Token

Smart Contract Audit Report
Prepared for Megapad

Date Issued: Jan 31, 2022
Project ID: AUDIT2021045
Version: v1.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021045
Version	v1.0
Client	Megapad
Project	Token
Auditor(s)	Patipon Suwanbol Ronnachai Chaipha Puttimet Thammasaeng
Author	Puttimet Thammasaeng
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jan 31, 2022	Full report	Puttimet Thammasaeng

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 Megapad, Inspex team conducted an audit to verify the security posture of the Token smart contracts between Jan 10, 2022 and Jan 11, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Token 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 2 medium, 1 low, 1 very low, and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved in the reassessment. Therefore, Inspex trusts that Token smart contracts have high-level protections in place to be safe from most attacks.

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

The Megapad is a decentralized finance platform aiming to provide easier access to funding and alternative investment options via launchpad and beyond.

The Megapad Token is the governance token of the Megapad platform that is based on OpenZeppelin's ERC20 with slow-release mechanism, which will be further integrated to other features in the future.

Scope Information:

Project Name	Token
Website	https://megapad.app
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 10, 2022 - Jan 11, 2022
Reassessment Date	Jan 27, 2022

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: 0807d0ae5ee58f18f9f18943d7c5b03c391bcf3e)

Contract	Location (URL)
MegapadToken.sol	https://github.com/megapadofficial/mep-token/blob/0807d0ae5e/contracts/MegapadToken.sol
ERC20.sol	https://github.com/megapadofficial/mep-token/blob/0807d0ae5e/contracts/lib/ERC20/ERC20.sol
ERC20Capped.sol	https://github.com/megapadofficial/mep-token/blob/0807d0ae5e/contracts/lib/ERC20/ERC20Capped.sol
ERC20SlowRelease.sol	https://github.com/megapadofficial/mep-token/blob/0807d0ae5e/contracts/lib/ERC20/ERC20SlowRelease.sol

Reassessment: (Commit: ded74b4a8b87849a826b9edf7d770650b1d3d512)

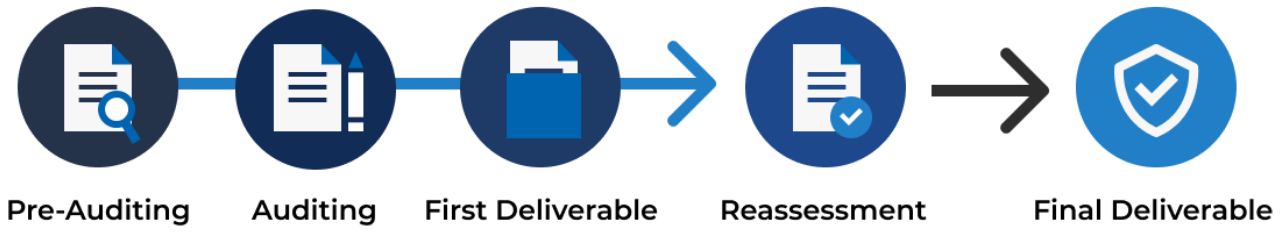
Contract	Location (URL)
MegapadToken.sol	https://github.com/megapadofficial/mep-token/blob/ded74b4a8b/contracts/MegapadToken.sol
ERC20.sol	https://github.com/megapadofficial/mep-token/blob/ded74b4a8b/contracts/lib/ERC20/ERC20.sol
ERC20Capped.sol	https://github.com/megapadofficial/mep-token/blob/ded74b4a8b/contracts/lib/ERC20/ERC20Capped.sol
ERC20SlowRelease.sol	https://github.com/megapadofficial/mep-token/blob/ded74b4a8b/contracts/lib/ERC20/ERC20SlowRelease.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

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
Use of Upgradable Contract Design
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
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

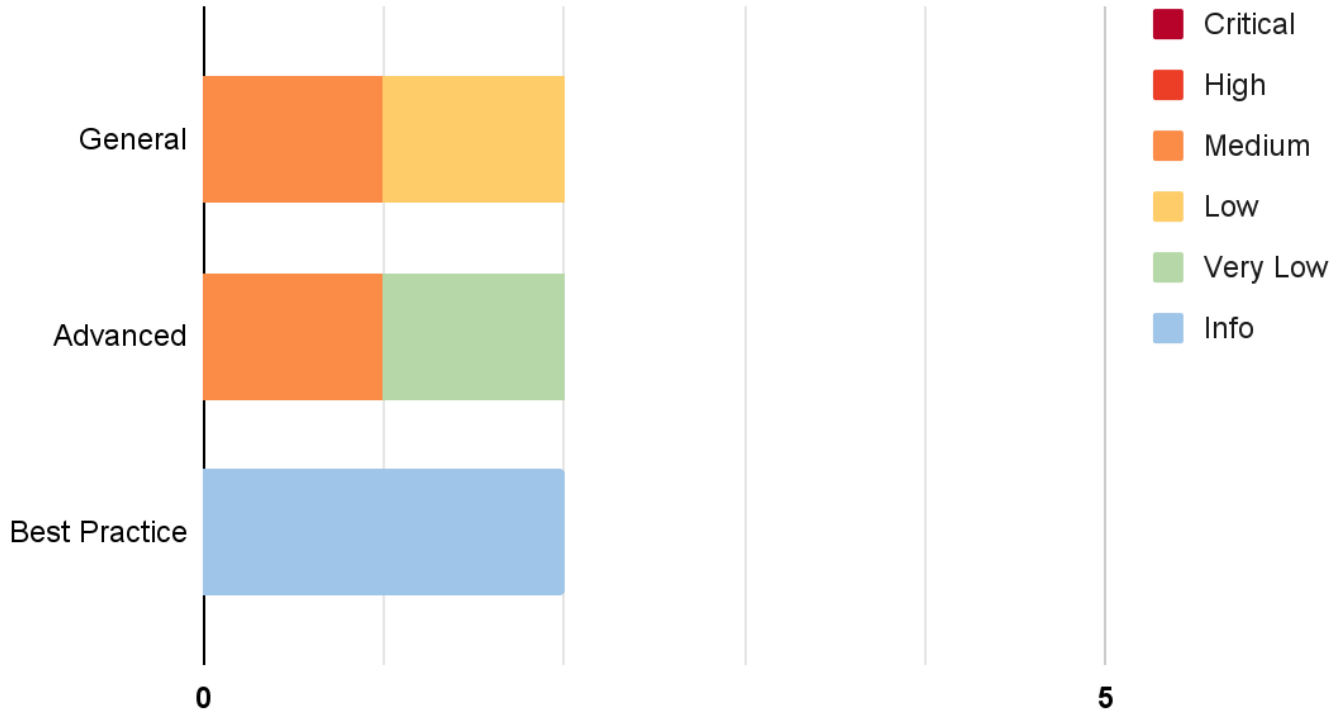
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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	Design Flaw in getUnlockedToClaim Function	Advanced	**Medium**	Resolved
IDX-003	Loop Over Unbounded Data Structure	General	**Low**	Resolved
IDX-004	Insufficient Logging for Privileged Functions	Advanced	**Very Low**	Resolved
IDX-005	Improper Function Visibility	Best Practice	**Info**	Resolved
IDX-006	Inexplicit Solidity Compiler Version	Best Practice	**Info**	Resolved

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

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	MegapadToken
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authority 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, this action can only be done by the contract owner.
Status	Resolved The MegapadToken team has resolved this issue in the commit `ded74b4a8b87849a826b9edf7d770650b1d3d512` by adding the `effectLockTime()` function to delay the change of `lockTime` state.

5.1.1. Description

The `lockTime` state variable can be updated any time by the controlling authority.

ERC20SlowRelease.sol

```
83 function setLockTime(uint256 _lockTime) public onlyOwner {
84     lockTime = _lockTime;
85 }
```

Changes in this variable can cause impacts to the users as claiming the amount of the locked token relies on the `lockTime` state variable, so the users should accept or be notified before the change is effective.

However, there is currently no constraint to prevent the authority from modifying this variable without notifying the users.

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. Inspex encourages removing the `setLockTime()` function.

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 reasonable amount of time

5.2. Design Flaw in getUnlockedToClaim Function

ID	IDX-002
Target	MegapadToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High All locked balances of several users will be frozen until the `block.timestamp` reaches the end time of lock balances (`_lockInfo[account].endTimestamp`). This means the platform's users will not be able to claim their partial unlocked token balances and the pool (`onlyPools`) will not be able to transfer tokens with lock through `transferWithLock()` to any users that the lock duration has not expired. **Likelihood:** Low This flaw can be encountered when `lockTime` is set as 0, and the lock duration of target address, the token receiver, is not ended.
Status	Resolved The MegapadToken team has resolved this issue in commit `ded74b4a8b87849a826b9edf7d770650b1d3d512` by adding the condition in the `setLockTime()` function to ensure that the `_lockTime` parameter won't be zero.

5.2.1. Description

The `getUnlockedToClaim()` function is a function to calculate the unlocked token amount that users could receive.

ERC20SlowRelease.sol

```
48 function getUnlockedToClaim(address account) public view returns (uint256) {
49     if(block.timestamp > _lockInfo[account].endTimestamp){
50         return _balancesLock[account];
51     }
52     else{
53         uint256 progressTime =
   block.timestamp.sub(_lockInfo[account].lastClaimTimestamp);
54         return _lockInfo[account].amount.mul(progressTime).div(lockTime);
55     }
56 }
```

The contract owner could possibly set the `lockTime` to be any value through the `setLockTime()` function such as setting it as 0.

ERC20SlowRelease.sol

```
83 function setLockTime(uint256 _lockTime) public onlyOwner {
84     lockTime = _lockTime;
85 }
```

This means when the platform's users claim a partial unlocked balance, the transaction will be reverted if `lockTime` was set to 0 due to the division by zero.

With the current design, the `getUnlockedToClaim()` function is executed when users and the authorized parties perform the `claimUnlocked()` and the `transferWithLock()` functions respectively.

As a result, when the contract owner set `lockTime` to 0. Any users will not be able to claim their partial unlocked balance, and also the pools will not be able to execute the `transferWithLock()` function to the user whose lock duration has not expired.

5.2.2. Remediation

Inspex suggests avoiding dividing the value that could possibly be set as 0. According to the platform's business design, the `lockTime` state variable could be set to 0 in the future to disable the `transferWithLock()` function.

So, this can be achieved by calculating the partial unlocked balance without using the `lockTime` state variable, for example:

ERC20SlowRelease.sol

```
48 function getUnlockedToClaim(address account) public view returns (uint256) {
49     if(block.timestamp >= _lockInfo[account].endTimestamp){
50         return _balancesLock[account];
51     }
52     else{
53         uint256 progressTime =
   block.timestamp.sub(_lockInfo[account].lastClaimTimestamp);
54         uint256 totalLockTime =
   _lockInfo[account].endTimestamp.sub(_lockInfo[account].lastClaimTimestamp);
55         return _lockInfo[account].amount.mul(progressTime).div(totalLockTime);
56     }
57 }
```

5.3. Loop Over Unbounded Data Structure

ID	IDX-003
Target	MegapadToken
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `removePool()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `pools_array` size will be raised until the `removePool()` is eventually unusable.
Status	Resolved The MegapadToken team has resolved this issue as recommended in the commit `ded74b4a8b87849a826b9edf7d770650b1d3d512`.

5.3.1. Description

The implemented `ERC20SlowRelease` abstract contract is a `ERC20` token standard with a mechanism to lock a portion of tokens when specific authorities distribute tokens to the platform's users, for example, transferring the token through the `transferWithLock()` function by a list of pool (the `onlyPools` modifier).

ERC20SlowRelease.sol

```
27 modifier onlyPools {
28     require(pools[msg.sender], "Not in pools member.");
29     _;
30 }
```

ERC20SlowRelease.sol

```
93 function transferWithLock(address recipient, uint256 amount) public onlyPools
   returns (bool) {
94         require(amount > 0, "amount: invalid amount");
95         _transferWithLock(_msgSender(), recipient, amount);
96         return true;
97 }
```

A list of pool addresses can be adjusted through the `addPool()` and `removePool()` functions.

ERC20SlowRelease.sol

```
165 // Add new Pool
166 function addPool(address pool_address) public onlyOwner {
167     require(pools[pool_address] == false, "poolExisted");
168     pools[pool_address] = true;
169     pools_array.push(pool_address);
170     emit PoolAdded(pool_address);
171 }
```

ERC20SlowRelease.sol

```
173 // Remove a pool
174 function removePool(address pool_address) public onlyOwner {
175     require(pools[pool_address] == true, "!pool");
176     // Delete from the mapping
177     delete pools[pool_address];
178     // 'Delete' from the array by setting the address to 0x0
179     for (uint256 i = 0; i < pools_array.length; i++) {
180         if (pools_array[i] == pool_address) {
181             pools_array[i] = address(0); // This will leave a null in the array
    and keep the indices the same
182             break;
183         }
184     }
185     emit PoolRemoved(pool_address);
186 }
```

However, with the current design of the `removePool()` function, the pool address is not truly removed from the `pools_array` state variable, it is instead marked as `address(0)`.

In addition, when adding a pool address through the `addPool()` function, the added address will be recorded as the last index of the `pools_array` state variable.

Hence, if new pools continue to be added to this contract, the loop iteration when executing the `removePool()` will keep enlarging and this function will eventually be unusable due to excessive gas usage.

5.3.2. Remediation

Inspex suggests making the contract capable of removing unnecessary/ended pools to reduce the loop round in the `removePool()` function.

For example, changing the data structure of the `pools` state variable from `mapping(address => bool) public pools` to `mapping(address => uint256) public pools` so that the mapping will be the pool address with the index in the `pools_array`.

ERC20SlowRelease.sol

```
23  mapping(address => uint256) public pools;
```

This means the `removePool()` function will no longer require the loop iteration to find the index in `pools_array` in order to remove it.

ERC20SlowRelease.sol

```
165 // Add new Pool
166 function addPool(address pool_address) public onlyOwner {
167     require(pools[pool_address] == 0, "poolExisted
168     pools_array.push(pool_address);
169     pools[pool_address] = pools_array.length;
170     emit PoolAdded(pool_address);
171 }
```

ERC20SlowRelease.sol

```
173 // Remove a pool
174 function removePool(address pool_address) public onlyOwner {
175     require(pools[pool_address] > 0, "pool not exist");
176     // 'Delete' from the array by setting the address to 0x0
177     pools_array[pools[pool_address] - 1] = address(0);
178     // Delete from the mapping
179     delete pools[pool_address];
180     emit PoolRemoved(pool_address);
181 }
```

Please note that the proposed example is concerned on the audited scope only, it might conflict with other existing contracts on the platform.

5.4. Insufficient Logging for Privileged Functions

ID	IDX-004
Target	MegapadToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged function's executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged function will be a malicious action.
Status	Resolved The MegapadToken team has resolved this issue as recommended in the commit `ded74b4a8b87849a826b9edf7d770650b1d3d512`.

5.4.1. Description

A privileged function that is executable by the controlling party is not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of this privileged function, allowing the controlling party to perform actions that cause big impacts on the platform.

For example, the owner can set the token lock time by executing the `setLockTime()` function in the `MegapadToken` contract, and no events are emitted.

ERC20SlowRelease.sol

```
83  function setLockTime(uint256 _lockTime) public onlyOwner {
84      lockTime = _lockTime;
85  }
```

5.4.2. Remediation

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

ERC20SlowRelease.sol

```
82  event SetLockTime(uint256 _lockTime);
83  function setLockTime(uint256 _lockTime) public onlyOwner {
84      lockTime = _lockTime;
85      emit SetLockTime(_lockTime);
86  }
```

5.5. Improper Function Visibility

ID	IDX-005
Target	MegapadToken
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The MegapadToken team has resolved this issue as recommended in the commit `ded74b4a8b87849a826b9edf7d770650b1d3d512`.

5.5.1. 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.

The following source code shows that the `transfer()` function of the MEP token is set to public and it is never called from any internal function.

ERC20.sol

```
113 function transfer(address recipient, uint256 amount) public virtual override
    returns (bool) {
114     _transfer(_msgSender(), recipient, amount);
115     return true;
116 }
```

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

Target	Function
ERC20.sol (L:113)	transfer()
ERC20.sol (L:132)	approve()
ERC20.sol (L:150)	transferFrom()
ERC20.sol (L:178)	increaseAllowance()

ERC20.sol (L:197)	decreaseAllowance()
ERC20SlowRelease.sol (L:83)	setLockTime()
ERC20SlowRelease.sol (L:93)	transferWithLock()
ERC20SlowRelease.sol (L:102)	claimUnlocked()
ERC20SlowRelease.sol (L:166)	addPool()
ERC20SlowRelease.sol (L:174)	removePool()

5.5.2. Remediation

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

ERC20.sol

```
113 function transfer(address recipient, uint256 amount) external virtual override
    returns (bool) {
114     _transfer(_msgSender(), recipient, amount);
115     return true;
116 }
```

5.6. Inexplicit Solidity Compiler Version

ID	IDX-006
Target	MegapadToken
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The MegapadToken team has resolved this issue as recommended in the commit `ded74b4a8b87849a826b9edf7d770650b1d3d512`.

5.6.1. Description

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

MegapadToken.sol

```
1 //SPDX-License-Identifier: Unlicense
2 pragma solidity ^0.8.6;
```

5.6.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.11 [2].

MegapadToken.sol

```
1 //SPDX-License-Identifier: Unlicense
2 pragma solidity 0.8.11;
```

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]

[2] “Ethereum - Release” [Online]. Available: https://github.com/ethereum/solidity/releases/tag/v0.8.11. [Accessed: 27-January-2022]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE