Security Audit Report for iBTC Contracts

Date: December 26, 2024 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	KikiFinance
Target	iBTC Contracts

Version History

Version	Date	Description
1.0	November 22, 2024	First release
2.0	December 26, 2024	Second release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-12-27 14:32:15+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 the iBTC Contracts [1] of KikiFinance. The iBTC Contracts enable users to deposit `XBTC` in exchange for `iBTC` on a 1:1 basis and delegate the staked `XBTC` to multiple validators via `StakeRouter` to earn rewards (typically `XSAT`), which are then distributed to `iBTC` holders.

Please note that the audit scope is limited to the contracts in the **contracts** folder. 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
iBTC Contracts	Version 1	3182d5ed920e3ce171ac105a66745bf51eb94980
	Version 2	816ee4eb61fb72e29c8237f6a83e561f6ea13310
	Version 3	8adb47dbcd5700274808d6962a87ca18e6cda34f
	Version 4	0015a50323d27930ab0b6790ee5d33542f98c350
	Version 5	4260fa2d7c2001fa3f7542b971974b4e4e9ac2a6

The contracts are deployed on the exSat network using the same `Version 5` code. The deployed contract addresses are listed in the following table.

Contract		Address
StakeRouter	Proxy	0xfe34D8B434324479Bb2850864e9B2966765B81Be
	Impl	0x6190b39e162232a633EbA445fA493F471e1D2B8c
iBTC	Proxy	0x8154Aaf094c2f03Ad550B6890E1d4264B5DdaD9A
	Impl	0x2c1ECEa8f2F56B91a69F11bB4f9dF09cabD3F3C1

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,

[1] https://github.com/KikiFinance/ibtc-contract/

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.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

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

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.
- **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 **five** recommendations and **one** note.

- High Risk: 2
- Low Risk: 1
- Recommendation: 5
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Flawed sorting logic in the `updateValidator` function	Software Security	Fixed
2	High	Flawed reward accounting logic in the `iBTC` contract	DeFi Security	Fixed
3	High	Flawed reward distribution logic due to split atomic operations	DeFi Security	Fixed
4	-	Traverse the `validators` array from the end when removing a validator	Recommendation	Fixed
5	-	Avoid using `address.transfer` for transferring native tokens	Recommendation	Fixed
6	-	Remove redundant operations in the `iBTC` contract	Recommendation	Fixed
7	-	Optimize the sorting algorithm to reduce gas consumption	Recommendation	Fixed
8	-	Implement non-zero value validation for key parameters	Recommendation	Fixed
9	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Flawed sorting logic in the `updateValidator` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `StakeRouter` contract, the `updateValidator` function allows operators to update the configuration of an existing validator and sort the `validators` array by priority. However, the sorting logic is flawed. For example, if there are four validators with priorities 50, 30, 20, and 10, and the priority of the second validator is updated to 5, the `validators` array are reordered to priorities 50, 5, 20, and 10, resulting in an evidently incorrect order.

```
function updateValidator(
    address _validator,
    uint256 _minStakePerTx,
```

```
        uint256 _maxStake,
        uint256 _priority
    ) external onlyOperator {
        require(_minStakePerTx <= _maxStake, "Minimum stake must be less than maximum stake");

        int256 index = getValidatorIndex(_validator);
        require(index >= 0, "Validator not found");

        Validator storage validator = validators[uint256(index)];

        if (validator.currentStake > 0) {
            require(
                validator.currentStake <= _maxStake,
                "Current stake must be less than or equal to the maximum stake"
            );
        }

        validator.minStakePerTx = _minStakePerTx;
        validator.maxStake = _maxStake;
        validator.priority = _priority;
        emit ValidatorUpdated(_validator, _minStakePerTx, _maxStake, _priority);
        _sortValidatorsByPriority(uint256(index));
    }
```

Listing 2.1: StakeRouter.sol

```
    function _sortValidatorsByPriority(uint256 startIndex) internal {
        for (uint256 i = startIndex; i > 0; i--) {
            if (validators[i].priority > validators[i - 1].priority) {
                Validator memory temp = validators[i];
                validators[i] = validators[i - 1];
                validators[i - 1] = temp;
            } else {
                break;
            }
        }
    }
```

Listing 2.2: StakeRouter.sol

Impact The `validators` array is incorrectly sorted.

Suggestion Revise the code logic accordingly.

2.2 DeFi Security

2.2.1 Flawed reward accounting logic in the `iBTC` contract

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the iBTC contract, the transfer and transferFrom functions invoke the hook function _beforeTokenTransfer to settle rewards for both the sender and the receiver (i.e., the from and to addresses). However, after updating token balances, the contract fails to update rewardDebt, which is critical for calculating pending rewards. This flaw leads to improper accounting, allowing an attacker to arbitrarily claim rewards through transfers.

For example, an attacker transfers 100 iBTC tokens to a controlled account with no token balance. The rewardDebt for the controlled account is set to zero in _beforeTokenTransfer before the balance update. The controlled account can then immediately invoke the claimReward function to claim XSAT tokens amounting to 100 * accRewardPerShare, violating the intended design.

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
    if (from != address(0)) {
        _settleReward(from);
    }
    if (to != address(0)) {
        _settleReward(to);
    }
    super._beforeTokenTransfer(from, to, amount);
}
```

Listing 2.3: iBTC.sol

```
function _settleReward(address userAddress) internal {
    require(xsatBalanceBefore == 0, "distributed rewards not yet finish");
    uint256 pending = _pendingReward(userAddress);
    if (pending > 0) {
        xsat.safeTransfer(userAddress, pending); // Transfer pending XSAT reward
        emit ClaimReward(userAddress, pending);
    }
    // Update the user's reward debt to reflect the latest accumulated reward per share
    rewardDebt[userAddress] = (balanceOf(userAddress) * accRewardPerShare) / PRECISION;
}
```

Listing 2.4: iBTC.sol

```
function _pendingReward(address userAddress) internal view returns (uint256) {
    uint256 userBalance = balanceOf(userAddress);
    uint256 accumulatedReward = (userBalance * accRewardPerShare) / PRECISION;
    if (accumulatedReward > rewardDebt[userAddress]) {
        return accumulatedReward - rewardDebt[userAddress];
    }
    return 0;
}
```

Listing 2.5: iBTC.sol

```
function claimReward() external nonReentrant {
    _settleReward(msg.sender);
}
```

Listing 2.6: iBTC.sol

Impact The attacker can arbitrarily claim rewards by transferring `iBTC` tokens to new addresses.

Suggestion Update the `rewardDebt` timely after the balance change.

2.2.2 Flawed reward distribution logic due to split atomic operations

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `iBTC` contract, rewards are accumulated in the `StakeRouter` contract and distributed to `iBTC` when the `finalizeRewardDistribution` function of the `StakeRouter` contract is invoked. The `iBTC` contract records its `XSAT` balance (stored in the `xsatBalanceBefore` variable) in the `prepareRewardDistribution function`, and calculates the amount of `XSAT` tokens received (based on the `xsatBalanceAfter` variable) in the `finalizeRewardDistribution` function after calling the corresponding function in the `StakeRouter` contract. However, these two operations should be atomic but are not in the current implementation, enabling an attacker to arbitrarily increase the reward per share in the `iBTC` contract.

Specifically, if the `finalizeRewardDistribution` function of the `StakeRouter` contract is executed successfully, the attacker can manipulate the `iBTC` contract's `XSAT` balance to be considered as rewards. Since the `finalizeRewardDistribution` function of the `StakeRouter` contract only requires its `XSAT` balance to be greater than zero, the attacker can transfer 1 wei of `XSAT` tokens to the `StakeRouter` contract to bypass this requirement. Consequently, the attacker can repeatedly invoke the `finalizeRewardDistribution` function of the `iBTC` contract to infinitely increase `accRewardPerShare`.

```
function prepareRewardDistribution() external nonReentrant {
    _prepareRewardDistribution();
}

// Finalizes reward distribution by updating accumulated rewards and recalculating rewards per share
function finalizeRewardDistribution() external nonReentrant {
    uint256 supply = totalSupply();
    require(supply > 0, "No iBTC in circulation for reward distribution");
    stakeRouter.finalizeRewardDistribution();

    // Get the updated XSAT balance after reward distribution
    uint256 xsatBalanceAfter = xsat.balanceOf(address(this));
    require(xsatBalanceAfter >= xsatBalanceBefore, "Balance error: Insufficient XSAT");

    // Calculate the newly received rewards and reset the balance tracker
    uint256 amount = xsatBalanceAfter - xsatBalanceBefore;
    xsatBalanceBefore = 0;

    // Update the accumulated reward per share for iBTC holders
    accRewardPerShare += (amount * PRECISION) / supply;

    emit RewardDistributed(amount);
```

```
}
```

Listing 2.7: iBTC.sol

```
function _pendingReward(address userAddress) internal view returns (uint256) {
    uint256 userBalance = balanceOf(userAddress);
    uint256 accumulatedReward = (userBalance * accRewardPerShare) / PRECISION;
    if (accumulatedReward > rewardDebt[userAddress]) {
        return accumulatedReward - rewardDebt[userAddress];
    }
    return 0;
}
```

Listing 2.8: iBTC.sol

```
function finalizeRewardDistribution() external onlyIBTC nonReentrant {
    // transfer all xsat reward to iBTC
    uint256 amount = xsat.balanceOf(address(this));
    require(amount > 0, "Balance error: Insufficient XSAT");

    xsat.safeTransfer(address(iBTC), amount);
    emit RewardDistributed(amount);
}
```

Listing 2.9: StakeRouter.sol

Impact The attacker can infinitely inflate the `accRewardPerShare` value, enabling them to drain all reward tokens.

Suggestion Revise the code logic accordingly.

2.3 Additional Recommendation

2.3.1 Traverse the `validators` array from the end when removing a validator

Status Fixed by `Version 2`

Introduced by `Version 1`

Description In the `StakeRouter` contract, the `removeValidator` function allows the operator to remove a validator whose `currentStake` is zero. According to the protocol's design, validators with smaller indexes have higher priority, so validators selected for removal typically have larger indexes and lower priority. However, the `getValidatorIndex` function traverses the `validators` array starting from index 0 to locate a validator with zero staked funds. This approach can fail due to gas exhaustion if the array is too large.

```
function removeValidator(address _validator) external onlyOperator {
    int256 index = getValidatorIndex(_validator);
    require(index >= 0, "Validator not found");

    uint256 validatorIndex = uint256(index);
    Validator storage validatorToRemove = validators[validatorIndex];

```

```
    require(
        validatorToRemove.currentStake == 0,
        "Cannot remove a validator with an active stake"
    );

    // Shift elements to the left to maintain order
    for (uint256 i = validatorIndex; i < validators.length - 1; i++) {
        validators[i] = validators[i + 1];
    }

    // Remove the last element
    validators.pop();
    emit ValidatorRemoved(_validator);
}
```

Listing 2.10: StakeRouter.sol

```
function getValidatorIndex(address _validator) internal view returns (int256) {
    for (uint256 i = 0; i < validators.length; i++) {
        if (validators[i].validatorAddress == _validator) {
            return int256(i);
        }
    }
    return -1; // Not found
}
```

Listing 2.11: StakeRouter.sol

Impact This can cause the `removeValidator` function to fail due to gas exhaustion if the array is too large.

Suggestion Traverse the `validators` array from the end.

2.3.2 Avoid using `address.transfer` for transferring native tokens

Status Fixed by `Version 2`

Introduced by `Version 1`

Description In the `iBTC` contract, the `withdrawBTC` function calls the `transfer` function to return native tokens to `msg.sender`. However, this can result in a transfer failure if the recipient is a contract and its `fallback` function contains complex logic that exceeds the gas limit imposed by transfer (i.e., 2300 gas units). It is recommended to use the `sendValue` function from OpenZeppelin's `Address` library to avoid such issues.

```
function withdrawBTC() external nonReentrant {
    uint256 totalAmount = _processWithdrawals();

    // Convert the XBTC to BTC using the XBTC contract's `withdraw` method
    ixbtc.withdraw(totalAmount);

    // Transfer the BTC to the user
    payable(msg.sender).transfer(totalAmount);

```

```
        emit Withdraw(msg.sender, totalAmount);
    }
```

Listing 2.12: iBTC.sol

Impact This may lead to unexpected transfer failures.

Suggestion Replace it with `Address.sendValue` from the OpenZeppelin library.

2.3.3 Remove redundant operations in the `iBTC` contract

Status Fixed by `Version 2`

Introduced by `Version 1`

Description In the `deposit`, `depositBTC`, and `requestWithdraw` functions, the initial invocation of the `_settleReward` function to handle reward settlement is redundant. This operation is already performed by the `_beforeTokenTransfer` function, which is called by the `_mint` or `_burn` functions.

```
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
        if (from != address(0)) {
            _settleReward(from);
        }
        if (to != address(0)) {
            _settleReward(to);
        }
        super._beforeTokenTransfer(from, to, amount);
    }
```

Listing 2.13: iBTC.sol

```
    function deposit(uint256 amount) public nonReentrant {
        _settleReward(msg.sender);

        require(amount > 0, "Amount must be greater than 0");
        xbtc.safeTransferFrom(msg.sender, address(this), amount);

        // Approve and stake the deposited XBTC with the StakeRouter
        xbtc.safeApprove(address(stakeRouter), amount);
        stakeRouter.deposit(amount);

        // Mint iBTC tokens at a 1:1 ratio with XBTC
        _mint(msg.sender, amount);

        // Update user's reward debt based on their new balance
        rewardDebt[msg.sender] = (balanceOf(msg.sender) * accRewardPerShare) / PRECISION;
        emit Deposit(msg.sender, amount);
    }

    function depositBTC() public payable nonReentrant {
        uint256 amount = msg.value;
        require(amount > 0, "Amount must be greater than 0");
        ixbtc.deposit{value: amount}();
```

```

        _settleReward(msg.sender);

        // Approve and stake the deposited XBTC with the StakeRouter
        xbtc.safeApprove(address(stakeRouter), amount);
        stakeRouter.deposit(amount);

        // Mint iBTC tokens at a 1:1 ratio with XBTC
        _mint(msg.sender, amount);

        // Update user's reward debt based on their new balance
        rewardDebt[msg.sender] = (balanceOf(msg.sender) * accRewardPerShare) / PRECISION;
        emit Deposit(msg.sender, amount);
    }

    // Allows users to request the withdrawal of XBTC by burning iBTC tokens
    function requestWithdraw(uint256 amount) external nonReentrant {
        uint256 userBalance = balanceOf(msg.sender);
        require(userBalance >= amount, "Insufficient iBTC balance");
        require(amount > 0, "Amount must be greater than zero");

        _settleReward(msg.sender);

        // Burn the corresponding amount of iBTC tokens
        _burn(msg.sender, amount);

        // Update user's reward debt to reflect their new balance
        rewardDebt[msg.sender] = (balanceOf(msg.sender) * accRewardPerShare) / PRECISION;

        // Create a new withdrawal request with an unlock timestamp determined by the StakeRouter
        uint256 unlockTimestamp = block.timestamp + stakeRouter.lockTime();
        userWithdrawals[msg.sender].push(WithdrawalRequest({
            amount: amount,
            unlockTimestamp: unlockTimestamp
        }));

        // Initiate withdrawal with the StakeRouter
        stakeRouter.withdraw(amount);

        emit WithdrawRequested(msg.sender, amount, unlockTimestamp);
    }
```

Listing 2.14: iBTC.sol

Impact Redundant operations can lead to more gas consumption.

Suggestion Remove redundant operations.

2.3.4 Optimize the sorting algorithm to reduce gas consumption

Status Fixed by `Version 3`

Introduced by `Version 2`

Description In the `StakeRouter` contract, the `updateValidator` function sorts validators by priority using the `_sortUpdatedValidatorsByPriority` function after reconfiguring a validator. The `_sortUpdatedValidatorsByPriority` function implements bubble sort, which has high computational complexity and leads to excessive gas usage. Notably, the priorities of the `validators` array from 0 to `index`-1 and from `index`+1 to `validators.length`-1 are already ordered. Therefore, only one side needs to be traversed during the sorting process.

```
function _sortUpdatedValidatorsByPriority() internal {
    uint256 n = validators.length;

    for (uint256 i = 1; i < n; i++) {
        Validator memory current = validators[i];
        uint256 j = i;

        while (j > 0 && validators[j - 1].priority < current.priority) {
            validators[j] = validators[j - 1];
            j--;
        }

        validators[j] = current;
    }
}
```

Listing 2.15: StakeRouter.sol

```
function updateValidator(
    address _validator,
    uint256 _minStakePerTx,
    uint256 _maxStake,
    uint256 _priority
) external onlyOperator {
    require(_minStakePerTx <= _maxStake, "Minimum stake must be less than maximum stake");

    int256 index = getValidatorIndex(_validator);
    require(index >= 0, "Validator not found");

    Validator storage validator = validators[uint256(index)];

    if (validator.currentStake > 0) {
        require(
            validator.currentStake <= _maxStake,
            "Current stake must be less than or equal to the maximum stake"
        );
    }

    validator.minStakePerTx = _minStakePerTx;
    validator.maxStake = _maxStake;
    validator.priority = _priority;

    emit ValidatorUpdated(_validator, _minStakePerTx, _maxStake, _priority);
    _sortUpdatedValidatorsByPriority();
}
```

Listing 2.16: StakeRouter.sol

Impact May lead to excessive gas consumption.

Suggestion Optimize the sorting algorithm.

2.3.5 Implement non-zero value validation for key parameters

Status Fixed by `Version 5`

Introduced by `Version 4`

Description In the `StakeRouter` contract, the configuration functions `setServiceFeeRecipient` and `setDefaultValidator` lack proper validation to ensure non-zero values for critical parameters.

- If `serviceFeeRecipient` is incorrectly set to the zero address while `serviceFeePercentage` is greater than zero, the `finalizeRewardDistribution` function will revert due to a failed requirement check, potentially leading to unexpected behavior.
- Similarly, if `defaultValidator` is set to the zero address, the `executeStakeTransfer` function will fail.

It is recommended to add zero-address validation in these functions to prevent such issues.

```
function setServiceFeeRecipient(address _serviceFeeRecipient) external onlyOperator {
    emit ServiceFeeRecipientUpdated(serviceFeeRecipient, _serviceFeeRecipient);
    serviceFeeRecipient = _serviceFeeRecipient;
}

function setDefaultValidator(address _defaultValidator) external onlyOperator {
    emit DefaultValidatorUpdated(defaultValidator, _defaultValidator);
    defaultValidator = _defaultValidator;
}
```

Listing 2.17: src/StakeRouter.sol

```
function finalizeRewardDistribution() external onlyIBTC nonReentrant {
    // Get the XSAT balance of the contract
    uint256 amount = xsat.balanceOf(address(this));
    require(amount > 0, "Balance error: Insufficient XSAT");

    // Calculate the service fee if it's not zero
    uint256 serviceFee = 0;
    if (serviceFeePercentage > 0) {
        // Check if serviceFeeRecipient is set
        require(serviceFeeRecipient != address(0), "Service fee recipient not set");
```

Listing 2.18: src/StakeRouter.sol

```
function executeStakeTransfer(address _user, address _fromValidator, uint256 _amount) external
     onlyIBTC nonReentrant {
    require(_amount > 0, "Amount must be greater than zero");

```

```
    // Ensure there's at least one validator
    require(defaultValidator != address(0), "Default validator not set");
```

Listing 2.19: src/StakeRouter.sol

Impact May result in some unexpected behaviors.

Suggestion Add non-zero value validation.

2.4 Note

2.4.1 Potential centralization risk

Introduced by `Version 1`

Description The protocol includes several privileged functions that modify critical configurations, such as upgrading the implementation, transferring `XSAT` in the `WIBTC` contract, and configuring validators in the `StakeRouter` 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