SMART CONTRACT AUDIT REPORT

for

BeamEx

Prepared By: Xiaomi Huang

PeckShield

June 6, 2023

Document Properties

Client	BeamEx
Title	Smart Contract Audit Report
Target	BeamEx
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Liu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 6, 2023	Xuxian Jiang	Final Release
1.0-rc	June 4, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the BeamEx 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 BeamEx

BeamEx is a perpetual DEX that arises from the successes of Beamswap, a leading DeFi hub on the Moonbeam Network, and is expanded with more opportunities for derivatives trading. As a decentralized spot and perpetual exchange, BeamEx brings advanced crypto trading, both swaps and leverage trades, at low fees and zero price impact. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of BeamEx

Item	Description
Name	BeamEx
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 6, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/BeamSwap/beamex-contracts.git (02bcafd)

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

- https://github.com/BeamSwap/beamex-contracts.git (a62fd80)

1.2 About PeckShield

PeckShield Inc. [11] 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

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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

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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 design and implementation of the `BeamEx` protocol smart contracts. 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	0	
Medium	3	
Low	3	
Informational	1	
Total	7	

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 a few issues of varying severities 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 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Logic in TokenManager::signalSetGov()	Business Logic	Confirmed
PVE-002	Medium	BnGMX Reduction Minimization with JIT StakedGMX Inflation	Business Logic	Resolved
PVE-003	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-006	Low	Incorrect Position Execution in PositionRouter	Business Logic	Resolved
PVE-007	Medium	BLP CooldownDuration Bypass in Liquidity Removal	Business Logic	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited Logic in TokenManager::signalSetGov()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `BeamEx` protocol has a built-in `TokenManager` contract, which is tasked with the voting and execution of managed tokens. While examining the admin-related logic, we notice an implicit assumption on the current governance as an active `signer` and this implicit assumption may not always hold.

To elaborate, we show below the `signalSetGov()` function. It implements a rather straightforward logic in setting the pending action. However, it comes to our attention that it immediately sets the corresponding `signedActions` mapping (with the `msg.sender` key) as true. In other words, it implicitly assumes the current governance is an active signer, which may not be the case.

```
    function signalSetGov(address _timelock, address _target, address _gov) external
        nonReentrant onlyAdmin {
        actionsNonce++;
        uint256 nonce = actionsNonce;
        bytes32 action = keccak256(abi.encodePacked("signalSetGov", _timelock, _target,
            _gov, nonce));
        _setPendingAction(action, nonce);
        signedActions[msg.sender][action] = true;
        emit SignalSetGov(_timelock, _target, _gov, action, nonce);
    }
```

Listing 3.1: `TokenManager::signalSetGov()`

Recommendation Resolve the implicit assumption of the current governance as the active signer.

Status The issue has been acknowledged by the team.

3.2 BnGMX Reduction Minimization with JIT StakedGMX Inflation

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RewardRouter`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To incentivize the long-time stakers without inflation, the `BeamEx` protocol has a so-called `multiplier points`. Specifically, when a user stakes the governance token, the user will receive `multiplier points` every second at a fixed rate of `100% APR`. When `GMX` or `Escrowed GMX` tokens are unstaked, the proportional amount of `multiplier points` are burnt. While reviewing the current unstaking logic, we notice the current implementation can be improved.

To elaborate, we show below the related `_unstakeGmx()` routine. As the name indicates, this routine is used to unstake `GMX` with the necessary support of burning the proportional `multiplier points`. However, it comes to our attention that the computed amount of `multiplier points` to burn may be manipulated to retain the majority of `multiplier points`.

```
    function _unstakeGmx(address _account, address _token, uint256 _amount) private {
        require(_amount > 0, "RewardRouter: invalid _amount");

        uint256 balance = IRewardTracker(stakedGmxTracker).stakedAmounts(_account);

        IRewardTracker(feeGmxTracker).unstakeForAccount(_account, bonusGmxTracker,
            _amount, _account);
        IRewardTracker(bonusGmxTracker).unstakeForAccount(_account, stakedGmxTracker,
            _amount, _account);
        IRewardTracker(stakedGmxTracker).unstakeForAccount(_account, _token, _amount,
            _account);

        uint256 bnGmxAmount = IRewardTracker(bonusGmxTracker).claimForAccount(_account,
            _account);
        if (bnGmxAmount > 0) {
            IRewardTracker(feeGmxTracker).stakeForAccount(_account, _account, bnGmx,
                bnGmxAmount);
        }

        uint256 stakedBnGmx = IRewardTracker(feeGmxTracker).depositBalances(_account,
            bnGmx);
        if (stakedBnGmx > 0) {
```

```
            uint256 reductionAmount = stakedBnGmx.mul(_amount).div(balance);
            IRewardTracker(feeGmxTracker).unstakeForAccount(_account, bnGmx,
                reductionAmount, _account);
            IMintable(bnGmx).burn(_account, reductionAmount);
        }

        emit UnstakeGmx(_account, _amount);
    }
```

Listing 3.2: `RewardRouter::_unstakeGmx()`

Here is an example list of steps that can avoid the burn of most `multiplier points`. For simplicity, let's assume the user `Malice` has staked `GMX` and he wishes to unstake `GMX` while minimizing the amount of `bnGMX` burnt.

1. `Malice` initially calls `IRewardTracker(stakedGmxTracker).stake(GMX, JIT_AMOUNT)` to increase the staked `GMX` balance, i.e., with the addition of `JIT_AMOUNT`.

2. `Malice` performs the unstaking call, i.e., `RewardRouter.unstakeGmx()`. Note the calculation of reduced `bnGMX` amount is shown as follows: `bnGMX = bnGMX balance * GMX amount unstaked / GMX balance`. Since the staked `GMX` balance is increased, a smaller `bnGMX` amount to burn is derived.

3. `Malice` calls `IRewardTracker(stakedGmxTracker).unstake(GMX, JIT_AMOUNT)` to unstake the JIT'ed `GMX` balance – `JIT_AMOUNT`.

Recommendation Revise the above unstaking logic to reliably compute the `bnGMX` amount to burn.

Status This issue has been resolved as the team confirms this contract is not used in the protocol.

3.3 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Treasury
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the `BeamEx` protocol has a constant need of swapping one asset to another. With that, the protocol has provided related helper routines to facilitate the asset conversion. Moreover, the `Treasury` contract has the support of adding liquidity into the pair. Our analysis shows the current liquidity-adding logic may be subject to possible manipulation.

```
    function addLiquidity() external onlyGov nonReentrant {
        require(!isLiquidityAdded, "Treasury: liquidity already added");
        isLiquidityAdded = true;

        uint256 busdAmount = busdReceived.mul(busdBasisPoints).div(BASIS_POINTS_DIVISOR);
        uint256 gmtAmount = busdAmount.mul(PRECISION).div(gmtListingPrice);

        IERC20(busd).approve(router, busdAmount);
        IERC20(gmt).approve(router, gmtAmount);

        IGMT(gmt).endMigration();

        IPancakeRouter(router).addLiquidity(
            busd, // tokenA
            gmt, // tokenB
            busdAmount, // amountADesired
            gmtAmount, // amountBDesired
            0, // amountAMin
            0, // amountBMin
            address(this), // to
            block.timestamp // deadline
        );

        IGMT(gmt).beginMigration();

        uint256 fundAmount = busdReceived.sub(busdAmount);
        IERC20(busd).transfer(fund, fundAmount);
    }
```

Listing 3.3: `Treasury::addLiquidity()`

To elaborate, we show above the related `addLiquidity()` routine. We notice it is routed to `UniswapV2`-like router in order to provide the desired liquidity. Apparently, the instant DEX price for liquidity addition is highly volatile and there is a need to consider the use of `TWAP` and further specify necessary restriction on possible slippage, so that it is not vulnerable to possible front-running attacks.

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.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been resolved as the team confirms this contract is not used in the protocol.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
```

```
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses'
        //  allowance to zero by calling 'approve(_spender, 0)' if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: `USDT Token Contract`

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfe()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In the following, we show the `approve()` routine from the `TokenManager` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(_token).approve(_spender, _amount)` (line 87) may revert as there is no return value in the USDT token contract's approve() implementation (but the IERC20 interface expects a return value)!

```
    function approve(address _token, address _spender, uint256 _amount, uint256 _nonce) external nonReentrant onlyAdmin {
        bytes32 action = keccak256(abi.encodePacked("approve", _token, _spender, _amount, _nonce));
        _validateAction(action);
        _validateAuthorization(action);

        IERC20(_token).approve(_spender, _amount);
        _clearAction(action, _nonce);
    }
```

Listing 3.6: TokenManager::approve()

Note this issue is also applicable to other routines, including GmxMigrator::migrate(), MigrationHandler::redeemUsdg(), GmxTimelock::approve(), and GmxTimelock::approve(),. For the safeApprove() support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom(). Note the transfer()/transferFrom()-related issue also affects a number of contracts, including GMT::withdrawToken(), GmxMigrator::migrate(), BalanceUpdater, BatchSender, PriceFeedTimelock, and Timelock.

Status This issue has been resolved as the team confirms the affected contracts are not used in the protocol.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the BeamEx protocol, there is a privileged admin account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and adding new allowed tokens). 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 the related privileged accesses in current contracts.

```
    function setMaxGlobalShortSize(address _vault, address _token, uint256 _amount)
        external onlyAdmin {
        IVault(_vault).setMaxGlobalShortSize(_token, _amount);
    }

    function removeAdmin(address _token, address _account) external onlyAdmin {
        IYieldToken(_token).removeAdmin(_account);
    }

    function setIsAmmEnabled(address _priceFeed, bool _isEnabled) external onlyAdmin {
        IVaultPriceFeed(_priceFeed).setIsAmmEnabled(_isEnabled);
    }

    function setIsSecondaryPriceEnabled(address _priceFeed, bool _isEnabled) external
        onlyAdmin {
        IVaultPriceFeed(_priceFeed).setIsSecondaryPriceEnabled(_isEnabled);
    }

    function setMaxStrictPriceDeviation(address _priceFeed, uint256
        _maxStrictPriceDeviation) external onlyAdmin {
        IVaultPriceFeed(_priceFeed).setMaxStrictPriceDeviation(_maxStrictPriceDeviation)
            ;
    }

    function setUseV2Pricing(address _priceFeed, bool _useV2Pricing) external onlyAdmin
        {
        IVaultPriceFeed(_priceFeed).setUseV2Pricing(_useV2Pricing);
    }

    function setAdjustment(address _priceFeed, address _token, bool _isAdditive, uint256
         _adjustmentBps) external onlyAdmin {
        IVaultPriceFeed(_priceFeed).setAdjustment(_token, _isAdditive, _adjustmentBps);
    }

    function setSpreadBasisPoints(address _priceFeed, address _token, uint256
        _spreadBasisPoints) external onlyAdmin {
        IVaultPriceFeed(_priceFeed).setSpreadBasisPoints(_token, _spreadBasisPoints);
    }
```

Listing 3.7: Example Privileged Functions in `GmxTimelock`

Note that if the privileged `admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 This issue has been mitigated as the team makes use of a `multisig` account to act as the privileged admin.

3.6 Incorrect Position Execution in PositionRouter

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PositionRouter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The `BeamEx` protocol has a `PositionRouter` contract to faciliate the interaction with the main `Vault`. While examining the current helper routines, we notice a specific one can be improved.

Specifically, this affected routine `executeIncreasePosition()` is designed to execute an operation to increase the user position. It comes to our attention that the inherent fee-collection call `_collectFees()` is wrongly provided with `msg.sender` as the position owner. To fix, the first argument to `_collectFees()` should be `request.account`, not the current `msg.sender`. Note the same issue is also applicable to another routine, i.e., `_createIncreaseOrder()`.

```
    function executeIncreasePosition(bytes32 _key, address payable _executionFeeReceiver) public nonReentrant returns (bool) {
        IncreasePositionRequest memory request = increasePositionRequests[_key];
        // if the request was already executed or cancelled, return true so that the executeIncreasePositions loop will continue executing the next request
        if (request.account == address(0)) { return true; }

        bool shouldExecute = _validateExecution(request.blockNumber, request.blockTime, request.account);
        if (!shouldExecute) { return false; }

        delete increasePositionRequests[_key];

        if (request.amountIn > 0) {
            uint256 amountIn = request.amountIn;

            if (request.path.length > 1) {
                IERC20(request.path[0]).safeTransfer(vault, request.amountIn);
```

```
                amountIn = _swap(request.path, request.minOut, address(this));
            }

            uint256 afterFeeAmount = _collectFees(msg.sender, request.path, amountIn, request.indexToken, request.isLong, request.sizeDelta);
            IERC20(request.path[request.path.length - 1]).safeTransfer(vault, afterFeeAmount);
        }

        _increasePosition(request.account, request.path[request.path.length - 1], request.indexToken, request.sizeDelta, request.isLong, request.acceptablePrice);

        _transferOutETHWithGasLimitIgnoreFail(request.executionFee, _executionFeeReceiver);

        emit ExecuteIncreasePosition(
            request.account,
            request.path,
            request.indexToken,
            request.amountIn,
            request.minOut,
            request.sizeDelta,
            request.isLong,
            request.acceptablePrice,
            request.executionFee,
            block.number.sub(request.blockNumber),
            block.timestamp.sub(request.blockTime)
        );

        _callRequestCallback(request.callbackTarget, _key, true, true);

        return true;
    }
```

Listing 3.8: `PositionRouter::executeIncreasePosition()`

Recommendation Revise the above affected routines to properly provide the user account, instead of `msg.sender`.

Status This issue has been resolved by following the above the suggestions.

3.7 BLP CooldownDuration Bypass in Liquidity Removal

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BlpManager
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The BeamEx protocol has a BlpManager contract that allows the minting and redemption of BLP, the platform's liquidity provider token. We notice there is a cooldown duration after minting BLP. The cooldown duration represents the time that needs to pass for the user before it can be redeemed. Our analysis shows that this cooldown enforcement can be bypassed.

To elaborate, we show below the related _removeLiquidity() routine. When the intended liquidity is requested for removal, this routine will validate the cooldown duration is passed. However, it can trivially bypassed by transfering the BLP to another new account and instructing the new account to perform the liquidity removal – without further being constrained by the cooldown duration.

```
    function _removeLiquidity(
        address _account,
        address _tokenOut,
        uint256 _glpAmount,
        uint256 _minOut,
        address _receiver
    ) private returns (uint256) {
        require(_glpAmount > 0, "BlpManager: invalid _glpAmount");
        require(
            lastAddedAt[_account].add(cooldownDuration) <= block.timestamp,
            "BlpManager: cooldown duration not yet passed"
        );

        // calculate aum before sellUSDG
        uint256 aumInUsdg = getAumInUsdg(false);
        uint256 glpSupply = IERC20(blp).totalSupply();

        uint256 usdgAmount = _glpAmount.mul(aumInUsdg).div(glpSupply);
        uint256 usdgBalance = IERC20(usdg).balanceOf(address(this));
        if (usdgAmount > usdgBalance) {
            IUSDG(usdg).mint(address(this), usdgAmount.sub(usdgBalance));
        }

        IMintable(blp).burn(_account, _glpAmount);

        IERC20(usdg).transfer(address(vault), usdgAmount);
        uint256 amountOut = vault.sellUSDG(_tokenOut, _receiver);
        require(amountOut >= _minOut, "BlpManager: insufficient output");
```

```
        emit RemoveLiquidity(
            _account,
            _tokenOut,
            _glpAmount,
            aumInUsdg,
            glpSupply,
            usdgAmount,
            amountOut
        );

        return amountOut;
    }
```

Listing 3.9: `BlpManager::_removeLiquidity()`

Recommendation Revise the `BLP` routine to honor the above cooldown duration as well.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `BeamEx` protocol, which is a perpetual DEX and arises from the successes of `Beamswap`, a leading `DeFi` hub on the `Moonbeam Network`. The protocol is expanded with more opportunities for derivatives trading. As a decentralized spot and perpetual exchange, `BeamEx` brings advanced crypto trading, both swaps and leverage trades, at low fees and zero price impact. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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

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