SMART CONTRACT AUDIT REPORT

for

Vovo Finance Glp Vault

Prepared By: Yiqun Chen

PeckShield

April 8, 2022

Document Properties

Client	Vovo Finance
Title	Smart Contract Audit Report
Target	Vovo Finance Glp Vault
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 8, 2022	Jing Wang	Final Release
1.0-rc	February 19, 2022	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Vovo Finance Glp Vault protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Vovo Finance Glp Vault

Vovo Finance provides passive investment returns with customizable risks. The GlpVault product of the protocol receives tokens from users and then uses the token to buy and stake GLP from GMX. Periodically, the vault collects the rewards (WETH and esGMX) and uses the WETH rewards to open a leverage trade on GMX, and stakes esGMX to earn more rewards. The basic information of the Vovo Finance Glp Vault protocol is as follows:

Table 1.1: Basic Information of The Vovo Finance Glp Vault Protocol

Item	Description
Issuer	Vovo Finance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 8, 2022

In the following, we list the reviewed files and the commit hash values used in this audit.

- https://github.com/VovoFinance/VovoProducts/blob/glpVault/contracts/products/GlpVault.sol (bfb0188)

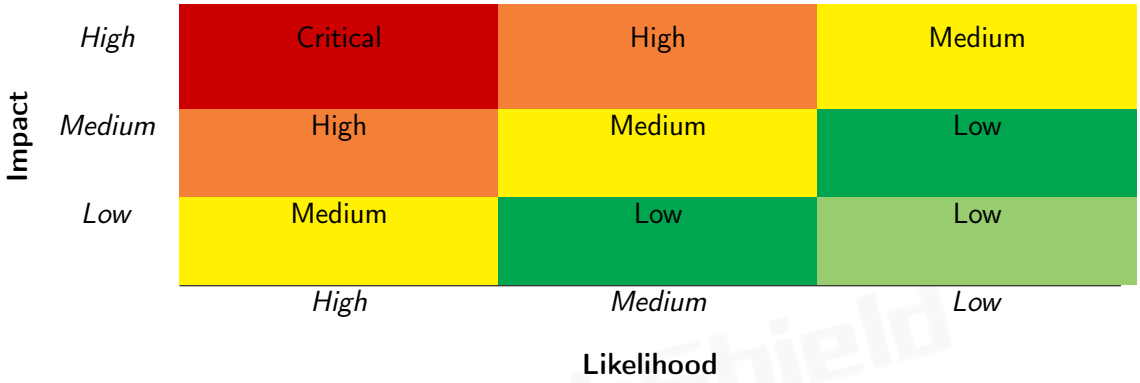
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/VovoFinance/VovoProducts/blob/glpVault/contracts/products/GlpVault.sol (7cd95b0)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Vovo Finance Glp Vault` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	1	■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined some issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Vovo Finance Glp Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	High	Improper Logic Of Handling withdrawAmount	Business Logic	Fixed
PVE-004	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GlpVault`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The `GlpVault` contract has a helper routine, i.e., `poke()`, that is designed to collect the rewards for staked `GLPs`, use trade profit to mint and stake more `GLPs` to earn reward. It has a rather straightforward logic to swap `WETH` tokens and mint `GLP` tokens by calling `mintAndStakeGlp()` to actually perform the intended token swap.

```
225   function poke() external nonReentrant {
226     require(keepers[msg.sender] || !isKeeperOnly, "!keepers");
227     require(lastPokeTime + pokeInterval < block.timestamp, "!poke time");
228     uint256 tokenReward = collectReward();
229     closeTrade();
230     if (tokenReward > 0) {
231       openTrade(tokenReward);
232     }
233     currentTokenReward = tokenReward;

235     uint256 glpAmount = 0;
236     uint256 wethBalance = IERC20(weth).balanceOf(address(this));
237     if (wethBalance > 0) {
238       IERC20(weth).safeApprove(glpManager, 0);
239       IERC20(weth).safeApprove(glpManager, wethBalance);
240       glpAmount = IRewardRouter(rewardRouter).mintAndStakeGlp(weth, wethBalance, 0, 0);
241     }
242     lastPokeTime = block.timestamp;
243     emit Poked(tokenReward, glpAmount);
```

```
244   }
```

Listing 3.1: GlpVault::poke()

To elaborate, we show above the poke() routine. We notice the token swap are routed to rewardRouter and do not specify any restriction (with _minGlp=0) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of UniswapV2. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense. Note another routine closeTrade() shares the same issue.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the team. And the team clarifies that they have confirmed with GMX team that there's no slippage when minting GLP, only the fee could be larger if there is front runner.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: GlpVault
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Vovo Finance Glp Vault protocol, there is a special administrative account, i.e., admin. This admin account plays a critical role in governing and regulating the system-wide operations (e.g., setting protocol-wide parameters, etc.). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `registerVault()` and `revokeVault()` functions in the `GlpVault` contract, which allow the `admin` to configure which `toVault` is allowed to be withdrawn to in the `withdrawToVault()` routine.

```
107     function registerVault(address fromVault, address toVault) external onlyAdmin {
108         withdrawMapping[fromVault][toVault] = true;
109         emit VaultRegistered(fromVault, toVault);
110       }

112       function revokeVault(address fromVault, address toVault) external onlyAdmin {
113         withdrawMapping[fromVault][toVault] = false;
114         emit VaultRevoked(fromVault, toVault);
115       }
```

Listing 3.2: `GlpVault::registerVault()and revokeVault()`

```
296     function withdrawToVault(uint256 shares, address vault) external nonReentrant {
297         require(vault != address(0), "!vault");
298         require(withdrawMapping[address(this)][vault], "Withdraw to vault not allowed");

300         uint256 glpAmount = (balance().mul(shares)).div(totalSupply());
301         _burn(msg.sender, shares);
302         IERC20(stakedGlp).approve(vault, glpAmount);
303         IGlpVault(vault).depositGlp(glpAmount);
304         uint256 receivedShares = IERC20(vault).balanceOf(address(this));
305         IERC20(vault).safeTransfer(msg.sender, receivedShares);

307         emit WithdrawGlp(msg.sender, glpAmount, 0);
308         emit WithdrawToVault(msg.sender, shares, vault, receivedShares);
309       }
```

Listing 3.3: `GlpVault::withdrawToVault()`

It is worrisome if the privileged `admin` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that they will use the `TimeLock` contract to be the admin of the `GlpVault` contract.

3.3 Improper Logic Of Handling withdrawAmount

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GlpVault`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The `GlpVault` contract provides several routines to help users to withdraw funds from the vault, including `withdraw()`, `withdrawToVault()` and `withdrawGlp()`. For example, the `withdraw()` routine is used to withdraw desired tokens from the vault by burning user shares. To elaborate, we show below the related routine.

```
206  function withdraw(uint256 shares, address tokenOut, uint256 minOut) external returns(
         uint256 withdrawAmount) {
207    require(shares > 0, "!shares");
208    uint256 glpAmount = (balance().mul(shares)).div(totalSupply());
209    _burn(msg.sender, shares);
210
211    uint256 tokenOutAmount = 0;
212    if (IERC20(tokenOut).isETH()) {
213      tokenOutAmount = IRewardRouter(rewardRouter).unstakeAndRedeemGlpETH(glpAmount,
           minOut, address(this));
214    } else {
215      tokenOutAmount = IRewardRouter(rewardRouter).unstakeAndRedeemGlp(tokenOut,
           glpAmount, minOut, address(this));
216    }
217    uint256 fee = tokenOutAmount.mul(withdrawalFee).div(FEE_DENOMINATOR);
218    IERC20(tokenOut).uniTransfer(rewards, fee);
219    withdrawAmount = tokenOutAmount.sub(fee);
220    emit Withdraw(tokenOut, msg.sender, withdrawAmount, fee);
221  }
```

Listing 3.4: `GlpVault::withdraw()`

We notice the `withdraw()` routine is using the `unstakeAndRedeemGlpETH()` and `unstakeAndRedeemGlp()` routines to unstake `GLP` and transfer `tokenOutAmount` of `tokenOut` to this vault. However, only `fee` of `tokenOut` is sent to `rewards`. The `withdrawAmount` of `tokenOut` that needs to be sent to the user is still left in the contract.

Recommendation Send the `withdrawAmount` of `tokenOut` to the user.

Status The issue has been fixed by this commit: `a7a92f7`.

3.4 Inconsistency Between Document and Implementation

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GlpVault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

There is a misleading comment embedded among lines of solidity code, which brings unnecessary hurdles to understand and/or maintain the software.

Specifically, if we examine the `GlpVault::withdraw()` routine, the preceding function summary indicates that this function is used to "Withdraw from this vault to another vault". However, the implemented logic indicates it is used to withdraw specific tokens to the user.

```
311  /**
312   * @notice Withdraw from this vault to another vault
313   * @param shares the number of this vault shares to be burned
314   * @param tokenOut the withdraw token
315   * @param minOut the minimum amount of tokenOut to withdraw
316   */
317  function withdraw(uint256 shares, address tokenOut, uint256 minOut) external returns(
         uint256 withdrawAmount) {
318    require(shares > 0, "!shares");
319    uint256 glpAmount = (balance().mul(shares)).div(totalSupply());
320    _burn(msg.sender, shares);
321
322    uint256 tokenOutAmount = 0;
323    if (IERC20(tokenOut).isETH()) {
324      tokenOutAmount = IRewardRouter(rewardRouter).unstakeAndRedeemGlpETH(glpAmount,
             minOut, address(this));
325    } else {
326      tokenOutAmount = IRewardRouter(rewardRouter).unstakeAndRedeemGlp(tokenOut,
             glpAmount, minOut, address(this));
327    }
328    uint256 fee = tokenOutAmount.mul(withdrawalFee).div(FEE_DENOMINATOR);
329    IERC20(tokenOut).uniTransfer(rewards, fee);
330    withdrawAmount = tokenOutAmount.sub(fee);
331    emit Withdraw(tokenOut, msg.sender, withdrawAmount, fee);
332  }
```

Listing 3.5: `GlpVault::withdraw()`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: `9a73ce2`.

4 | Conclusion

In this audit, we have analyzed the Vovo Finance GlpVault product design and implementation. The GlpVault product receives tokens from users and then use the token to buy and stake GLP from GMX. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.