STARTER

SMART CONTRACTS REVIEW

zokyo

June 26th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING STARTER

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 1 Critical issue: 1 resolved = 0 points deducted
- 3 High issues: 3 resolved = 0 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 3 Low issues: 2 resolved and 1 acknowledged = - 2 points deducted
- 4 Informational issues: 2 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 2 = 98

TECHNICAL SUMMARY

This document outlines the overall security of the Starter smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Starter smart contract/s codebase for quality, security, and correctness.

Contract Status

There was 1 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Starter team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Starter repository:
Repo: https://github.com/StarterLabsHQ/starterv2

Last commit - a9767ff43e7b2f61946953bd4552389653b0c73c

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- PoolBase.sol
- CorePool.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Starter smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Starter team and the Starter team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Out of range index in getTokenBurnFee() causes function reversion after 14 days	Critical	Resolved
2	Reentrance vulnerability in processRewards() and updateStakeLock() Functions	High	Resolved
3	Attacker Can Grief A Victim Creating Many Little Deposits On Victim's Behalf	High	Resolved
4	Reentrancy Guard Conflict in _processVaultRewards Private Function	High	Resolved
5	vaultRewardsPerWeight Should Be Updated When _processVaultRewards()	Medium	Resolved
6	Rounding Issue While Calculating liquidPercentage	Medium	Resolved
7	Staker Can Extend Expired Locks	Medium	Resolved
8	Missing Deposit ID Emission on New Deposit	Low	Resolved
9	Inadequate Input Validation in setStakingConfig Function	Low	Resolved
10	Centralization Risk Due to Privilege Management of updateLastInvestTimestamp Method	Low	Acknowledged
11	Actual Stakers (With Non Zero lockedUntil) Do Not Receive Yield	Informational	Acknowledged
12	Hardcoded Address in _unstake Function	Informational	Acknowledged
13	Use of require Statements Instead of Modifiers for Access Control	Informational	Resolved
14	Redundant Initialization of Variables	Informational	Resolved

Out of range index in `getTokenBurnFee()` causes function reversion after 14 days

Location: PoolBase.sol

The `getTokenBurnFee()` function in PoolBase.sol attempts to return `burnFees[feeCycle.length]` after 14 days of asset locking, which is out of range for the `burnFees` array. This out-of-bounds access results in the function reverting, causing subsequent `unstake` operations to fail and denying users access to their assets after the maturity period has passed.

```
920 function getTokenBurnFee(address _staker) public view returns
(uint256) {
921     User memory user = users[_staker];
922     for (uint256 i; i < feeCycle.length; i++) {
923     if (
924         (block.timestamp < user.lastUnstakedTimestamp + feeCycle[i])
||
925         block.timestamp < user.lastInvestTimestamp + feeCycle[i]
926     ) {
927         return burnFees[i];
928     }
929     }
930     return burnFees[feeCycle.length];
931 }
```

Recommendation:

To resolve this issue, the function should be modified to handle cases where the index `feeCycle.length` is out of range more gracefully. Instead of accessing `burnFees[feeCycle.length]` directly, consider returning a default burn fee value, or change the index to `feeCycle.length-1`.

Fix: Issue fixed at commit a9767ff4

Reentrance vulnerability in `processRewards()` and `updateStakeLock()` Functions

Location: PoolBase.sol

The `processRewards()` and `updateStakeLock()` functions in the `PoolBase.sol` contract are exposed to reentrancy attacks. This issue surfaced after the client attempted to fix the `_processVaultRewards()` function without adjusting its side effects in other parts of the codebase. As a result, malicious actors could exploit this vulnerability to manipulate the contract state and potentially drain funds or alter stakes.

Recommendation:

Integrate a reentrancy guard: Utilize a reentrancy guard, such as the `nonReentrant` modifier provided by OpenZeppelin that is already imported in the project, to prevent reentrant calls to vulnerable functions. This will effectively lock the contract during execution and prevent reentrancy attacks.

Fix: Issue fixed at commit a9767ff4

Attacker Can Grief A Victim Creating Many Little Deposits On Victim's Behalf

An attacker can create many small deposits on behalf of the victim using just dust amount (using the stakeFor() function) , with each subsequent stake a new deposit struct would be pushed to user.deposits (L563) , the gas cost of staking for the victim would be much cheaper in an L2 .
Therefore , now when the victim unstakes and _unstake is invoked , it calls the _processRewards which in turn calls getRewardLockPeriod() which executes a for loop →

```
for (i = 0; i < user.deposits.length; i++) {
      Deposit storage stakeDeposit = user.deposits[i];
      if (!stakeDeposit.isYield) {
        totalSum =
          totalSum +
          stakeDeposit.tokenAmount *
          (stakeDeposit.lockedUntil - stakeDeposit.lockedFrom);
      }
    }
```

Now since the user.deposits array is way bigger than it should be the for loop would consume lots of gas (or eventually revert) for the victim.

Recommendation:

Have a minimum stake amount which makes the griefing non-profitable

Reentrancy Guard Conflict in _processVaultRewards Private Function

Location: CorePool.sol

The private function `_processVaultRewards` in CorePool.sol is equipped with a `nonReentrant` modifier. This function is called by both internal and external functions that also use the `nonReentrant` modifier, causing a reentrancy guard conflict. Consequently, the function becomes non-executable due to the reentrancy lock it initiates. Adjusting the reentrancy guards or applying a different guard for `_processVaultRewards` is necessary to resolve this issue. This issue potentially leads to loss of funds.

Recommendation:

To resolve the reentrancy guard conflict in `_processVaultRewards`, you can consider the following recommendations:

- **Refactor the Callers:** Ensure that functions calling `_processVaultRewards` manage the reentrancy guard properly. This might involve only applying `nonReentrant` to external or public functions and ensuring that internal functions like `_processVaultRewards` are called in a guarded context.
- **Use Separate Reentrancy Guards:** If necessary, use a different reentrancy guard for `_processVaultRewards` to differentiate it from the guards used by its callers. This can be implemented by creating a new `ReentrancyGuard` instance or using a custom reentrancy mechanism for this function.

By implementing these changes, you can ensure that `_processVaultRewards` executes correctly without encountering reentrancy guard conflicts.

vaultRewardsPerWeight Should Be Updated When _processVaultRewards()

The vaultRewardsPerWeight is incremented whenever there are rewards transferred in to the core pool contract →

```
SafeERC20.safeTransferFrom(IERC20(buidl), msg.sender, address(this),
_rewardsAmount);

    vaultRewardsPerWeight += rewardToWeight(_rewardsAmount,
usersLockingWeight);
```

But it is not decremented when BUIDL is transferred out of the contract in _processVaultRewards() →

```
// transfer fails if pool BUIDL balance is not enough - which is a desired
behavior
    SafeERC20.safeTransfer(IERC20(buidl), _staker, pendingVaultClaim);
```

Recommendation:

Update the vaultRewardsPerWeight inside _processVaultRewards().

Rounding Issue While Calculating liquidPercentage

The liquidPercentage calculated inside _processRewards() (L773) is as follows →

```
Deposit memory newDeposit = Deposit({
        tokenAmount: pendingYield,
        lockedFrom: uint64(block.timestamp),
        lockedUntil: uint64(block.timestamp +
getRewardLockPeriod(_staker)), // staking yield for Reward Lock Period
        weight: depositWeight,
        liquidPercentage: (user.liquidWeight * 100) / user.totalWeight,
//AUDIT-round
        isYield: true
      });
```

Here a new deposit is being created (for rewards) and the liquidPercentage is user.liquidWeight*100 / user.totalWeight , if the totalWeight of the user is > 100*liquidWeight (quite possible if the user has staked a very little amount as liquid and rest non-liquid) then the liquidPercentage would round to 0. Therefore , a user with liquidWeight > 0 will receive 0 yield for his deposit due to the rounding.

Recommendation:

Use higher precision for calculating liquidPercentage.

Staker Can Extend Expired Locks

When a user stakes , his stake gets assigned a weight which is dependent on the length of the stake and the amount staked. A user can also update his stake which updates the deadline (lockedUntil) of his stake , so if the stake was previously 10 days long a user can extend that to 12 days.
A user might have a 5 day long stake which would end on day 5 , assuming he wants to update the stake at day 8 (3 days after the stake ended) the new weight of the stake would also account for the 3 days in between i.e. new weight will be calculated as (_updateStakeLock())→

```
int256 newWeight = (((stakeDeposit.lockedUntil - stakeDeposit.lockedFrom)
*
```

```
    WEIGHT_MULTIPLIER) /
    365 days +
    WEIGHT_MULTIPLIER) * stakeDeposit.tokenAmount;
```

The newWeight calculated will account for the entire 8 + new deadline days which includes the 3 days where the stake was not active and therefore rewards would be accounted for the entire 8 days (till the new deadline) in _processRewards()

Recommendation:

Update the logic to only account weight and rewards when the stake was active.

Missing Deposit ID Emission on New Deposit

Location: PoolBase.sol

When a new deposit is pushed to the `deposits` array using `user.deposits.push(deposit);`, the `depositId`, which corresponds to the index of this new deposit, is not emitted. This ID is necessary for subsequent functions like `_unstake()` and `_updateStakeLock()`, as it refers to the specific position of the deposit in the `deposits` array. The absence of emitting this `depositId` can lead to difficulty in tracking and managing deposits efficiently.

Recommendation:
To address this issue, it is recommended to emit an event whenever a new deposit is pushed to the `deposits` array. This event should include the `depositId` so that users can easily reference the index of their deposits for future operations like unstaking or updating stake locks. The implementation might look like this:

```
event DepositMade(address indexed user, uint256 depositId, uint256
amount);

function deposit(uint256 _amount) external {
    // Assume necessary validations and logic here

    uint256 depositId = user.deposits.length;
    user.deposits.push(deposit);

    emit DepositMade(msg.sender, depositId, _amount);

    // Continue with other logic if necessary
}
```

By emitting this event, users will have a better way to track their deposits and the corresponding IDs, improving overall usability and manageability.

Fix - Issue addressed and fixed in commit e935e74 by adding `emit Deposited(_staker, user.deposits.length - 1, pendingYield);` whenever `user.deposits` is pushed onto.

Inadequate Input Validation in `setStakingConfig` Function

Location: PoolBase.sol

The `setStakingConfig` function in the `PoolBase.sol` contract lacks proper input validation for the `_index` parameter. The `_index` parameter is used to update the values of `feeCycle` and `burnFees` arrays. Given that `feeCycle` is of length 4 and `burnFees` is of length 5, inadequate checks on the `_index` value can lead to out-of-bounds errors or unexpected behavior. This discrepancy in array lengths and the absence of checks for valid _index values could result in setting configurations that are not intended or are erroneous. It might as well be necessary to validate that `minStakeTime`, `_cycle` and `_fee` are within acceptable bounds.

Recommendation:
Add appropriate input validation for the `_index` parameter within the `setStakingConfig` function. The validation should ensure `_index` falls within the valid range for both the `feeCycle` and `burnFees` arrays. By doing this, you ensure that `_index` is checked against the lengths of both arrays, thereby preventing potential errors and maintaining the integrity of the staking configuration process. Additionally, consider discussing with the project owners if the restriction on changing the burn fee after the last cycle is an intentional strategic decision to ensure it aligns with the project's goals.

Fix - The client made an attempt to address the issue in commit e935e74. Mean while the issue in itself is addressed the change proposed introduced significant bugs that undermines the functionality of the codebase.

Centralization Risk Due to Privilege Management of `updateLastInvestTimestamp` Method

Location: PoolBase.sol

The current implementation of the `updateLastInvestTimestamp` function allows not only the contract owner but also any wallet listed in the `presales` mapping to update the last investment timestamp for users. This creates a centralization risk wherein the power to modify critical user-specific parameters. If any of these privileged wallets (owner or presales) are compromised, it could lead to unauthorized and potentially malicious modifications to user data. This design undermines the principles of decentralization and poses a significant concern to the contract and its users.

Recommendation:
To mitigate this centralization risk, it is recommended to implement a multisignature (multisig) scheme for both the owner and even the presales wallets. This approach would require multiple approvals for any privileged action, significantly reducing the likelihood of unauthorized changes even if one of the wallets is compromised. Additionally, the contract can be enhanced to include further checks and balances, such as time-lock mechanisms or community voting for critical updates, to ensure that user-specific parameters are modified transparently and securely.

Fix: The client recognized the risk and mentioned that a multisig scheme is being adopted to substitute the admin's wallet in functions with elevated privileges.

INFORMATIONAL-1 | ACKNOWLEDGED

Rewards Not Compounded For Users with Non-Zero lockedUntil

The pool contract only rewards stakers with yield who staked without a lock period (for these stakers liquidWeight is being assigned , for users who have locked their stake for a duration there liquidWeight is `0`) inside _unstake (L655-L656) . Though this may be a design choice it seems that stakers who actually staked for a duration and locked their stake don't

Recommendation:
The design choice should be acknowledged.

INFORMATIONAL-2 | ACKNOWLEDGED

Hardcoded Address in _unstake Function

Location: PoolBase.sol

The _unstake function in the PoolBase contract contains a hardcoded address for burning tokens. Hardcoding addresses can lead to issues when deploying the contract on different networks or when the address needs to change, as it requires recompilation and redeployment of the contract.

Location: The hardcoded address is found in the _unstake function:

address(`0x000000000000000000000000000000000000dEaD`)

Recommendation:
To avoid hardcoding addresses, declare the burn address as an immutable variable and initialize it via the constructor. This approach allows the address to be set at deployment time and remain unchanged thereafter, ensuring flexibility across different deployments.

Proposed Solution:
1. Add an immutable variable for the burn address.
2. Initialize the burn address in the constructor.

```
address public immutable burnAddress;
constructor(
    address _buidl,
    IPoolFactory _factory,
    address _poolToken,
    uint64 _initBlock,
    uint32 _weight,
    address _starterInfo,
    address _burnAddress
) {
    require(address(_factory) != address(0), "-2");
    require(_poolToken != address(0), "-3");
    require(_initBlock != 0, "-4");
    require(_weight != 0, "-5");
    require(_starterInfo != address(0), "-6");
    require(_burnAddress != address(0), "-21");
    starterInfo = IStarterInfo(_starterInfo);
    buidl = _buidl;
    factory = _factory;
    poolToken = _poolToken;
    lastYieldDistribution = _initBlock;
    weight = _weight;
    burnAddress = _burnAddress;
}
```

Use of require Statements Instead of Modifiers for Access Control

Location: CorePool.sol & PoolBase.sol

The CorePool contract uses require statements for access control in several functions. This approach reduces readability and maintainability. Instead, Solidity modifiers should be used to enforce access control, making the code cleaner and more understandable.

The PoolBase contract lacks the use of modifiers for access control in several functions (stakeFor, setWeight, clearHistory, setConfiguration, setInitialSettings). While require statements are used to enforce certain conditions, they do not provide the same level of readability and maintainability as dedicated access control modifiers.

Affected Code

The following require statements are used for access control:
Factory Owner Check:

```
require(factory.owner() == msg.sender, "-1");
```

Vault Address Check:

```
require(msg.sender == vault, "-3");
```

Pool Existence Check:

```
require(factory.poolExists(msg.sender), "-5");
```

Function stakeFor:

```
function stakeFor(
    address _staker,
    uint256 _amount,
    uint64 _lockUntil
) external override {
    require(_staker!= msg.sender, "-7");
    _stake(_staker, _amount, _lockUntil, false, 0);
    history.push(_staker);
}
```

Function setWeight:

```
function setWeight(uint32 _weight) external override {
    require(msg.sender == address(factory), "-8");
    emit PoolWeightUpdated(msg.sender, weight, _weight);
    weight = _weight;
}
```

Function clearHistory:

```
function clearHistory() external {
    require(msg.sender == factory.owner(), "-18");
    delete history;
}
```

Function setConfiguration:

```
function setConfiguration(
    uint256 _rewardPerWeightMultiplier,
    uint256 _yearStakeWeightMultiplier,
    uint256 _weightMultiplier,
    uint256 _liquidMultiplier
) external {
    require(msg.sender == factory.owner(), "-19");
    emit PoolConfigurationUpdated(...);
   ...
}
```

Function setInitialSettings:

```
function setInitialSettings(address _factory, address _poolToken)
external {
    require(msg.sender == factory.owner(), "-20");
    factory = IPoolFactory(_factory);
    poolToken = _poolToken;
}
```

Recommendation:

Define and use modifiers for these access control checks. This will improve code readability and maintainability.

```
modifier onlyFactoryOwner() {
    require(factory.owner() == msg.sender, "-1");
    _;
}
modifier onlyVault() {
    require(msg.sender == vault, "-3");
    _;
}
modifier onlyExistingPool() {
    require(factory.poolExists(msg.sender), "-5");
    _;
}
modifier onlyFactory() {
    require(msg.sender == address(factory), "Caller is not the factory");
    _;
}
```

Redundant Initialization of Variables

Location: PoolBase.sol

The code contains redundant initializations of variables to zero. In Solidity, the default value for uninitialized variables is zero, making these initializations unnecessary. Removing these redundant initializations can improve code readability and slightly reduce gas costs.

Location:
The for loop in the getTokenBurnFee function:
```
for (uint256 i = 0; i < feeCycle.length; i++) {
```
The totalSum variable in the getRewardLockPeriod function:
```
uint256 totalSum = 0;
```

Recommendation:
Remove the redundant initializations of variables to zero. This will make the code cleaner and more efficient.

	PoolBase.sol CorePool.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Starter team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Starter team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo