SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Metatime
Date: 19 June, 2023

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Metatime
Approved By	Marcin Ugarenko | Lead Solidity SC Auditor at Hacken OU
Tags	ERC20 token; Vesting; Proxy; Factory
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://metatime.com/en
Changelog	25.05.2023 - Initial Review 09.06.2023 - Second Review 19.06.2023 - Third Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Metatime (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

MetaTime is a multi purpose protocol with the following contracts:

- *MTC* – custom ERC-20 token that mints all initial supply to pools determined by the deployer. Additional minting is not allowed.
  It has the following attributes:
  - Name: Metatime
  - Symbol: MTC
  - Decimals: 18
  - Total supply: dynamic (documentation: 10b tokens)
- *Distributor* – a pool contract that stores and distributes locked tokens related to the ecosystem tokenomics, such as Marketing Pool, Team Pool and Charity Pool.
- *TokenDistributor* – a pool contract that stores and distributes locked tokens related to mass distribution, such as Seed Sale 1, Seed Sale 2 and Public Sale.
- *LiquidityPool* – represents the Liquidity Pool in the MTC Tokenomics. It transfers funds when needed according to the market making purposes.
- *StrategicPool* – used for burning purposes. It has a manual burning function and burning function that calculates the burn amount by using formula.
- *TokenDistributorWithNoVesting* – custom TokenDistributor contract implementation with some differences, such as absence of periodic distribution, and it is not a proxy logic contract.

Privileged roles

- The owner of the *Distributor* contract can claim claimable tokens from the pool.
- The owner of the *LiquidityPool* contract can transfer tokens from the pool.
- The owner of the *TokenDistributorWithNoVesting* contract can arbitrarily set claimable token amounts for users and sweep tokens balance after end time.
- The owner of the *StrategicPool* contract can withdraw and burn tokens from the contract's balance.

- The owner of the *TokenDistributor* contract can arbitrarily set claimable token amounts for users and sweep tokens balance after end time.

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 and environment descriptions are provided.

Code quality

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

- The code follows best practices.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- All system features are covered with tests.

Security score

As a result of the audit, the code contains **no** 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: **10**.

The system users should acknowledge all the risks summed up in the risks section of the report.

1
2
3
4
5
6
7
8
9
10
The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
25 May 2023	11	9	4	0
09 June 2023	8	3	0	0
19 June 2023	0	0	0	0

Risks

- The owner of the vesting contract can extract all contract tokens (after the token distribution end time plus 100 days), leaving users empty-handed if they did not claim their tokens.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Passed	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. Invalid Calculations; Missing Validation

Impact	High
Likelihood	Medium

The vesting logic in the Distributor and TokenDistributor contracts is invalid.

Dependencies between *endTime*, *DISTRIBUTION_RATE*, and *PERIOD* variables are not validated or checked correctly during initialization.

The *DISTRIBUTION_RATE* can be incorrect compared to the *endTime* and number of periods that will occur based on the *PERIOD* variable between *startTime* and *endTime*.

This can lead to insufficient release amounts for users or Denial of Service and Token Supply Manipulation in case the *DISTRIBUTION_RATE* is too large.

Inside the internal *_calculateClaimableAmount()* function, the time from which the calculations are performed is not limited and the *block.timestamp* is used even if it is greater than *endTime*. It is mitigated by the *require(block.timestamp < endTime, "Distribution has ended");* check in the *calculateClaimableAmount()* function, but fundamentally it is incorrect, and can lead to invalid calculations when misused.

Paths:
./contracts/core/Distributor.sol
./contracts/core/TokenDistributor.sol

Recommendation: add validation of the initialization parameters.

For example, check if: *BASE_DIVIDER / _distributionRate * _period == endTime - startTime*

Or calculate the distribution rate based on the period length and vesting duration.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

H02. Funds Lock

Impact	Medium
Likelihood	High

Contracts TokenDistributor and PrivateSaleTokenDistributor do not allow users to claim tokens after the end date, resulting in tokens that are locked and can only be claimed later by the owner.

Users should be able to claim their pending claimable tokens after the end date, as it is highly unlikely that they will call the claim() function at the exact last moment in order to withdraw all possible tokens.

