BLOCKSEC

Security Audit
Report for Yield Contracts

Date: December 27, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	KikiFinance
Target	Yield Contracts

Version History

Version	Date	Description
1.0	December 27, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-12-30 02:32:53+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 KikiFinance's Yield Contracts [1]. These contracts enable users to deposit `iBTC`, `WBTC`, or `BTC` into the `YieldPool` contract after exchanging them for `WBTC` at a 1:1 ratio.

Please note that the audit scope is limited to the contracts located in the **contracts** folder; other files are outside 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, therefore, are not included in the audit.

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
Yield Contracts	`Version 1`	`bbcf2ead9220ee8680f3410457ce5b0c22549b69`
	`Version 2`	`f62ea8c2ad28aab943709634564df83cadd8446d`
	`Version 3`	`489359e7b19e8ad3b7ce157db5af1f8925e87ba3`

The contracts are deployed on the exSat network using the same `Version 3` code. The deployed contract addresses are listed in the following table.

Contract		Address
YieldPool	Proxy	0x1FDF1BeD9CBe3e8e800b939F72Ab8D7840736e06
	Impl	0xb7ceb18E9B70515A1f7F006F655a2B2Af87A6404
WIBTC	-	0xC237fd3a90400082aD729fCc1cD88D179cb86e2b

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.

[1] https://github.com/KikiFinance/yield_contract/

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.

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**,

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

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

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.
- **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 **two** potential security issues. Besides, we have **one** note.

- High Risk: 1
- Medium Risk: 1
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Improper validation on redemption requests	DeFi Security	Fixed
2	High	Incorrect state update	DeFi Security	Fixed
3	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Improper validation on redemption requests

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `requestRedemption` function in the `YieldPool` contract enables users to request the redemption of their staked tokens (including `iBTC`, `WBTC` and `BTC`; `iBTC` will be used in the following examples for further illustration). While the function checks that the `amount` parameter does not exceed `userStakes[poolId][msg.sender]`, it does not account for portions of the stake that have already been requested for redemption. This oversight may lead to two potential issues:

1. **Bypassing the Time Lock Limitation:** Suppose Alice deposits 100 `iBTC` and invokes the `requestRedemption` function ten times with an amount of 100 each time. This could allow her to redeem up to 1000 `iBTC` at the end date. Furthermore, Alice could deposit an additional 900 `iBTC` later and instantly redeem this amount, bypassing the intended time lock.
2. **Potential DoS Issue:** If Alice unintentionally calls the `requestRedemption` function twice with an amount of 100 after depositing 100 `iBTC`, the `redeem` function, which withdraws the total amount from the `request` array, may prevent her from redeeming her funds. This is because the combined redemption requests exceed her stake, requiring her to deposit additional tokens to resolve the discrepancy.

```
129   function requestRedemption(uint256 poolId, uint256 amount) external nonReentrant {
130       require(poolInfos[poolId].redeemEnable, "Pool redeem disabled");
131       require(amount > 0, "Redemption amount must be greater than 0");
132       require(userStakes[poolId][msg.sender] >= amount, "Insufficient staked amount");
```

Listing 2.1: YiledPool.sol

```
167   function redeem(uint256 poolId) external nonReentrant {
168       require(poolInfos[poolId].redeemEnable, "Pool redeem disabled");
169       uint256 totalAmount = 0; // Explicitly initialize totalAmount
170       RedemptionRequest[] storage requests = userRedemptionRequests[poolId][msg.sender];
171       uint256 length = requests.length;
172       for (uint256 i = 0; i < length; ) {
173           if (requests[i].unlockTimestamp <= block.timestamp) {
174               totalAmount += requests[i].amount;
175               requests[i] = requests[length - 1];
176               requests.pop();
177               length--;
178           } else {
179               i++;
180           }
181       }
182       require(totalAmount > 0, "No unlocked withdrawal requests available");
183       require(wiBTCErc20.balanceOf(address(this)) >= totalAmount, "Insufficient balance");
184       wiBTC.withdraw(totalAmount);
185       iBTCErc20.safeTransfer(msg.sender, totalAmount);
186       userStakes[poolId][msg.sender] -= totalAmount;
187       poolInfos[poolId].currentDepositedAmount -= totalAmount;
188       emit Redeemed(msg.sender, totalAmount);
189   }
```

Listing 2.2: YiledPool.sol

Impact This may allow users to bypass the time lock limitation and could also result in potential DoS issues.

Suggestion Revise the logic accordingly.

2.1.2 Incorrect state update

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The `requestRedemption` function in the `YieldPool` contract iterates through the `requests` array to calculate the portion already requested. If the array length exceeds the `maxRedemptionRequests` variable, the function attempts to update the last element of the `requests` array stored in memory. However, this does not result in any changes to the contract's state variables, leading to behavior inconsistent with the function's intended purpose.

```
129   function requestRedemption(uint256 poolId, uint256 amount) external nonReentrant {
130       require(poolInfos[poolId].redeemEnable, "Pool redeem disabled");
131       require(amount > 0, "Redemption amount must be greater than 0");
132       RedemptionRequest[] memory requests = userRedemptionRequests[poolId][msg.sender];
133       uint256 totalRequested = 0;
134       if (userRedemptionRequests[poolId][msg.sender].length > 0) {
135           for (uint256 i = 0; i < requests.length; i++) {
136               totalRequested += requests[i].amount;
```

```
137             }
138         }
139         require(userStakes[poolId][msg.sender] >= (amount + totalRequested), "Insufficient staked
                amount");
140
141         if (requests.length >= maxRedemptionRequests) {
142             uint256 index = _findLastRequest(poolId, msg.sender);
143             require(
144                 requests[index].unlockTimestamp > block.timestamp,
145                 "all Redemption request already exists"
146             );
147             requests[index].amount += amount;
148             requests[index].unlockTimestamp = block.timestamp + lockTime;
149         } else {
150             uint256 unlockTimestamp = block.timestamp + lockTime;
151             userRedemptionRequests[poolId][msg.sender].push(
152                 RedemptionRequest({amount: amount, unlockTimestamp: unlockTimestamp})
153             );
154         }
155
156         emit RedemptionRequested(msg.sender, amount);
157     }
```

Listing 2.3: YiledPool.sol

Impact The contract's state variable is not updated as intended.

Suggestion Update the corresponding storage variable (i.e., `userRedemptionRequests`).

2.2 Note

2.2.1 Potential centralization risk

Introduced by `Version 1`

Description The Yield Contracts includes several privileged functions that modify critical configurations and transfer token from the protocol, such as the `transferXSAT` fucntion in the `WIBTC` contract, `setLockTime` and `setMaxRedemptionRequests` functions in the `YieldPool` contract. If the private key of a privileged role is lost or maliciously exploited, it could lead to significant losses for users.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS