HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: PAID Network (Master Ventures)
Date: 18 December, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for PAID Network (Master Ventures)
Approved By	Grzegorz Trawiński | Lead Solidity SC Auditor at Hacken OÜ Seher Saylik | Solidity SC Auditor at Hacken OÜ Eren Gonen | Solidity SC Auditor at Hacken OÜ
Tags	IDO, Vesting
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://paidnetwork.com/
Changelog	20.10.2023 - Initial Review 17.11.2023 - Second Review 18.11.2023 - Third Review 28.11.2023 - Fourth Review 15.12.2023 - Fifth Review 18.12.2023 - Sixth Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Introduction **5**
System Overview **5**
Executive Summary **7**
Risks **8**
Findings **9**
Critical 9
C01. withdrawPurchasedAmount() Can Be Called Without Emergency Event 9
C02. Insufficient IDO Token Balance Can Lead To Funds Loss 10
C03. Inconsistent Data In withdrawRedundantIDO() Function Locks Protocol Owner's Funds 12
C04. Inaccessible Funds Due to Ineffective withdrawRedundantIDOToken() Function in Emergency Situations 13
C05. withdrawPurchasedAmount() Function Allows Dual Benefits for Users 15
High 17
H01. Inconsistency in Purchase Amount Limitation Base On Allocation 17
H02. Incorrect Implementation of Upgradability Pattern 17
H03. Insufficient balance of IDO Tokens Due to Critical Flaw in updateTime() Function 18
H04. Funds Lock When vestingFrequency Is Zero 19
Medium 20
M01. Unrestricted Fee Configuration 20
M02. Centralization: Admin Control over TGE Date 21
M03. Mismatch Between Documentation and Implementation of Lockup Duration 22
Low 22
L01. Invalid Allowance Check 22
L02. Missing Zero Address Validation 23
L03. Missing Validation Check 23
L04. Static DOMAIN_SEPARATOR May Lead to Signature Replay Attacks On Forked Chains. 24
L05. Non Disabled Implementation Contract 24
Informational 25
I01. Usage of OpenZeppelin's Deprecated Functions 25
I02. Redundant Functions Declaration 25
I03. Redundant Variable Value Assignment 26
I04. Redundant Status Update 26
I05. Redundant Logic in isFailBeforeTGEDate() Function 26
I06. Inefficient Data Storage: Use of Memory Instead of Calldata 27
I07. Unused Variables 27
I08. Inconsistent Use of Pausability in Inherited Contracts 28
I09. Commented Code Parts 28
Disclaimers **30**
Appendix 1. Severity Definitions **31**
Risk Levels 31
Impact Levels 32
Likelihood Levels 32

Informational 32
Appendix 2. Scope **33**

Introduction

Hacken OÜ (Consultant) was contracted by PAID Network (Master Ventures) (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

PAID Network (Master Ventures) is a IDO Vesting protocol with the following contracts:

- *BasePausable* – a contract that is used by Vesting and Pool contracts to handle the pausable feature of the system.
- IgnitionFactory – factory contract to create new pool and vesting contracts.
- *Pool* – an IDO token contract that has different vesting stages based on user types and time periods. It allows users to deposit stable coins to the contract to participate in vesting.
- PoolStorage – storage contract of Pool that handles storing all the global variables.
- *Vesting* – The Vesting contract facilitates the gradual release of tokens to beneficiaries over time, following a vesting schedule.
- *VestingStorage* – storage contract of Vesting that handles storing all the global variables.
- *IgnitionList* – The IgnitionList contract provides functionality to verify whether a particular user belongs to a predefined list using the Merkle proof mechanism.
- *Errors* – The Errors library provides a set of constant error messages that can be used to represent various error scenarios.
- *PoolLogic* – The PoolLogic library provides utilities for handling various aspects of a token pool, particularly for initial decentralized offerings (IDOs). This library has a set of functions that can validate and calculate token-related values.
- *VestingLogic* – a logic contract that provides functions for calculating claimable amounts and verifying vesting information based on specific parameters.

Privileged roles

- The owner of the Pool contract:
    - can fund IDO tokens and start vesting
    - withdraw redundant IDO tokens from the contract
    - withdraw the purchased tokens from the contract
- The owner can revoke a vesting if upon creation such a parameter was provided. On revoking all vested tokens till the moment are automatically released to the beneficiary account.

- The owner of the Vesting contract:
  - Can set the IDO token address with setIDOToken().
  - Can change the funded status with setFundedStatus().
  - Can toggle claimable status with setClaimableStatus().
  - Can set the emergency canceled status with setEmergencyCancelled().
  - Can update the TGE date with updateTGEDate() to utmost 2 years later.
  - Can create vesting schedules for users with createVestingSchedule().
  - Can withdraw redundant IDO tokens with withdrawRedundantIDOToken().
- The admin of the Pool contract can:
  - set the Merkle proof
  - cancel the pool
  - update the TGE time before it arrives
  - update the Whale Open and Close times and Community Open and Close times
  - set claimable status and pause token claimings
  - claim token fee
  - claim participation fee
- The owner of the IgnitionFactory can:
  - Change the pool implementation address with setPoolImplementation()
  - Change the vesting implementation address with setVestingImplementation()
- The owner of the BasePausable can:
  - Pause the contract with pause()

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Technical description is provided.
- NatSpecs are provided.
- Project technical specifications are provided

Code quality

The total Code Quality score is **8** out of **10**.

- The development environment is configured.
- Some redundant declarations are found.
- Some possible code improvements were identified (I06, I07, I08).

Test coverage

- The test coverage metric was omitted from the final evaluation score exceptionally, due to inaccurate values provided by the coverage tool.

Security score

As a result of the audit, the code does not contain any severity issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.6**. The system users should acknowledge all the risks summed up in the risks section of the report.

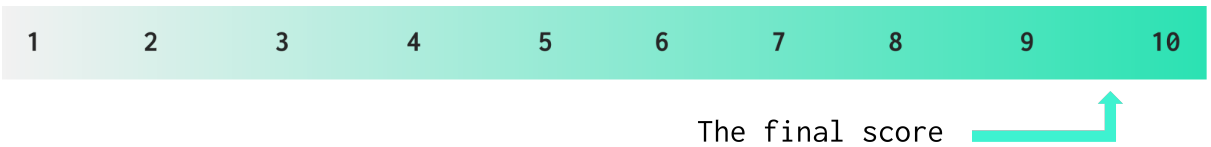

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
20 October 2023	4	3	2	2
17 November 2023	0	2	0	0

18 November 2023	0	0	0	0
28 November 2023	1	0	2	3
15 December 2023	0	0	0	1
18 December 2023	0	0	0	0

Risks

- The administrative role holds the authority to modify the Token Generation Event (TGE) date, adjust the durations for early and normal access, and defer the commencement of vesting for at most 2 years, thereby introducing the potential risk of altering critical project timelines and token release schedules.
- The owner privileged role has the authority to pause claiming earned IDO tokens for any period of time.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

■■■■ Critical

C01. withdrawPurchasedAmount() Can Be Called Without Emergency Event

Impact	High
Likelihood	High

The *withdrawPurchasedAmount()* function is intended to be an emergency function, allowing users to withdraw their deposited tokens in case of an emergency. However, due to a logic error in the requirement check, users can execute this function even when there is not an emergency situation.

The erroneous requirement check is as follows:

```
require(
    (isFailBeforeTGEDate() || !vesting.isEmergencyCancelled()) &&
    userInfo.withdrawn == 0,
    Errors.NOT_ALLOWED_TO_WITHDRAW_PURCHASED_AMOUNT
);
```

The *!vesting.isEmergencyCancelled()* check is inverted due to the '!' operator. This means that the function will proceed even if there is not an emergency situation, which is a direct violation of the intended requirement.
Also, the *withdrawPurchasedAmount()* function can be called only once, as it expects that userInfo's withdrawn property must be 0.

Users can exploit this vulnerability to:

1. Withdraw Tokens Prematurely: Users can withdraw their deposited tokens even without an emergency.
2. Claim More Than Allowed: After withdrawing their deposited tokens using the emergency function, users can still claim IDO tokens.
3. Misrepresent Deposits: Users can appear to have deposited more tokens than they actually have by exploiting the logic flaw.

Path: ./contracts/core/Pool.sol: withdrawPurchasedAmount()

Proof of Concept:

Immediate Double Benefit:

1. Users deposit tokens into the contract.
2. Execute the *withdrawPurchasedAmount()* function and withdraw their tokens.

3. After the lockup period is completed,execute the *claimIDOToken()* function and claim IDO tokens for the amount they have already withdrawn.

Incremental Benefit Through Re-deposit:

1. Assuming a max allocation of 1000 tokens and a user has only 500 tokens.
2. The user deposits 500 tokens into the contract.
3. Executes the *withdrawPurchasedAmount()* function, withdrawing the 500 tokens.
4. The user then re-deposits the same 500 tokens they just withdrew.
5. The user's principal in the contract now appears as 1000 tokens, even though they only ever deposited 500.
6. Since the user has already executed *withdrawPurchasedAmount()*, they cannot do so again due to the one-time execution check.
7. The user can now claim IDO tokens as if they deposited 1000 tokens, even though they only ever provided 500.

Recommendation: The primary issue stems from the incorrect condition check in the require statement. The condition should be rectified to ensure that the *withdrawPurchasedAmount()* function can only be executed during genuine emergency situations. Specifically, the condition *!vesting.isEmergencyCancelled()* should be corrected to properly reflect the intended logic.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: ´!´ operator removed from the require assertion.

C02. Insufficient IDO Token Balance Can Lead To Funds Loss

Impact	**High**
Likelihood	**High**

In the *fundIDOToken()* function, the owner has the authority to deposit promised funds. Once these tokens are funded, users will be able to claim the IDO tokens from the contract. Simultaneously, both the owner and administrators will be able to withdraw purchased tokens and accumulated fees. However, the variable that is used to limit total purchasable amount in all rounds, *totalRaiseAmount*, poses a potential risk that may lead to users losing their funds. The total purchased amount can be already exceeded during the Galaxy and Early access period because in those rounds the required checks are not implemented.

Moreover, users are able to deposit after the owner funds required corresponding IDO tokens to the Vesting contract.

Owner has two different options to fund IDO tokens.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

1. Owner executes the *fundIDOToken()* function by only transferring the required IDO tokens calculated based on the total purchased amount.

2. The owner executes the fundIDOToken function by only transferring the IDO tokens calculated based on the *totalRaiseAmount* value.

There are 3 main concerns within the explained implementation.

1. If the owner is funding the IDO tokens as taking *purchasedAmound* into account, some users will not receive earned IDO tokens due to insufficient balance. The reason for that is the pools are still active and users can deposit tokens after contract funding.
   The users are not allowed to withdraw their original deposited tokens either. The reason for that is the *withdrawPurchasedAmount* function cannot be called when the contract is in a funded state.
2. If the owner is funding the IDO tokens as taking *totalRaiseAmount* into account, it might fund less token than required and some users will not receive their IDO tokens. Moreover, since the deposits are still active, new-coming users will increase the IDO token amount required for Vesting.
   The users are not allowed to withdraw their original deposited tokens either. The reason for that is the *withdrawPurchasedAmount()* function cannot be called when the contract is in a funded state.
3. It is stated that the total purchased amount in all rounds should be restricted to the *totalRaiseAmount* variable. However, in Galaxy Whale Access and in Crowdfunding Whale Access rounds this variable is not considered and not used at all. Therefore, the total purchased amount can exceed the planned maximum limited amount that the owner can transfer to the Vesting contract.

Path: ./contracts/core/Pool.sol: fundIDOToken()

Proof of Concept:

1. Deploy the solution. Configure it that 1 IDO token price is equal to the 1 purchase token.
2. The owner sets the *totalRaiseAmount* to 1.000.000 tokens.
3. Whales purchase 1.100.000 tokens in the Galaxy and CrowdFunding rounds;

   ```
   Whale A: 300.000
   Whale B: 500.000
   Whale C: 300.000
   ```

4. The system owner funds IDO tokens according to the *totalRaiseAmount* value, which is 1.000.000 tokens.
5. Whale D purchases 400.000 tokens.
6. Vesting has started.

7. Whale A claims for the earned 300.000 IDO tokens.
8. Whale B claims for the earned 500.000 IDO tokens.
9. Whale D can only get 200.000 of their earned 400.000 IDO tokens due to insufficient balance. Whale D's initially deposited 400.000 purchase tokens are also locked in the contract or withdrawn by the owner because the emergency withdraw function cannot be called when the funding is completed and the contract is in claimable status.
10. Whale C cannot get any IDO tokens or purchased tokens.

Recommendation:

1. For all rounds, implement proper checks that validate if the total purchased amount is always less or equal to the total raise amount. The system must update the total purchased amount cumulatively in all rounds and use that value to check if it is exceeding the total raise amount.
2. Fund IDO tokens based on the total raise amount, not the total purchased amount since the deposits are still active after the funding.
3. Restrict the *withdrawRedundantIDOTokens()* function to be callable only when the deposits are completed and the vesting is started. With that way, it will be ensured that the users who deposited after the funding will receive their IDO tokens and the owner cannot access them.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The implemented solution allows either funding the total raise amount or if the deposits are completed, allows funding purchased amount. Now, all rounds are checked to ensure the total purchased amount is not exceeding the total raise amount.

C03. Inconsistent Data In withdrawRedundantIDO() Function Locks Protocol Owner's Funds

Impact	**High**
Likelihood	**High**

The function *withdrawRedundantIDO()* takes Vesting contract's IDO balance as a base to calculate the difference between total IDO amount that can be claimed by users and total funded IDO amount. The issue arises when the owner first sets claimable status to true for users to claim their IDO tokens and then withdraws the redundant IDO tokens left in the contract. When users start withdrawing (claiming) their IDO tokens, the IDO balance of the Vesting contract will be decreasing and the calculation presented below will result in locking the redundant funds in the Vesting contract. Ultimately, assuming majority of vested tokens are claimed, the function will revert with arithmetic underflow error.

```
redundantAmount =
IERC20(vesting.getIDOToken()).balanceOf(address(vesting)); -
getIDOTokenAmountByOfferedCurrency(purchasedAmount)
```

Path: ./contracts/core/Pool.sol: withdrawRedundantIDO()

Proof of Concept:

1. User A purchases 10.000 IDO tokens
2. The platform owner funds the Vesting contract with 15.000 IDO tokens (Vesting contract balance has now 15.000 tokens)
3. The owner enables claiming IDO tokens
4. User A withdraws 10.000 IDO tokens. Now, the new balance of Vesting contract is 5.000 tokens
5. The owner attempts to withdraw the redundant IDO tokens within the given calculation above. Observe that withdrawable redundant amount is equal to 5.000 - 10.000 = -5.000
   Since it is a negative value, the transaction reverts due to arithmetic underflow.

Recommendation: For the given issue two different solutions can be followed:
1. Rectify the token accounting within the *withdrawRedundantIDOToken()* function by adjusting the subtraction to occur between *totalFundedAmount* and *purchasedAmount*. To implement this, it is necessary to introduce an additional global variable: *totalFundedAmount* that keeps track of the funded total amount in the Vesting contract.

2. Alternatively, establish a mandatory order for function calls, stipulating that *withdrawRedundantIDOToken()* must be invoked before the initiation of the claiming process.

Found in: 9fef7b1

Status: Fixed (Revised commit: 86d5f36)

Remediation: A new implementation that is explained below is introduced.

1. If the project is canceled, the withdrawRedundantIDOToken function allows the owner to withdraw the entire IDO balance in Vesting.
2. For all the other scenarios, the withdrawRedundantIDOToken function allows the owner to withdraw only the difference between the total funded amount and the purchased amount.

C04. Inaccessible Funds Due to Ineffective withdrawRedundantIDOToken() Function in Emergency Situations

Impact	**High**
Likelihood	**High**

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

The *withdrawRedundantIDOToken()* function is designed to allow the withdrawal of excess IDO tokens from the Vesting contract in specific situations, such as pool failure or insufficient funding. However, a scenario reveals that this function becomes inoperative when the pool is canceled post-TGE, leading to the locking of funds in the Vesting contract. This issue arises due to the *vesting.setEmergencyCancelled(true)* call made during the *cancelPool()* operation, which inadvertently restricts the functionality of *withdrawRedundantIDOToken()* due to its *notEmergencyCancelled()* modifier. This action inadvertently affects the operational status of the *withdrawRedundantIDOToken()* function, which is governed by the *notEmergencyCancelled()* modifier. The *withdrawRedundantIDOToken()* function is designed to check the *emergencyCancelled* status; if this status is set to true, as is the case following the execution of *cancelPool()*, the contract will automatically revert any attempts to execute *withdrawRedundantIDOToken()*. Consequently, this results in the function becoming restricted and non-operational in situations where its needed.

```
modifier notEmergencyCancelled() {
    require(
        !vesting.isEmergencyCancelled(),
        Errors.NOT_ALLOWED_TO_DO_AFTER_EMERGENCY_CANCELLED
    );
    _;
}
```

```
function cancelPool(bool _permanentDelete) external onlyAdmin {
    (uint64 _TGEDate, , , , ) = vesting.getVestingInfo();
    if (block.timestamp >= _TGEDate) {
        require(
            block.timestamp <=
                (ignitionFactory.getLockupDuration() + _TGEDate),
            Errors.NOT_ALLOWED_TO_CANCEL_AFTER_LOCKUP_TIME
        );
        vesting.setEmergencyCancelled(true);
    }
    // This should be marked as cancel (paused === cancel)
    _pause();
    vesting.setClaimableStatus(false);
    emit CancelPool(address(this), _permanentDelete);
}
```

Path: ./contracts/core/Pool.sol: withdrawRedundantIDO(), cancelPool()

./contracts/core/Vesting.sol: withdrawRedundantIDOToken()

Proof of Concept:

1. Deploy the pool. Configure it that 1 IDO token price is equal to the 1 purchase token. The owner sets the *totalRaiseAmount* to 10,000 tokens.
2. Whale A purchases 5,000 tokens in the Galaxy round.
3. The system owner funds IDO tokens according to the *totalRaiseAmount* value, which is 10,000 tokens.
4. Forward time to after TGE date.
5. The administrator executes *pool.cancelPool()*
6. The administrator attempts to execute *pool.withdrawRedundantIDOToken()* after an emergency was declared. Observe that the function does not operate due to the *vesting.notEmergencyCancelled()* modifier.Note, that this results in 10,000 funds being locked inside the Vesting contract.

Recommendation: Remove the *notEmergencyCancelled(* modifier from the Vesting's *withdrawRedundantIDOToken()* function.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: The Client removed the mentioned modifier from the *withdrawRedundantIDOToken* function.

C05. withdrawPurchasedAmount() Function Allows Dual Benefits for Users

Impact	**High**
Likelihood	**High**

The *withdrawPurchasedAmount()* function is designed for use when a project fails, allowing users to withdraw their initially deposited funds. However, there is a dual benefit scenario in situations where a user has deposited tokens and the vesting contract is funded by a collaborator. After the Token Generation Event (TGE) date has passed, users will be able to claim their vested IDO (Initial DEX Offering) tokens. The admin has the authority to cancel the pool even after vesting has started. If users begin to claim their tokens and the admin decides to cancel the pool and set *setClaimableStatus()* to a "False" value. Consequently, users will be no longer able to claim, as the admin has changed the claim status to "False" and *vesting.claimIDOToken()* includes an *onlySatisfyClaimCondition()* modifier. So, *withdrawPurchasedAmount()* function will be activated and callable by the users. In this case, a user who has claimed IDO tokens can also withdraw their initial deposited funds from the pool contract. This scenario allows a user to both claim IDO tokens and withdraw their purchased token amount.

```
 function withdrawPurchasedAmount(
       address _beneficiary
   ) external nonReentrant {
       PurchaseAmount storage userInfo =
userPurchasedAmount[_msgSender()];

       uint principalAmount = userInfo.principal;
       uint feeAmount = userInfo.fee;
       uint amount = principalAmount + feeAmount;

       require(amount > 0, Errors.ZERO_AMOUNT_NOT_VALID);
       // @fix: There're 2 ways to withdraw purchased amount: Pool is
closed or Pool is failed at TGE Date
       require(
           (isFailBeforeTGEDate() || vesting.isEmergencyCancelled()) &&
               userInfo.withdrawn == 0,
           Errors.NOT_ALLOWED_TO_WITHDRAW_PURCHASED_AMOUNT
       );

       purchaseToken.safeTransfer(_beneficiary, amount);
       userInfo.withdrawn = amount;

       emit WithdrawPurchasedAmount(_msgSender(), _beneficiary, amount);
   }
```

Path: ./contracts/core/Pool.sol: withdrawPurchasedAmount(), cancelPool()

./contracts/core/Vesting.sol: claimIDOToken()

Proof of Concept:

1. User A deposits an amount X of tokens in the galaxy round.
2. The admin funds the vesting contract.
3. Forward time to after the TGE Date.
4. The admin sets the *setClaimableStatus()* to “True”.
5. User A executes *vesting.claimIDOToken()* and claims an amount Y of IDO tokens.
6. The admin decides to stop users from claiming IDO tokens and sets claimable status to false.
7. The admin executes *cancelPool()* and pauses the contract, activating emergency mode.
8. With the emergency active, User A is able to call *pool.withdrawPurchasedAmount()* and withdraw the total purchased amount X from the pool contract. As a result, User A not only claims Y amount of IDO tokens but also withdraws all the purchased tokens that were paid in the galaxy round.

Recommendation: Restrict the owner's actions: the owner should not be able to cancel the project after users start claiming their vested IDO tokens.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: The user will be able to withdraw their initial amount, minus the claimed IDO tokens. The *withdrawPurchasedAmount()* function has been reconfigured to reflect this change.

■■■ High

H01. Inconsistency in Purchase Amount Limitation Base On Allocation

Impact	Medium
Likelihood	High

The *_maxPurchaseBaseOnAllocations* variable, which signifies the upper limit for whale purchases in the galaxy round, presents a potential discrepancy in the implementation. While the code in the implementation only checks if the amount is under the limits only for per transaction, it does not account for the overarching maximum allowable purchase for the entire galaxy round, as defined by the mentioned variable. Consequently, there exists a misalignment between the defined limit for whales in the round and the actual enforcement of this constraint during transactions. This will result in instances where the cumulative purchases in the galaxy round exceed the intended threshold for whale investors.

Path: ./contracts/core/Pool.sol: *buyTokenInGalaxyPool*()

Proof of Concept:

1. Owner specifies the *_maxPurchaseBaseOnAllocations* variable as 10.000 tokens.
2. The user who has the whale access buys 10.000 tokens in Galaxy round.
3. The user repeats the transaction 10 times and purchases 100.000 tokens. The maximum purchase limit based on allocation (10.000) is exceeded.

Recommendation: Declare a mapping that tracks each whale's purchased amount in galaxy round and compare the *_maxPurchaseBaseOnAllocations* variable to that to validate that the whale privileged users' purchases are not exceeding the limit.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: A mapping called *whalePurchasedAmount* declared to save and track whales' purchases in Galaxy Pool.

H02. Incorrect Implementation of Upgradability Pattern

Impact	Medium
Likelihood	High

The smart contract system appears to be designed for upgradability, as evidenced by the inheritance from the *Initializable* contract. However, the deployment and initialization process is separated into two distinct transactions, which raises concerns about potential front-running attacks on the *initialize()* function. Additionally, while there are indications of the intention to use the *TransparentProxy* pattern as seen in the deployment script (factory.deploy.ts), this was commented out, suggesting that the upgradability pattern might not have been correctly implemented.

Front-running Attacks: If the *initialize()* function can be called by anyone before the intended initializer, it can lead to unauthorized control or unintended behavior of the contract.

Broken Upgradability: If the upgradability pattern is not correctly implemented, it might be impossible to upgrade the contract in the future without redeploying and migrating state, which can be costly and complex.

Path: ./contracts/core/IgnitionFactory.sol

Recommendation: Review the upgradability pattern, ensuring that if *TransparentProxy* or another proxy pattern is intended to be used, it is correctly implemented. Ensure that the proxy contract and the logic contract are correctly linked, and the initialization is done securely.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The factory contract is now initialized within Proxy in deployment scripts. Also, contracts inherit from OpenZepplin's *contracts-upgradeable* versions of external libraries.

H03. Insufficient balance of IDO Tokens Due to Critical Flaw in updateTime() Function

Impact	High
Likelihood	Medium

In the context of funding IDO tokens, the owner has two distinct options:

1.The owner can transfer the *totalRaiseAmount* at any given time before TGE date. This flexibility is acceptable as long as it occurs within a reasonable timeframe, considering that the total purchased amount cannot surpass the *totalRaiseAmount*.

2.Alternatively, the owner has the option to transfer only the purchased amount, subject to the condition that "*block.timestamp > communityCloseTime*" is met. This option assumes that all purchases have ended.

However, a concern arises in the *updateTime()* function. Here, the owner has the ability to redefine the community open and close dates. This action reactivates deposits and may lead to a loss of funds for some users. The reason being, the contract lacks sufficient IDO tokens, as the funding was done only considering purchases up to that time, and there are not enough IDO tokens for the purchases to be made thereafter.

Path: ./contracts/core/Pool.sol: updateTime()

Proof of Concept:

1. Deploy the Pool with initializing the IDOToken address as zero address.
2. Owner sets 1 IDO token price to 1 purchase token.
3. CrowdFunding KYC user A purchases 10.000 IDO tokens.
4. Owner funds the contract by transferring IDO tokens based on the total purchased amount which is 10.000 IDO tokens.
5. Owner sets new community open and close dates by calling the *updateTime()* function.
6. Deposits are reactivated now. CrowdFunding KYC user B purchases 10.000 tokens.
7. Time advances to the date Vesting ends. Note that Vesting contract's current balance is 10.000 IDO.
8. User B claims their 10.000 tokens. Note that Vesting contract's current balance is 0 IDO.
9. User A attempts to claim tokens. Observe that transaction reverts due to insufficient balance.

Recommendation: It is recommended to revoke the owner's authority to modify community open & close times. Alternatively, fund only the totalRaisedAmount instead of purchasedAmount as a base to calculate the amount to be funded.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: Now the owner is not allowed to change the community times if the raise type is private and the contract is funded.

H04. Funds Lock When vestingFrequency Is Zero

Impact	High

Likelihood	Medium

The *calculateClaimableAmount()* function is designed to determine the current distributable IDO tokens, taking into account various release intervals and percentages. The *TGEPercentage*, which represents tokens available until the Token Generation Event (TGE) date, is allocated first. The remaining tokens are then distributed incrementally, contingent on the completion of the TGE and vesting dates, and in accordance with the vesting frequency.

The owner has the ability to set the *vestingFrequency* variable to zero during the contract initialization. Setting it to zero implies that users can obtain IDO tokens right after the TGE date. However, due to a mistaken if statement logic in the function, it always returns "*TGEAmount - claimedAmount*" resulting in the distribution of only a portion of the IDO tokens allocated for the TGE duration. The remaining funds of users will be locked in the contract.

As evident in the code snippet below, even if the condition '*block.timestamp > TGEDate + vestingCliff*' is met, the if statement will always pass because *vestingFrequency* is always zero. Users will not be able to claim funds that are separated for the date after TGE.

```
        // In cliff time
        if (block.timestamp < TGEDate + vestingCliff || vestingFrequency
== 0) {
            return TGEAmount - claimedAmount;
        }
```

Path: ./contracts/logics/VestingLogic.sol: calculateClaimableAmount()

Proof of Concept:

1. Owner deploys the Pool contract and sets the *vestingFrequency* as 0 and *TGEPercentage* as 20%. 1 IDO token is accepted as equal to 1 purchase token price.
2. User A purchases 10.000 tokens.
3. The owner funds the contract with 10.000 IDO tokens.
4. The owner sets the claimable status to 'true'.
5. Time advances to the end of the entire Vesting period.
6. User A attempts to claim tokens. Observe that only 2.000 tokens are claimable (20% of claimable amount).

Recommendation: Remove the condition of "*vestingFrequency == 0*" from the if statement. Introduce a new if statement below it that allows withdrawing the remaining tokens when the TGE date is over if the *vestingFrequency* is set to zero.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: A solution that allows withdrawing all earnings when the vesting frequency is zero is introduced.

Medium

M01. Unrestricted Fee Configuration

Impact	Low
Likelihood	High

No restriction on the values of *galaxyParticipationFeePercentage* and *crowdfundingParticipationFeePercentage*, allowing them to be set to arbitrary and potentially harmful percentages, including 100%. This oversight poses a severe risk of misuse or exploitation, as it could lead to excessive fees or unintended financial consequences within the system.

Path: ./contracts/core/Pool.sol: initialize()

Recommendation: Implement constraints on these percentages to ensure they remain within reasonable and secure limits, safeguarding the financial integrity of the application.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The *galaxyParticipationFeePercentage* and *crowdfundingParticipationFeePercentage* variables now have minimum and maximum value control checks added to the initialize function.

M02. Centralization: Admin Control over TGE Date

Impact	Low
Likelihood	High

The contract allows the administrative role to modify the Token Generation Event (TGE) date, adjust early and normal access durations, and defer vesting commencement. This centralized control can alter project timelines and token release schedules.

The administrative role has the capability to:

- Modify the TGE date.
- Adjust early and normal access durations.
- Defer the start of vesting.

The ability to indefinitely modify the TGE date can lead to potential misuse, such as perpetually delaying the TGE and locking users' funds.

Path: ./contracts/core/Pool.sol: updateTGEDate()

Recommendation: There should be a maximum limit set for the TGE date. For instance, the administrative role should not be allowed to set a TGE date that is more than 1 years from the initial TGE date. This ensures that while there's flexibility to adjust the TGE date for genuine reasons, there's also a safeguard against unreasonable delays.

Found in: 6ff182a

Status: Fixed (Revised commit: 9cd4c55)

Remediation: The following control was added to function:

```
require(_newTGEDate <= TGEDate +
ignitionFactory.getMaximumTGEDateAdjustment(),
Errors.NOT_ALLOWED_TO_ADJUST_TGE_DATE_TOO_BIG)
```

According to this control, the admin cannot set a value greater than *`TGEDate + ignitionFactory.getMaximumTGEDateAdjustment()`*. However, the admin can still execute the function multiple times and bypass the intended check.

M03. Mismatch Between Documentation and Implementation of Lockup Duration

Impact	Low
Likelihood	High

The *`IgnitionFactory`* contract's documentation (IgnitionFactory.md) and NatSpec comments mention a lockup duration of 14 days. However, in the actual implementation, the *`LOCKUP_DURATION`* constant is set to 5 minutes. This discrepancy can lead to confusion and unintended behavior.

Path: ./contracts/core/IgnitionFactory.sol

Recommendation: Depending on the intended behavior, either update the *`LOCKUP_DURATION`* constant in the contract to reflect a 14-day duration or update the documentation to accurately describe the 5-minute lockup period.

Found in: 6ff182a

Status: Fixed (Revised commit: 9cd4c55)

Remediation: The *`LOCKUP_DURATION`* is set to 14 days.

Low

L01. Invalid Allowance Check

Impact	Low

Likelihood	Medium

Functions to buy tokens validate if the allowance is greater than the purchase amount without considering the additional fee. Given that fees are also being collected alongside the purchase amount, this omission creates an inconsistency in the system. This may cause users to spend unnecessary Gas as some transactions can revert.

Path: ./contracts/core/Pool.sol: *buyTokenInGalaxyPool*(), *buyTokenInCrowdfundingPool*()

Recommendation: Re-implement the *_verifyAllowance* function as followed:

```
require(allowance >= _purchaseAmount + _fee,
Errors.NOT_ENOUGH_ALLOWANCE);
```

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The solution implements now require assertion proposed in the issue's recommendation.

L02. Missing Zero Address Validation

Impact	Low
Likelihood	Low

The zero address validation check is not implemented for the following functions:

1. *initialize()*

Setting one of aforementioned parameters to zero address (0x0) results in breaking main business flow.

Paths: ./contracts/core/Pool.sol: *initialize()*

Recommendation: Implement zero address validation for the given parameters.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: Required zero address checks are provided.

L03. Missing Validation Check

Impact	Low

Likelihood	Low

The provided code lacks a check to ensure that *maxPurchaseAmountForKYCUser* is greater than *maxPurchaseAmountForNotKYCUser* to align with the system's intended logic.

There is no check if the galaxy pool fee is less than the crowdfunding pool fee.

Path: ./contracts/core/Pool.sol: initialize()

Recommendation: Implement the proper checks.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The mentioned check was added to the *initialize()* function.

L04. Static DOMAIN_SEPARATOR May Lead to Signature Replay Attacks On Forked Chains.

Impact	Medium
Likelihood	Low

The *Pool* contract initializes the *DOMAIN_SEPARATOR* using the network's *chainID* during its initialization. This *DOMAIN_SEPARATOR* remains static and does not account for potential post-deployment chain forks. Consequently, signatures could be replayed across both versions of the chain, leading to potential security risks.

In the *Pool* contract:

- Initialization: The domain separator is computed using the network's *chainID* during initialization.
- Signature Verification: The *_verifyFundAllowanceSignature* function checks if a user has signed the *DOMAIN_SEPARATOR*.
- Replay Attack: In the event of a chain fork, since the *chainID* might change and the *DOMAIN_SEPARATOR* remains static, an attacker could reuse signatures to potentially receive user funds on both chains.

Path: ./contracts/core/Pool.sol: initialize()

Recommendation: Implement a mechanism to detect changes in the *chainID* and regenerate the *DOMAIN_SEPARATOR* accordingly. This ensures that the *DOMAIN_SEPARATOR* remains unique even after a chain fork.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: Rather than calculating during initialization and relying on a static *DOMAIN_SEPARATOR*, the *_domainSeparatorV4()* function has been implemented. With the implementation of the *_domainSeparatorV4()* function, the *DOMAIN_SEPARATOR* will now be recalculated in the event of forks.

L05. Non Disabled Implementation Contract

Impact	Medium
Likelihood	Low

The upgradeable contracts do not disable initializers in the constructor, as recommended by the imported OpenZeppelin's Initializable contract. This means that anyone can call the initializer on the implementation contract to set the contract variables and assign the roles.

Path: ./contracts/core/Pool.sol: initialize()

Recommendation: Build a constructor function in the upgradeable contracts that calls the *disableInitializers()* function.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: The *disableInitializers()* function is implemented.

Informational

I01. Usage of OpenZeppelin's Deprecated Functions

The *BasePausable* contract raises a concern by utilizing a deprecated function from the OpenZeppelin library, specifically the use of *_setUpRole*, which is no longer recommended, potentially posing security and reliability risks. To rectify this, the deprecated function should be replaced with the recommended *_grantRole* function.

Path: ./contracts/core/BasePausable.sol: __BasePausable__init()

Recommendation: Use OpenZeppelin's *_grantRole()* function.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The *_grantRole()* function is implemented.

I02. Redundant Functions Declaration

The existing code exhibits redundancy and inefficiency by having separate functions named *_forwardParticipationFee* and

_forwardPurchaseTokenFunds. A more streamlined approach can be adopted by directly utilizing the *safeTransferFrom()* function, consolidating these two functions into a single, more comprehensively named function, such as *purchaseTokenTransfer()* or using just *safeTransferFrom()*. This enhancement would enhance code simplicity, reduce duplication, reduce the deployment costs and promote more efficient maintenance, aligning with best coding practices.

Path: ./contracts/core/Pool.sol: _forwardParticipationFee(), _forwardPurchaseTokenFunds()

Recommendation: Declare only one function for the same implementations.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The *_forwardParticipationFee* and *_forwardPurchaseTokenFunds* functions were removed.

I03. Redundant Variable Value Assignment

The *claimable* variable is declared in the *VestingStorage* contract as *true* and its value is updated as *true* in the *initialize()* function of Vesting contract. Redundant declarations cause spending unnecessary Gas.

Path: ./contracts/core/Vesting.sol: initialize()

Recommendation: The initialization of the *claimable* variable in the Vesting contract's *initialize()* function should be removed since it is already initialized in the *VestingStorage* contract.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: The variable is only initialized in the *Vesting* contract.

I04. Redundant Status Update

The *fundIDOToken()* function in the contract contains logic that allows the vesting funded status to be updated to *true* multiple times. This redundant operation can lead to unnecessary Gas consumption. In the provided *fundIDOToken()* function, the external call *vesting.setFundedStatus(true)* sets the vesting funded status to *true*. However, there is no check to determine if the status is already *true* before making this update. If the function is called multiple times, the status will be overridden with the same value, leading to wastage of Gas.

Path: ./contracts/core/Pool.sol: fundIDOToken()

Recommendation: Before updating the vesting funded status, check if it is already set to true. If it is, skip the update.

Found in: 6ff182a

Status: Fixed (Revised commit: ce3edb0)

Remediation: Within the current function implementation, funding can be called / executed only once.

I05. Redundant Logic in isFailBeforeTGEDate() Function

The *isFailBeforeTGEDate()* function in the contract contains logic that checks if the contract is paused or if the vesting is not funded by the TGE date. However, it seems that the check for the contract being funded is redundant. If the contract is not funded by the TGE date, the project is considered to have failed, and the contract can be paused using the *pause()* function.

Path: ./contracts/core/Pool.sol: isFailBeforeTGEDate()

Recommendation: It is advisable to streamline the function by only checking if the contract is in a paused state. The check for vesting not being funded by the TGE date should be eliminated.

Found in: 6ff182a

Status: Mitigated (Revised commit: ce3edb0)

Remediation: The Client stated that the intent is the project status should fail automatically without triggering the pause function.

I06. Inefficient Data Storage: Use of Memory Instead of Calldata

The smart contract uses the *memory* keyword for function parameters when *calldata* would be more gas-efficient. In Ethereum, *calldata* is a read-only storage location that holds the function arguments, making it cheaper in terms of Gas compared to *memory*.

When a function parameter is marked as memory, it creates a copy of the input data in a temporary location. This consumes more Gas than necessary, especially when the data is not modified within the function. On the other hand, *calldata* is a special storage location that is cheaper in terms of Gas because it is read-only and does not require copying data.

Path: ./contracts/core/Pool.sol: initialize(), buyTokenInGalaxyPool(), buyTokenInCrowdfundingPool(), _internalWhaleBuyToken(), _internalNormalUserBuyToken()

./contracts/extensions/IgnitionList.sol: _verifyUser()

Recommendation: It is recommended to mark the data type as *calldata* instead of *memory*.

Found in: 6ff182a

Status: Reported (Revised commit: ce3edb0)

Remediation: The changes are applied for the Pool contract but it is skipped for *IgnitionList*.

I07. Unused Variables

Within the *initialize()* function of the contract, several variables are defined and assigned values but are never utilized in any subsequent logic or operations. This can lead to confusion, increased Gas costs, and potential inefficiencies.

In the provided *initialize()* function, the following variables are declared and assigned values:

- galaxyPoolProportion
- earlyAccessProportion
- maxPurchaseAmountForGalaxyPool

However, post their declaration and assignment, these variables are not referenced or used in any subsequent operations or logic within the function or elsewhere in the contract.

Path: ./contracts/core/Pool.sol

Recommendation: If these variables are not required for any future logic or operations, they should be removed from the contract to save some Gas.

Found in: 6ff182a

Status: Reported (Revised commit : ce3edb0)

Remediation: The variables *galaxyPoolProportion* and *earlyAccessProportion* are used in the calculation of the *maxPurchaseAmountForEarlyAccess* variable, but the *maxPurchaseAmountForGalaxyPool* variable is declared and never used.

I08. Inconsistent Use of Pausability in Inherited Contracts

The contracts *Vesting.sol*, *IgnitionFactory.sol*, and *Pool.sol* all inherit from *BasePausable.sol*, which provides functionality to pause the contract. However, only the *Pool.sol* contract has implemented the pausability feature. This inconsistency can lead to confusion and potential misuse of the contracts.

Path: ./contracts/core/Pool.sol,
./contracts/core/IgnitionFactory.sol,
./contracts/core/Vesting.sol

Recommendation: If *Vesting.sol* and *IgnitionFactory.sol* are intended to be pausable, implement the pausability feature in these contracts,

ensuring that critical functions can be halted in emergencies. Alternatively, consider changing the inheritance of contracts.

Found in: 6ff182a

Status: Reported (Revised commit: ce3edb0)

Remediation: *BasePausable* is imported in the *IgnitionFactory* and *Vesting* contracts. However, the pausability features of *BasePausable* are not used within these contracts.

I09. Commented Code Parts

Following commented code parts were observed:

1. *Pool* lines 6, *EIP712Upgradeable* import

The presence of commented-out code indicates an unfinished implementation, potentially causing confusion for both developers and users and decreasing code readability.

Path: ./contracts/core/Pool.sol

Recommendation: Remove commented parts of code.

Found in: 9fef7b1

Status: Fixed (Revised commit: 0f8d17d)

Remediation: The *EIP712Upgradeable* contract is implemented.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting
Commit	6ff182a5499103feec5c980677be3c5ebe4f4968
Whitepaper	Not Provided
Requirements	Confidential
Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:26c9448ba3944f5afc7ed8838ec707cb460a54ce189673a4571c344201b121cf File: ./contracts/core/IgnitionFactory.sol SHA3 b22adf7223498d995404f7ce7e453701c69130d3296b4c439d5164051d2dcda4 File: ./contracts/core/Pool.sol SHA3:c005d0832a53c51e38364f7977365de80246a7c088e706d1b3a433bcba29ceda File: ./contracts/core/PoolStorage.sol SHA3:946f2b42e5012b054bed50484b3e108bf04098e0802fcd532b600923e6c924ea File: ./contracts/core/Vesting.sol SHA3:687e00c88561c63781fc68efa92b23d767640a21431e06191ead3ccf7f396274 File: ./contracts/core/VestingStorage.sol SHA3:f843453794d5ac360ebd6191b22885b7f636b5092b5a7cb226bd3cc8b46ffbde File: ./contracts/logics/PoolLogic.sol SHA3:7d5eb07b9665366343c82968fffe753095d508fee0fd9231e4edffcbcfa9c52a File: ./contracts/logics/VestingLogic.sol SHA3:7fa6fd231ae664df34dcf02b4dd95c8083b62d8c96eea912468be863825bf420 File: ./contracts/extensions/IgnitionList.sol SHA3:ce1fc6d5a784d80fa1562676a5a103be3ef07afe057d5214b70bb38c32ff08e5 File: ./contracts/helpers/Errors.sol SHA3:ce1fc6d5a784d80fa1562676a5a103be3ef07afe057d5214b70bb38c32ff08e5 File: ./contracts/interfaces/IERC20withDec.sol SHA3:59472cc1bef4439b0f2ca755758b92f08b6525dd0692d7b75f57903bb79d26df File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:1bc06a539876c49fe5028545f69ded70880eb54be7c4b1adc52dbdc02c576809 File: ./contracts/interfaces/IPool.sol SHA3:a996182800b3bb717525a5566bdbe6856375fce891b5a4a3a69440618e4a0cb2 File: ./contracts/interfaces/IVesting.sol SHA3:77d65e2ff7c1525d7c60c6abd82b2eecbd3010dc6202a1bbfc81c23ada93942d

Second review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting
Commit	ce3edb03e4c0752461f0d01edee69ba97a624874
Whitepaper	Not Provided
Requirements	Confidential
Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:06507c61af922c83474d0819e7ebcbe8810c8697a563e3af6123b66b361b5891 File: ./contracts/core/IgnitionFactory.sol SHA3 cc3123b821ed7c4996d81e43c8b1bb1abc2e123c1f95a88e4c5b738707e50e0b File: ./contracts/core/Pool.sol SHA3:4a85c835832d8cb06a00161e666974c08e26c787f4712899ec0a35bfa04cc507 File: ./contracts/core/PoolStorage.sol SHA3:79266fcad92de4e652f481280e4abdb6b3441f685c573ce5695e3924fba50232 File: ./contracts/core/Vesting.sol SHA3:7c0cdf7c823c3c0a3f4f09869dab3b6f04382dfb338f8f60c76e4938cb311ebe File: ./contracts/core/VestingStorage.sol SHA3:92e90bed3663c4da44b87fcb3364c1cf816015b2e70d1ebc71b1806ad24a57cd File: ./contracts/logics/PoolLogic.sol SHA3:1fad6b4873d39a4ab5a442ba80399066b5ee69a9af29f3b4219cbf16e964f29e File: ./contracts/logics/VestingLogic.sol SHA3:a560fb547dc4794743ae8526a359c0ea69d974cd8b17c84668aae8a3639ca058 File: ./contracts/extensions/IgnitionList.sol SHA3:6c23eee2f4836e546a9bbf183cc966259f9a6f4c6c8796423f41d10defea16fd File: ./contracts/helpers/Errors.sol SHA3:c273e13a3629b2596f3e4a5ffd7511b8282fc9448ae61065c6435f29f654e481 File: ./contracts/interfaces/IERC20withDec.sol SHA3:dd9778bc430745ef260584cbb89865e6584ef7f29c694ae02e23573efbce53d3 File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:f7fd8a0c1df3ac637a15a2f705321d9c17508d9c3bd508911b5a51e31ccc001e File: ./contracts/interfaces/IPool.sol SHA3:cf828ee31941fe9704f55396cfa408519f78a08c5e9dbeb1d7685b6e75786e23 File: ./contracts/interfaces/IVesting.sol SHA3:3b343bd80636a95a1db7042fe815368f913d37836fd913d8f5af43bf8b000f95

Third review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting

Commit	9cd4c55a3d2d3fb406f9c1021ba047ed4fa55a12
Whitepaper	Not Provided
Requirements	Confidential
Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:06507c61af922c83474d0819e7ebcbe8810c8697a563e3af6123b66b361b5891 File: ./contracts/core/IgnitionFactory.sol SHA3 5635284d559207e4054f1a87572818e7399f997ac4036118528c25121dc927f8 File: ./contracts/core/Pool.sol SHA3:e1a71c38188ef89671cd9e0ae3deba970949fed56a3552c9312d9aa630c7b7fd File: ./contracts/core/PoolStorage.sol SHA3:7b7701a8a23c41e414f05804642b19a1fd4da7c1aef5e4c3991c037ba3d7182f File: ./contracts/core/Vesting.sol SHA3:3178873b96d77ca4a7f0490d76ec99e7ec30bec61d0c29e84539797f44cd35ed File: ./contracts/core/VestingStorage.sol SHA3:cdd6f59ebf5b5590985224846538 64e9085617cbde0d885153ae04864a800897 File: ./contracts/logics/PoolLogic.sol SHA3:1fad6b4873d39a4ab5a442ba80399066b5ee69a9af29f3b4219cbf16e964f29e File: ./contracts/logics/VestingLogic.sol SHA3:a560fb547dc4794743ae8526a359c0ea69d974cd8b17c84668aae8a3639ca058 File: ./contracts/extensions/IgnitionList.sol SHA3:6c23eee2f4836e546a9bbf183cc966259f9a6f4c6c8796423f41d10defea16fd File: ./contracts/helpers/Errors.sol SHA3:c273e13a3629b2596f3e4a5ffd7511b8282fc9448ae61065c6435f29f654e481 File: ./contracts/interfaces/IERC20withDec.sol SHA3:dd9778bc430745ef260584cbb89865e6584ef7f29c694ae02e23573efbce53d3 File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:46f57cec10ef9cb2779f9044789168e79cb1c4380fe476e5f476b82051a24516 File: ./contracts/interfaces/IPool.sol SHA3:cf828ee31941fe9704f55396cfa408519f78a08c5e9dbeb1d7685b6e75786e23 File: ./contracts/interfaces/IVesting.sol SHA3:78f389b8ba1cffa2382485526520fe93a06c810d78a83e89cea9b196dd4e267e

Fourth review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting
Commit	9fef7b1e85522077c9325dbfc5bc6f235f77b26e

Whitepaper	Not Provided
Requirements	Confidential
Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:06507c61af922c83474d0819e7ebcbe8810c8697a563e3af6123b66b361b5891 File: ./contracts/core/IgnitionFactory.sol SHA3 5635284d559207e4054f1a87572818e7399f997ac4036118528c25121dc927f8 File: ./contracts/core/Pool.sol SHA3:e1a71c38188ef89671cd9e0ae3deba970949fed56a3552c9312d9aa630c7b7fd File: ./contracts/core/PoolStorage.sol SHA3:7b7701a8a23c41e414f05804642b19a1fd4da7c1aef5e4c3991c037ba3d7182f File: ./contracts/core/Vesting.sol SHA3:3178873b96d77ca4a7f0490d76ec99e7ec30bec61d0c29e84539797f44cd35ed File: ./contracts/core/VestingStorage.sol SHA3:cdd6f59ebf5b559098522484653864e9085617cbde0d885153ae04864a800897 File: ./contracts/logics/PoolLogic.sol SHA3:1fad6b4873d39a4ab5a442ba80399066b5ee69a9af29f3b4219cbf16e964f29e File: ./contracts/logics/VestingLogic.sol SHA3:a560fb547dc4794743ae8526a359c0ea69d974cd8b17c84668aae8a3639ca058 File: ./contracts/extensions/IgnitionList.sol SHA3:6c23eee2f4836e546a9bbf183cc966259f9a6f4c6c8796423f41d10defea16fd File: ./contracts/helpers/Errors.sol SHA3:c273e13a3629b2596f3e4a5ffd7511b8282fc9448ae61065c6435f29f654e481 File: ./contracts/interfaces/IERC20withDec.sol SHA3:dd9778bc430745ef260584cbb89865e6584ef7f29c694ae02e23573efbce53d3 File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:46f57cec10ef9cb2779f9044789168e79cb1c4380fe476e5f476b82051a24516 File: ./contracts/interfaces/IPool.sol SHA3:cf828ee31941fe9704f55396cfa408519f78a08c5e9dbeb1d7685b6e75786e23 File: ./contracts/interfaces/IVesting.sol SHA3:78f389b8ba1cffa2382485526520fe93a06c810d78a83e89cea9b196dd4e267e

Fifth review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting
Commit	0f8d17d12004412e709fdae2dd4a775782f7bce8
Whitepaper	Not Provided
Requirements	Confidential

Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:2b3e05d456222c6484a45085574d58987eade9533ab3392cf3228282ff015b53 File: ./contracts/core/Base.sol SHA3:2b3e05d456222c6484a45085574d58987eade9533ab3392cf3228282ff015b53 File: ./contracts/core/IgnitionFactory.sol SHA3 36ea32126df74dab82b5fd9320ef26bf0b858664234e3933b3877f5279ddadd9 File: ./contracts/core/Pool.sol SHA3:bb8340679f114991f70fc7a403a36f074a6b69acd10dfef269469785f1c5c9a7 File: ./contracts/core/PoolStorage.sol SHA3:da7e003d56c3f86e952b47dbc658ae8a143920cf950a0abcbd0440751e01a629 File: ./contracts/core/Vesting.sol SHA3:a2f0d85a811d9a7d46ad73f0ea7fc2719cc3bba61adbcac649cebe21abdea526 File: ./contracts/core/VestingStorage.sol SHA3:2a587a36a1505ad8a270d6f3753da2db7e8e0b70c052d23b2cb265c8d35b6721 File: ./contracts/logics/PoolLogic.sol SHA3:1fad6b4873d39a4ab5a442ba80399066b5ee69a9af29f3b4219cbf16e964f29e File: ./contracts/logics/VestingLogic.sol SHA3:8f9f82b58dd60d266c1ba70282f23521d331a7f1c74dfa3e71f265fe9ea70512 File: ./contracts/extensions/IgnitionList.sol SHA3:c88136a351a35edac06663742ccda077742f8c0e8b7f1dfa80737076cd420a5d File: ./contracts/helpers/Errors.sol SHA3:bf954b863431a2f04530650f46ee9638ed1386d32c4613231a73c3d6bf82e396 File: ./contracts/interfaces/IERC20withDec.sol SHA3:dd9778bc430745ef260584cbb89865e6584ef7f29c694ae02e23573efbce53d3 File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:46f57cec10ef9cb2779f9044789168e79cb1c4380fe476e5f476b82051a24516 File: ./contracts/interfaces/IPool.sol SHA3:cf828ee31941fe9704f55396cfa408519f78a08c5e9dbeb1d7685b6e75786e23 File: ./contracts/interfaces/IVesting.sol SHA3:3a3a5594a6659b2fae60b248ed41b36eedf28361ebcee01aba2fb8a50138e795

Sixth review scope

Repository	https://github.com/PAIDNetwork/ignition-sc-crowdfunding-pool/tree/feature/vesting
Commit	86d5f3636d73aa566b465637ec7a078cb73486a7
Whitepaper	Not Provided
Requirements	Confidential

Technical Requirements	Confidential
Contracts	File: ./contracts/core/BasePausable.sol SHA3:06507c61af922c83474d0819e7ebcbe8810c8697a563e3af6123b66b361b5891 File: ./contracts/core/Base.sol SHA3:2b3e05d456222c6484a45085574d58987eade9533ab3392cf3228282ff015b53 File: ./contracts/core/IgnitionFactory.sol SHA3 36ea32126df74dab82b5fd9320ef26bf0b858664234e3933b3877f5279ddadd9 File: ./contracts/core/Pool.sol SHA3:e2ba41b5cd0a73e6f4b632c08ebe767755071c833d8e97688f7dfc7eaff974ae File: ./contracts/core/PoolStorage.sol SHA3:ec1ea97dc9b157cbd634060c4e3fffd74caf65e3bf8a609f5b5fd2e4665fa529 File: ./contracts/core/Vesting.sol SHA3:a2f0d85a811d9a7d46ad73f0ea7fc2719cc3bba61adbcac649cebe21abdea526 File: ./contracts/core/VestingStorage.sol SHA3:2a587a36a1505ad8a270d6f3753da2db7e8e0b70c052d23b2cb265c8d35b6721 File: ./contracts/logics/PoolLogic.sol SHA3:1fad6b4873d39a4ab5a442ba80399066b5ee69a9af29f3b4219cbf16e964f29e File: ./contracts/logics/VestingLogic.sol SHA3:8f9f82b58dd60d266c1ba70282f23521d331a7f1c74dfa3e71f265fe9ea70512 File: ./contracts/extensions/IgnitionList.sol SHA3:c88136a351a35edac06663742ccda077742f8c0e8b7f1dfa80737076cd420a5d File: ./contracts/helpers/Errors.sol SHA3:bf954b863431a2f04530650f46ee9638ed1386d32c4613231a73c3d6bf82e396 File: ./contracts/interfaces/IERC20withDec.sol SHA3:dd9778bc430745ef260584cbb89865e6584ef7f29c694ae02e23573efbce53d3 File: ./contracts/interfaces/IIgnitionFactory.sol SHA3:46f57cec10ef9cb2779f9044789168e79cb1c4380fe476e5f476b82051a24516 File: ./contracts/interfaces/IPool.sol SHA3:cf828ee31941fe9704f55396cfa408519f78a08c5e9dbeb1d7685b6e75786e23 File: ./contracts/interfaces/IVesting.sol SHA3:3a3a5594a6659b2fae60b248ed41b36eedf28361ebcee01aba2fb8a50138e795