Paths:
./contracts/core/TokenDistributor.sol : claim();
./contracts/core/PrivateSaleTokenDistributor.sol : claim();

Recommendation: allow participants to claim tokens correctly after the distribution end date, or add a threshold period after the end date for users to claim in full.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

H03. Missing Validation; Data Consistency

Impact	High
Likelihood	Medium

Contracts TokenDistributor and PrivateSaleTokenDistributor allow calling setClaimableAmounts() more than once.

This results in miscalculations, as the actual total claimable amount will be greater due to past existing participants.

The check:

```
require(token.balanceOf(address(this)) >= totalClaimableAmount,
```

will not be valid.

In the worst case, a user vesting that has already been partially claimed can be updated with a new value, resulting in data inconsistency.

Paths:
./contracts/core/TokenDistributor.sol : setClaimableAmounts();
./contracts/core/PrivateSaleTokenDistributor.sol : setClaimableAmounts();

Recommendation: consider limiting the function call to be callable only once, or update the function logic to prevent data inconsistency.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

H04. Undocumented Functionality: Proxy System Architecture

Impact	Medium
Likelihood	High

The proxy systems defined by DistributorProxyManager, TokenDistributorProxyManager and InitializeProxy are implemented using an immutable logic address.

Therefore, it is not clear why the project is using custom proxy contracts and inheriting upgradeable contracts like Ownable2StepUpgradeable if the contracts are not supposed to be upgraded.

Paths:
./contracts/core/PrivateSaleTokenDistributor.sol
./contracts/utils/TokenDistributorProxyManager.sol
./contracts/utils/DistributorProxyManager.sol
./contracts/utils/InitializedProxy.sol
./contracts/core/Distributor.sol
./contracts/core/TokenDistributor.sol

Recommendation: it is recommended to use the Minimal Proxy ERC-1167 standard and the Clones library from OpenZeppelin as a way of implementing the Factory design pattern. Use non-upgradable versions of the Ownable2Step contract and use the Initializable contract only where necessary.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

Medium

M01. Best Practice Violation: Disable Initializers

Impact	High
Likelihood	Low

According to the OpenZeppelin documentation, upgradeable contracts should invoke the method _disableInitializers() in their constructor() to prevent them from being used.

However, said functionality is not implemented in all upgradeable contracts.

_disableInitializers() should be called in the constructor() of the Distributor and TokenDistributor contracts.

Paths:
./contracts/core/Distributor.sol
./contracts/core/LiquidityPool.sol
./contracts/core/PrivateSaleTokenDistributor.sol
./contracts/core/StrategicPool.sol
./contracts/core/TokenDistributor.sol
./contracts/utils/DistributorProxyManager.sol
./contracts/utils/TokenDistributorProxyManager.sol

Recommendation: follow OpenZeppelin's documentation regarding _disableInitializers in upgradeable contracts.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

M02. Best Practice Violation: Unchecked Transfer

Impact	High
Likelihood	Low

The ERC20 function transfer() is used repeatedly without the SafeERC20 wrapper.

Tokens may not follow the ERC20 standard and return false in case of transfer failure or not returning any value at all. This can lead to a Denial of Service or unexpected behavior when dealing with some tokens. Hence, it is a best practice to use the SafeERC20 wrapper when transferring tokens.

Paths:
./contracts/core/Distributor.sol: claim(), sweep();
./contracts/core/LiquidityPool.sol: _withdraw();
./contracts/core/PrivateSaleTokenDistributor.sol: claim(), sweep();
./contracts/core/StrategicPool.sol: _transfer();
./contracts/core/TokenDistributor.sol: claim(), sweep();

Recommendation: consider implementing the SafeERC20 library.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

M03. Contradiction: Third Party Integration

Impact	Medium
Likelihood	Medium

Although most contracts integrate OpenZeppelin's proxy features, not all of them seem to be used as implementations in a proxy architecture: PrivateSaleTokenDistributor, LiquidityPool and StrategicPool inherit from Owneable2StepUpgradeable.

The current code leads to confusion and may behave differently than expected.

Paths:
./contracts/core/PrivateSaleTokenDistributor.sol
./contracts/core/LiquidityPool.sol
./contracts/core/StrategicPool.sol

Recommendation: use Ownable2Step instead of Ownable2StepUpgradeable as the contracts are not supposed to be upgradeable.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

M04. Best Practice Violation - Checks-Effects-Interactions Pattern

Impact	High
Likelihood	Low

State variables are updated after the external calls to the token contract.

As explained in Solidity Security Considerations, it is best practice to follow the checks-effects-interactions pattern when interacting with external contracts to avoid reentrancy-related issues.

Paths:
./contracts/core/Distributor.sol: claim();
./contracts/core/PrivateSaleTokenDistributor.sol: claim();
./contracts/core/TokenDistributor.sol: claim();
./contracts/core/StrategicPool.sol: burnWithFormula(), burn();

Recommendation: follow the checks-effects-interactions pattern when interacting with external contracts.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1) (The mitigation step of introducing the nonReentrant modifier was performed, but the CEI violation was not resolved.

When CEI violations are resolved, the use of nonReentrant will be redundant and it will cost less Gas to call the function without it.)

M05. Division Before Multiplication

Impact	Low

Likelihood	High

The variable periodSinceLastClaim is calculated as a result of a division. Said variable is immediately multiplied afterward.

Since Solidity language does not have floating point numbers, performing divisions before multiplications results in a loss of precision.

Paths:
./contracts/core/Distributor.sol: _calculateClaimableAmount().
./contracts/core/TokenDistributor.sol: _calculateClaimableAmount().

Recommendation: it is recommended to perform divisions after multiplications to avoid loss of precision.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

M06. Missing Check: Loss of Funds

Impact	High
Likelihood	Low

In _submitPools, tokens are minted directly to the input pool addresses. However, there is no check that those addresses actually exist or are not the 0x0 address.

It is possible to lose funds if the 0x0 address or an un-existing pool address is used.

Path:
./contracts/core/MTC.sol: _submitPools();

Recommendation: it is recommended to add a check for each address to avoid 0x0, and that such pool addresses exist.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

M07. Wrong Event Data

Impact	Low
Likelihood	High

The function burnWithFormula() emits the event Burned with wrong parameters. Instead of passing the value of *amount*, it is using a hard-coded *1*.

Path:
./contracts/core/StrategicPool.sol: burnWithFormula()

Recommendation: pass correct values to event arguments.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

M08. Funds Lock

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

Some contracts accept Ether deposits but lack a withdrawal mechanism, which can result in funds being locked in the contract.

Paths:
./contracts/core/Distributor.sol
./contracts/core/TokenDistributor.sol

Recommendation: implement a withdrawal mechanism to allow the owner to retrieve deposited Ether if it is an expected behavior or remove ability to receive Ether.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

M09. Requirements Violation

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

A contradiction arises between the NatSpec notes and the content of several functions. The comments state the functions work for “a given address”, which is not reflected in the code.

The code should match the requirements provided by the customer.

Paths:
./contracts/core/Distributor.sol: getLeftClaimableAmount(), sweep(), _calculateClaimableAmount().
./contracts/core/PrivateSaleTokenDistributor.sol: sweep().
contracts/core/TokenDistributor.sol: sweep().

Recommendation: update the functions and/or their NatSpec so that they match.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

■ Low

L01. Unused Variables

Impact	Low
Likelihood	Medium

Unused variables are allowed in Solidity and do not pose a direct security issue. However, it is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease readability.

Paths:
./contracts/core/Distributor.sol : poolName, totalAmount;
./contracts/core/TokenDistributor.sol : poolName, totalAmount;

Recommendation: remove unused variables, or describe its usage.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L02. Floating Pragma

Impact	Medium
Likelihood	Low

As stated in SWC-103, contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Some contracts use Solidity 0.8.18 features, such as mapping key/values names and will not be compatible with previous versions.

Paths:
./contracts/*.sol

Recommendation: consider locking the pragma version in all contracts.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

L03. Missing Zero Address Validation

Impact	Low
Likelihood	Low

Additional checks against the 0x0 address should be included in the reported functions to avoid unexpected results.

Paths:
./contracts/core/PrivateSaleTokenDistributor.sol: setClaimableAmounts() → users[i];
./contracts/core/TokenDistributor.sol:setClaimableAmounts() → users[i];
./contracts/core/Distributor.sol: initialize() → token.
./contracts/core/TokenDistributor.sol: initialize() → token.
./contracts/utils/PoolFactory.sol: createTokenDistributor(), createDistributor() → token.
./contracts/core/LiquidityPool: constructor() → token.
./contracts/core/StrategicPool: constructor() → token.
./contracts/core/PrivateSaleTokenDistributor: constructor() → token.

Recommendation: it is recommended to add zero address checks.

Found in: 31a4e8c

Status: Fixed (Revised commit: 8b8d8e1)

L04. Missing Amount Validation

Impact	Low
Likelihood	Low

An additional check should be introduced in the function claim() to make sure that claimableAmount is not zero.

Path:
./contracts/core/PrivateSaleTokenDistributor.sol : claim();

Recommendation: consider adding a check that claimableAmount > 0.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L05. Missing NatSpecs: Burning Formula

Impact	Low
Likelihood	Low

The contract StrategicPool performs complex math operations to calculate the amount of tokens that should be burned in the *burnWithFormula()* function.

The calculations performed in the function are complex enough to require proper comments and documentation in code explaining how it works.

Path:
./contracts/core/StrategicPool.sol : calculateBurnAmount();

Recommendation: provide proper documentation in code about the calculations.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L06. Naming Consistency

Impact	Medium
Likelihood	Low

A variable named usersLength is used to represent _pools.length.

Using names that do not represent the variables can lead to confusion and decrease code readability.

Path:
./contracts/core/MTC.sol: _submitPools().

Recommendation: consider changing the name of usersLength to a new name that represents better _pools.length.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L07. Redundant Code: Unnecessary Getters

Impact	Medium
Likelihood	Low

Public variables do not need a getter function in order to be accessed. Unnecessary functions lead to bigger contract code and higher deployment costs.

Paths:
./contracts/core/Distributor.sol: getLeftClaimableAmount();
./contracts/core/StrategicPool.sol: getTotalBurnedAmount();
./contracts/core/TokenDistributor.sol: getLeftClaimableAmount();

Recommendation: remove redundant/unnecessary code.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L08. Variable That Should Be Constant

Impact	Medium

Likelihood	Low

Hard-coded variables that do not change their values during their lifecycle should be declared as constants in order to save Gas.

Path:
./contracts/core/StrategicPool.sol: S;

Recommendation: change variable to constant.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L09. Functions That Should Be External

Impact	Medium
Likelihood	Low

Public functions that are not called from inside the contract should be declared external to save Gas.

Paths:
./contracts/utils/DistributorProxyManager.sol: getPoolProxy(), addToWhitelist(), removeFromWhitelist();
./contracts/utils/MultiSigWallet.sol: submitTransaction(), confirmTransaction(), executeTransaction(), getOwners(), getTransactionCount(), getTransaction();
./contracts/utils/TokenDistributorProxyManager.sol: getPoolProxy(), addToWhitelist(), removeFromWhitelist();

Recommendation: change function visibility to external.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L10. Redundant SafeMath

Impact	Low
Likelihood	Low

The mentioned contract integrates the SafeMath library for uint256 while using the compiler ^0.8.0.

Prior to Solidity version 0.8.0, arithmetic overflows were not handled natively by the language, and developers were encouraged to use the SafeMath library as a safeguard against such errors.

However, with the release of Solidity version 0.8.0, the language introduced new arithmetic overflow and underflow protection features that made the SafeMath library redundant if using Solc versions above 0.8.0.

Path:
./contracts/core/StrategicPool.sol

Recommendation: consider removing the SafeMath integration.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L11. Non-Explicit Variable Visibilities

Impact	Low
Likelihood	Low

Variables without explicit visibility will be public by default.

Lack of variable visibility can lead to readability issues.

Path:
./contracts/core/StrategicPool.sol: n, S.

Recommendation: add explicit variables visibility consciously.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

L12. Wrong Import

Impact	Low
Likelihood	Medium

The upgradeable version of OpenZeppelin's Initialiable is imported instead of the regular one, which is more complex and thus increases the Gas cost unnecessarily.

Paths:
./contracts/core/Distributor.sol
./contracts/core/TokenDistributor.sol

Recommendation: import the regular Initializable contract instead of the upgradeable version.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

L13. Missing Check

Impact	Medium

Likelihood	Low

The function updatePoolParams() does not check that endTime > starTime.

The constructor() in PrivateSaleTokenDistributor does not check that endTime > starTime.

Paths:
./contracts/core/Distributor.sol: updatePoolParams().
./contracts/core/TokenDistributor.sol: updatePoolParams().
./contracts/core/PrivateSaleTokenDistributor: constructor().

Recommendation: add the missing check.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

L14. Repetitive Code

Impact	Low
Likelihood	Medium

The function updatePoolParams() introduces the check startTime > block.timestamp instead of reusing the modifier isSettable().

Paths:
./contracts/core/Distributor.sol: updatePoolParams().
./contracts/core/TokenDistributor.sol: updatePoolParams().

Recommendation: use isSettable() modifier.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

L15. Redundant Require Statement

Impact	Medium
Likelihood	Low

The function calculateClaimableAmount() introduces a redundant check *block.timestamp < endTime* in a “else” block scope that will not happen in block.timestamp > endTime.

As it is very unlikely that the user calls the method at the moment that block.timestamp == endTime, this check is too strict for time comparison and redundant due to its unlikelihood.

Redundant require statements lead to unnecessary Gas usage.

Path:

./contracts/core/TokenDistributor.sol: calculateClaimableAmount().

Recommendation: use block.timestamp >= endTime in the if case, and remove redundant require from the else case.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

L16. Redundant Code; Invalid Calculations

Impact	Low
Likelihood	Low

The *totalLockedAmount* calculations inside the *submitPools()* function are redundant as they are never used.

Additionally, there is an invalid calculation case in it; the function should increase the value of the variable totalLockedAmount according to the transferred amounts, but that is not always true. In the case that the pool address is 0x0, the contract skips the transfer but still increases the value of the variable.

Path:
./contracts/core/MTC.sol: submitPools();

Recommendation: remove redundant code.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

L17. Variables That Can Be Set Immutable

Impact	Low
Likelihood	Low

Use the *immutable* keyword on the *token* state variable to limit changes to its state and save Gas.

Paths:
./contracts/core/LiquidityPool
./contracts/core/StrategicPool
./contracts/core/PrivateSaleTokenDistributor

Recommendation: consider using the keyword immutable for said variable.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

Informational

I01. Solidity Style Guide: mixedCase

Local and State Variable names should be mixedCase: capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name.

Paths:
./contracts/core/Distributor.sol: PERIOD, DISTRIBUTION_RATE, BASE_DIVIDER;
./contracts/core/StrategicPool.sol: S.
./contracts/core/TokenDistributor.sol: PERIOD, DISTRIBUTION_RATE, BASE_DIVIDER;

Recommendation: follow the official Solidity guidelines.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

I02. Missing Events for Critical Value Updates

Events should be emitted after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract’s activity.

Paths:
./contracts/core/PrivateSaleTokenDistributor.sol: setClaimableAmounts();
./contracts/core/TokenDistributor.sol: setClaimableAmounts();
./contracts/utils/DistributorProxyManager.sol: addToWhitelist(), removeFromWhitelist();
contracts/utils/TokenDistributorProxyManager.sol: addToWhitelist(), removeFromWhitelist();
./contracts/core/Distributor: initialize() → PoolParamsUpdated().
./contracts/core/TokenDistributor: initialize() → PoolParamsUpdated().

Recommendation: consider emitting *events* in said functions.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

I03. Non-Explicit Variable Unit Sizes

Variable types uint and bytes are used without explicitly setting their size in the whole contract MultiSigWallet.

Paths:
./contracts/utils/MultiSigWallet.sol
./contracts/utils/TokenDistributorProxyManager.sol: createPoolProxy();
./contracts/utils/DistributorProxyManager.sol: createPoolProxy();

Recommendation: set variable size explicitly for uint.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

I04. Style Guide: Order of Functions

The provided projects should follow the official guidelines. Functions should be grouped according to their *visibility* and ordered:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External
5. Public
6. Internal
7. Private

Paths:
./contracts/core/Distributor.sol
./contracts/core/StrategicPool.sol.
./contracts/core/TokenDistributor.sol
./contracts/utils/DistributorProxyManager.sol
./contracts/utils/InitializedProxy.sol
./contracts/utils/TokenDistributorProxyManager.sol

Recommendation: follow the official Solidity guidelines.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

I05. Bad Variable Naming

Variables should have descriptive and conscious names. Some variable names in the project do not describe its function and cause confusion to readers.

Path:
./contracts/core/StrategicPool.sol : n, S, LP, MB, M;

Recommendation: give variables names consciously according to their functions, or document such variables in code.

Found in: 31a4e8c

Status: Fixed (Revised commit: 16b2fc4)

I06. Style Guide: Order of Layout

The provided projects should follow the official guidelines. Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables

3. Events
4. Modifiers
5. Functions

Paths:
./contracts/core/PrivateSaleTokenDistributor.sol
./contracts/core/TokenDistributor.sol

Recommendation: follow the official Solidity guidelines.

Found in: 16b2fc4

Status: Mitigated

I07. Redundant Function Call

The calls to _transferOwnership() in contracts that inherit Ownable2Step are redundant since ownership is already set to the deployer during that contract constructor.

Paths:
./contracts/core/LiquidityPool.sol: constructor().
./contracts/core/MTC.sol: constructor().
./contracts/core/StrategicPool: constructor().
./contracts/core/PrivateSaleTokenDistributor.sol: constructor().
./contracts/utils/PoolFactory.sol: constructor().

Recommendation: it is recommended to remove the redundant call to _transferOwnership().

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

I08. Unused Code

The event CanClaim() is unused and thus should be removed from the code.

Path:
./contracts/core/Distributor.sol: CanClaim().

Recommendation: it is recommended to remove unused code.

Found in: 16b2fc4

Status: Fixed (Revised commit: 8b8d8e1)

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.

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 won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

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/Metatime-Technology-Inc/pool-contracts
Commit	31a4e8c
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	File: contracts/core/Distributor.sol SHA3: 6d4f231cab4ad5b13d78f2f96454593d88b0f157db2d14a7353629a9fc26371f File: contracts/core/LiquidityPool.sol SHA3: d9ee7308dee55c8552ae0ffa5b28c905ac98ce31fa18a03e5e93875c5a8cf8e6 File: contracts/core/MetatimeToken.sol SHA3: 9fae5927cd569e69eaf54d7e78b12ad30f8319ed6db267487fbd9531892b52d0 File: contracts/core/MTC.sol SHA3: 34555276e470ff9fbd7d533f56ba1d74273ffc322b0fc5be5e7b056c05f1b5cb File: contracts/core/PrivateSaleTokenDistributor.sol SHA3: d5f317ae5001fa1d98e7ae440a60ffcaa6b21fcfc60fb4ed707ab67dd3c2b281 File: contracts/core/StrategicPool.sol SHA3: 4a25c61aeb987a9719b052dc3d26dc63300650b8c44079b814f5c9179913f90a File: contracts/core/TokenDistributor.sol SHA3: fdf8399853595350738b0c6ece187b47d0c568e5c6c45d00a528658b5b3701fc File: contracts/interfaces/IMetatimeToken.sol SHA3: 7da6c4e7bf7ef1406b7c5d27096dbca3ce0a9c164cb7c95d6416135a2345dfb9 File: contracts/libs/Trigonometry.sol SHA3: 161073a88c43a3e6698e696df15b8cc6c4a9c9e1c3a3ef63ce068aaf5920c05c File: contracts/utils/DistributorProxyManager.sol SHA3: ba2a7e8f71e3353518c37f7c9a2bac729e00bb1df40a4b678eac878e4317503a File: contracts/utils/InitializedProxy.sol SHA3: abc17b68cf590f1e259a6c7e5f74cff96eb4ed0b48978d303d34ccb300ffdc80 File: contracts/utils/MultiSigWallet.sol SHA3: 319d1422b2039a249d01cbf117e07187d0cc78c95920e2e2d2acde604095a0ea File: contracts/utils/TokenDistributorProxyManager.sol SHA3: 28deeac92c6db8b940245c27a3d54e34833fa9bf8415233a792c786216654d09

Second review scope

Repository	https://github.com/Metatime-Technology-Inc/pool-contracts
Commit	16b2fc4
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	File: contracts/core/Distributor.sol SHA3: 734f96a52557c9f525f20ab4fbff48f290333b3908439466d2768c8eddc8dd72 File: contracts/core/LiquidityPool.sol SHA3: 6c1b4bbf7b3eadcc0a46942abc1b95e5ecd7c1830049cb24856a4b567c7b5ebf File: contracts/core/MTC.sol SHA3: df2567b143bf3963c13bea985025f9f0ca4e001559ffb627ec9fc9ecce2361d6 File: contracts/core/PrivateSaleTokenDistributor.sol SHA3: 8f9a4747cf3d254c83ce6d5d4b61917f3479f080c16ef5ac5c00f4d67dd7b35f File: contracts/core/StrategicPool.sol SHA3: fbf50528b22099a671910be8f90e73bc74800521fe71d82088e8da43355aac0d File: contracts/core/TokenDistributor.sol SHA3: e30856c4f4a42035d41861882036b41a83a85dd0b0d9c4a91df81e649ea32dec File: contracts/interfaces/IDistributor.sol SHA3: fc2baa2c2e25b363dece2e94ee300ac0bf0d8c3714c14447f400900047bfeb4d File: contracts/interfaces/IMTC.sol SHA3: b96e803785d4b5b98d68cd7fbc07237989091694d10a15e8c3a8430bc5712de0 File: contracts/interfaces/ITokenDistributor.sol SHA3: 37cdd7900f05ae6b94695dc5c19e58525ce8c03d12d0e2ed5099c46e0fdba8c7 File: contracts/libs/Trigonometry.sol SHA3: 55de5daea153ae0715d2f0edd243065700559650b2b178882838d568d55eecf9 File: contracts/utils/MultiSigWallet.sol SHA3: 86ee2cc45bdbd85fd3bb81ee06d0b0780ee17ee177919fb372d68126b8baa3eb File: contracts/utils/PoolFactory.sol SHA3: a58fce4d056a6dc32dc3d2476b93fc93ee71bd5a1d68f68a68b84865ff208652

Third review scope

Repository	https://github.com/Metatime-Technology-Inc/pool-contracts
Commit	8b8d8e1
Whitepaper	Link
Requirements	Link

Technical Requirements	Link
Contracts	File: contracts/core/Distributor.sol SHA3: ebcc474de0a16c8893fec0fe3a8138f06f2670007f707c3107eb8f9be9cd43e1 File: contracts/core/LiquidityPool.sol SHA3: 58da818a293e1c2d1b6b3726bd1d7c9db1369289eedb6932794f21ec3fa82822 File: contracts/core/MTC.sol SHA3: 8513c2b3466a0f5d696ce16421db5b85ba4e48265faaa1ee802132531ec07f47 File: contracts/core/StrategicPool.sol SHA3: 947ad170d56636506c822e9150aa187861c735a22f0459b53dc0822992ca88c2 File: contracts/core/TokenDistributor.sol SHA3: 45571bc526916fb19168f1a0ab82521020eec7efd752949327d1bef8aa64f079 File: contracts/core/TokenDistributorWithNoVesting.sol SHA3: cd8c4681b6ac0644cab3da65a3f32942e4b45e4b6e56e411cb677132322c842d File: contracts/interfaces/IDistributor.sol SHA3: 8859352023ac2abb134e2055bcf9219ed0419bfecce42d333d8008da9b9221b5 File: contracts/interfaces/IMTC.sol SHA3: 25abe8bc06d7904fe412078dda0bf2f433653d9ce32dd45f902caa36380e9dc8 File: contracts/interfaces/ITokenDistributor.sol SHA3: 71cec0c60c87efeb39d3b8a3720b5a31d1973313446849e0550801449763813f File: contracts/libs/Trigonometry.sol SHA3: 260290f6fc7484cbf53b745e7a26b07a32988ef7e5fbd84f7c18ad296cffd3cf File: contracts/utils/PoolFactory.sol SHA3: 7da934f46ec62d5d53ec1bf12310c47f333b0994ec7faaf36fd47039de98ca68