Security Audit Report for Guru

Date: February 26, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Guru
Target	Guru

Version History

Version	Date	Description
1.0	February 26, 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 audited file is provided as a **ZIP** archive.

The auditing process is iterative. Specifically, we would audit the files that fix the discovered issues. If there are new issues, we will continue this process. The MD5 hashes of the audited files 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	MD5 Hash
Guru	`Version 1`	`26f6cda68fe4285b45cb162e731971b1`
	`Version 2`	`970f950c003bc33808b792021d848b90`

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.

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 [1] and Common Weakness Enumeration [2]. 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.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

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

Chapter 2 Findings

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

- High Risk: 3
- Low Risk: 1
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	High	Lack of check in function `_executeSingleSwap()`	DeFi Security	Fixed
2	High	Potential signature replay risk in function `withdraw()`	DeFi Security	Fixed
3	High	Potential manipulation of cooldown period by fund manager	DeFi Security	Fixed
4	Low	Inconsistent capital adjustment during share transfer	DeFi Security	Fixed
5	-	Lack of checks in function `setMultisig()`	Recommendation	Fixed
6	-	Lack of check for _deposit.tvlDelta in function `depositAsset()`	Recommendation	Fixed
7	-	Potential centralization risk	Note	-
8	-	Off-Chain validation of fund name and symbol	Note	-
9	-	Fund manager margin withdrawal allowed by protocol	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of check in function `_executeSingleSwap()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SwapHelper`, the function `_executeSingleSwap()` performs a low-level call to a router for executing a token swap. However, the function does not verify whether the call succeeds or not.

As a result, if the swap operation fails, the function will continue execution without reverting. This means the protocol will still charge the swap fee, even though no tokens are actually swapped.

```
function _executeSingleSwap(Swap memory _swap) internal {
    uint256 tokenInBalanceBefore = _swap.tokenIn.balanceOf(address(this));
    uint256 tokenOutBalanceBefore = _swap.tokenOut.balanceOf(address(this));
```

```
        // Approve the router to spend the tokenIn
        _swap.tokenIn.forceApprove(address(_swap.router), _swap.amountToSend);


        // Forward the call to the router and ignore the return data (it gets optimized out)
        (bool success, ) = _swap.router.call(_swap.callData);
        success;


        ERC20(weth).safeTransfer(feeCollector, _swap.swapFee);


        uint256 tokenInBalanceAfter = _swap.tokenIn.balanceOf(address(this));
        uint256 tokenOutBalanceAfter = _swap.tokenOut.balanceOf(address(this));


        emit SwapExecuted(
            msg.sender,
            address(_swap.tokenIn),
            address(_swap.tokenOut),
            tokenInBalanceBefore - tokenInBalanceAfter,
            tokenOutBalanceAfter - tokenOutBalanceBefore,
            _swap.router
        );
    }
```

Listing 2.1: SwapHelper.sol

Impact If the swap fails, the function does not revert, causing the protocol to collect swap fees despite the transaction failing.

Suggestion Add a check to ensure that the transaction reverts if the swap operation fails.

2.1.2 Potential signature replay risk in function `withdraw()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `GuruFund` contract, function `withdraw()` receives a signature for authorization. Anyone can call this function with a valid signature, including both regular users and the `fund manager`. While repeated withdrawals by regular users do not pose a risk, the `fund manager` is required to maintain a margin deposit as a security measure enforced by the protocol.

However, because the same signed `payload` can be reused multiple times, the `fund manager` can repeatedly invoke function `withdraw()` before the signature expires. This effectively bypasses off-chain validation, allowing the `fund manager` to withdraw their entire margin deposit, which should not be allowed. This could compromise the protocol's security by removing the required financial commitment from the `fund manager`.

```
function withdraw(
    SignedPayload calldata _signedWithdrawPayload
) external nonReentrant verifyingSignature(_signedWithdrawPayload) {
    FundAction.Withdraw memory _userWithdrawal = abi.decode(
        _signedWithdrawPayload.data,
        (FundAction.Withdraw)
    );


    // 1. Burn tokens
    _burn(msg.sender, _userWithdrawal.burnAmount);


    // 2. Update invested capital
    unchecked {
        investedCapital[msg.sender] -= _userWithdrawal
            .amountsValue
            .investedCapital;
    }


    // 3. Execute swaps
    _executeSwaps(_userWithdrawal.swaps);


    // 4. Handle ETH transfers and fees
    _executeWithdrawalTransfers(_userWithdrawal.amountsWei);


    emit Withdrawn(
        msg.sender,
        _userWithdrawal.burnAmount,
        _userWithdrawal.amountsWei,
        _userWithdrawal.amountsValue,
        _userWithdrawal.tvlDelta
    );
}
```

Listing 2.2: GuruFund.sol

Impact The lack of replay protection in function `withdraw()` enables a `fund manager` to withdraw their entire margin deposit by reusing the same signature, bypassing protocol-imposed restrictions and potentially undermining trust in the system.

Suggestion Implement a `nonce` mechanism to ensure each signature can only be used once.

2.1.3 Potential manipulation of cooldown period by fund manager

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the GuruFund contract, the function updateMinDepositCooldown() allows the fund manager (i.e., contract owner) to modify minUserDepositCooldown at any time without restriction. This variable sets the mandatory waiting period for users before they can withdraw their funds. The timing of this cooldown is crucial because the protocol calculates a user's profit and loss (PnL) at the time of withdrawal.

The incentive structure for the fund manager is problematic. When users withdraw funds at a profit, the fund manager earns a portion of the gains as fees. Conversely, no fees are collected from withdrawals made at a loss. This discrepancy creates a potential for abuse: the fund manager could extend the cooldown period strategically when users are likely to incur losses, thus preventing them from withdrawing their funds. By delaying withdrawals until market conditions improve and users' balances turn profitable, the fund manager can then shorten the cooldown period to allow withdrawals and collect fees. Such manipulative tactics provide the fund manager with an undue advantage, compromising the fairness and integrity of the contract from the users' perspective.

```
function updateMinDepositCooldown(
  uint256 _newMinCooldown
) external onlyOpen onlyOwner {
  minUserDepositCooldown = _newMinCooldown;
  emit MinUserDepositCooldownUpdated(_newMinCooldown);
}
```

Listing 2.3: GuruFund.sol

Impact A malicious fund manager might manipulate withdrawal restrictions, delaying exits during unfavorable market conditions to prevent losses, and hastening them when profitable, allowing fee extraction. This selective adjustment leads to unethical fund management and could cause significant financial harm to users.

Suggestion Set a maximum limit on the cooldown period or require off-chain signer validation for modifications.

2.1.4 Inconsistent capital adjustment during share transfer

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In the GuruFund contract, when shares are transferred, the investedCapital of the recipient and sender are not updated. However, in the withdraw() function, the users' investedCapital is decreased when they withdraw, even though the investedCapital balance isn't adjusted during a transfer. This could lead to inconsistent or incorrect behavior, although the transaction won't revert due to the unchecked subtraction.

```
  function withdraw(
      SignedPayload calldata _signedWithdrawPayload
  ) external nonReentrant verifyingSignature(_signedWithdrawPayload) {
      FundAction.Withdraw memory _userWithdrawal = abi.decode(
          _signedWithdrawPayload.data,
```

```
            (FundAction.Withdraw)
        );


        // 1. Burn tokens
        _burn(msg.sender, _userWithdrawal.burnAmount);


        // 2. Update invested capital
        unchecked {
            investedCapital[msg.sender] -= _userWithdrawal
                .amountsValue
                .investedCapital;
        }


        // 3. Execute swaps
        _executeSwaps(_userWithdrawal.swaps);


        // 4. Handle ETH transfers and fees
        _executeWithdrawalTransfers(_userWithdrawal.amountsWei);


        emit Withdrawn(
            msg.sender,
            _userWithdrawal.burnAmount,
            _userWithdrawal.amountsWei,
            _userWithdrawal.amountsValue,
            _userWithdrawal.tvlDelta
        );
    }


    /**
     * @notice Executes the withdrawal transfers, including fees.
     * @param amountsWei The withdrawal amounts in wei units
     */
    function _executeWithdrawalTransfers(
        WithdrawalAmounts memory amountsWei
    ) internal {
        if (amountsWei.grossPnl <= 0) {
            _unwrapETH(amountsWei.netOutput);
        } else {
            unchecked {
                _unwrapETH(
                    amountsWei.netOutput +
                        amountsWei.protocolFee +
                        amountsWei.guruFee
                );
            }


```

```
        _safeTransferETH(fundFactory.multisig(), amountsWei.protocolFee);
        _safeTransferETH(owner(), amountsWei.guruFee);
    }
    _safeTransferETH(msg.sender, amountsWei.netOutput);
}
```

Listing 2.4: GuruFund.sol

Impact The variable `investedCapital` not properly updated during share transfer, which is incorrect.

Suggestion Revise the logic to ensure that the `investedCapital` variable is correctly updated during share transfer.

2.2 Additional Recommendation

2.2.1 Lack of checks in function `setMultisig()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `FundFactory`, the function `setMultisig()` allows the `owner` to update the multisig address. However, it lacks validation to prevent setting `_newMultisig` to `address(0)` or the current multisig address. This could lead to an invalid or redundant multisig update.

```
function setMultisig(address _newMultisig) public onlyOwner {
    multisig = _newMultisig;
    emit MultisigUpdated(_newMultisig);
}
```

Listing 2.5: FundFactory.sol

Suggestion Add checks to ensure `_newMultisig` is neither `address(0)` nor the current multisig address before updating.

2.2.2 Lack of check for _deposit.tvlDelta in function `depositAsset()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `depositAsset()` function is responsible for depositing an asset into the fund. One of the parameters, `_deposit.tvlDelta`, is of type `int256` and represents the change in the total value locked (`TVL`) in the fund. However, in the current implementation, this value is being directly cast from `int256` to `uint256` without verification. Since the `tvlDelta` can be a negative value, casting a negative number to `uint256` will result in an overflow and produce an extremely large value. This overflow would lead to incorrect fund accounting and possibly cause unexpected behavior, such as improper calculations of the total invested capital.

```
function depositAsset(
    SignedPayload calldata _signedAssetDeposit
) external onlyOwner nonReentrant verifyingSignature(_signedAssetDeposit) {
    FundAction.AssetDeposit memory _deposit = abi.decode(
        _signedAssetDeposit.data,
        (FundAction.AssetDeposit)
    );


    /// 1. Update asset index
    require(
        assets[_deposit.assetIndex] == ERC20(address(0)) ||
            assets[_deposit.assetIndex] == _deposit.asset,
        AssetIndexAlreadyOccupied(
            _deposit.assetIndex,
            assets[_deposit.assetIndex]
        )
    );


    assets[_deposit.assetIndex] = _deposit.asset;


    /// 2. Transfer deposit in
    _deposit.asset.safeTransferFrom(
        msg.sender,
        address(this),
        _deposit.amount
    );


    /// 3. Mint fund tokens
    _mint(msg.sender, _deposit.mintAmount);


    /// 4. Update invested capital
    investedCapital[msg.sender] += uint256(_deposit.tvlDelta);


    emit DepositedAsset(_deposit.asset, _deposit.amount, _deposit.tvlDelta);
}
```

Listing 2.6: GuruFund.sol

Suggestion Add a check to ensure that `_deposit.tvlDelta` is positive before performing the cast.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description The protocol relies on an off-chain trusted signer to sign the `payload` for critical operations, ensuring parameter validation. When users perform an operation, they must provide a valid signature to call the corresponding function. Once the signature is successfully verified, the payload is parsed into parameters and used in the execution logic. All parameter validations are conducted off-chain, including price calculations, the `price oracle` it relies on, and slippage control during swaps. Note that the correctness of these parameters are not in the scope of our audit. During the audit, we assume these components are secure and trustworthy. Additionally, the `fund manager`'s operations depend on their own market judgment, meaning users still bear the risk of losses when depositing funds into the contract. The whitelist of assets that the `fund manager` can purchase is also provided and verified off-chain by the protocol.

2.3.2 Off-Chain validation of fund name and symbol

Introduced by `Version 1`

Description In the `FundFactory` contract, the function `createFund()` initializes a new `fund` using `name` and `symbol` as identifiers. However, the contract itself does not enforce uniqueness for these parameters, relying instead on off-chain validation. Specifically, the `_signedPayload` is signed by a trusted signer, and the function verifies the signature. The off-chain logic ensures that the fund's `name` and `symbol` are unique before signing. Since this validation occurs outside the smart contract, its integrity is assumed but not within the contract's scope.

2.3.3 Fund manager margin withdrawal allowed by protocol

Introduced by `Version 1`

Description To mitigate malicious actions by fund managers, the `createFund()` function requires them to deposit a certain amount of native tokens as margin when creating a new fund. The protocol allows fund managers to withdraw this margin, but when they do, users of the corresponding fund are notified through the client-side interface.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS