specifically, the function `previewRedeem()` calculates the fees using the formula: $\lceil \text{assets} \times (\text{exitFeeBasisPoints} + \text{treasuryExitFeeBasisPoints}) \rceil$, which sums the two fee rates first before applying the ceiling function. In contrast, the function `redeem()` calculates the fee separately: $\lceil \text{asset} \times \text{exitFeeBasisPoints} \rceil + \lceil \text{asset} \times \text{treasuryExitFeeBasisPoints} \rceil$, applying the ceiling function to each type of fee. Consequently, this inconsistent calculation may cause the function `previewRedeem()` to return a value greater than what the user receives during actual redemption.

```
542   function previewRedeem(uint256 shares) public view virtual override returns (uint256) {
543       uint256 assets = super.previewRedeem(shares);
544       // nosemgrep basic-arithmetic-underflow
545       return assets - _feeOnRaw(assets, getTotalExitFeeBasisPoints());
546   }
```

Listing 2.7: mainnet-contracts/src/PufferVaultV5.sol

```
567   function getTotalExitFeeBasisPoints() public view virtual returns (uint256) {
568       VaultStorage storage $ = _getPufferVaultStorage();
569       return $.exitFeeBasisPoints + $.treasuryExitFeeBasisPoints;
570   }
```

Listing 2.8: mainnet-contracts/src/PufferVaultV5.sol

Impact The inconsistency can lead to unexpected behavior and user confusion when the actual redemption amount differs from the previewed amount.

Suggestion Align the fee calculation logic in the function `previewRedeem()` with that of the function `redeem()` to ensure accurate estimations for users.

Feedback from the project As with the feedback of Issue 2.1.1, this is negligible and has no practical impact.

2.2 Recommendation

2.2.1 Add non-zero checks for deposit and withdrawal operations

Status Confirmed

Introduced by `Version 1`

Description In the contract `PufferVaultV5`, the deposit and withdrawal operations lack non-zero value checks on the minted shares or withdrawn assets. For instance, in the function `depositETH()`, if a deposit amount results in zero shares, the depositor loses their assets without receiving any shares in return. The functions `depositStETH()` and `deposit()` exhibit the same issue.

Similarly, the functions `redeem()` and `withdraw()` lack a zero check on the withdrawn assets. Users may burn shares without receiving assets.

```
141   function depositETH(address receiver) public payable virtual markDeposit restricted returns (
          uint256) {
142       uint256 maxAssets = maxDeposit(receiver);
143       if (msg.value > maxAssets) {
144           revert ERC4626ExceededMaxDeposit(receiver, msg.value, maxAssets);
145       }
146
147       uint256 shares = previewDeposit(msg.value);
148       _mint(receiver, shares);
```

Listing 2.9: mainnet-contracts/src/PufferVaultV5.sol

```
281   function withdraw(uint256 assets, address receiver, address owner)
282       public
283       virtual
284       override
285       revertIfDeposited
286       restricted
287       returns (uint256)
288   {
289       uint256 maxAssets = maxWithdraw(owner);
290       if (assets > maxAssets) {
291           revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
292       }
293       VaultStorage storage $ = _getPufferVaultStorage();
294
295       uint256 treasuryExitFeeBasisPoints = $.treasuryExitFeeBasisPoints;
296
297       uint256 assetsWithFeeIncluded = _assetsWithFee(assets, $.exitFeeBasisPoints +
              treasuryExitFeeBasisPoints);
298
299       uint256 treasuryFee = _feeOnRaw(assetsWithFeeIncluded, treasuryExitFeeBasisPoints);
300
301       uint256 shares = super.previewWithdraw(assetsWithFeeIncluded);
```

Listing 2.10: mainnet-contracts/src/PufferVaultV5.sol

Suggestion Add non-zero checks for deposit and withdrawal operations to prevent unexpected operations.

Feedback from the project Users would only acquire zero assets or shares in cases involving tiny quantities like 2 wei. This is not realistic, and it isn't worth adding logic that would increase gas costs for a situation that will never happen.

2.2.2 Implement a distinct error for the function `setTreasuryExitFeeBasisPoints()`

Status Confirmed

Introduced by `Version 1`

Description Both the functions `setTreasuryExitFeeBasisPoints()` and `_setExitFeeBasisPoints()` use the same error `InvalidExitFeeBasisPoints` despite serving different purposes. It is advised to use a distinct error for the function `setTreasuryExitFeeBasisPoints()`.

```
482   function setTreasuryExitFeeBasisPoints(uint96 newTreasuryExitFeeBasisPoints, address
          newTreasury)
483       external
484       restricted
485   {
486       // 2.5% is the maximum exit fee
487       if (newTreasuryExitFeeBasisPoints > _MAX_EXIT_FEE_BASIS_POINTS) {
488           revert InvalidExitFeeBasisPoints();
489       }
```

Listing 2.11: mainnet-contracts/src/PufferVaultV5.sol

```
649   function _setExitFeeBasisPoints(uint256 newExitFeeBasisPoints) internal virtual {
650       VaultStorage storage $ = _getPufferVaultStorage();
651       // 2.5% is the maximum exit fee
652       if (newExitFeeBasisPoints > _MAX_EXIT_FEE_BASIS_POINTS) {
653           revert InvalidExitFeeBasisPoints();
654       }
```

Listing 2.12: mainnet-contracts/src/PufferVaultV5.sol

Suggestion Implement a distinct error message to improve code clarity and protocol usability.

Feedback from the project These functions will be called very rarely, only by us, and with much care. So if the error were to happen, it would be easy for us to locate it.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS