BLOCKSEC

Security Audit

Report for Multistrategy Contracts

Date: October 23, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Goat Protocol
Target	Multistrategy Contracts

Version History

Version	Date	Description
1.0	October 23, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-10-23 12:13:10+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

This audit focuses on the Multistrategy feature of Goat Protocol [1]. The Goat Protocol is a decentralized yield optimizer that allows users to earn yield on their digital assets by automatically compounding rewards. The Multistrategy feature of Goat Protocol aggregates multiple strategies, such as the AaveAdapter. Specifically, only the following contracts in the repository are included in the scope of this audit. Other files are not within the scope of this audit.

- src/infra/multistrategy/Multistrategy.sol
- src/infra/multistrategy/adapters/AaveAdapter.sol
- src/infra/multistrategy/adapters/CurveLendAdapter.sol
- src/infra/multistrategy/adapters/GoatProtocolAdapter.sol
- src/abstracts/MultistrategyAdminable.sol
- src/abstracts/MultistrategyManageable.sol
- src/abstracts/StrategyAdapter.sol
- src/abstracts/StrategyAdapterAdminable.sol

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
Multistrategy Contracts	`Version 1`	`fc8b41e021d992310f8c8f9f5dc02af2d9cddeee`
	`Version 2`	`6e690cf9c14d251e4c891607307d47ea3916d3d0`

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 war-

[1] https://github.com/goatfi/contracts/tree/multistrategy

ranties 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**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

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

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

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 **six** potential security issues. Besides, we have **two** recommendations and **two** notes.

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

ID	Severity	Description	Category	Status
1	Low	The function `_calculateAmountToBeWithdrawn()` will revert if slippage is set to 100%	Software Security	Fixed
2	Low	Inconsistent implementations between functions `previewWithdraw()` and `previewRedeem()`	DeFi Security	Fixed
3	Low	Lack of revoking allowance in function `pause()`	DeFi Security	Fixed
4	Low	Unused functions `_pause()` and `_unpuase()` in contract `StrategyAdapterAdminable`	DeFi Security	Fixed
5	Medium	Profit distribution can be delayed unexpectedly	DeFi Security	Confirmed
6	Low	Lack of overriding function `_decimalsOffset()`	DeFi Security	Confirmed
7	-	Add threshold checks in function `setStrategyMinDebtDelta()`	Recommendation	Confirmed
8	-	Lack of check on `_slippageLimit` in function `setSlippageLimit()`	Recommendation	Fixed
9	-	Functions `previewRedeem()` and `previewWithdraw()` will return values accounting for the slippages	Note	-
10	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 The function `_calculateAmountToBeWithdrawn()` will revert if slippage is set to 100%

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, the parameter `slippageLimit` in the contract `StrategyAdapter` is used to prevent exceeding the expected amount. The `slippageLimit` can be set to

MAX_SLIPPAGE, which indicates no slippage, as described in the comment on Line 17. However, when the slippageLimit is set to MAX_SLIPPAGE, the function _calculateAmountToBeWithdrawn will revert due to a division-by-zero error, potentially resulting in a Denial of Service (DoS) issue.

```
17  /// @dev 100% in BPS, setting the slippage to 100% means no slippage protection.
18  uint256 constant MAX_SLIPPAGE = 10_000;
```

Listing 2.1: src/abstracts/StrategyAdapter.sol

```
82  function setSlippageLimit(uint256 _slippageLimit) external onlyOwner {
83      require(_slippageLimit <= MAX_SLIPPAGE, Errors.SlippageLimitExceeded(_slippageLimit));
84
85      slippageLimit = _slippageLimit;
86
87      emit SlippageLimitSet(_slippageLimit);
88  }
```

Listing 2.2: src/abstracts/StrategyAdapter.sol

```
175 function _calculateAmountToBeWithdrawn(uint256 _repayAmount, uint256 _strategyGain) internal
        view returns (uint256) {
176     uint256 exceedingDebt = IMultistrategy(multistrategy).debtExcess(address(this));
177     if(exceedingDebt > 0 && _repayAmount > 0) {
178         uint256 exceedingDebtWithSlippage = exceedingDebt.mulDiv(MAX_SLIPPAGE, MAX_SLIPPAGE -
                slippageLimit);
179         return Math.min(_repayAmount, exceedingDebtWithSlippage) + _strategyGain;
180     }
181
182     return _strategyGain;
183 }
```

Listing 2.3: src/abstracts/StrategyAdapter.sol

Impact Potential DoS when slippageLimit is set to MAX_SLIPPAGE.

Suggestion Revise the function setSlippageLimit to prevent the MAX_SLIPPAGE from being set, or handle the edge case in _calculateAmountToBeWithdrawn.

2.2 DeFi Security

2.2.1 Inconsistent implementations between functions previewWithdraw() and previewRedeem()

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The function previewWithdraw() calculates the amount of shares that will be burned for a given withdrawal assets amount, accounting for the slippage if there are insufficient assets. The function previewRedeem() calculates the amount of assets that will be transferred to the user for a given amount of shares that will be burned, accounting for the slippage if there are insufficient assets.

One invariant is that for the same amount of shares burned, the user should get the same amount of assets after taking the slippage into consideration. However, the current implementation doesn't follow this invariant. For example:

1. The current asset:share ratio is 1000:1000 (including the virtual decimals), and the slippage is 20%.
2. If a user calls `redeem()` to redeem 100 shares and there are insufficient assets, the `previewRedeem()` function will return 100 * (1-20%) = 80 assets.
3. However, if the user calls `withdraw()` with 80 assets, the `previewWithdraw()` will return 80 * (1+20%) = 96 shares.

```
90  function previewWithdraw(uint256 _assets) public view override returns (uint256) {
91      uint256 shares = _convertToShares(_assets, Math.Rounding.Ceil);
92      if(_assets <= _liquidity()) {
93          return shares;
94      } else {
95          // Return the number of shares required at the current rate, accounting for slippage.
96          return shares.mulDiv(MAX_BPS + slippageLimit, MAX_BPS, Math.Rounding.Ceil);
97      }
98  }
```

Listing 2.4: src/infra/multistrategy/Multistrategy.sol

```
101 function previewRedeem(uint256 _shares) public view override returns (uint256) {
102     uint256 assets = _convertToAssets(_shares, Math.Rounding.Floor);
103     if(assets <= _liquidity()) {
104         return assets;
105     } else {
106         // Return the number of assets redeemable at the maximum permitted slippage.
107         return assets.mulDiv(MAX_BPS - slippageLimit, MAX_BPS, Math.Rounding.Floor);
108     }
109 }
```

Listing 2.5: src/infra/multistrategy/Multistrategy.sol

Impact This inconsistency may lead to unfair issues or potentially destabilize the economic model.

Suggestion Revise the code logic.

2.2.2 Lack of revoking allowance in function `pause()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StrategyAdapter`, the function `unpause()` calls `_giveAllowances()` to set an unlimited allowance for the target external contract. However, the function `pause()` does not revoke the allowance accordingly, resulting in redundant invocation in function `unpause()`.

```
116 /// @inheritdoc IStrategyAdapter
117 function panic() external onlyGuardian {
```

```
118         _emergencyWithdraw();
119         _revokeAllowances();
120         _pause();
121     }
122
123     /// @inheritdoc IStrategyAdapter
124     function pause() external onlyGuardian {
125         _pause();
126     }
127
128     /// @inheritdoc IStrategyAdapter
129     function unpause() external onlyOwner {
130         _unpause();
131         _giveAllowances();
132     }
```

Listing 2.6: src/abstracts/StrategyAdapter.sol

Impact Redundant invocation of `_giveAllowances()` in the function `unpause()` after the function `pause()` is invoked.

Suggestion Add invocation `_revokeAllowances()` in the function `pause()`.

2.2.3 Unused functions `_pause()` and `_unpuase()` in contract `StrategyAdapterAdminable`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `StrategyAdapterAdminable` is inherited from the contract `Pausable` but doesn't use the internal functions `_pause()` and `_unpause()`, which means it lacks the pause-related capabilities.

Impact The function cannot be paused.

Suggestion Implement external functions that use functions `_pause()` and `_unpause()` or mark the contract as abstract.

2.2.4 Profit distribution can be delayed unexpectedly

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The contract `Multistrategy` allows multiple strategies to be integrated and independently report profits and losses. All profits reported will be locked and released gradually with the following formula: `lockedFundsRatio = (block.timestamp - lastReport) * lockedProfitDegradation`. However, since the `lastReport` is updated each time a strategy reports, previously locked profits can be potentially delayed.

```
292   /// @notice Calculates the free funds available in the contract.
293   /// @return The amount of free funds available.
294   function _freeFunds() internal view returns (uint256) {
295       return totalAssets() - _calculateLockedProfit();
296   }
297
298   /// @notice Calculate the current locked profit.
299   ///
300   /// This function performs the following actions:
301   /// - Calculates the locked funds ratio based on the time elapsed since the last report and
          the locked profit degradation rate.
302   /// - If the locked funds ratio is less than the degradation coefficient, it computes the
          remaining locked profit by reducing it proportionally.
303   /// - If the locked funds ratio is greater than or equal to the degradation coefficient, it
          returns zero indicating no locked profit remains.
304   ///
305   /// @return The calculated current locked profit.
306   function _calculateLockedProfit() internal view returns (uint256) {
307       uint256 lockedFundsRatio = (block.timestamp - lastReport) * lockedProfitDegradation;
308
309       if(lockedFundsRatio < DEGRADATION_COEFFICIENT) {
310           return lockedProfit - lockedFundsRatio.mulDiv(lockedProfit, DEGRADATION_COEFFICIENT);
311       }
312       return 0;
313   }
```

Listing 2.7: src/infra/multistrategy/Multistrategy.sol

```
452   function _report(uint256 _debtRepayment, uint256 _gain, uint256 _loss) internal {
453       uint256 strategyBalance = IERC20(asset()).balanceOf(msg.sender);
454       require(!(_gain > 0 && _loss > 0), Errors.GainLossMismatch());
455       require(strategyBalance >= _debtRepayment + _gain, Errors.InsufficientBalance(
              strategyBalance, _debtRepayment + _gain));
456
457       uint256 profit = 0;
458       uint256 feesCollected = 0;
459       if(_loss > 0) _reportLoss(msg.sender, _loss);
460       if(_gain > 0) {
461           feesCollected = _gain.mulDiv(performanceFee, MAX_BPS);
462           profit = _gain - feesCollected;
463       }
464
465       uint256 debtToRepay = Math.min(_debtRepayment, _debtExcess(msg.sender));
466       if(debtToRepay > 0) {
467           strategies[msg.sender].totalDebt -= debtToRepay;
468           totalDebt -= debtToRepay;
469       }
470
471       uint256 newLockedProfit = _calculateLockedProfit() + profit;
472       if(newLockedProfit > _loss) {
473           lockedProfit = newLockedProfit - _loss;
474       } else {
475           lockedProfit = 0;
```

```
476         }
477
478         strategies[msg.sender].lastReport = block.timestamp;
479         lastReport = block.timestamp;
480
481         if(debtToRepay + _gain > 0) IERC20(asset()).safeTransferFrom(msg.sender, address(this),
                debtToRepay + _gain);
482         if(feesCollected > 0) IERC20(asset()).safeTransfer(protocolFeeRecipient, feesCollected);
483
484         emit StrategyReported(msg.sender, debtToRepay, profit, _loss);
485     }
```

Listing 2.8: src/infra/multistrategy/Multistrategy.sol

Impact Users' profits may be delayed.

Suggestion Revise the logic to prevent the issue when a new report is submitted.

Feedback from the project This is intended behavior. We acknowledge that a part of the profit that was realized at day 0 can still be in "distribution" for longer than 7 days if new reports come in. But it is a way to average out the rewards for 7 days and keep the code simple.

We could introduce a buffer that keeps the profits and once harvested it distributes them the next 7 days. But we think it is a more complex solution (code wise and infra wise, as we need to maintain the harvest worker).

2.2.5 Lack of overriding function `_decimalsOffset()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Currently, the `Multistrategy` vault uses the default decimals offset to defend against the share's price inflation attack [1]. By default, the decimals offset is 0, which means the `convertToShares()` will be calculated as `assets * (total_supply + 1) / (total_assets + 1)`.

According to Openzeppelin's documentation [2], the default settings can make the inflation attack able to absorb users' assets but are non-profitable. However, the calculation only considered a single deposit from innocent users. If there are multiple deposits after inflation, attackers can make a profit. For example, suppose the current asset:share is 1:1 including the virtual decimals) with zero slippage:

1. An attacker deposits 1 token, gets one share, and donates 98 tokens. The asset:share ratio becomes 100:2.
2. Alice deposits 49 tokens and gets zero shares. The asset:share ratio becomes 149:2.
3. Bob deposits 74 tokens and gets zero shares. The asset:share ratio becomes 223:2.
4. The attacker redeems one share and gets 111 tokens, making profits.

[1] https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks

[2] https://docs.openzeppelin.com/contracts/5.x/erc4626

Impact Attackers might absorb users' assets and get profits.

Suggestion Increasing the return value of `_decimalsOffset()` (e.g., 3) will make the attack magnitudinous harder.

Feedback from the project All multistrategies are required to have an initial deposit by the deployer / DAO of $100 before going live to the users. We'd like the receipt token to be the same "size" as the deposit asset. 1 ETH = 1 receiptETH at launch. Due to UX. Setting the decimal offset to 3 would set 1 ETH = 1000 receiptETH at launch.

2.3 Additional Recommendation

2.3.1 Add threshold checks in function `setStrategyMinDebtDelta()`

Status Confirmed

Introduced by `Version 1`

Description In the current implementation, the only restriction in the `setStrategyMinDebtDelta()` is that `maxDebtDelta >= _minDebtDelta`. However, there is no threshold check for `minDebtDelta`. When the `minDebtDelta` is set too high, the function `_creditAvailable` will consistently return 0, resulting in unexpected behavior.

```
215   function setStrategyMinDebtDelta(address _strategy, uint256 _minDebtDelta) external
216       onlyManager
217       onlyActiveStrategy(_strategy)
218   {
219       require(strategies[_strategy].maxDebtDelta >= _minDebtDelta, Errors.InvalidDebtDelta());
220
221       strategies[_strategy].minDebtDelta = _minDebtDelta;
222
223       emit StrategyMinDebtDeltaSet(_strategy, _minDebtDelta);
224   }
```

Listing 2.9: src/infra/multistrategy/Multistrategy.sol

Suggestion Add a threshold check for `minDebtDelta`.

Feedback from the project We won't add thresholds as the delta is "amount of tokens". Different multistrategies would require different deltas. We also want to be pretty dynamic. A multistrategy on mainnet during high gas seasons (consistent +50 gwei for weeks) would require a high minDebtDelta.

2.3.2 Lack of check on `_slippageLimit` in function `setSlippageLimit()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, the function `setSlippageLimit()` of contract `MultistrategyManageable` doesn't check if `_slippageLimit` <= `MAX_BPS`, which may lead to underflow and revert later.

```
129   function setSlippageLimit(uint256 _slippageLimit) external onlyManager {
130       slippageLimit = _slippageLimit;
131       emit SlippageLimitSet(slippageLimit);
132   }
```

Listing 2.10: src/infra/multistrategy/Multistrategy.sol

Suggestion Check `_slippageLimit` <= `MAX_BPS` in function `setSlippageLimit()`.

2.4 Note

2.4.1 Functions `previewRedeem()` and `previewWithdraw()` will return values accounting for the slippages

Introduced by `Version 1`

Description Currently, the functions `previewRedeem()` and `previewWithdraw()` will return values that account for the slippages, which is specified by `slippageLimit`. Though it follows the specification of EIP-4626, this implementation should be notified by the entities interacting with them.

```
90    function previewWithdraw(uint256 _assets) public view override returns (uint256) {
91        uint256 shares = _convertToShares(_assets, Math.Rounding.Ceil);
92        if(_assets <= _liquidity()) {
93            return shares;
94        } else {
95            // Return the number of shares required at the current rate, accounting for slippage.
96            return shares.mulDiv(MAX_BPS + slippageLimit, MAX_BPS, Math.Rounding.Ceil);
97        }
98    }
```

Listing 2.11: src/infra/multistrategy/Multistrategy.sol

```
101   function previewRedeem(uint256 _shares) public view override returns (uint256) {
102       uint256 assets = _convertToAssets(_shares, Math.Rounding.Floor);
103       if(assets <= _liquidity()) {
104           return assets;
105       } else {
106           // Return the number of assets redeemable at the maximum permitted slippage.
107           return assets.mulDiv(MAX_BPS - slippageLimit, MAX_BPS, Math.Rounding.Floor);
108       }
109   }
```

Listing 2.12: src/infra/multistrategy/Multistrategy.sol

2.4.2 Potential centralization risks

Introduced by `Version 1`

Description There are several important functions in the protocol, which are only callable by the owner, manager, or guardians. If their private keys are lost or compromised, it could lead to losses for the protocol and users.

Feedback from the Project We're aware of this. Owner of the contract will be a 12h or 24h timelock only callable by the Protocol Multisig. Manager will be a Multisig with a minimum of 2/3 signatures. Guardians can either be an EOA or a Multisig.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS