SMART CONTRACT AUDIT REPORT

for

Thruster

Prepared By: Xiaomi Huang

PeckShield

February 29, 2024

Document Properties

Client	Thruster
Title	Smart Contract Audit Report
Target	Thruster
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 29, 2024	Xuxian Jiang	Final Release
1.0-rc	February 22, 2024	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 `Thruster` 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 Thruster

`Thruster` is a `DEX` that provides critical fair launch, liquidity, and social infrastructure for builders, yield seekers, and traders on `Blast`. It provides the tried and true reliability of `DEXes` and adds additional informational and interactive components to make the on-chain experience dramatically better. Harnessing the power of multiple liquidity pool types and `Blast` native yield, `Thruster LPTs` are fungible and are composable for a wide variety of utility, including on-chain leverage and collateralization, and yield strategies. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Thruster

Item	Description
Name	Thruster
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 29, 2024

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

- https://github.com/ThrusterX/thruster-protocol.git (0568a06)

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

- https://github.com/ThrusterX/thruster-protocol.git (d4580c5)

1.2 About PeckShield

PeckShield Inc. [12] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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

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

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) [10], 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 `Thruster` protocol. 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	1	■
Low	4	■ ■ ■ ■
Informational	0	
Total	5	

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 1 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Enforcement of Implicit Assumption in ThrusterTreasure::enterTickets()	Business Logic	Resolved
PVE-002	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
PVE-003	Low	Implicit Assumption Enforcement in ThrusterRouter::AddLiquidity()	Coding Practices	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Enforcement of Implicit Assumption in ThrusterTreasure::enterTickets()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ThrusterTreasure`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The `Thruster` protocol has a core `ThrusterTreasure` constract with a lottery game implementation that uses entropy to determine winners. While examining the logic to enter tickets for active current round, we notice the need to revisit current logic.

In the following, we shows the implementation of the related `enterTickets()` routine. It has a rather straightforward logic in entering tickets into the current active round of `Thruster Treasure`. It has an implicit assumption that a user can only enter tickets once per round. However, this implicit assumption can be made explicit by enforcing the following requirement, i.e., `require(entered[msg.sender][currentRound_].ticketStart==ticketEnd)`.

```
    function enterTickets(uint256 _amount, bytes32[] calldata _proof) external {
        uint256 currentRound_ = currentRound;
        require(winningTickets[currentRound_][0].length == 0, "ET");
        bytes32 node = keccak256(abi.encodePacked(msg.sender, _amount));
        require(MerkleProof.verify(_proof, root, node), "IP");
        uint256 ticketsToEnter = _amount - cumulativeTickets[msg.sender];
        require(ticketsToEnter > 0, "NTE");
        uint256 currentTickets_ = currentTickets;
        Round memory round = Round(currentTickets_, currentTickets_ + ticketsToEnter,
            currentRound_);
        entered[msg.sender][currentRound_] = round;
        cumulativeTickets[msg.sender] = _amount; // Ensure user can only enter tickets
            once, no partials
```

```
        currentTickets += ticketsToEnter;
        emit EnteredTickets(msg.sender, currentTickets_, currentTickets_ +
            ticketsToEnter, currentRound_);
    }
```

Listing 3.1: `ThrusterTreasure::enterTickets()`

Recommendation Improve the above routine to explicitly enforce that a user can only enter tickets once per round.

Status This issue has been fixed by the following commit: `f46a4a0`.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ThrusterTreasure`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `ThrusterTreasure` as an example, the `claimPrizesForRound()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 114) start before effecting the update on internal state (line 118), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
102    function claimPrizesForRound(uint256 roundToClaim) external {
103        require(roundStart[roundToClaim] + MAX_ROUND_TIME >= block.timestamp, "ICT");
104        require(winningTickets[roundToClaim][0].length > 0, "NWT");
```

```
        Round memory round = entered[msg.sender][roundToClaim];
        require(round.ticketEnd > round.ticketStart, "NTE");
        uint256 maxPrizeCount_ = maxPrizeCount;
        for (uint256 i = 0; i < maxPrizeCount_; i++) {
            Prize memory prize = prizes[roundToClaim][i];
            uint256[] memory winningTicketsRoundPrize = winningTickets[roundToClaim][i];
            for (uint256 j = 0; j < winningTicketsRoundPrize.length; j++) {
                uint256 winningTicket = winningTicketsRoundPrize[j];
                if (round.ticketStart <= winningTicket && round.ticketEnd >
                    winningTicket) {
                    _claimPrize(prize, msg.sender, winningTicket);
                }
            }
        }
        entered[msg.sender][roundToClaim] = Round(0, 0, roundToClaim); // Clear user's
            tickets for the round
        emit CheckedPrizesForRound(msg.sender, roundToClaim);
    }
```

Listing 3.2: `ThrusterTreasure::claimPrizesForRound()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed by the following commit: `b7823c5`.

3.3 Implicit Assumption Enforcement in ThrusterRouter::AddLiquidity()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ThrusterRouter`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the `ThrusterRouter` contract, there is a routine that is used to add liquidity into a pair. This routine is named `addLiquidity()` and is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external virtual override ensure(deadline) returns (uint256 amountA, uint256 amountB, uint256 liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin, amountBMin);
        address pair = ThrusterLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IThrusterPair(pair).mint(to);
    }
```

Listing 3.3: `ThrusterRouter::addLiquidity()`

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {
        // create the pair if it doesn't exist yet
        if (IThrusterFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IThrusterFactory(factory).createPair(tokenA, tokenB);
        }
        (uint256 reserveA, uint256 reserveB) = ThrusterLibrary.getReserves(factory, tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint256 amountBOptimal = ThrusterLibrary.quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, "ThrusterRouter: INSUFFICIENT_B_AMOUNT");
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint256 amountAOptimal = ThrusterLibrary.quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, "ThrusterRouter: INSUFFICIENT_A_AMOUNT");
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
```

```
    }
```

Listing 3.4: `ThrusterRouter::_addLiquidity()`

It comes to our attention that the `ThrusterRouter` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `ThrusterRouter` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

Status The issue has been confirmed.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ThrusterTreasure`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

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 this section, we examine the `transfer()` routine and possible 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. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
```

```
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.5: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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.

In current implementation, if we examine the `ThrusterTreasure::retrieveTokens()` routine that is designed to withdraw the funds from the treasury contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 197).

```
192     function retrieveTokens(address _recipient, address _token, uint256 _amount)
            external onlyOwner {
193         IERC20Rebasing token = IERC20Rebasing(_token);
194         if (_amount == 0) {
195             _amount = token.balanceOf(address(this));
196         }
197         token.transfer(_recipient, _amount);
198         emit WithdrawPrizes(_recipient, _token, _amount);
199     }
```

Listing 3.6: ThrusterTreasure::retrieveTokens()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed by the following commit: `d4580c5`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Thruster protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., configure various parameters, set up prizes, withdraw funds, and execute privileged operations). 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 setMaxPrizeCount(uint256 _maxPrizeCount) external onlyOwner {
        maxPrizeCount = _maxPrizeCount;
        emit SetMaxPrizeCount(_maxPrizeCount);
    }

    /**
     * Claims the Blast native yield
     * @param _recipient - The address to claim the yield to
     * @param _amountWETH - The amount of WETH to claim
     * @param _amountUSDB - The amount of USDB to claim
     */
    function claimYield(address _recipient, uint256 _amountWETH, uint256 _amountUSDB)
        external onlyOwner {
        WETH.claim(_recipient, _amountWETH);
        USDB.claim(_recipient, _amountUSDB);
    }

    /**
     * Sets the prize for a round
     * @param _round - The round to set the prize for
     * @param _prizeIndex - The index of the prize to set
     * @param _amountWETH - The amount of WETH to set
     * @param _amountUSDB - The amount of USDB to set
     * @param _numWinners - The number of winners for the prize
     */
    function setPrize(uint256 _round, uint64 _prizeIndex, uint256 _amountWETH, uint256
        _amountUSDB, uint64 _numWinners)
        external
        onlyOwner
    {
        require(_round >= currentRound, "ICR");
```

```
        require(_prizeIndex < maxPrizeCount, "IPC");
        depositPrize(msg.sender, _amountWETH, _amountUSDB);
        prizes[_round][_prizeIndex] = Prize(_amountWETH, _amountUSDB, _numWinners, _prizeIndex, uint64(_round));
    }...
    function retrieveTokens(address _recipient, address _token, uint256 _amount) external onlyOwner {
        IERC20Rebasing token = IERC20Rebasing(_token);
        if (_amount == 0) {
            _amount = token.balanceOf(address(this));
        }
        token.transfer(_recipient, _amount);
        emit WithdrawPrizes(_recipient, _token, _amount);
    }

    /**
     * Retrieve ETH from the contract
     * @param _recipient - The address to retrieve the ETH to
     * @param _amount - The amount of ETH to retrieve
     */
    function retrieveETH(address payable _recipient, uint256 _amount) external onlyOwner {
        if (_amount == 0) {
            _amount = address(this).balance;
        }
        _recipient.transfer(_amount);
        emit WithdrawPrizes(_recipient, address(0), _amount);
    }
```

Listing 3.7: Example Privileged Functions in `ThrusterTreasure`

Note that if the privileged `owner` 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.

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 to act as the privileged owner.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Thruster` protocol, which is a `DEX` that provides critical fair launch, liquidity, and social infrastructure for builders, yield seekers, and traders on `Blast`. It provides the tried and true reliability of `DEXes` and adds additional informational and interactive components to make the on-chain experience dramatically better. Harnessing the power of multiple liquidity pool types and `Blast` native yield, `Thruster LPTs` are fungible and are composable for a wide variety of utility, including on-chain leverage and collateralization, and yield strategies. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[14] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.