SMART CONTRACTS REVIEW

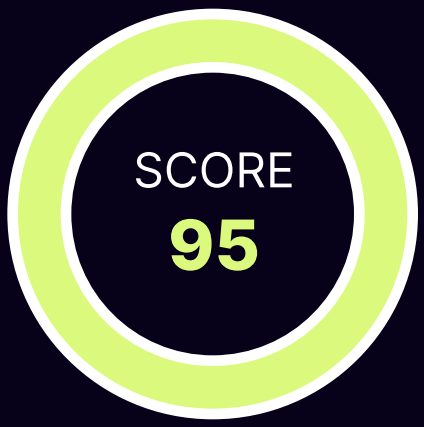
zokyo

March 25th 2024 | v. 1.0

Security Audit Score

PASS

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



TECHNICAL SUMMARY

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

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

Contract Status

There were 0 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 Planar 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 Planar repository:
Repo: https://github.com/planarfinance/audit-quotation

Last commit - 7786bc1a071e074196c1d7c50a94560671c52047

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

- ./XPlaneToken.sol
- ./UniswapV2ERC20.sol
- ./Blast.sol
- ./Refund.sol
- ./HyperPoolFactory.sol
- ./BalanceFetcher.sol
- ./PlanarLiquidityPool.sol
- ./libraries/VestingWallet.sol
- ./libraries/UniswapV2Library.sol
- ./libraries/VestingWallet2.sol
- ./Multicall.sol
- ./Presale.sol
- ./Launchpad.sol
- ./PlanarLiquidityPoolRouter.sol
- ./DividendsV2.sol
- ./ProtocolEarnings.sol
- ./PositionHelper.sol
- ./LpNFTPool.sol
- ./PlanarLiquidityPoolFactory.sol
- ./YieldBooster.sol
- ./LpNFTPoolFactory.sol
- ./PlaneToken.sol
- ./Multicall2.sol
- ./HyperPool.sol
- ./UniswapInterfaceMulticall.sol
- ./PlanarMaster.sol
- ./FairAuction.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 Planar 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 Planar team and the Planar 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	Incomplete reward transfer due to balance shortfall	Medium	Acknowledged
2	Potential Gas Limit Issues in Contract Functions Due to High Number of User Positions	Medium	Acknowledged
3	Nonce Verification Missing in permit Function	Medium	Unresolved
4	updateBeneficiary function lacks critical validation checks	Medium	Unresolved
5	Incorrect Implementation of revoke Function in VestingWallet Contract	Medium	Unresolved
6	Lack of Epoch End Block Consideration in pendingRewards Function	Medium	Unresolved
7	Lack of Minimum Fee Validation in setDefaultFee Function	Low	Unresolved
8	Lack of accountHash check in isContract function	Low	Unresolved
9	Lack of Minimum Waiting Period for pendingRewards Function Calls in LpNFTPool Contract	Low	Unresolved
10	Potential for Zero Allocation in updateAllocations Function Without Validation	Low	Unresolved
11	Possibility of zero amount addition to staking positions	Low	Unresolved
12	Use of single step ownership transfer	Low	Unresolved
14	Addressing the Approve Race Condition with safeIncreaseAllowance and safeDecreaseAllowance in Smart Contracts	Low	Unresolved
15	Ensuring Secure Token Transfers in Smart Contracts with OpenZeppelin's SafeTransfer	Low	Unresolved

#	Title	Risk	Status
16	Centralized Control and Lack of Transparency	Low	Unresolved
17	Optimizing Gas Usage in Smart Contracts with Caching Techniques	Informational	Unresolved
18	Optimizing Gas Usage in Smart Contracts by Leveraging Default Boolean Values in Solidity	Informational	Unresolved
19	Inefficient ETH Transfer Method	Informational	Unresolved
20	Cache the _tokens length	Informational	Unresolved
21	Potential Security Risk in using emergencyWithdrawFunds Function	Informational	Unresolved
22	Lack of Ownership Check in addRewards Function Contradicts Documentation	Informational	Resolved

Incomplete reward transfer due to balance shortfall

The _safeRewardsTransfer function within the LpNFTPool contract is designed to transfer reward tokens safely to users, guarding against rounding errors or situations where the contract's token balance is insufficient to cover all pending rewards. However, the current implementation does not account for scenarios where the contract's balance of planeToken is lower than the amount intended for transfer. In such cases, the function reduces the transfer amount to the available balance, potentially leaving a portion of the owed rewards unclaimed. This situation can lead to users not receiving the full rewards they are entitled to.

Recommendation:

Consider implementing a mechanism that tracks the shortfall in reward payments.Introduce a mapping to track the leftover rewards for each user. When the transfer amount is adjusted due to a balance shortfall, record the difference between the intended and actual transferred amounts in the user's shortfall record.

Note#2: `_safeRewardsTransfer` is an internal function which is called as part of a chain of events in `harvestPosition` or `harvestPositionTo` external functions. These functions call `_updatePool` to update balance of the LpNFT Pool before `_safeRewardsTransfer`. Since technically, the user gets particular percentage of the rewards owed by the LpNFT from the PlanerMaster contract, practically, it is not possible for the LpNFT Pool to have less balance than a user is owing. The capping in `_safeRewardsTransfer` function is merely a safety valve. We're keeping the code as is for this.

Potential Gas Limit Issues in Contract Functions Due to High Number of User Positions

Description:
The LpNFTpool contains several functions (`mergePositions`, `harvestAllPositions`, `withdrawFromAllPositions`, etc.) that iterate over user's staking positions or token IDs. In scenarios where a user has a large number of positions or tokens, these functions are prone to hitting the block gas limit, resulting in transactions that fail to execute. This not only hinders the user's ability to interact with the contract as intended but also may lead to situations where users are unable to consolidate, harvest, or withdraw from their positions without resorting to less efficient methods.

Scenario:
Consider a user who, over time, has accumulated a significantly large number of staking positions or NFTs within the system. If this user attempts to use a function like mergePositions to consolidate these positions into one or `harvestAllPositions` to claim rewards across all positions, the transaction could require an amount of gas that exceeds the block gas limit, making it impossible to execute the transaction successfully. This scenario can lead to frustration for users and could potentially lock users out of their assets or rewards under certain conditions.

Recommendation:

1. **Implement Batch Processing:** Consider adding functionality that allows users to specify subsets of positions or tokens for processing within a single transaction. By operating on smaller batches, users can avoid the gas limit issue while still being able to perform the desired actions on all their positions over several transactions.

Note#3: We don't have `harvestAllPositions` or `withdrawFromAllPositions` in LpNFT Pool contract. We shall leave `mergePositions` as it is and maybe consider batch processing mergers on frontend.

Nonce Verification Missing in permit Function

Source: ./UniswapV2ERC20.sol
Description:
The permit function does not include a check to verify that the nonce used in the signature is correct and has not been used before. This omission can lead to potential signature replay attacks, where an attacker could reuse a valid signature to perform unauthorized operations.

Recommendation:
It is recommended to include a nonce verification step in the permit function to ensure that each nonce is used only once. This can be achieved by checking the current nonce for the owner against the provided nonce in the function call and then incrementing the nonce after a successful verification and approval.

```
    mapping (address => uint)                    public nonces;
function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r,
bytes32 s, uint nonce) external {
    require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
    require(nonce == nonces[owner], 'UniswapV2: INVALID_NONCE'); // Nonce verification
    bytes32 digest = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonce, deadline))
        )
    );
    address recoveredAddress = ecrecover(digest, v, r, s);
    require(recoveredAddress != address(0) && recoveredAddress == owner, 'UniswapV2:
INVALID_SIGNATURE');
    _approve(owner, spender, value);
    nonces[owner] += 1; // Increment nonce after successful approval
}
```

updateBeneficiary function lacks critical validation checks

Source: ./libraries/VestingWallet.sol , ./libraries/VestingWallet2.sol
Description:
The updateBeneficiary function in the VestingWallet contract allows the contract owner to update the share of a beneficiary. However, this function lacks several critical validations that are necessary to ensure the integrity and security of the contract's operations. Specifically, the function does not validate:

- The contract has not been bootstrapped (if applicable to the contract's logic).
- The new share amount is different from the current share amount for the beneficiary.
- The beneficiary has not performed certain actions that should lock their share from being updated (e.g., transferring USD in some contexts).
- The new share amount is greater than zero.
- The beneficiary has been previously added and is not being set for the first time through this function.

Recommendation:
```
function updateBeneficiary(address wallet, uint256 newShare) external onlyOwner {

  // Assuming `bootstrapped` is a state variable indicating if the contract has been bootstrapped
  require(!bootstrapped, 'Cannot update beneficiary as contract has been bootstrapped');
  require(beneficiariesShare[wallet] != newShare, 'New share cannot be the same as old share');
  // Assuming there's a way to check if a beneficiary has transferred USD or any similar condition
  require(!hasTransferredUSD[wallet], 'Beneficiary should have not transferred USD');
  require(newShare > 0, 'Share cannot be smaller or equal to 0');
  require(beneficiariesShare[wallet] != 0 || _beneficiariesWallet.contains(wallet), 'Beneficiary has not been added');
```

```
  _release();

  totalShare = totalShare.sub(beneficiariesShare[wallet]).add(newShare);
  require(totalShare <= MAX_TOTAL_SHARE, "Allocation too high");
  beneficiariesShare[wallet] = newShare;
  if (newShare == 0) _beneficiariesWallet.remove(wallet);
  else _beneficiariesWallet.add(wallet);
}
```

Incorrect Implementation of revoke Function in VestingWallet Contract

Source: ./libraries/VestingWallet.sol , ./libraries/VestingWallet2.sol
Description:
The revoke function in the VestingWallet contract is implemented to transfer all remaining tokens in the contract back to the owner upon revocation. This implementation does not account for tokens that have already been vested to beneficiaries, potentially allowing the owner to reclaim tokens that are rightfully owned by the beneficiaries. This behavior deviates from the expected functionality where only unvested tokens should be returned to the owner, leaving vested tokens accessible to the beneficiaries.

Recommendation:
To align with the expected behavior of a vesting contract, the revoke function should be modified to calculate the amount of unvested tokens at the time of revocation and only transfer this amount back to the owner. The calculation should consider the total amount of tokens, the amount already released to beneficiaries, and the vesting schedule to determine the unvested portion. Additionally, it is recommended to introduce a state variable to track whether the contract has been revoked and prevent further token releases post-revocation. This approach ensures that vested tokens remain accessible to beneficiaries, while only unvested tokens are returned to the owner, respecting the rights of all parties involved.

```
 bool public revoked = false; // State variable to track revocation status

   event Revoked(); // Event to indicate revocation

   // Modified revoke function
   function revoke() external onlyOwner {
      require(!revoked, "Contract already revoked");
      require(!nonRevocable, "Revoke not allowed");

      uint256 vestedAmount = releasable(); // Calculate the vested amount
      uint256 unvestedAmount = planeToken.balanceOf(address(this)).sub(vestedAmount); //
Calculate unvested amount
```

```
        planeToken.transfer(owner(), unvestedAmount); // Transfer unvested tokens back to
owner

        revoked = true; // Mark contract as revoked
        emit Revoked();
    }
```

Note#4 : There could be multiple VestingWallet2 deployments in our project, each one have it's own settings depending on the terms between Planar and the Beneficiaries. In certain cases, we expect certain behaviour (say, not to dump tokens in the market or to keep the marketing going) from the Beneficiaries to keep their Vesting going, which is revocable on breach to protect the Protocol from adversary Beneficiaries. In the case of Beneficiaries who set an irrevocable Vesting term with us, `nonRevocable` flag is set and the owner can no longer withdraw tokens from the contract. So the contract is made flexible on purpose.

Lack of Epoch End Block Consideration in pendingRewards Function

Source: LpNFTPool.sol
Description:
The pendingRewards function calculates the pending rewards for a staking position without considering the end of the epoch block (endOfEpochBlock). This omission can lead to the calculation of rewards beyond the intended reward distribution period, potentially resulting in over-distribution of rewards. The function recalculates accRewardsPerShare based on the current block timestamp, last reward time, reserve, and pool emission rate without checking if the current period is within the active epoch defined for reward distribution.

Recommendation:
To ensure that rewards are accurately calculated and distributed within the designated epochs, it is recommended to incorporate a check for the endOfEpochBlock within the pendingRewards function. This check should ensure that rewards are only calculated up to the end of the active epoch. If the current block number exceeds the endOfEpochBlock, the calculation should use the endOfEpochBlock as the upper limit for the rewards period. This modification will prevent the potential over-distribution of rewards and ensure that the reward distribution aligns with the intended epochs set by the contract owner or protocol.

```
function pendingRewards(uint256 tokenId) external view returns (uint256) {
    StakingPosition storage position = _stakingPositions[tokenId];

uint256 accRewardsPerShare = _accRewardsPerShare;
    (,,uint256 lastRewardTime, uint256 reserve, uint256 poolEmissionRate) =
master.getPoolInfo(address(this));
    uint256 endOfEpochBlock = master.getEndOfEpochBlock(address(this)); // Assume this
function exists or a similar mechanism to retrieve endOfEpochBlock

    // Ensure rewards are calculated within the epoch
    uint256 effectiveLastRewardBlock = lastRewardTime < endOfEpochBlock ? lastRewardTime :
endOfEpochBlock;
```

```
    if ((reserve > 0 || _currentBlockTimestamp() > effectiveLastRewardBlock) &&
_lpSupplyWithMultiplier > 0) {
        uint256 duration = _currentBlockTimestamp().sub(effectiveLastRewardBlock);
        uint256 tokenRewards = duration.mul(poolEmissionRate).add(reserve);
        accRewardsPerShare =
accRewardsPerShare.add(tokenRewards.mul(1e18).div(_lpSupplyWithMultiplier));
    }

    return
position.amountWithMultiplier.mul(accRewardsPerShare).div(1e18).sub(position.rewardDebt)
        .add(position.pendingXPlaneRewards).add(position.pendingPlaneRewards);
}
```

LOW-1 | UNRESOLVED

Lack of Minimum Fee Validation in setDefaultFee Function

Source: ./HyperPoolFactory.sol

Description:
The setDefaultFee function in the HyperPoolFactory contract allows the contract owner to set a default fee for hyper pools. However, the function only checks that the new fee does not exceed the MAX_DEFAULT_FEE but does not validate a minimum fee threshold. This oversight could potentially allow setting the default fee to 0, which might not align with the protocol's economic model or intentions,

Recommendation:
uint256 public constant MIN_DEFAULT_FEE = 1; // Example minimum fee (0.01%), adjust as necessary

```
function setDefaultFee(uint256 newFee) external onlyOwner {
    require(newFee >= MIN_DEFAULT_FEE && newFee <= MAX_DEFAULT_FEE, "Fee must be between min and max");
    defaultFee = newFee;
    emit SetDefaultFee(newFee);
}
```

Note#5: HyperPool is a permission-less protocol to have other projects to build their Staking and Reward distribution strategies. `defaultFee` is kept to 0 in the beginning to attract protocols and users with no cost and more options.

LOW-2 | UNRESOLVED

Lack of accountHash check in isContract function

Source: BalanceFetcher.sol

Description:
The isContract function, as currently implemented, checks if an address contains code by examining the size of the code at that address. This method, however, can yield misleading results due to specific Ethereum Virtual Machine (EVM) behaviors. Specifically:

- During contract creation, the contract's code is not yet stored at the address, resulting in isContract returning false even for addresses that will imminently hold contract code.
- If a contract was previously deployed at an address and then self-destructed, the code size check would return false, disregarding the historical presence of a contract.
- Addresses where contracts will be deployed in the future but currently hold no code will also result in a false return value.

Recommendation:
To address these limitations and improve the reliability of contract detection, it's recommended to use the extcodehash function instead. This approach considers the hash of the code at an address, which provides a more accurate indication of whether the address is a contract, including for contracts in creation and addresses of destroyed contracts.

```
/**
 * @dev Returns true if `account` is a contract.
 *
 * [IMPORTANT]
 * ====
 * It is unsafe to assume that an address for which this function returns
 * false is an externally-owned account (EOA) and not a contract.
 *
 * Among others, `isContract` will return false for the following
 * types of addresses:
 *
```

```
 *  - an externally-owned account
 *  - a contract in construction
 *  - an address where a contract will be created
 *  - an address where a contract lived, but was destroyed
 * ====
 */
function isContract(address account) internal view returns (bool) {
    // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
    // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is
returned
    // for accounts without code, i.e. `keccak256('')`
    bytes32 codehash;
    bytes32 accountHash =
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
    // solhint-disable-next-line no-inline-assembly
    assembly {
        codehash := extcodehash(account)
    }
    return (codehash != accountHash && codehash != 0x0);
}

```
```

Note#6 : It's a read-only contract intended for frontend only. As long as it returns data of a limited static list of contracts, we think it's okay to keep it as is.
```

Lack of Minimum Waiting Period for pendingRewards Function Calls in LpNFTPool Contract

Source: LpNFTPool.sol , HyperPool.sol

Description:
The pendingRewards function in the LpNFTPool contract calculates the pending rewards for a staking position based on the current block timestamp and the last reward time. However, it lacks a mechanism to enforce a minimum waiting period between successive calls. This could potentially allow users to call this function excessively without any temporal restrictions

Recommendation:
To mitigate this issue, it is recommended to introduce a minimum waiting period for calling the pendingRewards function. This can be achieved by maintaining a mapping to track the last time the pendingRewards function was called for each tokenId and requiring that a certain amount of time (e.g., one hour) has passed before the function can be called again for the same tokenId. This approach ensures that there is a controlled frequency of reward calculations,

```
// SPDX-License-Identifier: MIT
pragma solidity =0.7.6;

// Import statements...

contract LpNFTPool is ReentrancyGuard, ILpNFTPool, ERC721("Planar Locked Position NFT", "lpNFT"), Blast {
  // Existing variable and function declarations...

  mapping(uint256 ⇒ uint256) private lastRewardCalculationTime; // Tracks the last time rewards were calculated for each tokenId
  uint256 private constant SECONDS_PER_HOUR = 3600; // Define the minimum waiting period

  // Existing constructor and function definitions...
```

```
  /**
   * @dev Returns pending rewards for a position with a minimum waiting period enforcement
   */
  function pendingRewards(uint256 tokenId) external view returns (uint256) {
    require(_currentBlockTimestamp() > lastRewardCalculationTime[tokenId] +
SECONDS_PER_HOUR, "Wait for the minimum period before recalculating");

    StakingPosition storage position = _stakingPositions[tokenId];
    uint256 accRewardsPerShare = _accRewardsPerShare;
    (,,uint256 lastRewardTime, uint256 reserve, uint256 poolEmissionRate) =
master.getPoolInfo(address(this));

    // Existing logic to recompute accRewardsPerShare if not up to date...

    return
position.amountWithMultiplier.mul(accRewardsPerShare).div(1e18).sub(position.rewardDebt)
      .add(position.pendingXPlaneRewards).add(position.pendingPlaneRewards);
  }

  // Additional or modified functions as necessary...
}

```
```
```

Potential for Zero Allocation in updateAllocations Function Without Validation

Source: PlaneToken.sol

Description:
The updateAllocations function allows the owner to update the allocations for farming and legacy holders. However, the function lacks validation to prevent either farmingAllocation_ or legacyAllocation_ from being set to zero. This oversight could lead to scenarios where one or both allocations are unintentionally set to zero, potentially disrupting the intended distribution of rewards and affecting the incentive mechanisms of the platform.

Recommendation:
To mitigate these issues and enhance the function's robustness, it is recommended to implement the following changes:

- Validation Checks: Introduce validation checks to ensure that neither farmingAllocation_ nor legacyAllocation_ can be set to zero unless explicitly intended as part of the platform's strategy. This could involve requiring that each allocation is greater than a certain minimum threshold.
- Explicit Zero Allocation Handling: If there are valid scenarios where an allocation might need to be set to zero, implement explicit handling and documentation for these cases to ensure that such actions are deliberate and well-understood.
- Adjustable Minimum Thresholds: Consider implementing adjustable minimum thresholds for allocations that can be modified by the owner or through governance mechanisms. This allows for flexibility while still preventing accidental zero allocations.

```
function updateAllocations(uint256 farmingAllocation_, uint256 legacyAllocation_) external onlyOwner {
    // apply emissions before changes
    emitAllocations();

 // total sum of allocations can't be > 100%
    uint256 totalAllocationsSet = farmingAllocation_.add(legacyAllocation_);
    require(totalAllocationsSet <= 100, "updateAllocations: total allocation is too high");
```

```
    // Ensure neither allocation is set to zero unintentionally
    require(farmingAllocation_ > 0 && legacyAllocation_ > 0, "updateAllocations: allocations must be greater than zero");

    // set new allocations
    farmingAllocation = farmingAllocation_;
    legacyAllocation = legacyAllocation_;

    emit UpdateAllocations(farmingAllocation_, legacyAllocation_, treasuryAllocation());
}
```

LOW-5 | UNRESOLVED

Possibility of zero amount addition to staking positions

The addToPosition function within the LpNFTPool contract allows for the addition of positions with a zero amount due to the absence of a check after calling _transferSupportingFeeOnTransfer. This function is intended to add a specified amount to an existing staking position, provided the amount is greater than zero. However, if the token being staked has a transfer fee, the actual amount added to the staking position could be reduced to zero by the transfer mechanism, bypassing the initial non-zero check.

Recommendation:
Consider moving the amount check after the token has been transferred.

LOW-6 | UNRESOLVED

Use of single step ownership transfer

The contracts in the codebase use the Ownable contract which allows changing the owner address. However, this contract does not implement a 2-step-process for transferring ownership. If the admin's address is set incorrectly, this could potentially result in critical functionalities becoming locked.

Recommendation:
Consider implementing a two-step pattern. Utilize OpenZeppelin's Ownable contract.

LOW-7 | UNRESOLVED

Inaccuracy in the decimal assumption

The Presale contract has a hardcoded constant, MIN_TOTAL_RAISED_FOR_MAX_PLANE, which assumes the usage of a stablecoin (USDC) with 6 decimals. This design choice introduces a vulnerability when the contract is deployed on blockchain networks where the stablecoin or sale token has a different decimal configuration. The discrepancy in decimal handling can lead to significant miscalculations in the token sale mechanics, potentially allowing tokens to be sold for far less than their intended price.

Recommendation:
Consider dynamically adjusting the MIN_TOTAL_RAISED_FOR_MAX_PLANE value based on the decimal configuration of the sale token at the time of contract deployment.

Addressing the Approve Race Condition with `safeIncreaseAllowance` and `safeDecreaseAllowance` in Smart Contracts

Location:XplaneToken, line 252

The provided `approveUsage` function is designed to set an approval amount for a specific usage by an external contract or entity. However, it directly sets the approval amount to a new value without considering the potential risks associated with approval race conditions. This vulnerability is inherent in the ERC-20 token standard's approve function and can similarly affect custom approval mechanisms like the one described.

The Race Condition Explained:

The race condition occurs when a user attempts to adjust an already granted allowance. Suppose a user wants to change an allowance from 100 to 200 tokens. They send a transaction to update this allowance, but just before this transaction is confirmed, a malicious actor uses the original allowance of 100 tokens. If the update transaction then goes through, the malicious actor could potentially spend up to 300 tokens (100 before the allowance change and 200 after), exploiting the gap between these transactions.

This issue arises because the `approve` function (or similar custom functions, as in the `approveUsage` example) replaces the existing allowance with a new value, without locking the allowance during the update process. This creates a window where the allowances can be misused.

Solution with `safeIncreaseAllowance` and `safeDecreaseAllowance`:

A safer approach involves using two functions: `safeIncreaseAllowance` and `safeDecreaseAllowance`,

- `safeIncreaseAllowance` safely increases the allowance by a certain amount, ensuring that the increase is additive to the current allowance, reducing the risk of the race condition.
- `safeDecreaseAllowance` safely decreases the allowance by a specified amount, ensuring that the decrease cannot be exploited to spend more than intended.

Ensuring Secure Token Transfers in Smart Contracts with OpenZeppelin's SafeTransfer

Location: Presale.sol, line 322, 324
The use of `.transfer()` for token transactions, while common, can pose significant security risks, particularly due to its inability to handle token transfer failures gracefully. This limitation becomes apparent in the context of ERC-20 tokens, where transfers might fail for various reasons, such as lack of allowance or insufficient balance. To mitigate these risks and ensure robust error handling, it's advisable to adopt OpenZeppelin's `safeTransfer` function from its ERC-20 library.

The provided `_safeClaimTransfer` function attempts to securely transfer PROJECT_TOKENs by checking the contract's balance and transferring the lesser of the desired amount or the available balance. However, it relies on the basic `transfer` method, which returns a boolean value to indicate success or failure. This approach requires explicit, manual checking of the transfer outcome, as demonstrated by the `require` statement to revert the transaction if `transferSuccess` is false.

Recommendation:
To enhance security and reliability, replace the basic `transfer` call with OpenZeppelin's `safeTransfer` method.

Centralized Control and Lack of Transparency

Description:
The presale contract contains functions `(emergencyWithdrawFunds, burnUnsoldTokens, setUsersDiscount)` that provide the owner with significant control over the contract's operation and token distribution. While necessary for administration and emergency situations, these functions could be misused or lead to centralization concerns.

Scenario:
The contract owner could withdraw tokens or funds unexpectedly, change discount rates arbitrarily, or burn unsold tokens in a manner that affects the presale outcome. Such actions could undermine trust in the presale process and potentially harm participants.

Recommendation:
Implement a multi-signature wallet or a decentralized autonomous organization (DAO) mechanism for critical functions, requiring consensus among multiple stakeholders.

Optimizing Gas Usage in Smart Contracts with Caching Techniques

Locations:
BalanceFetcher.sol, line 22
DividendsV2.sol, line 278
FairAuction.sol, line 288
Multicall.sol, line 14
Multicall2.sol, line 22, line 57
PlanarLiquidityPoolRouter.sol, line 228, 242
Presale.sol, line 270
Refund.sol, line 41
UniswapInterfaceMulticall.sol, line 30
Uniswapv2library.sol, line 48

When executing smart contracts, especially on networks where transaction costs can be significant, it's crucial to optimize for gas efficiency. A common inefficiency arises during iterations, where accessing the length property of an array multiple times within a loop can lead to unnecessary gas consumption. This occurs because each retrieval of the array's length is a state access that costs gas.

In the provided example, the function `getBalances` iterates over an array of token addresses to fetch the balance of each token for a given owner. The inefficiency stems from repeatedly accessing `_tokens.length` within the loop condition.

Solution:
A more gas-efficient approach involves caching the array's length outside of the loop. By storing the length in a local variable before entering the loop, the contract only pays for a single state access, regardless of the number of iterations. This modification can significantly reduce gas costs for transactions, particularly for functions that are called frequently or iterate over large arrays.

```
function getBalances(address _owner, address[] calldata _tokens) external view returns
(uint256[] memory balances) {
    uint256 length = _tokens.length; // Cache the length
    balances = new uint256[](length);
```

```
    for(uint256 i = 0; i < length; i++) { // Use cached length
        if(!isContract(_tokens[i])) {
            continue;
        }
        try IERC20(_tokens[i]).balanceOf(_owner) returns(uint256 balance) {
            balances[i] = balance;
        } catch {}
    }
}
```

This technique ensures the function is optimized for gas efficiency without compromising functionality. It's a simple yet effective way to reduce the gas cost of loops that access array lengths or similar repeatable state reads.

Optimizing Gas Usage in Smart Contracts by Leveraging Default Boolean Values in Solidity

Location: Presale.sol, line 319

In Solidity, the initialization of variables in smart contracts can have a direct impact on gas consumption, especially when these variables are set to default values. The boolean type in Solidity is initialized to `false` by default. Recognizing and utilizing this default behavior can lead to more efficient gas usage in your smart contract code.

In the context of the `_safeClaimTransfer` function provided, the variable `transferSuccess` is explicitly initialized to `false`:

```
bool transferSuccess = false;
```

While this is a clear and explicit way to define the variable's initial state, it is technically unnecessary and results in slightly higher gas costs during contract execution. This is because the Solidity compiler allocates storage for the variable and sets its initial value, even though it automatically defaults to `false`.

Recommendation:

To optimize gas consumption, you can omit the explicit initialization of `transferSuccess` and rely on Solidity's default value assignment. Here's how the optimized version of the function could look:

```
function _safeClaimTransfer(address to, uint256 amount) internal {
    uint256 balance = PROJECT_TOKEN.balanceOf(address(this));
    bool transferSuccess; // No need to initialize to false
    ...
}
```

Inefficient ETH Transfer Method

Source: ./Refund.sol

Description:

The _safeTransferETH function in the smart contract uses a less efficient method for transferring ETH, which can lead to excessive gas usage.

Recommendation:

```
function _safeTransferETH(address to, uint256 amount) internal {
    bool success;

    /// @solidity memory-safe-assembly
    assembly {
        // Transfer the ETH and store if it succeeded or not.
        success := call(gas(), to, amount, 0, 0, 0, 0)
    }

    require(success, "ETH_TRANSFER_FAILED");
}
```

INFORMATIONAL-4 | UNRESOLVED

Cache the _tokens length

Source: BalanceFetcher.sol

Cache the value of _tokens.length to save gas as its a Calldata not memory

```
 for(uint256 i = 0; i < _tokens.length; i++) {
            if(!isContract(_tokens[i])) {
                continue;
            }
            try IERC20(_tokens[i]).balanceOf(_owner) returns(uint256 balance) {
                balances[i] = balance;
            } catch {}
        }
```

INFORMATIONAL-5 | UNRESOLVED

Potential Security Risk in using emergencyWithdrawFunds Function

Description:
The emergencyWithdrawFunds function present in most contracts of the project allows the contract owner to withdraw funds (both ETH and ERC20 tokens) to their address. While this function includes an onlyOwner modifier ensuring that only the contract owner can call it, there is a potential risk of misuse leading to a rug pull scenario if not properly safeguarded.

Recommendation:

- Implement a multi-signature requirement for calling this function.
- Introduce a time lock or delay mechanism to provide transparency before funds are withdrawn.
- Clearly document the conditions under which this function can be used and communicate them to the community.

Lack of Ownership Check in `addRewards` Function Contradicts Documentation

Location: Hyperpool.sol, line `402`

The `addRewards` function, as per the provided code snippet, is intended for exclusive use by the owner of a Hyper Pool to add reward tokens to the pool. This functionality is crucial for managing the pool's reward distribution and aligns with the governance model outlined in the project documentation. The documentation explicitly states that "Only the owner of a Hyper Pool can fill it with reward tokens," emphasizing the need for ownership verification within the function to adhere to the specified access control policy.

However, the current implementation of the `addRewards` function lacks any form of ownership check or validation.

Expected Behavior:

The `addRewards` function should incorporate an ownership check to ensure that only the owner of the Hyper Pool can invoke it. This is critical for maintaining the integrity and security of the pool's reward distribution mechanism, as outlined in the project's documentation.

Observed Misbehavior:

The function currently allows any external entity to add rewards to the pool, disregarding the ownership requirement detailed in the documentation.

Note#The addRewards function is meant to be an external function without access restriction, irrespective of the documentation (which was an error). Anybody can add Reward Tokens specified by the Owner. The protocol can charge a fee (Maximum 1%) for non-Owner deposits. You may find this exemption logic in HyperPoolFactory.sol

	./XPlaneToken.sol **./UniswapV2ERC20.sol** **./Blast.sol** **./Refund.sol** **./HyperPoolFactory.sol** **./BalanceFetcher.sol** **./PlanarLiquidityPool.sol** **./libraries/VestingWallet.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

	./libraries/UniswapV2Library.sol **./libraries/VestingWallet2.sol** **./Multicall.sol** **./Presale.sol** **./Launchpad.sol** **./PlanarLiquidityPoolRouter.sol** **./DividendsV2.sol** **./ProtocolEarnings.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

	./PositionHelper.sol **./LpNFTPool.sol** **./PlanarLiquidityPoolFactory.sol** **./YieldBooster.sol** **./LpNFTPoolFactory.sol** **./PlaneToken.sol** **./Multicall2.sol** **./HyperPool.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

	./UniswapInterfaceMulticall.sol ./PlanarMaster.sol ./FairAuction.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 Planar 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 Planar team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo