Cyberscope

A TAC Security Company

Audit Report
Five Pillars

September 2025

Repositories :

https://github.com/fivepillarstoken/5PT-Token

https://github.com/fivepillarstoken/InvestmentManager

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Overview

The **InvestmentManager** contract is a core component of the **Five Pillars ecosystem**, designed to manage investments, distribute rewards, and facilitate participation in a tiered pool system.

It integrates **OpenZeppelin's SafeERC20** for secure token handling and **Ownable2Step** for role-based access control.

The contract supports:

- Deposits
- Reward calculations (daily, referral, and pool-based)
- Fee collection
- Whitelist management for nine pools (0–8)

Users progress to higher pools by accumulating personal deposits and referrals, with the system incentivizing growth to unlock higher reward potential. Importantly, users **earn rewards from all pools they are active in**, not just the latest or highest-tier pool, maximizing their reward opportunities.

Below is a detailed breakdown of the flow of key operations, focusing on how user actions (e.g., deposits, claims) propagate through the system.

1. Deposit Operation

The **deposit** function allows users to invest Five Pillars tokens (via `IFivePillarsToken`) and participate in the ecosystem.

Eligibility Check

- Ensures current timestamp > `startTimestamp` and at least **4 hours (`depositDelay`)** have passed since the last deposit.
- For first-time deposits: minimum **1 token (10^18 wei)**.
- `referer` address:
    - Must not be the investor themselves.
    - Must not be zero for subsequent deposits unless previously set.

Token Transfer and Fee Calculation

- **Investor Portion**:
  `(amount * (BASIS_POINTS - depositFeeInBp)) / BASIS_POINTS`
  Burned via `fivePillarsToken.burnFrom`.
- **Fee Portion**:
  `(amount * depositFeeInBp) / BASIS_POINTS`
  Transferred to the contract for ETH swapping.

Referer Updates

- For first deposits:
    - Set `InvestorInfo.referer`.
    - Validate no circular chains.
- **Direct referer**:
    - Increment `directRefsDeposit` by investor portion.
    - If first deposit, increment `directRefsCount`.
- **Indirect referers (up to 9 levels)**:
    - Increment `downlineRefsDeposit`.
    - If first deposit, increment `downlineRefsCount`.

Pool and Reward Updates

- Update `totalDeposit` and global `totalDepositAmount`.
- For first deposits:
    - Add investor to `_investors`.
    - Adjust `onlyWhitelistedInvestorsCount` if in pools 7 or 8.
- **Daily Rewards**:
  `(totalDeposit * 0.3%) * endedRounds`
  Update `lastDailyReward` and add to `accumulatedReward`.
- **Pool Eligibility (0–6)**:
    - Check criteria (`personalInvestRequired`, `directRefsRequired`, `totalDirectInvestRequired`).
    - If eligible:
        - Set `isInvestorInPool = true`.
        - Increment `participantsCount`.
        - Update `poolRewardPerInvestorPaid`.
        - Activate pool if inactive.
- **Reward Allocation**:
    - Pools 0–4: **17.5%**
    - Pools 5–6: **10%**
    - Pools 7–8: **20%**
- Investors earn from **all qualified pools**.
- Update `lastUpdatePoolRewardTimestamp`.

Event Emission

- `Deposit(investorAddress, referer, toInvestor)`

2. Claim Rewards Operation

The **claimReward** function allows investors to claim accumulated rewards (daily, referral, and pool-based), with **50% redistributed** back to the ecosystem.

Reward Update (`_updateInvestorRewards`)

- **Daily Rewards**: `(totalDeposit * 0.3%) * endedRounds`
- **Referral Rewards**:
    - `(directRefsDeposit * 0.025%) * endedRounds`
    - `(downlineRefsDeposit * 0.00675%) * endedRounds` (if in pool 2)
- **Pool Rewards**: `(rewardPerInvestorStored - poolRewardPerInvestorPaid)` per eligible pool
- Update reward trackers.
- Ensure `accumulatedReward >= 1 token`.

Fee and Redistribution

- **Split accumulatedReward**:
    - Fee: `(accumulatedReward * claimFeeInBp) / BASIS_POINTS` (default 10%)
    - Investor Portion: remainder
- **Redistribution**:
    - 50% of investor portion is redistributed.
- Mint investor portion (minus redistribution).
- Reset `accumulatedReward` to 0.

Redistribution as Deposit

- Redistributed portion treated as a **new deposit**:
    - May add investor to `_investors`.
    - Update referer structures and pools.

    - Adjust `totalDeposit` and `totalDepositAmount`.

Event Emission

- `Redistribute(investorAddress, toRedistribute)`
- `ClaimReward(investorAddress, toInvestor)`

3. Fee Swapping and Treasury Distribution

The **swapAndSendFees** function, **owner-only**, converts accumulated fees into ETH and distributes them to two treasuries.

Flow

1. Ensure contract holds fees.
2. Approve DEX router (PancakeSwap).
3. Swap tokens for ETH: `swapExactTokensForETH(accumulatedFees, minSwapPrice)`
4. Split ETH:
    - 70% → `treasury`
    - 30% → `treasury2`
5. Transfer ETH (revert if failed).

Event Emission

- `SwapAndSendFees(accumulatedFees, firstTreasuryAmount, secondTreasuryAmount)`

4. Pool Criteria Update

The **setPoolCriteria** function, **owner-only**, updates criteria for pools 0–6.

Rules

- At least **30 days** since last update.
- Pool IDs must be `< 7`.
- Criteria deviation ≤ 50% from previous.
- Maintain hierarchy (`pool[i] ≤ pool[i+1]`).

Batch Processing

- `isUpdateCriteriaActive = true`
- Iterate `_investors` (limited by `checkCountLimit`):
    - Remove ineligible investors.
    - Add eligible investors.
    - Adjust `participantsCount` and pool states.
- On completion:
    - Reset batch variables.
    - Emit `PoolsCriteriaUpdated`.

5. Whitelist Management

The **setWhitelist** function, **owner-only**, manages pools 7 and 8.

Add to Whitelist

- Must not already be whitelisted.
- Adjust `onlyWhitelistedInvestorsCount`.
- Set `isInvestorInPool = true`, increment `participantsCount`.
- Activate pool if inactive.

Remove from Whitelist

- Must already be whitelisted.
- Adjust `onlyWhitelistedInvestorsCount`.
- Remove from pool, adjust rewards.
- Deactivate pool if empty.

Event Emission

- `WhitelistUpdated(investor, poolId, add)`

Review

Repositories	https://github.com/fivepillarstoken/5PT-Token https://github.com/fivepillarstoken/InvestmentManager
Commits	2478b01e8d230fd4ce6756555eecb4e67d4ff74f 59b5dd477629c3fa7fdb3e71abdaade1dbed2702

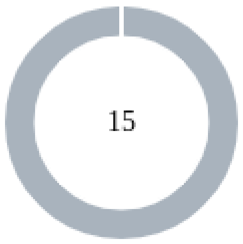
Audit Updates

Initial Audit	09 Sep 2025

Source Files

Filename	SHA256
InvestmentManager.sol	1f95b43f1fb8935d4b09e742cc2fd348bb0ace3841a7c6c54aa7147ef43bbd35
FivePillarsToken.sol	a5a356adab79862490b2474dcdc81de1bacaebf43488c67d4f1ce1cb2be3bcdb
interfaces/IPancakeRouter02.sol	aeab1d290f0fbce259d5fc278584f4c9d5fe51c8e028128eb6fcff90084097ba
interfaces/IPancakeRouter01.sol	a60878668592c407e86c3da4e25afc26a3281092f253fd59ebc06ac44868f25e
interfaces/IFivePillarsToken.sol	3a4ae184e1e7879f3b7fb46eea14cb91549dedbb49ba2deaef64937295e9e6ba

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	0	0	0	0
● Medium	0	0	0	0
● Minor / Informative	0	15	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	AAO	Accumulated Amount Overflow	Acknowledged
●	BT	Burns Tokens	Acknowledged
●	CO	Code Optimization	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	CRD	Cyclic Referral Dependency	Acknowledged
●	IDI	Immutable Declaration Improvement	Acknowledged
●	IPAR	Inconsistent Pool Amount Records	Acknowledged
●	MT	Mints Tokens	Acknowledged
●	PLPI	Potential Liquidity Provision Inadequacy	Acknowledged
●	RRA	Redundant Repeated Approvals	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L06	Missing Events Access Control	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged

●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L20	Succeeded Transfer Check	Acknowledged

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	InvestmentManager.sol#L807,833
Status	Acknowledged

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

Specifically, the contract increments the variable `curReward` for every reward pool proportionally to the deposited amount. This means that subsequent deposits will earn more rewards per period ended. These rewards will be minted and potentially deposited in the contract again. This will further amplify the `curReward` amount leading to excessive minting until the variable potentially overflows.

Shell
```
 poolInfo.curReward += amount * poolInfo.share /
BASIS_POINTS;
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

BT - Burns Tokens

Criticality	Minor / Informative
Location	InvestmentManager.sol#L571 FivePillarsToken.sol#L51
Status	Acknowledged

Description

The contract owner has the authority to burn tokens from a specific address. The owner may take advantage of it by calling the `burn` function. As a result, the targeted address will lose the corresponding tokens.

Shell
```
fivePillarsToken.burnFrom(investorAddress,
toInvestor);
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

CO - Code Optimization

Criticality	Minor / Informative
Location	InvestmentManager.sol
Status	Acknowledged

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, performs fewer operations or improves its readability.

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	InvestmentManager.sol#L395,409,425,475,645 FivePillarsToken.sol#L34,44,51
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Shell
```
function setDepositFee(uint256 newDepositFeeInBp) external
onlyOwner {...}
function setWhitelist(address investor, uint8 poolId, bool
add) external onlyOwner {...}
function setPoolCriteria(
uint8[] calldata poolIds,
PoolCriteria[] calldata criteriaOfPools,
uint256 checkCountLimit
) external onlyOwner {...}

function swapAndSendFees(uint256 minSwapPrice) external
onlyOwner {...}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

CRD - Cyclic Referral Dependency

Criticality	Minor / Informative
Location	InvestmentManager.sol#L657
Status	Acknowledged

Description

The contract uses conditional statements to prevent a user from being their own referrer within a referral chain of up to nine levels. However, this approach may be ineffective, as users can bypass the restriction by using multiple wallet addresses to create indirect self-referrals.

Specifically, the contract may incentivize users to split their funds accross multiple accounts to build a chain of self-referrals, where all accounts qualify for pool 2. As such, all accounts in the chain are eligible for downline rewards from the last deposit. This strategy may yield rewards that exceed the expected rewards from a single deposit.

Shell
```
function _checkRefererCirculation(address referer)
internal view {
address directReferer = referer;
if (referer != address(0)) {
for (uint i = 0; i < 9; i++) {
referer = accountToInvestorInfo[referer].referer;
if (referer == address(0)) break;
if (referer == directReferer) revert
RefererCirculationDetected();
}
}

}
```

Recommendation

To mitigate indirect self-referral mechanisms, implement a robust referral validation mechanism that tracks and requires a minimum deposit amount ensuring the uniqueness of participants across the referral chain. This approach helps prevent users from exploiting the system by using multiple wallet addresses, thereby preserving the integrity of the referral structure.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	InvestmentManager.sol#L269,270,271,274
Status	Acknowledged

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variable that saves gas when it is defined.

```
Shell
treasury
treasury2
dexRouter

startTimestamp
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IPAR - Inconsistent Pool Amount Records

Criticality	Minor / Informative
Location	InvestmentManager.sol#L784
Status	Acknowledged

Description

During deposit operations, the contract assigns users and their referrals to sequential reward pools. When a user is allocated to a pool, their deposited amount is used to increase the pool's `curReward` variable`. However, if a user performs a deposit that qualifies them for multiple pools simultaneously, the full deposited amount is credited across all eligible pools at once.

In contrast, if the user deposits incrementally, becoming eligible for each pool seperately, only a portion of the total amount is credited to each pool at the moment they qualify. This discrepancy leads to inconsistent reward allocations, where the same total deposit can yield different rewards depending on the deposit strategy used.

Shell
```
poolInfo.curReward += amount * poolInfo.share /
BASIS_POINTS;
```

Recommendation

The team is advised to consider allocating only the proportional portion of the deposit to each pool when multiple pools are activated simultaneously. This ensures accurate reward distribution and prevents users from gaining an unintended advantage based on their deposit method.

MT - Mints Tokens

Criticality	Minor / Informative
Location	FivePillarsToken.sol#L44 InvestmentManager.sol#L620,641
Status	Acknowledged

Description

The contract owner has authority to mint tokens. In particular, the contract does not enforce a capping mechanism on the minted supply. As a result, the contract tokens may be highly inflated.

Shell
```
fivePillarsToken.mint(address(this), fee);

fivePillarsToken.mint(investorAddress,
toInvestor);
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	InvestmentManager.sol#L881
Status	Acknowledged

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

Shell
```
IPancakeRouter01(dexRouter).swapExactTokensForETH(
accumulatedFees,
amountOutMin,
path,
address(this),
block.timestamp

);
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	InvestmentManager.sol#L877
Status	Acknowledged

Description

The contract is designed to `approve` token swaps during the contract's operation by calling the _approve function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

Shell
```
fivePillarsToken.approve(dexRouter,
accumulatedFees);
```

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	InvestmentManager.sol#L180
Status	Acknowledged

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

Shell
```
modifier NotInPoolCriteriaUpdate {
        if (isUpdateCriteriaActive) revert
PoolCriteriaUpdateNotEnded();
        _;

    }
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	FivePillarsToken.sol#L38
Status	Acknowledged

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

Shell
```
investmentManager = manager
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	InvestmentManager.sol#L677,679,684,689
Status	Acknowledged

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

Shell
```
uint256 roundReward = investorInfo.totalDeposit *
30000 / BASIS_POINTS

return (roundReward * endedRounds, roundReward)
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	InvestmentManager.sol#L723
Status	Acknowledged

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```Shell
uint256 reward
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	InvestmentManager.sol#L570
Status	Acknowledged

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

Shell
```
fivePillarsToken.transferFrom(investorAddress,
address(this), fee)
```

Recommendation

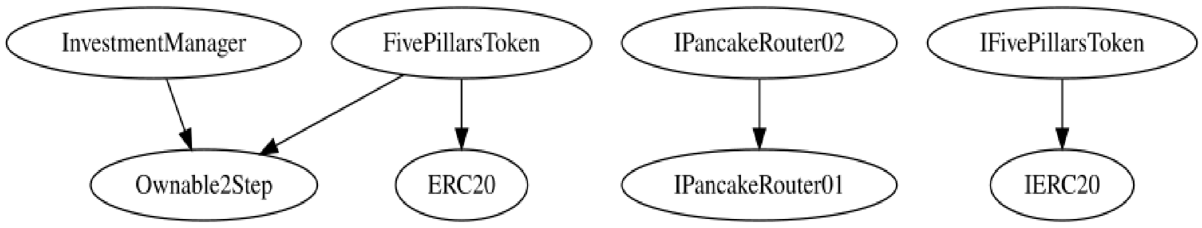
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

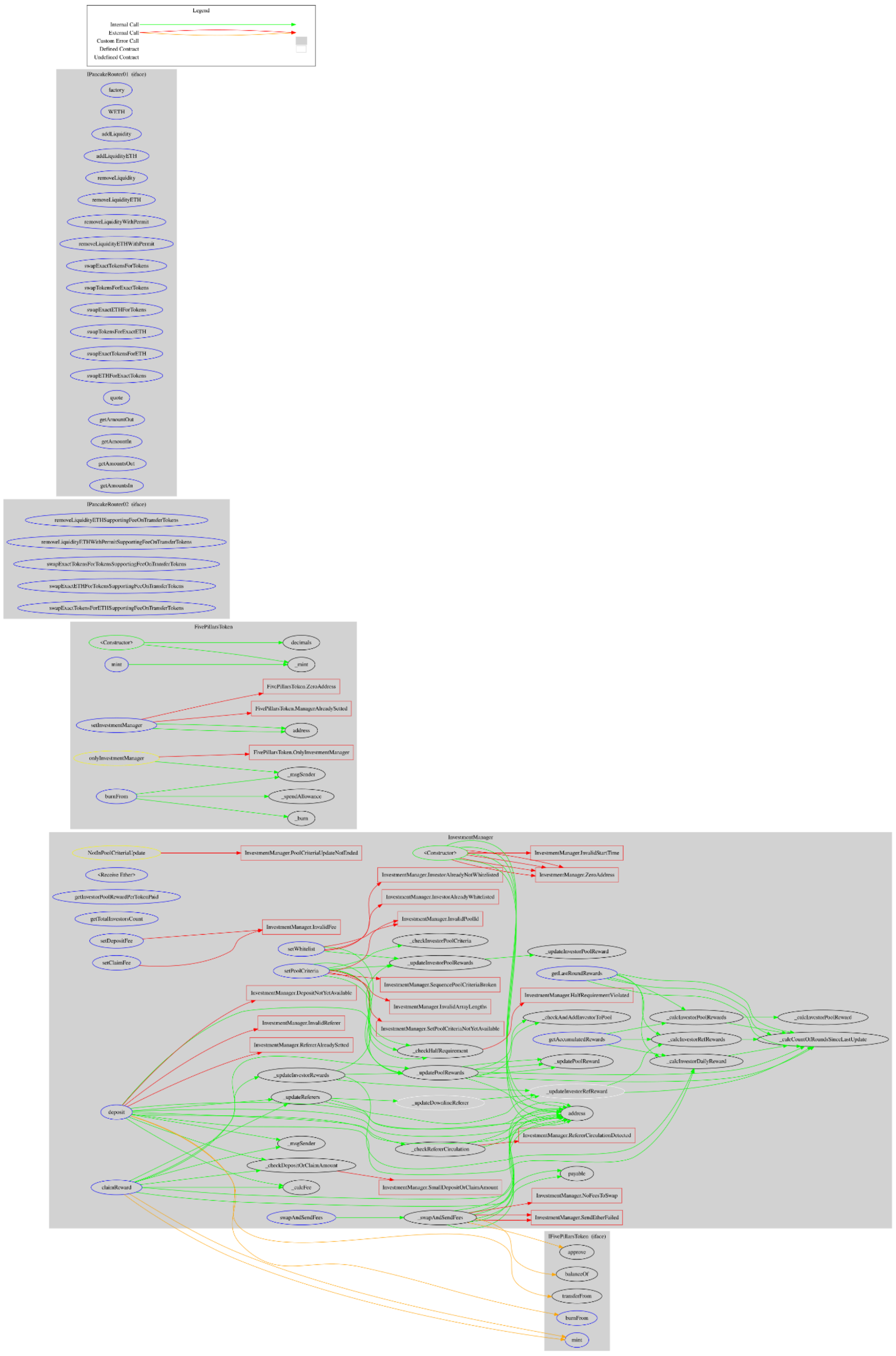
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
InvestmentManager	Implementation	Ownable2Step		
		Public	✓	Ownable
		External	Payable	-
	getAccumulatedRewards	External		-
	getLastRoundRewards	External		-
	getInvestorPoolRewardPerTokenPaid	External		-
	getTotalInvestorsCount	External		-
	setDepositFee	External	✓	onlyOwner
	setClaimFee	External	✓	onlyOwner
	setWhitelist	External	✓	onlyOwner
	setPoolCriteria	External	✓	onlyOwner
	deposit	External	✓	NotInPoolCriteriaUpdate
	claimReward	External	✓	NotInPoolCriteriaUpdate
	swapAndSendFees	External	✓	onlyOwner
	_checkDepositOrClaimAmount	Internal		
	_checkHalfRequirement	Internal		
	_checkRefererCirculation	Internal		
	_calcCountOfRoundsSinceLastUpdate	Internal		
	_calcInvestorDailyReward	Internal		

	_calcInvestorRefRewards	Internal		
	_calcInvestorPoolRewards	Internal		
	_calcInvestorPoolReward	Internal		
	_calcFee	Internal		
	_updateInvestorPoolRewards	Internal	✓	
	_updateInvestorPoolReward	Internal	✓	
	_updateInvestorRefReward	Internal	✓	
	_updateInvestorRewards	Internal	✓	
	_updateDownlineReferer	Internal	✓	
	_updateReferers	Internal	✓	
	_updatePoolRewards	Internal	✓	
	_updatePoolRewards	Internal	✓	
	_updatePoolReward	Internal	✓	
	_checkInvestorPoolCriteria	Internal		
	_checkAndAddInvestorToPool	Internal	✓	
	_swapAndSendFees	Internal	✓	
FivePillarsToken	Implementation	ERC20, Ownable2Step		
		Public	✓	ERC20 Ownable
	setInvestmentManager	External	✓	onlyOwner
	mint	External	✓	onlyInvestmentManager
	burnFrom	External	✓	onlyInvestmentManager

Inheritance Graph



Flow Graph



Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
IPancakeRouter01 (iface)
factory
WETH
addLiquidity
addLiquidityETH
removeLiquidity
removeLiquidityETH
removeLiquidityWithPermit
removeLiquidityETHWithPermit
swapExactTokensForTokens
swapTokensForExactTokens
swapExactETHForTokens
swapTokensForExactETH
swapExactTokensForETH
swapETHForExactTokens
quote
getAmountOut
getAmountIn
getAmountsOut
getAmountsIn
IPancakeRouter02 (iface)
removeLiquidityETHSupportingFeeOnTransferTokens
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens
FivePillarsToken
<Constructor>
decimals
mint
_mint
FivePillarsToken.ZeroAddress
FivePillarsToken.ManagerAlreadySetted
setInvestmentManager
address
onlyInvestmentManager
FivePillarsToken.OnlyInvestmentManager
_msgSender
burnFrom
_spendAllowance
_burn
InvestmentManager
NotInPoolCriteriaUpdate
InvestmentManager.PoolCriteriaUpdateNotEnded
<Constructor>
InvestmentManager.InvalidStartTime
InvestmentManager.ZeroAddress
<Receive Ether>
InvestmentManager.InvestorAlreadyNotWhitelisted
getInvestorPoolRewardPerTokenPaid
InvestmentManager.InvestorAlreadyWhitelisted
getTotalInvestorsCount
InvestmentManager.InvalidPoolId
InvestmentManager.InvalidFee
setDepositFee
_checkInvestorPoolCriteria
setWhitelist
_updateInvestorPoolReward
setClaimFee
_updateInvestorPoolRewards
setPoolCriteria
getLastRoundRewards
InvestmentManager.SequencePoolCriteriaBroken
InvestmentManager.DepositNotYetAvailable
InvestmentManager.HalfRequirementViolated
InvestmentManager.InvalidArrayLengths
InvestmentManager.InvalidReferer
_checkAndAddInvestorToPool
_calcInvestorPoolRewards
_calcInvestorPoolReward
InvestmentManager.SetPoolCriteriaNotYetAvailable
InvestmentManager.RefererAlreadySetted
getAccumulatedRewards
_calcInvestorRefRewards
_calcCountOfRoundsSinceLastUpdate
_checkHalfRequirement
_updatePoolReward
_calcInvestorDailyReward
_updateInvestorRewards
_updatePoolRewards
_updateInvestorRefReward
_updateReferers
_updateDownlineReferer
deposit
address
InvestmentManager.RefererCirculationDetected
_msgSender
_checkRefererCirculation
_checkDepositOrClaimAmount
payable
claimReward
_calcFee
InvestmentManager.SmallDepositOrClaimAmount
InvestmentManager.NoFeesToSwap
swapAndSendFees
_swapAndSendFees
InvestmentManager.SendEtherFailed
IFivePillarsToken (iface)
approve
balanceOf
transferFrom
burnFrom
mint

Summary

Five Pillars Token contract implements a token and utility mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

About Cyberscope

Cyberscope is a TAC blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A ***TAC Security*** *Company*

The Cyberscope team

cyberscope.io