BLOCKSEC

Security Audit Report for Vault Contracts

Date: April 22, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Chapter 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

Chapter 2 Findings **5**

2.1 DeFi Security 5

2.1.1 Potential front-run attacks to seize the idle yield 5

2.2 Additional Recommendation 7

2.2.1 Optimize gas usage 7

2.2.2 Revise the sanity check 8

2.3 Note 8

2.3.1 The protocol supports wrapped tokens of `WETH` and `USDB` on Blast 8

2.3.2 No locked yield accrues in the initial window period 9

2.3.3 Potential centralization risks 9

Report Manifest

Item	Description
Client	Particle Platform
Target	Vault Contracts

Version History

Version	Date	Description
1.0	March 27, 2024	First Release
1.1	April 22, 2024	Update the commit hash

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-04-22 07:34:02+0000

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 the Vault Contracts of Particle Platform [1]. The Particle Platform is a yield vault protocol that runs specifically on the Blast network. The protocol offers users two distinct yield options for depositing and earning yield:

- Fixed rate yield: Users can earn yield at a fixed rate until the lock maturity date. They also pay position fees to support the variable rate positions.
- Variable rate yield: Users can earn idle yield based on their proportional contribution to the total principal over time.

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
Vault Contracts	`Version 1`	`e0398b59a6990c2d49506e1e490e53188140373f`
	`Version 2`	`9214216ae8d378d18448de6427d423943e3bcca8`

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.

[1] https://github.com/Particle-Platforms/vault-contract/tree/main/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.

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

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

- **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 find **one** potential security issue. Besides, we also have **two** recommendations and **three** notes.

- Medium Risk: 1
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Potential front-run attacks to seize the idle yield	DeFi Security	Fixed
2	-	Optimize gas usage	Recommendation	Fixed
3	-	Revise the sanity check	Recommendation	Fixed
4	-	The protocol supports wrapped tokens of WETH and USDB on Blast	Note	-
5	-	No locked yield accrues in the initial window period	Note	-
6	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential front-run attacks to seize the idle yield

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The locked yield is calculated upon opening a fixed-rate position, and the equivalent number of TOKEN will be minted to the FixedRate contract. This will cause the idle yield, represented by the difference between the TOKEN's underlying token balance and totalSupply, to decrease. The idle yield can be reduced to zero, potentially preventing the closing of variable rate positions from collecting yield.

```
function getIdleYield() public view returns (uint256 idleYield) {
    idleYield =
        IERC20(IYieldStrippedToken(TOKEN).token()).balanceOf(TOKEN) -
        IYieldStrippedToken(TOKEN).totalSupply();
}
```

Listing 2.1: contracts/Vault.sol

Therefore, an attacker holding variable rate positions can initiate a front-run attack to seize others' share of the idle yield. When a user sends a transaction to close variable rate positions, the attacker could front-run it by opening a fixed rate position that depletes the idle yield. This prevents the user from receiving due yield. Subsequently, the attacker can close the fixed-rate position, burning the unmatured locked yield. This restores the idle yield, allowing the attacker's variable rate position to become eligible for more yield.

```
function mintFixedRate(
    uint256 amount,
    uint256 minLockedYield
) external override nonReentrant returns (address owner, uint256 lockedYield, uint256
     maturityTimestamp) {
    require(amount > 0, "Vault: invalid value");

    // update the yield estimate if needed
    _estimateYield();

    // create fixed rate contract
    FixedRate fixedRate = new FixedRate(TOKEN, address(this), msg.sender, GUARDIAN);
    owner = address(fixedRate);

    // take the fee
    if (positionFee > 0) {
        uint256 fee = (amount * positionFee) / BASIS;
        balance.positionFee += fee;
        amount -= fee;
        // drip the fee to this countract
        IERC20(TOKEN).safeTransferFrom(msg.sender, address(this), fee);
    }

    // get current amount to lock
    lockedYield = getYieldToLock(amount);

    // check slippage
    require(lockedYield >= minLockedYield, "Vault: slippage");

    // check enough yield to lock
    require(lockedYield <= getIdleYield(), "Vault: overspend");

    // update the balance storage
    balance.fixedRatePrincipal += amount;
    balance.lockedYield += lockedYield;

    // mint
    uint256 id = _nextId++;
    fixedRateTerms[id] = FixedRateTerm({
        owner: owner,
        principal: amount,
        yield: lockedYield,
        maturityTimestamp: maturityTimestamp = block.timestamp + MATURITY
    });

    // mint the yield to the fixed rate contract
    _mintYield(owner, lockedYield);

    // route the fund to the fixed rate contract
    IERC20(TOKEN).safeTransferFrom(msg.sender, owner, amount);
```

```
        fixedRate.mint(id, amount, lockedYield, maturityTimestamp); // record keeping for the
            contract

        emit MintFixedRate(owner, id, amount, lockedYield);
    }
```

Listing 2.2: contracts/Vault.sol

Impact In the most extreme case, the attacker could potentially drain all idle yield intended for all variable rate positions.

Suggestion Revise the code logic accordingly.

2.2 Additional Recommendation

2.2.1 Optimize gas usage

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are a few areas with potential for gas usage optimization:

1. The `Vault` contract creates a new `FixedRate` or `VariableRate` contract each time a position is opened. Adopting proxy patterns for contract deployment could significantly enhance gas efficiency, thus reducing the gas costs users incur for transactions.

```
    function mintFixedRate(
        uint256 amount,
        uint256 minLockedYield
    ) external override nonReentrant returns (address owner, uint256 lockedYield, uint256
         maturityTimestamp) {
        require(amount > 0, "Vault: invalid value");

        // update the yield estimate if needed
        _estimateYield();

        // create fixed rate contract
        FixedRate fixedRate = new FixedRate(TOKEN, address(this), msg.sender, GUARDIAN);

        …
    }
```

Listing 2.3: contracts/Vault.sol

```
    function mintVariableRate(uint256 amount) external override nonReentrant returns (
         address owner) {
        require(amount > 0, "Vault: invalid value");

        // update the yield estimate if needed
        _estimateYield();

        // create variable rate contract
        VariableRate variableRate = new VariableRate(TOKEN, address(this), msg.sender,
            GUARDIAN);

        …
    }
```

Listing 2.4: contracts/Vault.sol

2. In the `mintFixedRate` function, the check to ensure that `lockedYield` is not larger than idle yield is already performed within the `_mintYield` function. Therefore, this redundant check can be safely omitted.

```
    function _mintYield(address recipient, uint256 yield) internal {
        require(yield <= getIdleYield(), "Vault: overmint yield");
        IYieldStrippedToken(TOKEN).mint(recipient, yield);
    }
```

Listing 2.5: contracts/YieldManager.sol

Impact N/A

Suggestion Revise the code logic accordingly.

2.2.2 Revise the sanity check

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `positionFee` specifies the ratio of the fee deducted from the principal when users lock tokens for a fixed rate. The sanity check in the function should strictly ensure that `positionFee` is less than `BASIS`, according to the design.

```
function updatePositionFee(uint256 fee) external override onlyOwner nonReentrant {
    require(fee <= BASIS, "Vault: invalid position fee");
    positionFee = fee;
    emit UpdatePositionFee(fee);
}
```

Listing 2.6: Vault.sol

Impact The principal can be fully used to pay position fees if `positionFee` is erroneously set to `BASIS`.

Suggestion Revise the sanity check on the `fee` parameter.

2.3 Note

2.3.1 The protocol supports wrapped tokens of `WETH` and `USDB` on Blast

Description The protocol accumulates yield from auto-rebasing balances of `TOKEN`'s underlying tokens. `TOKEN` is the wrapped token developed by the protocol for `WETH` or `USDB`, specifically for the Blast network. Users should acquire `TOKEN` to deposit into the `Vault` and earn yield.

2.3.2 No locked yield accrues in the initial window period

Description Since `yieldEstimate.currentRate` is uninitialized and defaults to 0 during the `Vault` contract's initial estimate window, users who lock in at a fixed rate will not accrue any yield. Users should be aware of such edge cases, which could potentially lead to losses. Additionally, it is important to note that the protocol permits users to specify a minimum locked yield to mint when opening a position.

2.3.3 Potential centralization risks

Description The owner of the protocol contracts has the privilege to modify critical configurations such as the yield curve parameters and the position fee ratio. This introduces potential centralization risks. Should an attacker compromise the owner's account, they could potentially incapacitate the protocol.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS