BinaryOption

Smart Contract Audit Report
Prepared for EvryNet

Date Issued:	Apr 1, 2022
Project ID:	AUDIT2022021
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022021
Version	v1.0
Client	EvryNet
Project	BinaryOption
Auditor(s)	Patipon Suwanbol Puttimet Thammasaeng
Author(s)	Patipon Suwanbol
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Apr 1, 2022	Full report	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 BinaryOption smart contracts between Mar 22, 2022 and Mar 23, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of BinaryOption 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 high, and 3 info-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 BinaryOption 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".

BinaryOption is a contract that allows the user to predict the price of a specific coin with an oracle price source. It predicts whether the price from now on will rise or fall within a specific range of time.

Scope Information:

Project Name	BinaryOption
Website	https://evrynet.io/
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	Options

Audit Information:

Audit Method	Whitebox
Audit Date	Mar 22, 2022 - Mar 23, 2022
Reassessment Date	Mar 31, 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 contract was audited and reassessed by Inspex in detail:

Initial Audit: (Commit: e23abba9df7e97baef1ecfcf8631356041b9c1f8)

Contract	Location (URL)
BinaryOption	https://github.com/Evry-Finance/evry-finance-binary-options/blob/e23abba9df/contracts/BinaryOption.sol

Reassessment: (Commit: 9da99071ca854d911738ccca0b2dfa5ed451cab7)

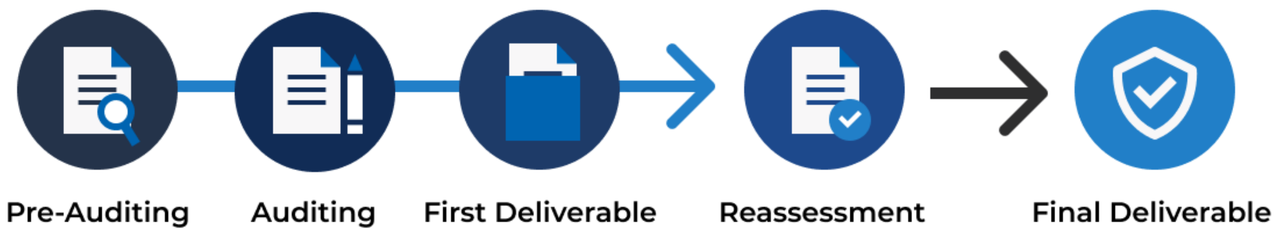
Contract	Location (URL)
BinaryOption	https://github.com/Evry-Finance/evry-finance-binary-options/blob/9da99071ca/contracts/BinaryOption.sol

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
Smart Contract with Unpublished Source Code
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
Centralized Control of State Variable
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 (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

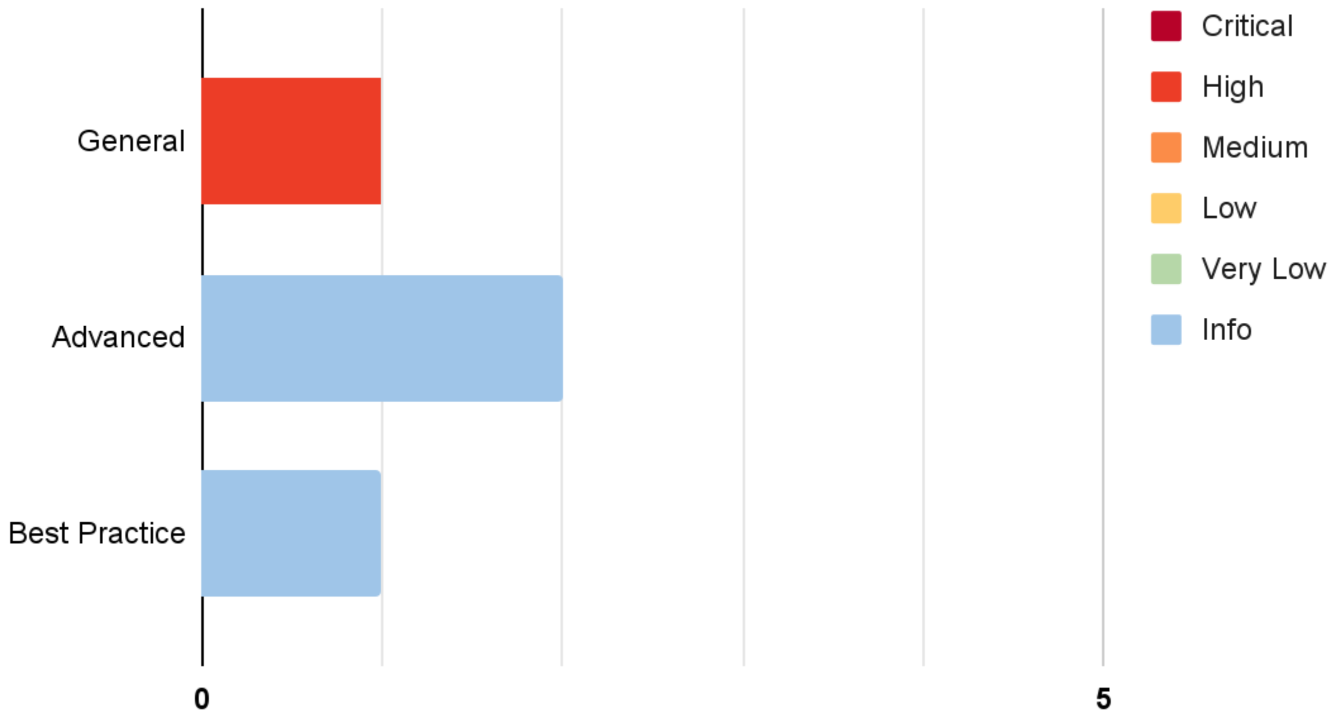
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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**.

Likelihood Impact	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 4 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	High	Resolved *
IDX-002	Improper Start and End Prediction Round Mechanism	Advanced	Info	No Security Impact
IDX-003	Missing Input Validation in Constructor	Advanced	Info	Resolved
IDX-004	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

* 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	BinaryOption
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High 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, this action can only be done by the contract owner.
Status	Resolved * EvryNet team has confirmed that a Timelock contract with 24-hour delay will be deployed and used for the affected roles. At the time of the reassessment, the contracts are not yet deployed. The platform users should confirm that only the Timelock contract has the privileged roles before using the platform.

5.1.1. Description

Critical state variables can be updated at 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 currently no constraint to prevent the authorities from modifying these variables without notifying the users.

For example, the contract owner can change the oracle price source to any contract that favors to a specific options, resulting in price manipulation as shown in the following source code:

BinaryOption.sol

```
function setOracle(address _oracle) external whenPaused onlyAdmin {
  require(_oracle != address(0), "BinaryOption: Cannot be zero address");
  oracleLatestRoundId = 0;
  oracle = AggregatorV3Interface(_oracle);

  // Dummy check to make sure the interface implements this function properly
  oracle.latestRoundData();
```

```

    emit NewOracle(_oracle);
}
```

The controllable privileged state update functions are as follows:

Target	Function	Modifier
BinaryOption.sol(L: 258)	setOracle()	onlyAdmin
BinaryOption.sol(L: 273)	setOracleUpdateAllowance()	onlyAdmin
BinaryOption.sol(L: 518)	setBufferIntervalAndLockInSeconds()	onlyAdmin
BinaryOption.sol(L: 537)	setMinBetAmount()	onlyAdmin
BinaryOption.sol(L: 548)	setTreasuryFee()	onlyAdmin
BinaryOption.sol(L: 559)	setOperator()	onlyAdmin
BinaryOption.sol(L: 570)	setAdmin()	onlyOwner

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 mechanism to delay the changes for a reasonable amount of time

In addition, if the timelock is used to mitigate this issue, the modifier of the `unpause()` function should be changed to `onlyAdminOrOperator` or `onlyOperator` modifier as an example to exclude this emergency function from being delayed.

5.2. Improper Start and End Prediction Round Mechanism

ID	IDX-002
Target	BinaryOption
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact EvryNet team has acknowledged this issue since the `bufferSeconds` state will be set to 30 seconds which cannot combine the end, lock and start round in the same function.

5.2.1. Description

In `BinaryOption` contract, after the genesis round is started and locked through both the `genesisStartRound()` and the `genesisLockRound()` functions, the `lockAndStartNextRound()` function is used to update the round's locked price, which is used as the criteria to finalize the user's prediction result.

BinaryOption.sol

```
/**
   * @notice Start the next round n, lock price for round n-1, end round n-2
   * @dev Callable by operator
   */
function lockAndStartNextRound() external whenNotPaused onlyOperator {
    require(
      genesisStartOnce && genesisLockOnce,
      "BinaryOption: Can only run after genesisStartRound and genesisLockRound is triggered"
    );

    (uint80 currentRoundId, int256 currentPrice) = _getPriceFromOracle();

    oracleLatestRoundId = uint256(currentRoundId);

    // CurrentEpoch refers to previous round (n-1)
    _safeLockRound(currentEpoch, currentRoundId, currentPrice);

    // Increment currentEpoch to current round (n)
    currentEpoch = currentEpoch + 1;
    _startRound(currentEpoch);
```

```
}
```

In order to allow the winners of the latest locked round (`currentEpoch - 1`) to claim the reward, the authorized party is required to execute the `endRound()` function to validate that round.

BinaryOption.sol

```
/**
  * @notice End round
  * @dev Callable by operator
  */
function endRound() external whenNotPaused onlyOperator {
    require(
      genesisStartOnce && genesisLockOnce,
      "BinaryOption: Can only run after genesisStartRound and genesisLockRound is triggered"
    );

    (uint80 currentRoundId, int256 currentPrice) = _getPriceFromOracle();

    _safeEndRound(currentEpoch - 1, currentRoundId, currentPrice);
    _calculateRewards(currentEpoch - 1);
    claimTreasury();
}
```

Hence, the `lockAndStartNextRound()` function can be called without ending the previous round first. This means the user's predictions will be invalid because that invested round can be simply skipped by repeatedly calling the `lockEndAndStartNextRound()` function. This is because the `currentEpoch` is updated every time the function is called, so the `endRound()` function will be eventually unable to update the two previous round (`currentEpoch` - 2).

However, the users can claim their investment back anyway from those invalid rounds, so there is no impact on this issue.

5.2.2. Remediation

Inspex suggests combining the `lockAndStartNextRound()` and `endRound()` functions together as one function to ensure that whenever the next round is started, the previous round is locked and ended synchronously, for example, the `lockEndAndStartNextRound()` function is consisted of the `lockAndStartNextRound()` and the `endRound()` functions:

BinaryOption.sol

```
function lockEndAndStartNextRound() external whenNotPaused onlyOperator {
    require(
      genesisStartOnce && genesisLockOnce,
      "BinaryOption: Can only run after genesisStartRound and genesisLockRound
```

```
is triggered"
    );

    (uint80 currentRoundId, int256 currentPrice) = _getPriceFromOracle();

    oracleLatestRoundId = uint256(currentRoundId);

    // CurrentEpoch refers to previous round (n-1)
    _safeLockRound(currentEpoch, currentRoundId, currentPrice);
    _safeEndRound(currentEpoch - 1, currentRoundId, currentPrice);
    _calculateRewards(currentEpoch - 1);
    claimTreasury();

    // Increment currentEpoch to current round (n)
    currentEpoch = currentEpoch + 1;
    _startRound(currentEpoch);
}
```

5.3. Missing Input Validation in Constructor

ID	IDX-003
Target	BinaryOption
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved EvryNet team has resolved this issue as suggested in commit `9da99071ca854d911738ccca0b2dfa5ed451cab7` by validating the `_bufferSeconds` and `_lockInSeconds` states.

5.3.1. Description

According to the business design, the `bufferSeconds` state is used to limit the range of time that the operator can lock the round's price and close that round.

BinaryOption.sol

```
/**
   * @notice Lock round
   * @param epoch: epoch
   * @param roundId: roundId
   * @param price: price of the round
   */
  function _safeLockRound(
    uint256 epoch,
    uint256 roundId,
    int256 price
  ) internal {
    require(rounds[epoch].startTimestamp != 0, "BinaryOption: Can only lock round after round has started");
    require(block.timestamp >= rounds[epoch].lockTimestamp, "BinaryOption: Can only lock round after lockTimestamp");
    require(
      block.timestamp <= rounds[epoch].lockTimestamp + bufferSeconds,
      "BinaryOption: Can only lock round within bufferSeconds"
    );
    Round storage round = rounds[epoch];
    round.lockPrice = price;
```

```
326     round.lockOracleId = roundId;
327 
328     emit LockRound(epoch, roundId, round.lockPrice);
329   }
```

BinaryOption.sol

```
372 /**
373    * @notice End round
374    * @param epoch: epoch
375    * @param roundId: roundId
376    * @param price: price of the round
377    */
378 function _safeEndRound(
379     uint256 epoch,
380     uint256 roundId,
381     int256 price
382   ) internal {
383     require(rounds[epoch].lockTimestamp != 0, "BinaryOption: Can only end round after round has locked");
384     require(block.timestamp >= rounds[epoch].closeTimestamp, "BinaryOption: Can only end round after closeTimestamp");
385     require(
386       block.timestamp <= rounds[epoch].closeTimestamp + bufferSeconds,
387       "BinaryOption: Can only end round within bufferSeconds"
388     );
389     Round storage round = rounds[epoch];
390     round.closePrice = price;
391     round.closeOracleId = roundId;
392     round.oracleCalled = true;
393     emit EndRound(epoch, roundId, round.closePrice);
394 }
```

Moreover, the `bufferSeconds` state must be less than the `intervalSeconds` and the `lockInSeconds` state, and the `lockInSeconds` state must be less than the `intervalSeconds` state as shown in the following source code in lines 523-525.

BinaryOption.sol

```
514 /**
515    * @notice Set buffer, interval and lock (in seconds)
516    * @dev Callable by admin
517    */
518 function setBufferIntervalAndLockInSeconds(
519     uint256 _bufferSeconds,
520     uint256 _intervalSeconds,
521     uint256 _lockInSeconds
522   ) external whenPaused onlyAdmin {
```

```
    require(_bufferSeconds < _intervalSeconds, "BinaryOption: bufferSeconds must be lower than intervalSeconds");
    require(_bufferSeconds < _lockInSeconds, "BinaryOption: bufferSeconds must be lower than lockInSeconds");
    require(_lockInSeconds < _intervalSeconds, "BinaryOption: lockInSeconds must be lower than intervalSeconds");
    bufferSeconds = _bufferSeconds;
    intervalSeconds = _intervalSeconds;
    lockInSeconds = _lockInSeconds;

    emit NewBufferIntervalAndLockInSeconds(_bufferSeconds, _intervalSeconds, _lockInSeconds);
}
```

However, in the constructor, there is no validation to ensure that the `bufferSeconds`, `intervalSeconds` and `lockInSeconds` states are set correctly.

BinaryOption.sol

```
/**
   * @notice Constructor
   * @param _oracleAddress: oracle address
   * @param _adminAddress: admin address
   * @param _operatorAddress: operator address
   * @param _intervalSeconds: number of time within an interval
   * @param _bufferSeconds: buffer of time for resolution of price
   * @param _minBetAmount: minimum bet amounts (in wei)
   * @param _oracleUpdateAllowance: oracle update allowance
   * @param _treasuryFee: treasury fee (1000 = 10%)
   * @param _lockInSeconds:
   */
constructor(
    ERC20 _betToken,
    address _oracleAddress,
    address _adminAddress,
    address _operatorAddress,
    uint256 _intervalSeconds,
    uint256 _bufferSeconds,
    uint256 _minBetAmount,
    uint256 _oracleUpdateAllowance,
    uint256 _treasuryFee,
    uint256 _lockInSeconds
  ) {
    require(_treasuryFee <= MAX_TREASURY_FEE, "BinaryOption: Treasury fee too high");

    betToken = _betToken;
    oracle = AggregatorV3Interface(_oracleAddress);
```

```
    adminAddress = _adminAddress;
    operatorAddress = _operatorAddress;
    intervalSeconds = _intervalSeconds;
    bufferSeconds = _bufferSeconds;
    minBetAmount = _minBetAmount;
    oracleUpdateAllowance = _oracleUpdateAllowance;
    treasuryFee = _treasuryFee;
    lockInSeconds = _lockInSeconds;
}
```

As a result, the `bufferSeconds`, `intervalSeconds` and `lockInSeconds` states can be set to any value, which breaks the business design.

However, the incorrect configuration for the `bufferSeconds`, `lockInSeconds` and `lockInSeconds` states can be reset again through the `setBufferIntervalAndLockInSeconds()` function. Therefore, there is no impact on this issue.

5.3.2. Remediation

Inspex suggests adding the input validation in the constructor to ensure that the `bufferSeconds` state is less than the `lockInSeconds` state, and the `lockInSeconds` state is less than the `intervalSeconds` state respectively, for example:

BinaryOption.sol

```
/**
   * @notice Constructor
   * @param _oracleAddress: oracle address
   * @param _adminAddress: admin address
   * @param _operatorAddress: operator address
   * @param _intervalSeconds: number of time within an interval
   * @param _bufferSeconds: buffer of time for resolution of price
   * @param _minBetAmount: minimum bet amounts (in wei)
   * @param _oracleUpdateAllowance: oracle update allowance
   * @param _treasuryFee: treasury fee (1000 = 10%)
   * @param _lockInSeconds:
   */
constructor(
    ERC20 _betToken,
    address _oracleAddress,
    address _adminAddress,
    address _operatorAddress,
    uint256 _intervalSeconds,
    uint256 _bufferSeconds,
    uint256 _minBetAmount,
    uint256 _oracleUpdateAllowance,
    uint256 _treasuryFee,
```

```
    uint256 _lockInSeconds
  ) {
    require(_treasuryFee <= MAX_TREASURY_FEE, "BinaryOption: Treasury fee too high");
    require(_bufferSeconds < _lockInSeconds, "BinaryOption: bufferSeconds must be lower than lockInSeconds");
    require(_lockInSeconds < _intervalSeconds, "BinaryOption: lockInSeconds must be lower than intervalSeconds");

    betToken = _betToken;
    oracle = AggregatorV3Interface(_oracleAddress);
    adminAddress = _adminAddress;
    operatorAddress = _operatorAddress;
    intervalSeconds = _intervalSeconds;
    bufferSeconds = _bufferSeconds;
    minBetAmount = _minBetAmount;
    oracleUpdateAllowance = _oracleUpdateAllowance;
    treasuryFee = _treasuryFee;
    lockInSeconds = _lockInSeconds;
}
```

5.4. Inexplicit Solidity Compiler Version

ID	IDX-004
Target	BinaryOption
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved EvryNet team has resolved this issue as suggested in commit `9da99071ca854d911738ccca0b2dfa5ed451cab7` by explicitly defining the Solidity version as the latest stable version (v0.8.13).

5.4.1. 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.

BinaryOption.sol

```
//contracts/BinaryOption.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.9;
```

5.4.2. 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 v0.8.13 (https://docs.soliditylang.org/en/v0.8.13/).

BinaryOption.sol

```
//contracts/BinaryOption.sol
// SPDX-License-Identifier: MIT

pragma solidity 0.8.13;
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

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

CYBERSECURITY PROFESSIONAL SERVICE