Earn Other Fixed APR

Smart Contract Audit Report
Prepared for EvryNet

Date Issued:	May 9, 2022
Project ID:	AUDIT2022020
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022020
Version	v1.0
Client	EvryNet
Project	Earn Other Fixed APR
Auditor(s)	Natsasit Jirathammanuwat Wachirawit Kanpanluk
Author(s)	Wachirawit Kanpanluk
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	May 9, 2022	Full report	Wachirawit Kanpanluk

Contact Information

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

1. Executive Summary

As requested by EvryNet, Inspex team conducted an audit to verify the security posture of the Earn Other Fixed APR smart contracts on Mar 17, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Earn Other Fixed APR 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 and 1 low-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that Earn Other Fixed APR 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

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".

There are three features for Earn Other that allow users to deposit their assets to get reward tokens.

- **Earn Other Fixed APR** allows the users to earn a fixed amount of the reward token by locking their deposit token in the contract. The user can harvest the reward at any time.
- **Earn Other Fixed APR and Lock Reward** allows the users to earn a fixed amount of the reward token by locking their deposit token until the expiration of the specified period. The user cannot harvest the reward until the expiration of the locking period.
- **Earn Other Fixed APR and Lock Reward with Changeable Ratio** allows the users to deposit their token and earn the reward token by locking their deposit token until the expiration of the specified period. The user cannot harvest the reward until the reward ratio is changed at the end of the locking period.

Scope Information:

Project Name	Earn Other Fixed APR
Website	https://evrynet.io/
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	Yield Farming

Audit Information:

Audit Method	Whitebox
Audit Date	Mar 17, 2022
Reassessment Date	May 6, 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: 58541bf095f935c8c59f882ed913b2530e7cf496)

Contract	Location (URL)
EarnOtherFixedAPR	https://github.com/Evry-Finance/evry-finance-farm/blob/58541bf095/contracts/EarnOtherFixedAPR.sol
EarnOtherFixedAPRLockReward	https://github.com/Evry-Finance/evry-finance-farm/blob/58541bf095/contracts/EarnOtherFixedAPRLockReward.sol
EarnOtherFixedAPRLockRewardWithChangeableRatio	https://github.com/Evry-Finance/evry-finance-farm/blob/58541bf095/contracts/EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

Reassessment: (Commit: 14f6ff663c7ca157d422e6aaa2bd7748b3200e94)

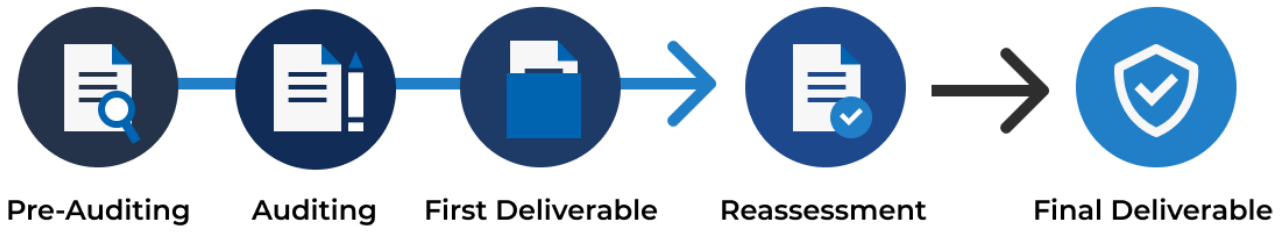
Contract	Location (URL)
EarnOtherFixedAPR	https://github.com/Evry-Finance/evry-finance-farm/blob/14f6ff663c/contracts/EarnOtherFixedAPR.sol
EarnOtherFixedAPRLockReward	https://github.com/Evry-Finance/evry-finance-farm/blob/14f6ff663c/contracts/EarnOtherFixedAPRLockReward.sol
EarnOtherFixedAPRLockRewardWithChangeableRatio	https://github.com/Evry-Finance/evry-finance-farm/blob/14f6ff663c/contracts/EarnOtherFixedAPRLockRewardWithChangeableRatio.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 testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) (https://inspex.gitbook.io/testing-guide/) which covers most prevalent risks in smart contracts. The following audit items were checked during the auditing activity:

Audit Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states

Audit Category	Testing Items
	6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) 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**.

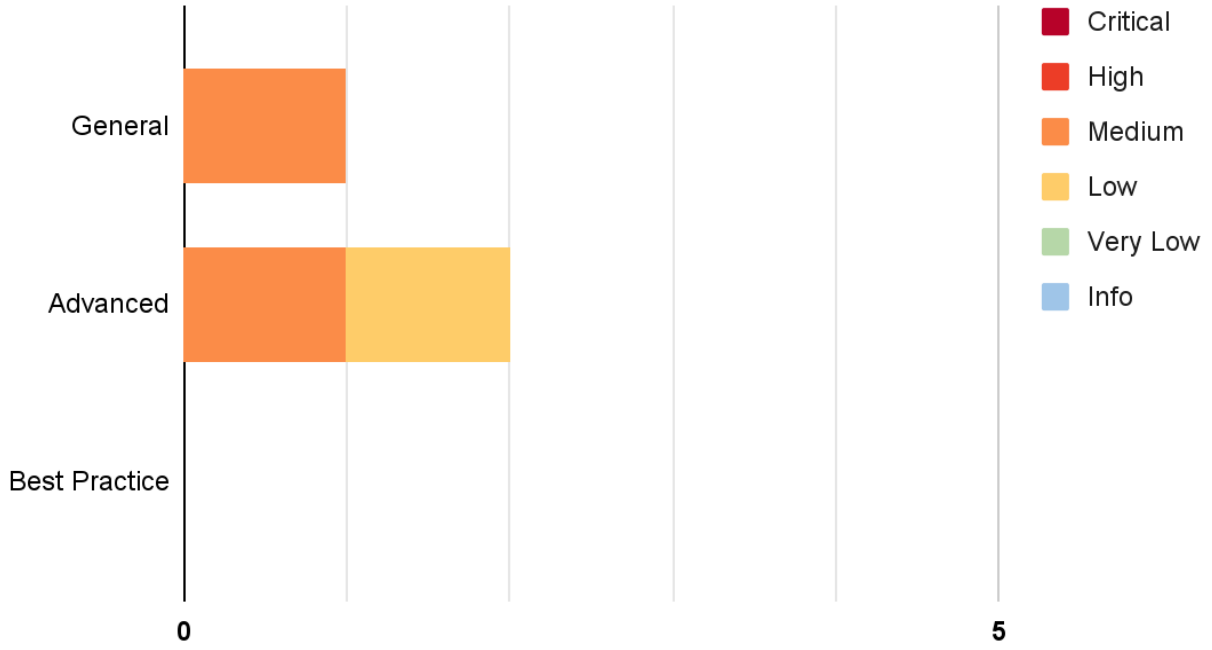
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

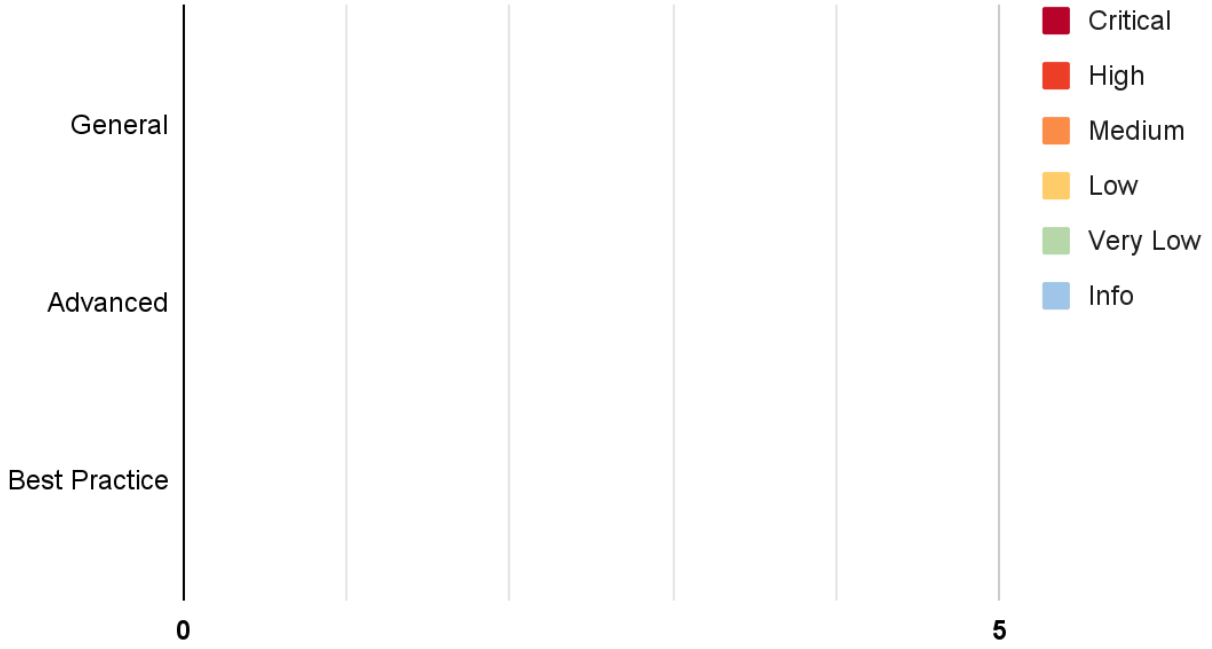
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



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 the Reward Ratio	General	Medium	Resolved *
IDX-002	Incorrect Reward Balance	Advanced	Medium	Resolved
IDX-003	Improper Reward Amount Verification	Advanced	Low	Resolved *

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

5. Detailed Findings Information

5.1. Centralized Control of the Reward Ratio

ID	IDX-001
Target	EarnOtherFixedAPRLockRewardWithChangeableRatio
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** High The owner can call the `updateTokenRatio()` function to change the reward ratio to any rate at the end of the reward distribution. Moreover, if the reward ratio is set to zero and the staked token is the same as the reward token, the owner can withdraw all tokens in the contract by calling the `recoveryReward()` function. **Likelihood:** Low There is nothing to restrict the changes from being done by the owner. However, only the owner role can call these functions to change the states.
Status	Resolved * EvryNet team has mitigated this issue by using the condition which checks whether the staked token will not be the same as the reward token in commit `14f6ff663c7ca157d422e6aaa2bd7748b3200e94`.

5.1.1. Description

In the `EarnOtherFixedAPRLockRewardWithChangeableRatio` contract, the owner has to call the `updateTokenRatio()` function at the end of the staking period to manually set the new token ratio. However, the new token ratio can be controlled by the `_ratio` parameter from the owner as shown below.

EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

```
215 function updateTokenRatio(uint256 _ratio) external onlyOwner {
216   require(block.number > endBlock, "not allow before end period");
217   require(!isUpdateTokenRatio, "token ratio is already updated");
218
219   preActualTokenRatio = actualTokenRatio;
220
221   tokenRatio = _ratio;
222   actualTokenRatio = _ratio;
223
224   if (rewardToken.decimals() > stakedToken.decimals()) {
225     actualTokenRatio = tokenRatio.mul(10**(rewardToken.decimals() -
```

```
stakedToken.decimals()));
226   }
227
228   if (rewardToken.decimals() < stakedToken.decimals()) {
229     actualTokenRatio = tokenRatio.div(10**(stakedToken.decimals() -
rewardToken.decimals()));
230   }
231
232   rewardPerBlock =
apr.mul(actualTokenRatio).div(BLOCK_PER_DAY.mul(DAY_PER_YEAR));
233   rewardDebt = rewardDebt.mul(actualTokenRatio).div(preActualTokenRatio);
234
235   isUpdateTokenRatio = true;
236
237   emit UpdateTokenRatio(_ratio, rewardDebt);
238 }
```

Moreover, if the `_ratio` parameter is zero, the `rewardDebt` state will be calculated and set to zero. When `rewardDebt` state is zero and the staked token is the same as the reward token, the owner can call the `recoveryReward()` function to withdraw all tokens from the contract as shown in lines 208 - 209.

EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

```
202 function recoveryReward(address _for) external onlyOwner {
203   require(block.number > endBlock, "not allow before end period");
204
205   uint256 rewardBalance = rewardToken.balanceOf(address(this));
206
207   if (rewardBalance > rewardDebt) {
208     uint256 transferAmount = rewardBalance.sub(rewardDebt);
209     rewardToken.safeTransfer(_for, transferAmount);
210
211     emit RecoveryReward(_for, transferAmount);
212   }
213 }
```

5.1.2. Remediation

Inspex suggests removing the `_ratio` parameter and using a price oracle for calculating the token ratio instead in the `updateTokenRatio()` function, for example in line 221:

EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

```
215 function updateTokenRatio() external onlyOwner {
216   require(block.number > endBlock, "not allow before end period");
217   require(!isUpdateTokenRatio, "token ratio is already updated");
218
219   preActualTokenRatio = actualTokenRatio;
220
221   tokenRatio = getTokenRatioFromPriceOracle();
222   actualTokenRatio = tokenRatio;
223
224   if (rewardToken.decimals() > stakedToken.decimals()) {
225     actualTokenRatio = tokenRatio.mul(10**(rewardToken.decimals() -
    stakedToken.decimals()));
226   }
227
228   if (rewardToken.decimals() < stakedToken.decimals()) {
229     actualTokenRatio = tokenRatio.div(10**(stakedToken.decimals() -
    rewardToken.decimals()));
230   }
231
232   rewardPerBlock =
    apr.mul(actualTokenRatio).div(BLOCK_PER_DAY.mul(DAY_PER_YEAR));
233   rewardDebt = rewardDebt.mul(actualTokenRatio).div(preActualTokenRatio);
234
235   isUpdateTokenRatio = true;
236
237   emit UpdateTokenRatio(_ratio, rewardDebt);
238 }
```

5.2. Incorrect Reward Balance

ID	IDX-002
Target	EarnOtherFixedAPR EarnOtherFixedAPRLockReward EarnOtherFixedAPRLockRewardWithChangeableRatio
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The staked tokens are counted and paid out as reward tokens. Furthermore, the owner can use the `recoveryReward()` function to retrieve the staked tokens that have been calculated as excessive rewards. **Likelihood:** Low It is possible that the staked token is the same as the reward token, so the user token will be paid out as the reward only if the balance of rewards in the contract is insufficient.
Status	Resolved The EvryNet team has resolved this issue by adding the condition which checks whether the balance of the contract is greater than the `rewardDebt` combine with the `totalStaked` values in commit `14f6ff663c7ca157d422e6aaa2bd7748b3200e94`.

5.2.1. Description

In the `EarnOtherFixedAPR`, `EarnOtherFixedAPRLockReward` and `EarnOtherFixedAPRLockReward` contracts, the staked token can be the same as the reward token (line 68-69).

EarnOtherFixedAPR.sol

```
57  constructor(
58      ERC20 _stakedToken,
59      ERC20 _rewardToken,
60      uint256 _cap,
61      uint256 _apr,
62      uint256 _tokenRatio,
63      uint256 _startBlock,
64      uint256 _endBlock,
65      bool _isLockWithdraw,
66      address _admin
67    ) {
68      stakedToken = _stakedToken;
69      rewardToken = _rewardToken;
```

```
70      cap = _cap;
71      apr = _apr;
72      tokenRatio = _tokenRatio;
73      startBlock = _startBlock;
74      endBlock = _endBlock;
75      isLockWithdraw = _isLockWithdraw;
76      claimableBlock = _startBlock;
77      actualTokenRatio = _tokenRatio;
78
79      if (_rewardToken.decimals() > _stakedToken.decimals()) {
80        actualTokenRatio = _tokenRatio.mul(10**(_rewardToken.decimals() -
    _stakedToken.decimals()));
81      }
82
83      if (_rewardToken.decimals() < _stakedToken.decimals()) {
84        actualTokenRatio = _tokenRatio.div(10**(_stakedToken.decimals() -
    _rewardToken.decimals()));
85      }
86
87      rewardPerBlock =
    _apr.mul(actualTokenRatio).div(BLOCK_PER_DAY.mul(DAY_PER_YEAR));
88      rewardPerBlockPrecisionFactor =
    RATIO_PRECISION.mul(PERCENTAGE_PRECISION).mul(APR_PRECISION);
89
90      transferOwnership(_admin);
91  }
```

When the staked token is the same as the reward token, the balance of reward in the contract will be higher than it should be. This will lead to functions working improperly, for example:

In the `recoveryReward()` function at line 170, the contract owner can claim the exceeded reward from the contract which is more than it should be since the contract uses the staked token as the reward token.

EarnOtherFixedAPR.sol

```
167 function recoveryReward(address _for) external onlyOwner {
168     require(block.number > endBlock, "not allow before end period");
169
170     uint256 unusedReward =
    rewardToken.balanceOf(address(this)).sub(rewardDebt);
171     rewardToken.safeTransfer(_for, unusedReward);
172     emit RecoveryReward(_for, unusedReward);
173 }
```

In the `deposit()` function at line 114, the validation can be passed with the balance of pending reward in the contract being less than expected since the staked token will be counted as the reward as well.

EarnOtherFixedAPR.sol

```
93  function deposit(uint256 _amount) external nonReentrant {
94      require(block.number >= startBlock, "not allow before start period");
95      require(block.number < endBlock, "not allow after end period");
96      require(totalStaked < cap, "pool capacity is reached");
97
98      harvest();
99
100     uint256 _remaining = cap.sub(totalStaked);
101     uint256 _preDeposit = stakedToken.balanceOf(address(this));
102
103     if (_remaining < _amount) {
104       stakedToken.safeTransferFrom(msg.sender, address(this), _remaining);
105     } else {
106       stakedToken.safeTransferFrom(msg.sender, address(this), _amount);
107     }
108
109     uint256 _postDeposit = stakedToken.balanceOf(address(this));
110     uint256 _actualAmount = _postDeposit.sub(_preDeposit);
111     totalStaked = totalStaked.add(_actualAmount);
112     rewardDebt = rewardDebt.add(_calculateReward(_actualAmount,
    endBlock.sub(_lastRewardBlock())));
113
114     require(rewardDebt <= rewardToken.balanceOf(address(this)), "insufficient
    reward reserve");
115
116     UserInfo storage user = userInfo[msg.sender];
117     user.amount = user.amount.add(_actualAmount);
118     user.lastRewardBlock = _lastRewardBlock();
119
120     emit Deposit(msg.sender, _actualAmount);
121 }
```

In the `harvest()` function at line 133, the reward will be transferred using the staked token. This can cause other users unable to claim the reward as expected.

EarnOtherFixedAPR.sol

```
129 function harvest() public {
130     UserInfo storage user = userInfo[msg.sender];
131     if (user.amount > 0 && block.number > claimableBlock) {
132       uint256 _reward = calculateReward(user.amount, user.lastRewardBlock);
133       rewardToken.safeTransfer(msg.sender, _reward);
134       rewardDebt = rewardDebt.sub(_reward);
135       emit Harvest(msg.sender, _reward);
136     }
137     user.lastRewardBlock = block.number;
138 }
```

The list of functions that use the incorrect reward balance is as follows:

File	Contract	Function
EarnOtherFixedAPR.sol (L:93)	EarnOtherFixedAPR	deposit()
EarnOtherFixedAPR.sol (L:167)	EarnOtherFixedAPR	recoveryReward()
EarnOtherFixedAPRLockReward.sol (L:178)	EarnOtherFixedAPRLockReward	recoveryReward()
EarnOtherFixedAPRLockRewardWithChangeableRatio.sol (L:202)	EarnOtherFixedAPRLockRewardWithChangeableRatio	recoveryReward()

5.2.2. Remediation

Inspex suggests adding a condition in the `deposit()` and the `recoveryReward()` functions in case the staked token is the same as the reward token, for example:

Adding a condition in line 114 - 118 of the `deposit()` function.

EarnOtherFixedAPR.sol

```
93  function deposit(uint256 _amount) external nonReentrant {
94      require(block.number >= startBlock, "not allow before start period");
95      require(block.number < endBlock, "not allow after end period");
96      require(totalStaked < cap, "pool capacity is reached");
97
98      harvest();
99
100     uint256 _remaining = cap.sub(totalStaked);
101     uint256 _preDeposit = stakedToken.balanceOf(address(this));
102
```

```
103     if (_remaining < _amount) {
104       stakedToken.safeTransferFrom(msg.sender, address(this), _remaining);
105     } else {
106       stakedToken.safeTransferFrom(msg.sender, address(this), _amount);
107     }
108 
109     uint256 _postDeposit = stakedToken.balanceOf(address(this));
110     uint256 _actualAmount = _postDeposit.sub(_preDeposit);
111     totalStaked = totalStaked.add(_actualAmount);
112     rewardDebt = rewardDebt.add(_calculateReward(_actualAmount,
    endBlock.sub(_lastRewardBlock())));
113 
114     if(stakedToken == rewardToken){
115         require(rewardDebt <=
    rewardToken.balanceOf(address(this)).sub(totalStaked), "insufficient reward
    reserve");
116     } else {
117         require(rewardDebt <= rewardToken.balanceOf(address(this)),
    "insufficient reward reserve");
118     }
119 
120     UserInfo storage user = userInfo[msg.sender];
121     user.amount = user.amount.add(_actualAmount);
122     user.lastRewardBlock = _lastRewardBlock();
123 
124     emit Deposit(msg.sender, _actualAmount);
125 }
```

Adding a condition in line 171 - 175 of the `recoveryReward()` function.

EarnOtherFixedAPR.sol

```
167 function recoveryReward(address _for) external onlyOwner {
168     require(block.number > endBlock, "not allow before end period");
169 
170     uint256 unusedReward;
171     if(stakedToken == rewardToken){
172         unusedReward =
    rewardToken.balanceOf(address(this)).sub(rewardDebt).sub(totalStaked);
173     } else {
174         unusedReward = rewardToken.balanceOf(address(this)).sub(rewardDebt);
175     }
176     rewardToken.safeTransfer(_for, unusedReward);
177     emit RecoveryReward(_for, unusedReward);
178 }
```

5.3. Improper Reward Amount Verification

ID	IDX-003
Target	EarnOtherFixedAPRLockReward EarnOtherFixedAPRLockRewardWithChangeableRatio
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium When the contract owner has not transferred sufficient reward tokens into the contract, the staked token can not be withdrawn by calling the `withdraw()` function. However, the users can get their staked token back by discarding their pending rewards when the reward distribution period has ended by calling the `emergencyWithdraw()` function. When this happens, the users will lose the opportunity to gain the rewards with their staked tokens during the reward distribution period. **Likelihood:** Low It is possible that the reward token will be insufficient when the contract owner does not transfer enough reserve of reward token into the contract to cover all users' deposits since this action has to be done manually by the contract owner. However, the users can check the reward reserve before depositing the token, and the reward cannot be transferred out by the owner before the reward distribution period has ended.
Status	Resolved * The EvryNet team has resolved this issue in `EarnOtherFixedAPRLockReward` contract as suggested and mitigated this issue in `EarnOtherFixedAPRLockRewardWithChangeableRatio` contract by using the condition which checks whether the staked token will not be the same as the reward token in commit `14f6ff663c7ca157d422e6aaa2bd7748b3200e94`.

5.3.1. Description

The `EarnOtherFixedAPRLockReward` and the `EarnOtherFixedAPRLockRewardWithChangeableRatio` contracts allow users to stake their tokens in the contract for a specific amount of time to gain a reward. At the end of the reward distribution period, the users should be able to withdraw their staked token and reward token by using the `withdraw()` function, but if the reward in the contract is not enough for the users, the calling of `rewardToken.safeTransfer()` function in line 142 will revert the transaction, so the users cannot withdraw their staked token.

EarnOtherFixedAPRLockReward.sol

```
133 function withdraw() external nonReentrant {
134   require(block.number > endBlock, "not allow before end period");
135   UserInfo storage user = userInfo[msg.sender];
136
137   if (user.amount > 0) {
138     uint256 withdrawAmount = user.amount;
139     uint256 rewards = _calculateReward(user.amount, user.lastRewardBlock);
140     rewards = user.rewards.add(rewards);
141
142     rewardToken.safeTransfer(msg.sender, rewards);
143     stakedToken.safeTransfer(msg.sender, withdrawAmount);
144
145     user.amount = 0;
146     user.rewards = 0;
147     user.lastRewardBlock = endBlock;
148
149     totalStaked = totalStaked.sub(withdrawAmount);
150     rewardDebt = rewardDebt.sub(rewards);
151
152     emit Withdraw(msg.sender, withdrawAmount, rewards);
153   }
154 }
```

Furthermore, the owner can use the `updateTokenRatio()` function in the `EarnOtherFixedAPRLockRewardWithChangeableRatio` contract to update the `rewardDebt` with the new token ratio as shown in line 233. If the new token ratio is higher than the previous one, the reward token in the contract may be insufficient for the users.

EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

```
215 function updateTokenRatio(uint256 _ratio) external onlyOwner {
216   require(block.number > endBlock, "not allow before end period");
217   require(!isUpdateTokenRatio, "token ratio is already updated");
218
219   preActualTokenRatio = actualTokenRatio;
220
221   tokenRatio = _ratio;
222   actualTokenRatio = _ratio;
223
224   if (rewardToken.decimals() > stakedToken.decimals()) {
225     actualTokenRatio = tokenRatio.mul(10**(rewardToken.decimals() -
    stakedToken.decimals()));
226   }
227
228   if (rewardToken.decimals() < stakedToken.decimals()) {
229     actualTokenRatio = tokenRatio.div(10**(stakedToken.decimals() -
```

```
rewardToken.decimals()));
230   }
231
232   rewardPerBlock =
apr.mul(actualTokenRatio).div(BLOCK_PER_DAY.mul(DAY_PER_YEAR));
233   rewardDebt = rewardDebt.mul(actualTokenRatio).div(preActualTokenRatio);
234
235   isUpdateTokenRatio = true;
236
237   emit UpdateTokenRatio(_ratio, rewardDebt);
238 }
```

Nevertheless, the users can still use the `emergencyWithdraw()` function to withdraw their staked token by discarding their pending reward when the reward distribution period has ended.

The following table contains list of affected functions:

File	Contract	Function
EarnOtherFixedAPRLockReward.sol (L:91)	EarnOtherFixedAPRLockReward	deposit()
EarnOtherFixedAPRLockRewardWithChangeableRatio.sol (L:94)	EarnOtherFixedAPRLockRewardWithChangeableRatio	deposit()
EarnOtherFixedAPRLockRewardWithChangeableRatio.sol (L:215)	EarnOtherFixedAPRLockRewardWithChangeableRatio	updateTokenRatio()

5.3.2. Remediation

Inspex suggests adding a condition in the `deposit()` and the `updateTokenRatio()` functions to confirm that there is enough reward for the users:

Adding a condition in line 115 - 119 of the `deposit()` function.

EarnOtherFixedAPRLockReward.sol

```
91  function deposit(uint256 _amount) external nonReentrant {
92    require(block.number >= startBlock, "not allow before start period");
93    require(block.number < endBlock, "not allow after end period");
94    require(totalStaked < cap, "pool capacity is reached");
95    UserInfo storage user = userInfo[msg.sender];
96
97    uint256 remaining = cap.sub(totalStaked);
98    uint256 preAmount = stakedToken.balanceOf(address(this));
99
100   if (remaining < _amount) {
101     stakedToken.safeTransferFrom(address(msg.sender), address(this),
```

```
remaining);
  } else {
    stakedToken.safeTransferFrom(address(msg.sender), address(this), _amount);
  }

  uint256 postAmount = stakedToken.balanceOf(address(this));
  uint256 actualAmount = postAmount.sub(preAmount);
  totalStaked = totalStaked.add(actualAmount);
  uint256 lastRewardBlock = block.number;

  rewardDebt = rewardDebt.add(

actualAmount.mul((endBlock.sub(lastRewardBlock)).mul(rewardPerBlock)).div(rewardPerBlockPrecisionFactor)
  );

  if(stakedToken == rewardToken){
      require(rewardDebt <= rewardToken.balanceOf(address(this)).sub(totalStaked), "insufficient reward reserve");
  } else {
      require(rewardDebt <= rewardToken.balanceOf(address(this)), "insufficient reward reserve");
  }

  if (user.amount > 0 && block.number > startBlock) {
    uint256 rewards = _calculateReward(user.amount, user.lastRewardBlock);
    user.rewards = user.rewards.add(rewards);
  }

  user.amount = user.amount.add(actualAmount);
  user.lastRewardBlock = lastRewardBlock;

  emit Deposit(msg.sender, actualAmount);
}
```

Adding a condition in line 235 - 239 of the `updateTokenRatio()` function.

EarnOtherFixedAPRLockRewardWithChangeableRatio.sol

```
215 function updateTokenRatio(uint256 _ratio) external onlyOwner {
216   require(block.number > endBlock, "not allow before end period");
217   require(!isUpdateTokenRatio, "token ratio is already updated");
218
219   preActualTokenRatio = actualTokenRatio;
220
221   tokenRatio = _ratio;
222   actualTokenRatio = _ratio;
223
224   if (rewardToken.decimals() > stakedToken.decimals()) {
225     actualTokenRatio = tokenRatio.mul(10**(rewardToken.decimals() -
    stakedToken.decimals()));
226   }
227
228   if (rewardToken.decimals() < stakedToken.decimals()) {
229     actualTokenRatio = tokenRatio.div(10**(stakedToken.decimals() -
    rewardToken.decimals()));
230   }
231
232   rewardPerBlock =
    apr.mul(actualTokenRatio).div(BLOCK_PER_DAY.mul(DAY_PER_YEAR));
233   rewardDebt = rewardDebt.mul(actualTokenRatio).div(preActualTokenRatio);
234
235   if(stakedToken == rewardToken){
236       require(rewardDebt <=
    rewardToken.balanceOf(address(this)).sub(totalStaked), "insufficient reward
    reserve");
237   } else {
238       require(rewardDebt <= rewardToken.balanceOf(address(this)), "insufficient
    reward reserve");
239   }
240
241   isUpdateTokenRatio = true;
242
243   emit UpdateTokenRatio(_ratio, rewardDebt);
244 }
```

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

inspeX

CYBERSECURITY PROFESSIONAL SERVICE