SMART CONTRACT AUDIT REPORT

for

Thena Finance

Prepared By: Xiaomi Huang

PeckShield

March 25, 2023

Document Properties

Client	Thena Finance
Title	Smart Contract Audit Report
Target	Thena Finance
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	March 25, 2023	Xuxian Jiang	Final Release
1.0-rc	March 22, 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 Thena 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 Thena

Thena is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from Solidly/Velodrome with a unique AMM. The DEX is compatible with all the standard features as popularized by UniswapV2 with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps, as well as a built-in NFT-based voting mechanism and associated token emissions. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Thena Finance

Item	Description
Name	Thena Finance
Website	https://thena.fi/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 25, 2023

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

- https://github.com/ThenafiBNB/THENA-Contracts.git (52d42ca)

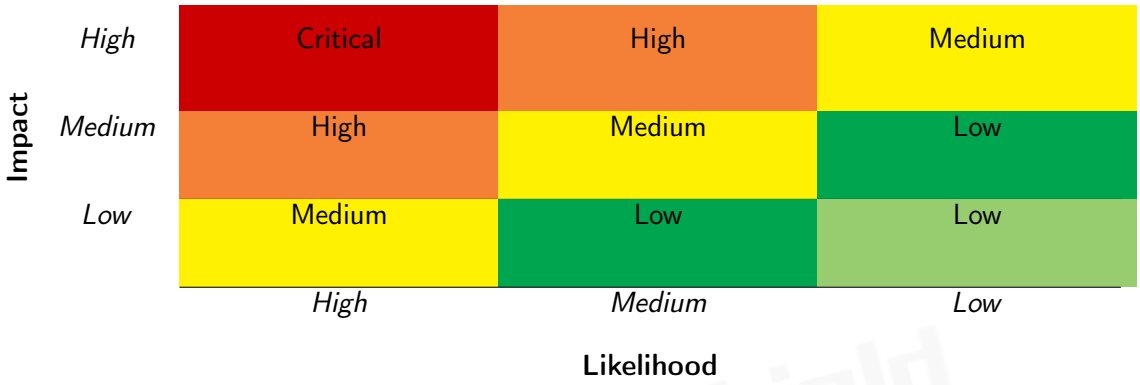
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/ThenafiBNB/THENA-Contracts.git (07106e7)

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



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

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

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) [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 Thena 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	5	■■■■■
Informational	0	
Total	8	

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 and 5 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect Delegate/Voting Balance Accounting in VotingEscrow	Business Logic	Resolved
PVE-002	Low	Improper VotingEscrow Query in RewardsDistributor	Coding Practices	Resolved
PVE-003	Low	Revisited Proposal State For Cancellation in Governor	Business Logic	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Medium	Improved Airdrop Logic in AirdropClaim::setUserInfo()	Business Logic	Resolved
PVE-006	Low	Suggested Adherence Of Checks Effects Interactions Pattern	Business Logic	Resolved
PVE-007	Medium	Killed Gauges Still Eligible For Rewards	Business Logic	Resolved
PVE-008	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 Incorrect Delegate/Voting Balance Accounting in VotingEscrow

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `VotingEscrow`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Thena` protocol has a core `VotingEscrow` contract that escrows the governance tokens in the form of an `ERC-721 NFT`. It also has a built-in delegation feature that allows a user to delegate the voting power to another user. In the process of reviewing the delegation feature, we notice the current implementation is flawed.

In particular, we show below the logic of a core routine that implements the delegation feature. As the name indicates, this `_moveAllDelegates()` routine records the changes of the owner's NFTs as part of the delegate operation. However, it comes to our attention that the previously delegated NFTs are being duplicated in `srcRepNew` when `nextSrcRepNum = srcRepNum-1`, which could seriously affect the voting balance calculation. Note another routine `_moveTokenDelegates()` shares the same issue.

```
1310    function _moveAllDelegates(
1311        address owner,
1312        address srcRep,
1313        address dstRep
1314    ) internal {
1315        // You can only re-delegate what you own
1316        if (srcRep != dstRep) {
1317            if (srcRep != address(0)) {
1318                uint32 srcRepNum = numCheckpoints[srcRep];
1319                uint[] storage srcRepOld = srcRepNum > 0
```

```
1320                     ? checkpoints[srcRep][srcRepNum - 1].tokenIds
1321                     : checkpoints[srcRep][0].tokenIds;
1322                 uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
1323                 uint[] storage srcRepNew = checkpoints[srcRep][nextSrcRepNum].tokenIds;
1324                 // All the same except what owner owns
1325                 for (uint i = 0; i < srcRepOld.length; i++) {
1326                     uint tId = srcRepOld[i];
1327                     if (idToOwner[tId] != owner) {
1328                         srcRepNew.push(tId);
1329                     }
1330                 }
1331
1332                 numCheckpoints[srcRep] = srcRepNum + 1;
1333             }
1334
1335             if (dstRep != address(0)) {
1336                 uint32 dstRepNum = numCheckpoints[dstRep];
1337                 uint[] storage dstRepOld = dstRepNum > 0
1338                     ? checkpoints[dstRep][dstRepNum - 1].tokenIds
1339                     : checkpoints[dstRep][0].tokenIds;
1340                 uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
1341                 uint[] storage dstRepNew = checkpoints[dstRep][
1342                     nextDstRepNum
1343                 ].tokenIds;
1344                 uint ownerTokenCount = ownerToNFTokenCount[owner];
1345                 require(
1346                     dstRepOld.length + ownerTokenCount <= MAX_DELEGATES,
1347                     "dstRep would have too many tokenIds"
1348                 );
1349                 // All the same
1350                 for (uint i = 0; i < dstRepOld.length; i++) {
1351                     uint tId = dstRepOld[i];
1352                     dstRepNew.push(tId);
1353                 }
1354                 // Plus all that's owned
1355                 for (uint i = 0; i < ownerTokenCount; i++) {
1356                     uint tId = ownerToNFTokenIdList[owner][i];
1357                     dstRepNew.push(tId);
1358                 }
1359
1360                 numCheckpoints[dstRep] = dstRepNum + 1;
1361             }
1362         }
1363     }
```

Listing 3.1: `VotingEscrow::_moveAllDelegates()`

Recommendation Revise the above delegate logic to properly record the set of NFTs being delegated.

Status This issue has been resolved as the team confirms the above delegate logic is not used

anymore.

3.2 Improper VotingEscrow Query in RewardsDistributor

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardsDistributor`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To incentivize the long-time stakers with inflation, the `Thena` protocol has a built-in `RewardsDistributor` contract to calculate inflation and adjust emission balances accordingly. While reviewing the current logic, we notice three different routines can be improved.

To elaborate, we show below one example `ve_for_at()` routine. This routine is proposed to calculate the voting power of the given `NFT` at the specific timestamp. It comes to our attention that the resulting `int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.ts))))` (line 140) may be negative. However, when it is negative, the type cast to `uint` makes it positive, which leads to an incorrect calculation of voting power (in this case, the resulting voting power should be 0.). The same issue is also applicable to two other routines `_checkpoint_total_supply()` and `_claim()`.

```
135     function ve_for_at(uint _tokenId, uint _timestamp) external view returns (uint) {
136         address ve = voting_escrow;
137         uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
138         uint epoch = _find_timestamp_user_epoch(ve, _tokenId, _timestamp, max_user_epoch
                );
139         IVotingEscrow.Point memory pt = IVotingEscrow(ve).user_point_history(_tokenId,
                epoch);
140         return Math.max(uint(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.
                ts))))), 0);
141     }
```

Listing 3.2: `RewardsDistributor::ve_for_at()`

Recommendation Revise the above three routines to properly compute the user's voting power.

Status This issue has been fixed in the following commit: `07106e7`.

3.3 Revisited Proposal State For Cancellation in Governor

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Governor, L2Governor`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Thena` protocol has a built-in governance to facilitate the protocol operation and management. In particular, each protocol has its own lifecycle and its associated protocol state. While reviewing the possible protocol states, we notice the current protocol cancellation operation makes use of an incorrect protocol state.

To elaborate, we show below the related code snippet `_cancel()`, which validates the current state not in `Canceled`, `Expired`, and `Executed`. Our analysis shows that the state of `Expired` here should be replaced with `Defeated` – as the current `state()` routine never returns the `Expired` state.

```
374     function _cancel(
375         address[] memory targets,
376         uint256[] memory values,
377         bytes[] memory calldatas,
378         bytes32 descriptionHash
379     ) internal virtual returns (uint256) {
380         uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);
381         ProposalState status = state(proposalId);
383         require(
384             status != ProposalState.Canceled && status != ProposalState.Expired &&
                    status != ProposalState.Executed,
385             "Governor: proposal not active"
386         );
387         _proposals[proposalId].canceled = true;
389         emit ProposalCanceled(proposalId);
391         return proposalId;
392     }
```

Listing 3.3: `L2Governor::_cancel()`

Moreover, another related routine `execute()` is invoked to execute the protocol actions. We notice one specific validation – `require(status == ProposalState.Succeeded || status == ProposalState.Queued)` (line 301), which potentially checks the `Queued` state. However, the `state()` routine never returns the `Queued` state.

```
291     function execute(
292         address[] memory targets,
293         uint256[] memory values,
294         bytes[] memory calldatas,
295         bytes32 descriptionHash
296     ) public payable virtual override returns (uint256) {
297         uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);

299         ProposalState status = state(proposalId);
300         require(
301             status == ProposalState.Succeeded  status == ProposalState.Queued,
302             "Governor: proposal not successful"
303         );
304         _proposals[proposalId].executed = true;

306         emit ProposalExecuted(proposalId);

308         _beforeExecute(proposalId, targets, values, calldatas, descriptionHash);
309         _execute(proposalId, targets, values, calldatas, descriptionHash);
310         _afterExecute(proposalId, targets, values, calldatas, descriptionHash);

312         return proposalId;
313     }
```

Listing 3.4: `L2Governor::execute()`

Recommendation Revise the above two routines to properly examine possible protocol states.

Status This issue has been fixed in the following commit: `07106e7`.

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).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

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

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.5: 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()`.

```
38      /**
39       * @dev Deprecated. This function has issues similar to the ones found in
40       * {IERC20-approve}, and its usage is discouraged.
41       *
42       * Whenever possible, use {safeIncreaseAllowance} and
43       * {safeDecreaseAllowance} instead.
44       */
45      function safeApprove(
46          IERC20 token,
47          address spender,
48          uint256 value
49      ) internal {
50          // safeApprove should only be called when setting an initial allowance,
51          // or when resetting it to zero. To increase and decrease it, use
52          // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53          require(
54              (value == 0)  (token.allowance(address(this), spender) == 0),
55              "SafeERC20: approve from non-zero to non-zero allowance"
56          );
57          _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
                spender, value));
```

```
58      }
```

Listing 3.6: SafeERC20::safeApprove()

In the following, we show the setUserInfo() routine from the AirdropClaim contract. If the USDT token is supported as token, the unsafe version of token.approve(ve, 0) (line 122) may revert as there is no return value in the USDT token contract's approve() implementation (but the IERC20 interface expects a return value)!

```
92      function setUserInfo(address _who, address _to, uint256 _amount) external onlyMerkle
            nonReentrant returns(bool status) {

94          require(_who != address(0), 'addr 0');
95          require(_to != address(0), 'addr 0');
96          require(_amount > 0, 'amnt 0');
97          require(usersFlag[_who] == false, '!flag');
98          require(init, 'not init');


101         uint256 _vestedAmount = _amount * VESTED_SHARE / PRECISION;
102         uint256 _theInstantAmount = _amount * LINEAR_DISTRO / PRECISION;
103         uint256 _theLockedLinearAmount = _theInstantAmount;
104         uint256 _tokenPerSec = _theLockedLinearAmount * PRECISION / DISTRIBUTION_PERIOD;

106         UserInfo memory _user = UserInfo({
107             totalAmount: _amount,
108             initAmount: _theInstantAmount,
109             vestedAmount: _vestedAmount,
110             lockedAmount: _theLockedLinearAmount,
111             tokenPerSec: _tokenPerSec,
112             lastTimestamp: startTimestamp,
113             claimed: _theInstantAmount + _vestedAmount,
114             to: _to
115         });

117         users[_who] = _user;
118         usersFlag[_who] = true;

120         // send out init amount
121         token.safeTransfer(_to, _theInstantAmount);
122         token.approve(ve, 0);
123         token.approve(ve, _vestedAmount);
124         IVotingEscrow(ve).create_lock_for(_vestedAmount, 2 * 364 * 86400 , _who);

126         status = true;
127     }
```

Listing 3.7: AirdropClaim::setUserInfo()

Note this issue is also applicable to other routines in AirdropClaimTheNFT and NFTSalesSplitter contracts. 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()`.

Status This issue has been confirmed and the team clarifies that the supported tokens are expected to have the full ERC20-compliance.

3.5 Improved Airdrop Logic in AirdropClaim::setUserInfo()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AirdropClaim`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Thena` protocol supports the airdrop mechanism that divides the airdropped amount into three parts: `initAmount`, `vestedAmount`, and `lockedAmount`. We notice the current logic to compute these three parts can be improved.

In the following, we show the implementation of the related `setUserInfo()` routine. This routine properly computes the `initAmount` and `vestedAmount` (lines 101 – 102), but not `lockedAmount` (line 103). A correct assignment to `lockedAmount` should be the following: `uint256 _theLockedLinearAmount = _amount - _vestedAmount -_theInstantAmount`.

```
92      function setUserInfo(address _who, address _to, uint256 _amount) external onlyMerkle
            nonReentrant returns(bool status) {

94          require(_who != address(0), 'addr 0');
95          require(_to != address(0), 'addr 0');
96          require(_amount > 0, 'amnt 0');
97          require(usersFlag[_who] == false, '!flag');
98          require(init, 'not init');


101         uint256 _vestedAmount = _amount * VESTED_SHARE / PRECISION;
102         uint256 _theInstantAmount = _amount * LINEAR_DISTRO / PRECISION;
103         uint256 _theLockedLinearAmount = _theInstantAmount;
104         uint256 _tokenPerSec = _theLockedLinearAmount * PRECISION / DISTRIBUTION_PERIOD;

106         UserInfo memory _user = UserInfo({
107             totalAmount: _amount,
108             initAmount: _theInstantAmount,
109             vestedAmount: _vestedAmount,
110             lockedAmount: _theLockedLinearAmount,
111             tokenPerSec: _tokenPerSec,
```

```
112             lastTimestamp: startTimestamp,
113             claimed: _theInstantAmount + _vestedAmount,
114             to: _to
115         });

117         users[_who] = _user;
118         usersFlag[_who] = true;

120         // send out init amount
121         token.safeTransfer(_to, _theInstantAmount);
122         token.approve(ve, 0);
123         token.approve(ve, _vestedAmount);
124         IVotingEscrow(ve).create_lock_for(_vestedAmount, 2 * 364 * 86400 , _who);

126         status = true;
127     }
```

Listing 3.8: `AirdropClaim::setUserInfo()`

Recommendation Improve the above airdrop Logic by computing the right `lockedAmount`.

Status This issue has been fixed in the following commit: `07106e7`.

3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MerkleTree/MerkleTreeTHENFT`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

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` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `PairFees` as an example, the `withdrawStakingFees()` function (see the code snippet below) is

provided to externally call a contract to execute the intended operation. 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 52 or 56) start before effecting the update on the internal state (line 53 or 57), 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.

```
95      function withdrawStakingFees(address recipient) external {
96          require(msg.sender == pair);
97          if (toStake0 > 0){
98              _safeTransfer(token0, recipient, toStake0);
99              toStake0 = 0;
100         }
101         if (toStake1 > 0){
102             _safeTransfer(token1, recipient, toStake1);
103             toStake1 = 0;
104         }
105     }
```

Listing 3.9: `PairFees::withdrawStakingFees()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle or utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Note that `MerkleTree/MerkleTreeTHENFT` contracts share the same issue.

Status This issue has been fixed in the following commit: `07106e7`.

3.7 Killed Gauges Still Eligible For Rewards

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `VoterV2_1`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Thena` protocol creates a gauge for the supported pool and the created gauge can be killed or revived based on the community needs. While reviewing the current gauge-killing logic, we notice a killed gauge is still eligible for rewards!

To elaborate, we show below the related `killGaugeTotally()`. While it properly resets multiple storage states for a killed gauge, it does not remove the associated `supplyIndex` and `weights`, making it still eligible for rewards.

```
440     function killGaugeTotally(address _gauge) external {
441         require(msg.sender == emergencyCouncil, "not emergency council");
442         require(isAlive[_gauge], "gauge already dead");
443         isAlive[_gauge] = false;
444         claimable[_gauge] = 0;
445         address _pool = poolForGauge[_gauge];
446         internal_bribes[_gauge] = address(0);
447         external_bribes[_gauge] = address(0);
448         gauges[_pool] = address(0);
449         poolForGauge[_gauge] = address(0);
450         isGauge[_gauge] = false;
451         isAlive[_gauge] = false;
452         claimable[_gauge] = 0;
453         emit GaugeKilled(_gauge);
454     }
```

Listing 3.10: VoterV2_1::killGaugeTotally()

Recommendation Revise the above logic to properly remove a current gauge.

Status This issue has been fixed in the following commit: 07106e7.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Thena protocol, there is a privileged owner 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.

```
93      ///@notice set distribution address (should be GaugeProxyL2)
94      function setDistribution(address _distribution) external onlyOwner {
95          require(_distribution != address(0), "zero addr");
96          require(_distribution != DISTRIBUTION, "same addr");
97          DISTRIBUTION = _distribution;
98      }
99
100     ///@notice set gauge rewarder address
101     function setGaugeRewarder(address _gaugeRewarder) external onlyOwner {
```

```
102         require(_gaugeRewarder != address(0), "zero addr");
103         require(_gaugeRewarder != gaugeRewarder, "same addr");
104         gaugeRewarder = _gaugeRewarder;
105     }
106
107     ///@notice set extra rewarder pid
108     function setRewarderPid(uint256 _pid) external onlyOwner {
109         require(_pid >= 0, "zero");
110         require(_pid != rewarderPid, "same pid");
111         rewarderPid = _pid;
112     }
```

Listing 3.11: Example Privileged Functions in GaugeV2

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.

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 resolved as the team makes use of the Thena Multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Thena protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from Solidly/Velodrome with a unique AMM. The DEX is compatible with all the standard features as popularized by UniswapV2 with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps, as well as a built-in NFT-based voting mechanism and associated token emissions. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

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

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