Token Generation

Smart Contract Audit Report
Prepared for EvryNet

Date Issued: Nov 9, 2021
Project ID: AUDIT2021018-2
Version: v2.0
Confidentiality Level: Public

Report Information

Project ID	AUDIT2021018-2
Version	v2.0
Client	EvryNet
Project	Token Generation
Auditor(s)	Weerawat Pawanawiwat Suvicha Buakhom Patipon Suwanbol Peeraphut Punsuwan
Author	Peeraphut Punsuwan Patipon Suwanbol
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Nov 9, 2021	Update issue id	Peeraphut Punsuwan
1.0	Oct 29, 2021	Full report	Peeraphut Punsuwan Patipon Suwanbol

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by EvryNet, Inspex team conducted an audit to verify the security posture of the Token Generation smart contracts between Sep 20, 2021 and Sep 30, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Token Generation smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 medium, 1 very low, and 4 info-severity issues. With the mitigation solutions and fixes confirmed by the project team, while 1 very low issue was acknowledged by the team. Therefore, Inspex trusts that Token Generation smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

EvryNet is an intelligent financial services platform providing infrastructure that enables developers and businesses to build an unlimited number of Centralised/Decentralised Finance (CeDeFi) applications, interoperable with many of the world's leading blockchains for "evryone".

Token Generation manages the generation and distribution of $EVRY via different distribution models.

Scope Information:

Project Name	Token Generation
Website	https://evrynet.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum, Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 20, 2021 - Sep 30, 2021
Reassessment Date	Oct 14, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 4120ed4afaf2ba01b222f469498922277d47cd73)

Contract	Location (URL)
PerformanceDistribution	https://github.com/Evry-Finance/evry-finance-toolkit/blob/4120ed4afa/contracts/wallet/PerformanceDistribution.sol
ReleaseController	https://github.com/Evry-Finance/evry-finance-toolkit/blob/4120ed4afa/contracts/wallet/ReleaseController.sol
TimeBasedDistribution	https://github.com/Evry-Finance/evry-finance-toolkit/blob/4120ed4afa/contracts/wallet/TimeBasedDistribution.sol
EvryToken	https://public.inspex.co/audit/EvryNet_TokenGeneration/EvryToken.sol
Timelock	https://github.com/Evry-Finance/evry-finance-toolkit/blob/4120ed4afa/contracts/Timelock.sol

Please note that the EvryToken contract was initially in EvryNet's private repository. The file has been uploaded to Inspex's storage for public access.

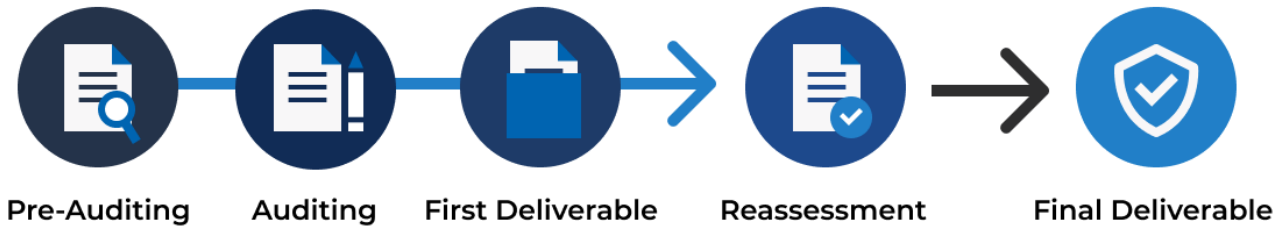
Reassessment: (Commit: 166f5ba9b610cccb6b349300e2227de74c1b44a7)

Contract	Location (URL)
PerformanceDistribution	https://github.com/Evry-Finance/evry-finance-toolkit/blob/166f5ba9b6/contracts/wallet/PerformanceDistribution.sol
ReleaseController	https://github.com/Evry-Finance/evry-finance-toolkit/blob/166f5ba9b6/contracts/wallet/ReleaseController.sol
TimeBasedDistribution	https://github.com/Evry-Finance/evry-finance-toolkit/blob/166f5ba9b6/contracts/wallet/TimeBasedDistribution.sol
EvryToken	https://github.com/Evry-Finance/evry-token/blob/5a2f08fd76/contracts/EvryToken.sol
Timelock	https://github.com/Evry-Finance/evry-finance-toolkit/blob/166f5ba9b6/contracts/Timelock.sol

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Use of Upgradable Contract Design
Improper Kill-Switch Mechanism

Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

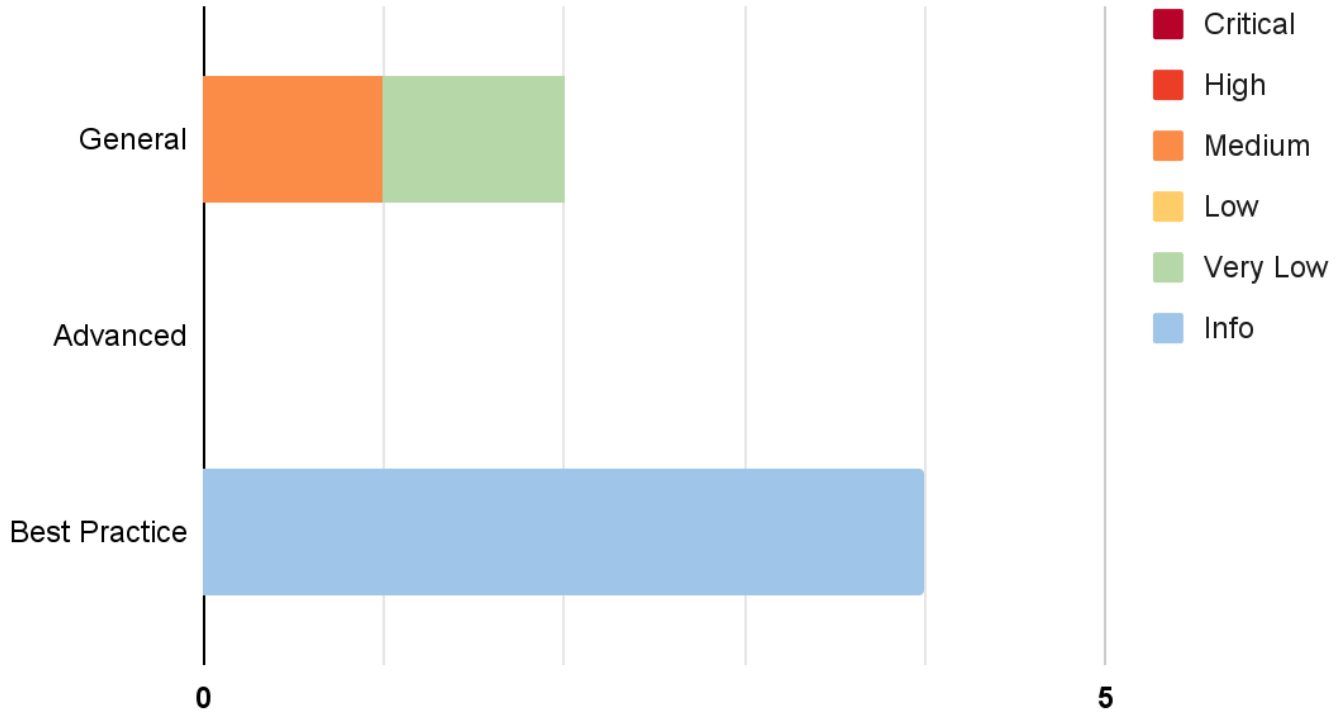
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 6 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	**Medium**	Resolved *
IDX-002	Insufficient Logging for Privileged Functions	General	**Very Low**	**Acknowledged**
IDX-003	Unused Function Parameter	Best Practice	**Info**	Resolved
IDX-004	Improper Function Visibility	Best Practice	**Info**	No Security Impact
IDX-005	Code Improvement in _getPeriodTimes() Function	Best Practice	**Info**	Resolved
IDX-006	Inexplicit Solidity Compiler Version	Best Practice	**Info**	No Security Impact

* The mitigations or clarifications by EvryNet can be found in Chapter 5.

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	PerformanceDistribution ReleaseController TimeBasedDistribution
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, these actions can only be performed by the contract owner.
Status	Resolved * EvryNet team has confirmed that timelock will be used to own the contracts. This will allow the users to monitor the changes to the smart contracts and act accordingly before the changes are applied. The smart contracts are not yet deployed at the time of the reassessment. Therefore, Inspex suggests the platform users to confirm the usage of the timelock before using the platform.

5.1.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
PerformanceDistribution.sol (L:13)	PerformanceDistribution	addReward()	onlyOwner
PerformanceDistribution.sol	PerformanceDistribution	removeMember()	onlyOwner

(L:141)			
PerformanceDistribution.sol (L:194)	PerformanceDistribution	transferExceedAmount()	onlyOwner
ReleaseController.sol (L:78)	ReleaseController	transferExceedAmount()	onlyOwner
TimeBasedDistribution.sol (L:85)	TimeBasedDistribution	transferExceedAmount()	onlyOwner
TimeBasedDistribution.sol (L:92)	TimeBasedDistribution	addMember()	onlyOwner
TimeBasedDistribution.sol (L:215)	TimeBasedDistribution	removeMemberAllocation()	onlyOwner
TimeBasedDistribution.sol (L:242)	TimeBasedDistribution	updateMemberAllocation()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L:54)	PerformanceDistribution, ReleaseController, TimeBasedDistribution	renounceOwnership()	onlyOwner
@openzeppelin/contracts/access/Ownable.sol (L:63)	PerformanceDistribution, ReleaseController, TimeBasedDistribution	transferOwnership()	onlyOwner

Please note that the `Ownable` contract is inherited from the OpenZeppelin library.

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a `Timelock` contract to delay the changes for a sufficient amount of time, e.g., 24 hours

5.2. Insufficient Logging for Privileged Functions

ID	IDX-002
Target	PerformanceDistribution ReleaseController TimeBasedDistribution
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	**Acknowledged** EvryNet team has acknowledged this issue and decided not to fix it in this release.

5.2.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts to the platform.

For example, the owner can transfer an excess amount by executing `transferExceedAmount()` function in the `PerformanceDistribution` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
PerformanceDistribution.sol (L:194)	PerformanceDistribution	transferExceedAmount()	onlyOwner
ReleaseController.sol (L:78)	ReleaseController	transferExceedAmount()	onlyOwner
TimeBasedDistribution.sol (L:85)	TimeBasedDistribution	transferExceedAmount()	onlyOwner

5.2.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

ReleaseController.sol

```
249 event TransferExceedAmount(address _to);
250   function transferExceedAmount(address _to) external onlyOwner {
251     require(_to != address(0), "PerformanceDistribution: cannot transfer exceed
    amount to zero address");
252     uint256 totalBalance = token.balanceOf(address(this)).add(distributed);
253     require(totalBalance > distributionCap, "PerformanceDistribution: balance
    is not exceed");
254     emit TransferExceedAmount(address _to);
255     token.safeTransfer(_to, totalBalance.sub(distributionCap));
256   }
```

5.3. Unused Function Parameter

ID	IDX-003
Target	PerformanceDistribution
Category	Smart Contract Best Practice
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved EvryNet team has resolved this issue in commit 166f5ba9b610cccb6b349300e2227de74c1b44a7 by modifying the `addReward()` function and using the `lastUpdateBlock` parameter in the `DistributionInfo` struct.

5.3.1. Description

In the `PerformanceDistribution` contract, the `lastUpdateBlock` parameter is accepted in the `addReward()` function, and the value of the parameter is checked in conditions at lines 72 and 73. However, this parameter is not used anywhere else in the function, and does not have any effect on the state of the contract.

PerformanceDistribution.sol

```
61  function addReward(
62      address _for,
63      uint256 amount,
64      uint256 lastUpdateBlock
65  ) external nonReentrant onlyOwner {
66      require(token.balanceOf(address(this)) >= amount, "addReward:: exceed
    balance");
67      require(distributionBalance >= amount, "addReward:: exceed balance");
68      require(_for != address(0), "addReward: invalid address");
69
70      uint256 minimumLastUpdateBlock = block.number
    .sub(blockPerDay.mul(distributionFrequency));
71      uint256 maximumLastUpdateBlock = block.number
    .add(blockPerDay.mul(distributionFrequency));
72      require(lastUpdateBlock >= minimumLastUpdateBlock, "addReward: exceed min
    lastUpdateBlock");
73      require(lastUpdateBlock <= maximumLastUpdateBlock, "addReward: exceed max
    lastUpdateBlock");
74
75      uint256 rewardPerPeriod = amount.div(distributionPeriod);
```

```
76
77      if (distributionInfo[_for].length == 0) {
78          members.push(_for);
79      }
80
81      distributionInfo[_for].push(
82          DistributionInfo({
83              amount: amount,
84              remainingAmount: amount,
85              rewardPerPeriod: rewardPerPeriod,
86              lastUpdateBlock: block.number
87          })
88      );
89
90      distributionBalance = distributionBalance.sub(amount);
91
92      emit LogAddReward(_for, amount, rewardPerPeriod, lastUpdateBlock);
93  }
```

5.3.2. Remediation

Inspex suggests editing the function to follow the intended business design, for example:

PerformanceDistribution.sol

```
61  function addReward(
62      address _for,
63      uint256 amount,
64      uint256 lastUpdateBlock
65  ) external nonReentrant onlyOwner {
66      require(token.balanceOf(address(this)) >= amount, "addReward:: exceed
    balance");
67      require(distributionBalance >= amount, "addReward:: exceed balance");
68      require(_for != address(0), "addReward: invalid address");
69
70      uint256 minimumLastUpdateBlock = block.number
    .sub(blockPerDay.mul(distributionFrequency));
71      uint256 maximumLastUpdateBlock = block.number
    .add(blockPerDay.mul(distributionFrequency));
72      require(lastUpdateBlock >= minimumLastUpdateBlock, "addReward: exceed min
    lastUpdateBlock");
73      require(lastUpdateBlock <= maximumLastUpdateBlock, "addReward: exceed max
    lastUpdateBlock");
74
75      uint256 rewardPerPeriod = amount.div(distributionPeriod);
76
77      if (distributionInfo[_for].length == 0) {
78          members.push(_for);
79      }
80
81      distributionInfo[_for].push(
82          DistributionInfo({
83              amount: amount,
84              remainingAmount: amount,
85              rewardPerPeriod: rewardPerPeriod,
86              lastUpdateBlock: lastUpdateBlock
87          })
88      );
89
90      distributionBalance = distributionBalance.sub(amount);
91
92      emit LogAddReward(_for, amount, rewardPerPeriod, lastUpdateBlock);
93  }
```

5.4. Improper Function Visibility

ID	IDX-004
Target	Timelock
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact EvryNet team has acknowledged this best practice recommendation.

5.4.1. Description

Functions with `public` visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

For example, the `pendingAdminConfirm()` function has `public` visibility, but it is never called from any internal function.

Timelock.sol

```
44  function pendingAdminConfirm() public {
45      require(msg.sender == pendingAdmin, "Timelock pendingAdminConfirm must call
    from pendingAdmin");
46      admin = msg.sender;
47      pendingAdmin = address(0);
48
49      emit NewAdmin(admin);
50  }
```

The following table contains all functions that have `public` visibility and are never called from any internal function:

File	Contract	Function
Timelock.sol (L: 44)	Timelock	pendingAdminConfirm()
Timelock.sol (L: 52)	Timelock	setPendingAdmin()
Timelock.sol (L: 59)	Timelock	queueTransaction()

Timelock.sol (L: 79)	Timelock	cancelTransaction()
Timelock.sol (L: 90)	Timelock	executeTransaction()

5.4.2. Remediation

Inspex suggests changing all functions' visibility to `external` if they are not called from any internal function as shown in the following example:

Timelock.sol

```
44  function pendingAdminConfirm() external {
45      require(msg.sender == pendingAdmin, "Timelock pendingAdminConfirm must call
    from pendingAdmin");
46      admin = msg.sender;
47      pendingAdmin = address(0);
48
49      emit NewAdmin(admin);
50  }
```

5.5. Code Improvement in _getPeriodTimes() Function

ID	IDX-005
Target	PerformanceDistribution
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved EvryNet team has resolved this issue in commit 166f5ba9b610cccb6b349300e2227de74c1b44a7 by modifying the `_getPeriodTimes()` function to calculate without recursion.

5.5.1. Description

In the `distributeReward()` function of the `PerformanceDistribution` contract, the `_getPeriodTimes()` is used to calculate the number of remaining periods at line 99.

PerformanceDistribution.sol

```
95  function distributeReward() external nonReentrant {
96      require(block.number >= nextDistributionBlock, "distributeReward: already
    update");
97
98      // get remaining period
99      uint256 periodTimes = _getPeriodTimes
    (block.number.sub(nextDistributionBlock), 1);
100
101     // distribute by one period * remaining period (times)
102     for (uint256 i = 0; i < periodTimes; i++) {
103         for (uint256 j = 0; j < members.length; j++) {
104             _distributeReward(members[j]);
105         }
106
107         // update block before next period distribution
108         nextDistributionBlock = nextDistributionBlock
    .add(blockPerDay.mul(distributionFrequency));
109     }
110 }
```

The `_getPeriodTimes()` function is a recursive function used to determine the most number of periods that do not exceed the number of blocks provided in the parameter, plus 1.

PerformanceDistribution.sol

```
131 function _getPeriodTimes(uint256 blockNum, uint256 times) internal returns
    (uint256) {
132     uint256 blockPerPeriod = blockPerDay.mul(distributionFrequency);
133
134     if (blockPerPeriod.mul(times) <= blockNum) {
135         return _getPeriodTimes(blockNum, times.add(1));
136     } else {
137         return times;
138     }
139 }
```

However, these recursive callings of the function have redundant calculations that could be simplified for better readability and gas saving.

5.5.2. Remediation

Inspex suggests improving the `_getPeriodTimes()` function by calculating the value in one function calling instead of using recursion, for example:

PerformanceDistribution.sol

```
131 function _getPeriodTimes(uint256 blockNum) internal returns (uint256) {
132     uint256 blockPerPeriod = blockPerDay.mul(distributionFrequency);
133
134     return blockNum.div(blockPerPeriod).add(1);
135 }
```

5.6. Inexplicit Solidity Compiler Version

ID	IDX-006
Target	Timelock
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact EvryNet team has acknowledged this best practice recommendation.

5.6.1. Description

The Solidity compiler version declared in the `Timelock` contracts was not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

Timelock.sol

```
1 //contracts/Timelock.sol
2 // SPDX-License-Identifier: MIT
3
4 pragma solidity ^0.7.6;
```

5.6.2. Remediation

Inspex suggests fixing the solidity compiler of the `Timelock` contract to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.7 is v0.7.6, for example.

Timelock.sol

```
1 //contracts/Timelock.sol
2 // SPDX-License-Identifier: MIT
3
4 pragma solidity 0.7.6;
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE