Security Audit Report for Bhavish Prediction Contracts

Date: October 24, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.10.24 19:12:10 +08'00'

Contents

Report Manifest

Item	Description
Client	Bhavish Finance
Target	Bhavish Prediction Contracts

Version History

Version	Date	Description
1.0	October 24, 2022	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the Bhavish Prediction Contracts [1] of the Bhavish Protocol. It is a prediction market that enables the users to make predictions on the future prices of various assets. The users may make profits if the predictions are correct. During the audit, some extra files (that were not included in the initial commit) were added, which are out of the audit scope. Specifically, the files covered in this audit are specified in the following table:

Folder Name	Contract File Name
Automation	ChainlinkPredictionOps.sol
	GelatoPredictionOps.sol
	EquitiesPredictionOpsManager.sol
	PredictionOpsManager
Impl	BhavishAdministrator.sol
	BhavishValidContracts.sol
	PriceManager.sol
Impl/BhavishSDK	AbstractBhavishSDK.sol
	ERC20SDK.sol
	NativeSDK.sol
Impl/BhavishPrediction	AbstractLossy.sol
	AbstractNative.sol
	AbstractPrediction.sol
	BhavishPredictionStorage.sol
	AbstractNoLoss.sol
Impl/BhavishPrediction/Equities	AbstractEP.sol
	LossyEP.sol
	NativeEP.sol
	NoLosseEP.sol
Impl/BhavishPrediction/Crypto	AbstractCP.sol
	LossyCP.sol
	NativeCP.sol
	NoLossCP.sol
Impl/BhavishRealTimePrediction	AbstractRT.sol

Continued on next page

[1] https://github.com/Bhavish-finance/prediction-contract

Continued from previous page

Folder Name	Contract File Name
Impl/BhavishRealTimePrediction	LossyRT.sol
	NativeRT.sol
	NoLossRT.sol
Integrations	Gasless/BaseRelayRecipient.sol
	Gasless/IRelayRecipient.sol
	MinimalForwarder/MinimalForwarder.sol
	Swap/BhavishSwap.sol
	YieldFarming/YieldFarming.sol
Interface	IBhavishAdministrator.sol
	IBhavishERC20SDK.sol
	IBhavishPrediction.sol
	IBhavishSDK.sol
	IPriceManager.sol
Pool	AbstractPool.sol
	BhavishLossyPool.sol
	BhavishNoLossPool.sol
	BhavishReInvest.sol
Rewards	BhavishPredictionRewards.sol
Tokens	BhavishGameToken.sol
	BhavishNoLossGameToken.sol
	BhavishRewardToken.sol
Vault	BaseVault.sol
	BhavishLossyVault.sol
	BhavishNativeVault.sol
	VaultProtector.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
predict-contract	`Version 1`	`6791e2f5bcaf0b8294bc6262ba051ed933d9ff53`
	`Version 2`	`3c7841b6e60082b22d2a8bfae54f6f6f1bf1962d`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report

in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic

- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **twelve** potential issues. We also have **three** recommendations and **five** notes.

- High Risk: 6
- Medium Risk: 3
- Low Risk: 3
- Recommendation: 3
- Note: 5

ID	Severity	Description	Category	Status
1	High	Lack of access control to make predictions	Software Security	Fixed
2	Low	Being unable to remove markets	Software Security	Fixed
3	Low	Unhandled corner case when calculating rewards	Software Security	Confirmed
4	Medium	No sanity checks for closed markets	Software Security	Fixed
5	Medium	Incorrect argument passed to the `refundUsers` function	Software Security	Fixed
6	Low	Ineffective prediction market updating	Software Security	Fixed
7	High	No access control for the `reinvest` function	DeFi Security	Fixed
8	High	Potential precision loss in share calculation	DeFi Security	Fixed
9	High	Token balance manipulation in the `AbstractPool` contract	DeFi Security	Fixed
10	Medium	Locked admin fees	DeFi Security	Fixed
11	High	Inconsistent state update	DeFi Security	Fixed
12	High	Incorrect calculation for token balance changes	DeFi Security	Fixed
13	-	Remove redundant checks	Recommendation	Acknowledged
14	-	Avoid mixed use of `msg.sender` and the `msgSender` function	Recommendation	Fixed
15	-	Remove redundant calculation	Recommendation	Fixed
16	-	Do not change the multiplier parameter	Note	-
17	-	Ensure the functionality of the price manager	Note	-
18	-	Design of the `BhavishNoLossPool` contract	Note	-
19	-	Stateless assumption of the `MinimalForwarder` contract	Note	-
20	-	RT market is closed on receiving invalid results	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Lack of access control to make predictions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AbstractLossy` and `AbstractNoLoss` contracts, the public functions `predictUp` and `predictDown` do not have any access control. It means that anyone can make predictions for other users without their permissions.

```
30    function predictUp(
31        uint256 _predictRoundId,
32        address _userAddress,
33        uint256 _amount
34    ) external override whenNotPaused nonReentrant {
35        token.safeTransferFrom(_userAddress, address(this), _amount);
36        _predictUp(_predictRoundId, _userAddress, _amount);
37    }
38
39    function predictDown(
40        uint256 _predictRoundId,
41        address _userAddress,
42        uint256 _amount
43    ) external override whenNotPaused nonReentrant {
44        token.safeTransferFrom(_userAddress, address(this), _amount);
45        _predictDown(_predictRoundId, _userAddress, _amount);
46    }
```

Listing 2.1: AbstractLossy.sol

```
33    function predictUp(
34        uint256 _predictRoundId,
35        address _userAddress,
36        uint256 _amount
37    ) external override whenNotPaused nonReentrant {
38        token.safeTransferFrom(_userAddress, address(this), _amount);
39        _predictUp(_predictRoundId, _userAddress, _amount);
40    }
41
42    /**
43     * @notice Bet Bear position
44     * @param _predictRoundId Round Id
45     * @param _userAddress Address of the user
46     */
47    function predictDown(
48        uint256 _predictRoundId,
49        address _userAddress,
50        uint256 _amount
51    ) external override whenNotPaused nonReentrant {
52        token.safeTransferFrom(_userAddress, address(this), _amount);
53        _predictDown(_predictRoundId, _userAddress, _amount);
54    }
```

Listing 2.2: AbstractNoLoss.sol

Note that the `ERC20SDK` contract also has the similar problem. The `predict` function in the contract does not have any access control either.

```
35    function predict(
36        PredictionStruct memory _predStruct,
```

```
37        address _userAddress,
38        address _provider,
39        uint256 _amount
40    ) external override {
41        IBhavishPredictionERC20 bhavishPrediction = _getERC20PredictionMap(
42            predictionMap[_predStruct.underlying][_predStruct.strike]
43        );
44
45        require(address(bhavishPrediction) != address(0), "Prediction Market for the asset is not
              present");
46        require(activePredictionMap[bhavishPrediction], "Prediction Market for the asset is not
              active");
47
48        address userAddress_;
49        if (address(msg.sender).isContract()) {
50            userAddress_ = _userAddress;
51        } else {
52            require(msg.sender == _userAddress, "Buyer and msg.sender cannot be different");
53            userAddress_ = msg.sender;
54        }
55
56        if (_predStruct.directionUp) bhavishPrediction.predictUp(_predStruct.roundId, userAddress_,
               _amount);
57        else bhavishPrediction.predictDown(_predStruct.roundId, userAddress_, _amount);
58
59        _populateProviderInfo(_provider, _amount);
60    }
```

Listing 2.3: ERC20SDK.sol

Impact Anyone can arbitrarily make predictions for other users.

Suggestion Apply some access control mechanism.

2.1.2 Being unable to remove markets

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `EquitiesPredictionOpsManager` and the `PredictionOpsManager` contracts are used for off-chain bots to automatically execute rounds for the prediction markets. The markets must be registered for query and execution using the `setPredicitionMarket` function. However, there is no way to disable or remove a registered prediction market.

```
16    constructor(IBhavishEquitiesPrediction[] memory _bhavishPrediction) {
17        for (uint256 i = 0; i < _bhavishPrediction.length; i++) {
18            setPredicitionMarket(_bhavishPrediction[i]);
19        }
20    }
21
22    function setPredicitionMarket(IBhavishEquitiesPrediction _bhavishPredicition) public onlyOwner
          {
```

```
23        require(address(_bhavishPredicition) != address(0), "Invalid predicitions");
24
25        predictionMarkets.push(_bhavishPredicition);
26    }
```

Listing 2.4: EquitiesPredictionOpsManager.sol

```
12    constructor(IBhavishPrediction[] memory _bhavishPrediction) {
13        for (uint256 i = 0; i < _bhavishPrediction.length; i++) {
14            setPredicitionMarket(_bhavishPrediction[i]);
15        }
16    }
17
18    function setPredicitionMarket(IBhavishPrediction _bhavishPredicition) public onlyOwner {
19        require(address(_bhavishPredicition) != address(0), "Invalid predicitions");
20
21        predictionMarkets.push(_bhavishPredicition);
22    }
```

Listing 2.5: PredictionOpsManager.sol

Impact Unused prediction markets cannot be removed.

Suggestion Add a governance function to remove unused prediction markets.

2.1.3 Unhandled corner case when calculating rewards

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `_calculateRewards` function of the `AbstractPrediction` contract (at line 343 and 353), the condition where `rewardAmount <= treasuryAmt` is not considered. Because `treasuryAmt` is derived from `round.totalAmount`, `rewardAmount` of the round may be smaller than `treasuryAmt`. For example, in the `AbstractNoLoss` contract, `rewardAmount` is either `upPredictAmount` or `downPredictAmount`. In this case, `treasuryAmt` can be larger than `rewardAmount`, so the fees are charged regardless of `rewardAmount`.

```
331    function _calculateRewards(uint256 _predictRoundId) internal {
332        Round memory round = bhavishPredictionStorage.getPredictionRound(_predictRoundId);
333        require(round.roundState == RoundState.ENDED, "Round is not ended");
334
335        uint256 rewardAmount = _getRoundRewardAmount(round);
336        uint256 treasuryAmt = (round.totalAmount * treasuryFee) / (10**decimals);
337
338        uint256 rewardBaseCalAmount;
339        // Bull wins
340        if (round.endPrice > round.startPrice) {
341            rewardBaseCalAmount = round.upPredictAmount;
342            // reward amount can be zero while treasury can be greater than reward for few cases
343            if (rewardAmount > 0 && rewardAmount > treasuryAmt) rewardAmount = rewardAmount -
                    treasuryAmt;
344            // case when there are no bets on winning side. loosing side bets should be moved to
                    treasury
```

```
345             if (rewardBaseCalAmount == 0) {
346                 treasuryAmt = round.downPredictAmount;
347                 rewardAmount = 0;
348             }
349         }
350         // Bear wins
351         else if (round.endPrice < round.startPrice) {
352             rewardBaseCalAmount = round.downPredictAmount;
353             if (rewardAmount > 0 && rewardAmount > treasuryAmt) rewardAmount = rewardAmount -
                    treasuryAmt;
354             // case when there are no bets on winning side. loosing side bets should be moved to
                    treasury
355             if (rewardBaseCalAmount == 0) {
356                 treasuryAmt = round.upPredictAmount;
357                 rewardAmount = 0;
358             }
359         }
360         // draw or tie
361         else {
362             rewardBaseCalAmount = 0;
363             rewardAmount = 0;
364         }
365
366         treasuryAmount += treasuryAmt;
367         bhavishPredictionStorage.setRewardAmountForRound(_predictRoundId, rewardAmount,
                rewardBaseCalAmount);
368
369         _updateCalculateRewards(rewardAmount + treasuryAmt, treasuryAmt);
370
371         emit RewardsCalculated(_predictRoundId, rewardBaseCalAmount, rewardAmount, treasuryAmt);
372     }
```

Listing 2.6: AbstractPrediction.sol

Impact It may lead to unexpected results due to the incorrect calculations of the fees and the reward amount.

Suggestion Take the corner case into consideration in the `_calculateRewards` function.

Feedback from the Project This will be a smaller amount, so users get the rewards without treasury fee. We are happy to give this smaller winning amount without treasury fee in noLoss.

2.1.4 No sanity checks for closed markets

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_placeBet` function of the `AbstractRT` contract, the close timestamp of the market is not considered. Specifically, there exists a case that the market has closed (i.e., the close timestamp of the market has reached) but not resolved (i.e., the `resolveMarket` function is not called). In such a case, users can place bets on a market that should have been closed. Because the result has been revealed (i.e., the return value of `resulfFor` function can be queried), users can always make correct predictions.

```
167    function _placeBet(
168        uint256 _marketId,
169        uint256 _outcomeId,
170        uint256 _amount,
171        address _provider
172    ) internal {
173        Market storage market = markets[_marketId];
174        require(block.timestamp >= market.opensAtTimestamp, "market not opened yet");
175        require(market.state == MarketState.OPEN, "event not open for prediction");
176        require(_outcomeId < market.outcomeIds.length, "invalid outcome");
177
178        market.balance += _amount;
179        market.outcomes[_outcomeId].amount += _amount;
180        market.outcomes[_outcomeId].traderStakes[msgSender()] += _amount;
181        userMarkets[msgSender()].push(_marketId);
182
183        emit BetPlaced(msgSender(), _marketId, _outcomeId, _amount);
184        emit ProviderInfo(_provider, _amount);
185    }
```

Listing 2.7: AbstractRT.sol

Impact It may allow the users to always predict correct results.

Suggestion Prevent users from placing bets on markets that have been closed.

2.1.5 Incorrect argument passed to the `refundUsers` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `performRefund` function of the `BhavishLossyVault` contract is used to refund multiple rounds in a batch for the vault. However, when the function calls the `refundUsers` function of the Bhavish SDK, the round ID is always the first item in the `roundIds` array (see line 76 in the following code snippet).

```
70    function performRefund(uint256[] calldata roundIds) external override onlyOperator(msg.sender)
          {
71        uint256 beforeBalance = address(this).balance;
72
73        for (uint256 i = 0; i < roundIds.length; i++) {
74            bhavishSDK.refundUsers(
75                IBhavishSDK.PredictionStruct(assetPair.underlying, assetPair.strike, 0, false),
76                roundIds[0]
77            );
78        }
79
80        uint256 totalRefundAmount = address(this).balance - beforeBalance;
81        vaultDeposit.totalDeposit += totalRefundAmount;
82    }
```

Listing 2.8: BhavishLossyVault.sol

Impact The `performRefund` function would only refund the first round provided in the `roundIds` parameter.

Suggestion Fix the `roundIds` iteration logic.

2.1.6 Ineffective prediction market updating

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description `AbstractBhavishSDK` is the parent contract for all the SDKs, including `NativeSDK` and `ERC20SDK`. Specifically, the `AbstractBhavishSDK` contract records the active prediction markets in a variable named `activePredictionMap`. However, in the `updatePredictionMarket` function of the `AbstractBhavishSDK` contract, when the prediction contract for a given pair is updated, the `activePredictionMap` variable is not updated accordingly. As a result, the newly updated prediction market cannot be used.

```
56    function updatePredictionMarket(
57        IBhavishPrediction _bhavishPrediction,
58        bytes32 _underlying,
59        bytes32 _strike
60    ) external onlyAdmin(msg.sender) {
61        require(address(predictionMap[_underlying][_strike]) != address(0), "Prediction market
              doesn't exist");
62        predictionMap[_underlying][_strike] = _bhavishPrediction;
63    }
```

Listing 2.9: AbstractBhavishSDK.sol

As shown in the below code snippet, any prediction on the underlying and the strike pair would check `activePredictionMap`. Therefore, invoking the `predict` function on an updated market would revert because `predictionMap` is modified while `activePredictionMap` is not updated accordingly.

```
43    function predict(
44        PredictionStruct memory _predStruct,
45        address _userAddress,
46        address _provider
47    ) external payable override {
48        IBhavishPrediction bhavishPrediction = predictionMap[_predStruct.underlying][_predStruct.
              strike];
49
50        require(address(bhavishPrediction) != address(0), "Prediction Market for the asset is not
              present");
51        require(activePredictionMap[bhavishPrediction], "Prediction Market for the asset is not
              active");
52
53        address userAddress_;
54        if (address(msg.sender).isContract()) {
55            userAddress_ = _userAddress;
56        } else {
57            require(msg.sender == _userAddress, "Buyer and msg.sender cannot be different");
58            userAddress_ = msg.sender;
59        }
60
```

```
61        if (_predStruct.directionUp) bhavishPrediction.predictUp{ value: msg.value }(_predStruct.
              roundId, userAddress_);
62        else bhavishPrediction.predictDown{ value: msg.value }(_predStruct.roundId, userAddress_);
63
64        _populateProviderInfo(_provider, msg.value);
65    }
```

Listing 2.10: NativeSDK.sol

Impact The prediction market update would be ineffective and the corresponding underlying-strike pair would be unusable.

Suggestion Properly update the `activePredictionMap` variable.

2.2 DeFi Security

2.2.1 No access control for the `reinvest` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `reinvest` function of the `BhavishReInvest` contract does not have any access control.

```
20    function reinvest(address _user) external {
21        uint256 beforeBalance = address(this).balance;
22        pool.claimWinningRewards(_user);
23        uint256 afterBalance = address(this).balance;
24        pool.deposit{ value: (afterBalance - beforeBalance) }(_user);
25    }
```

Listing 2.11: BhavishReInvest.sol

Impact User rewards can be arbitrarily claimed.

Suggestion Use a whitelist or disable arbitrarily claiming rewards for others.

2.2.2 Potential precision loss in share calculation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is a precision loss in the `convertToShares` function of the `AbstractPool` contract. At line 117 in the following code snippet, there would be a precision loss because the number of shares is calculated by `_amount / 10**yieldToken.decimals()`. The `_amount` parameter represents the "amount of tokens", which may have the same precision as the yield token. In this case, the number of shares being calculated is actually "the number of tokens" without any precision. For example, if `_amount` is `1e18` (which means 1 token), the number of shares would be exactly `1` without any precision. Hence there exist three cases which may lead to financial losses to the users, as follows:

- If there is no share in the pool (i.e., `poolData.shares` is zero), the number of shares being calculated will always be zero if the deposit amount is less than `10**yieldToken.decimals()`.
- Even if the number of shares in the pool is not zero, the deposits that are not exactly integer multiples of `10**yieldToken.decimals()` would suffer from precision losses as well. Because the decimals part would be lost.
- An attacker can reinvest other users' rewards which are less than `10**yieldToken.decimals()` (see Issue 2.2.1) to make the rewards distributed to all the stakers (i.e., their numbers of shares are greater than zero).

```
112   function convertToShares(uint256 _amount) public view returns (uint256) {
113       IERC20Extended yieldToken = IERC20Extended(farm.getYieldToken());
114       uint256 supply = poolData.shares;
115       return
116           (_amount == 0 || supply == 0)
117               ? _amount.mulDiv(1e0, 10**yieldToken.decimals(), Math.Rounding.Down)
118               : _amount.mulDiv(supply, totalAssets(), Math.Rounding.Down);
119   }
```

Listing 2.12: AbstractPool.sol

Impact May lead to financial losses to the users.

Suggestion Revise the share calculation.

2.2.3 Token balance manipulation in the `AbstractPool` contract

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `getProfit` function of the `AbstractPool` contract, the return value of the function is calculated by the yield token balance of this contract minus the total reserves recorded in the contract (i.e., values recorded in `poolData`). Therefore, the result can be manipulated by directly transferring the yield token to the contract. The manipulation may cause negative effects like the issue in the Compound Protocol [1].

```
165function getProfit() public view returns (uint256 profit) {
166    profit =
167        IERC20Extended(farm.getYieldToken()).balanceOf(address(this)) -
168        (poolData.totalLiquidity + poolData.providerPoolProfit + poolData.rewardPool + poolData.
               protocolPool);
169}
```

Listing 2.13: AbstractPool.sol

Impact Token balance manipulation may lead to unexpected results.

Suggestion N/A

[1] https://blog.openzeppelin.com/compound-comprehensive-protocol-audit/#ceth-and-cerc20-underlying-balances-can-be-manipulated

2.2.4 Locked admin fees

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `BhavishLossyPool` contract allows users to swap the native token for `BhavishGameToken`. The `depositForAdmin` function is used to record the admin fees, while the `withdrawForAdmin` function is used to withdraw those fees. However, the admin fees would be locked due to the flawed logic of the `withdrawForAdmin` function.

Specifically, this contract maintains two accounting systems, i.e., the token accounting system and the native accounting system stored in a mapping variable named `providers`, respectively. When one user deposit, the contract will mint `BhavishGameToken` to the user and make `providers[user].amount` increase simultaneously. However, the `depositForAdmin` function only adds the `providers[user].amount` without minting `BhavishGameToken`, which means the corresponding balance of the token accounting system remains unchanged.

```
49  function depositForAdmin(uint256 _amount) external {
50    require(predictionContracts[msg.sender], "invalid caller");
51    ProviderDetails storage provider = providers[admin];
52    provider.amount += _amount;
53}
```

Listing 2.14: BhavishLossyPool.sol

As a result, when invoking the `withdrawForAdmin` function, zero `balance` would be transferred to the user (at line 133 of Listing 2.16).

```
55  function withdrawForAdmin(address _admin) external nonReentrant {
56    require(_admin == address(admin), "cannot withdraw");
57    _withdraw(_admin);
58}
```

Listing 2.15: BhavishLossyPool.sol

```
130  function _withdraw(address _user) internal {
131    ProviderDetails storage provider = providers[_user];
132    require(provider.date + poolData.liquidityLockupDuration <= block.timestamp, "cannot withdraw
           with in lockup");
133    uint256 balance = _getAmountToTransfer(_user);
134    uint256 rewards = getAPYRewards(_user);
135
136    if (provider.date + poolData.rewardLockupDuration <= block.timestamp) {
137        _withdrawFromFarm(rewards + balance, _user);
138        // deduct provider pool profit
139        emit ClaimAPY(_user, rewards);
140    } else {
141        _withdrawFromFarm(balance, _user);
142        poolData.protocolPool += rewards;
143    }
144    // update pool data
145    poolData.totalLiquidity -= getBalance(_user) / poolData.multiplier;
```

```
146     poolData.providerPoolProfit -= rewards;
147     provider.amount = 0;
148     poolData.shares -= provider.shares;
149     provider.shares = 0;
150
151     afterWithdraw();
152
153     emit Withdraw(msg.sender, balance);
154 }
```

Listing 2.16: AbstractPool.sol

```
19  function _getAmountToTransfer(address _user) internal view override returns (uint256) {
20    return balanceOf(_user) / poolData.multiplier;
21  }
```

Listing 2.17: BhavishLossyPool.sol

Impact The admin fees in the `BhavishLossyPool` contract will be locked.

Suggestion Revise the code logic.

2.2.5 Inconsistent state update

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `BhavishNativeVault` contract, the `userDeposits` variable is updated at the end of the `withdraw` function. However, this step is missing in the `withdrawAsset` function of the `BhavishLossyVault`.

```
105     function withdraw(uint256 _shares) external nonReentrant {
106       uint256 assetAmount = _withdrawFromVault(_shares);
107       _safeTransfer(msg.sender, assetAmount);
108       vaultDeposit.userDeposits[msg.sender] = convertToAssets(vaultDeposit.userShares[msg.sender])
              ;
109   }
```

Listing 2.18: BhavishNativeVault.sol

```
95    function withdrawAsset(uint256 shares) external nonReentrant {
96      uint256 _amount = convertToAssets(shares);
97
98      require(_amount <= bgToken.balanceOf(address(this)), "cannot withdraw > contract balance");
99
100     _withdrawFromVault(shares);
101   }
```

Listing 2.19: BhavishLossyVault.sol

Impact The `BhavishLossyVault` may not function as expected.

Suggestion Make the state update be consistent with each other.

2.2.6 Incorrect calculation for token balance changes

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AbstractLossy` prediction market contract, the market refunds users with GameToken (through the `_amountTransfer` function).

```
82  function _amountTransfer(address _user, uint256 _amount) internal override {
83    token.safeTransfer(_user, _amount);
84  }
```

Listing 2.20: AbstractLossy.sol

However, in the `performRefund` function of the `BhavishLossyVault` contract, `totalRefundAmount` is calculated based on `address(this).balance` rather than the token balance. The same issue also exists in the `performClaim` function of the `BhavishLossyVault`.

```
70  function performRefund(uint256[] calldata roundIds) external override onlyOperator(msg.sender) {
71    uint256 beforeBalance = address(this).balance;
72
73    for (uint256 i = 0; i < roundIds.length; i++) {
74        bhavishSDK.refundUsers(
75            IBhavishSDK.PredictionStruct(assetPair.underlying, assetPair.strike, 0, false),
76            roundIds[0]
77        );
78    }
79
80    uint256 totalRefundAmount = address(this).balance - beforeBalance;
81    vaultDeposit.totalDeposit += totalRefundAmount;
82}
```

Listing 2.21: BhavishLossyVault.sol

```
61  function performClaim(uint256[] calldata roundIds) external override onlyOperator(msg.sender) {
62    uint256 beforeBalance = address(this).balance;
63
64    bhavishSDK.claim(IBhavishSDK.PredictionStruct(assetPair.underlying, assetPair.strike, 0, false
          ), roundIds);
65
66    uint256 claimedAmount = address(this).balance - beforeBalance;
67    vaultDeposit.totalDeposit += claimedAmount;
68}
```

Listing 2.22: BhavishLossyVault.sol

Impact The token balance changes of the contract cannot be correctly calculated.

Suggestion Revise the code logic.

2.3 Additional Recommendation

2.3.1 Remove redundant checks

Status Acknowledged

Introduced by Version 1

Description In the AbstractEP contract, the checks curRound.roundState != RoundState.CANCELLED in line 186 and line 190 are unnecessary because they are checked in line 181.

```
167    function closeCurrentRound() private returns (uint256 closingPrice) {
168        Round memory curRound = bhavishPredictionStorage.getPredictionRound(currentRoundId);
169
170        // currentRoundId refers to current round n
171        // fetch price to end current round and start new round
172        (closingPrice, ) = bhavishPriceManager.getPrice(
173            assetPair.underlying,
174            assetPair.strike,
175            curRound.roundEndTimestamp
176        );
177
178        if (
179            curRound.roundState == RoundState.CREATED ||
180            curRound.roundState == RoundState.ENDED ||
181            curRound.roundState == RoundState.CANCELLED
182        ) return closingPrice;
183
184        // End and Disperse current round
185        // skip for non current rounds
186        if (curRound.roundState != RoundState.CANCELLED && closingPrice != 0) {
187            _endRound(currentRoundId, closingPrice);
188
189            _calculateRewards(currentRoundId);
190        } else if (curRound.roundState != RoundState.CANCELLED && closingPrice == 0) {
191            _cancelRound(currentRoundId);
192        }
193    }
```

Listing 2.23: AbstractEP.sol

Impact N/A

Suggestion Remove redundant checks.

2.3.2 Avoid mixed use of msg.sender and the msgSender function

Status Fixed in Version 2

Introduced by Version 1

Description In the AbstractRT contract, the msgSender function is used to support the *gasless transaction feature* [2]. However, there is a mixed usage of msg.sender and the msgSender function. Specifically, the _placeBet internal function can collaborate with gasless transactions feature, while the claim function cannot (otherwise, the claimed rewards would be locked in the delegate contract).

[2]The concept is similar to EIP-2771: https://eips.ethereum.org/EIPS/eip-2771, which allows a trusted contract to delegate user requests.

```
167    function _placeBet(
168        uint256 _marketId,
169        uint256 _outcomeId,
170        uint256 _amount,
171        address _provider
172    ) internal {
173        Market storage market = markets[_marketId];
174        require(block.timestamp >= market.opensAtTimestamp, "market not opened yet");
175        require(market.state == MarketState.OPEN, "event not open for prediction");
176        require(_outcomeId < market.outcomeIds.length, "invalid outcome");
177
178        market.balance += _amount;
179        market.outcomes[_outcomeId].amount += _amount;
180        market.outcomes[_outcomeId].traderStakes[msgSender()] += _amount;
181        userMarkets[msgSender()].push(_marketId);
182
183        emit BetPlaced(msgSender(), _marketId, _outcomeId, _amount);
184        emit ProviderInfo(_provider, _amount);
185    }
186
187    function getRewards(uint256 _marketId, address _user) public view override returns (uint256
         rewards, bool claimed) {
188        Market storage market = markets[_marketId];
189        MarketOutcome storage outcome = market.outcomes[market.resolution.outcomeId];
190        claimed = outcome.claimed[msg.sender];
191
192        if (market.state == MarketState.RESOLVED && !claimed)
193            rewards = (outcome.traderStakes[_user] * market.reward) / outcome.amount;
194    }
195
196    function claim(uint256 _marketId) external {
197        Market storage market = markets[_marketId];
198        require(market.state != MarketState.OPEN, "event still open");
199        MarketOutcome storage outcome = market.outcomes[market.resolution.outcomeId];
200
201        require(outcome.claimed[msg.sender] == false, "user already claimed");
202        require(outcome.traderStakes[msg.sender] > 0, "no predict amount for user");
203        (uint256 rewards, ) = getRewards(_marketId, msg.sender);
204
205        outcome.claimed[msg.sender] = true;
206        uint256 traderAmount = outcome.traderStakes[msg.sender];
207
208        _claim(_marketId, traderAmount, rewards);
209    }
```

Listing 2.24: AbstractRT.sol

Impact May lead to misunderstanding or misuse.

Suggestion Revise the code logic.

2.3.3 Remove redundant calculation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are unnecessary calculation in the `_calcAmount` function of the `BhavishLossyPool` contract.

```
23    function getAPYRewards(address _provider) public view override returns (uint256 rewards) {
24        if (_calcAmount(_provider) > (balanceOf(_provider) / poolData.multiplier)) {
25            rewards = _calcAmount(_provider) - (balanceOf(_provider) / poolData.multiplier);
26        }
27    }
28
29    function getBalance(address _provider) public view override returns (uint256 amount) {
30        amount = balanceOf(_provider);
31    }
32
33    function _calcAmount(address _provider) internal view returns (uint256) {
34        uint256 amount = convertToAssets(providers[_provider].shares);
35        if (amount > providers[_provider].amount)
36            //  APY on deposited amount + bg token balance
37            return (balanceOf(_provider) / poolData.multiplier) + amount - providers[_provider].
                  amount;
38        return balanceOf(_provider) / poolData.multiplier;
39    }
```

Listing 2.25: BhavishLossyPool.sol

Impact Redundant calculation may cause extra gas usage and logical misleading.

Suggestion Remove redundant calculation.

2.4 Note

2.4.1 Do not change the multiplier parameter

Description In the `BhavishLossyPool` and `BhavishNoLossPool` contracts, the `multiplier` parameter should not be modified. Because the modification can drastically affect the users' token amount for both minting and burning.

```
19    function _getAmountToTransfer(address _user) internal view override returns (uint256) {
20        return balanceOf(_user) / poolData.multiplier;
21    }
22
23    function getAPYRewards(address _provider) public view override returns (uint256 rewards) {
24        if (_calcAmount(_provider) > (balanceOf(_provider) / poolData.multiplier)) {
25            rewards = _calcAmount(_provider) - (balanceOf(_provider) / poolData.multiplier);
26        }
27    }
28
29    function getBalance(address _provider) public view override returns (uint256 amount) {
30        amount = balanceOf(_provider);
```

```
31    }
32
33    function _calcAmount(address _provider) internal view returns (uint256) {
34        uint256 amount = convertToAssets(providers[_provider].shares);
35        if (amount > providers[_provider].amount)
36            // APY on deposited amount + bg token balance
37            return (balanceOf(_provider) / poolData.multiplier) + amount - providers[_provider].
                  amount;
38        return balanceOf(_provider) / poolData.multiplier;
39    }
40
41    function afterDeposit(address _user) internal override {
42        _mint(_user, msg.value * poolData.multiplier);
43    }
```

Listing 2.26: BhavishLossyPool.sol

2.4.2 Ensure the functionality of the price manager

Description The `AbstractEP` contract retrieves the price of the NYSC equities from the price manager. However, as of this writing, Chainlink only officially provides price information for the selected NYSC equities on Polygon [3]. Though there are third-party price sources, the validity of these prices may not be ensured. It is important to guarantee the functionality of the price manager.

2.4.3 Design of the `BhavishNoLossPool` contract

Description The `BhavishNoLossPool` contract inherits from `AbstractPool` and `BhavishNoLossGameToken`. As a pool, it allows users to swap the native token for GameToken to predict prices in the markets. The pool maintains two accounting systems, i.e., the GameToken ERC20 and the user deposit, respectively. It is worth noting that, in the `BhavishNoLossPool` contract, the users can withdraw their deposits even if they lose all GameTokens in the prediction games.

Specifically, the `_withdraw` function is designed to withdraw the deposit based on the user balance. The user balance comes from the return value of the `_getAmountToTransfer` function. This value is recorded in a storage variable, i.e., `providers[_user].amount`, which will only be updated when there is a deposit. Since the `BhavishNoLossGameToken` contract does not implement the transfer hooks to modify `providers[_user].amount`, a user can withdraw all his principal with zero GameToken balance.

```
125function withdraw() external nonReentrant {
126    require(providers[msg.sender].amount > 0, "no liquidity for user");
127    _withdraw(msg.sender);
128}
129
130function _withdraw(address _user) internal {
131    ProviderDetails storage provider = providers[_user];
132    require(provider.date + poolData.liquidityLockupDuration <= block.timestamp, "cannot withdraw
          with in lockup");
133    uint256 balance = _getAmountToTransfer(_user);
```

[3] See: https://docs.chain.link/docs/data-feeds/price-feeds/addresses/?network=polygon

```
134     uint256 rewards = getAPYRewards(_user);
135
136     if (provider.date + poolData.rewardLockupDuration <= block.timestamp) {
137         _withdrawFromFarm(rewards + balance, _user);
138         // deduct provider pool profit
139         emit ClaimAPY(_user, rewards);
140     } else {
141         _withdrawFromFarm(balance, _user);
142         poolData.protocolPool += rewards;
143     }
144     // update pool data
145     poolData.totalLiquidity -= getBalance(_user) / poolData.multiplier;
146     poolData.providerPoolProfit -= rewards;
147     provider.amount = 0;
148     poolData.shares -= provider.shares;
149     provider.shares = 0;
150
151     afterWithdraw();
152
153     emit Withdraw(msg.sender, balance);
154 }
```

Listing 2.27: AbstractPool.sol

```
24     function _getAmountToTransfer(address _user) internal view override returns (uint256) {
25         return providers[_user].amount;
26     }
```

Listing 2.28: BhavishNoLossPool.sol

Feedback from the Project The design of the NoLoss market is a capital-protected market. Let's consider a user deposits 100 MATIC, and for example, let's say it generates 1 MATIC per day as a reward (AAVE). The user is playing with his reward amount instead of original deposit of 100 MATIC.

2.4.4 Stateless assumption of the `MinimalForwarder` contract

Description The `execute` function of the `MinimalForwarder` contract can be utilized for arbitrary calls. It is not considered as an issue in this report because this contract is designed to be stateless, which means the contract has no crypto asset and the execution privilege needs to be strictly restricted.

```
41     function execute(ForwardRequest calldata req, bytes calldata signature)
42         public
43         payable
44         returns (bool, bytes memory)
45     {
46         require(verify(req, signature), "MinimalForwarder: signature does not match request");
47         require(msg.value == req.value, "Mismatched request and actual value");
48
49         _nonces[req.from] = req.nonce + 1;
50
51         (bool success, bytes memory returndata) = req.to.call{ gas: req.gas, value: req.value }(
52             abi.encodePacked(req.data, req.from)
53         );
```

```
54        // Validate that the relayer has sent enough gas for the call.
55        // See https://ronan.eth.link/blog/ethereum-gas-dangers/
56        assert(gasleft() > req.gas / 63);
57        /*
58        // This change is added by bhavish team
59        // to show the error in explorer for failed txns
60        // otherwise the txn in explorer just shows Fail
61        */
62        require(success, string(returndata));
63
64        return (success, returndata);
65    }
```

Listing 2.29: MinimalForwarder.sol

2.4.5 RT market is closed on receiving invalid results

Description The `AbstractRT` contract provides the base contract for the "real time prediction market". The real time prediction market relies on the `Reality.eth` project for reflecting off-chain facts to the on-chain contracts. Users can make predictions for the results before the final answer is revealed.

Specifically, in the real time prediction market, a market is created with a single associated question. The answer to this question would be revealed when the market is closed (but before the market is resolved).

- When a market is created, the `createMarket` function will invoke the `askQuestionWithMinBond` function of the `Reality.eth` project to attach the associated question.
- When the market is resolved, the `resolveMarket` function will invoke the `resultFor` function to retrieve the final result for the question.

Note that the `resultFor` function may return invalid results to the `resolveMarket` function. However, the invalid result is not handled in the code logic (e.g., may need refund for an invalid result).

```
144    function resolveMarket(uint256 _marketId) external override onlyOperator(msg.sender) {
145        Market storage market = markets[_marketId];
146        require(block.timestamp >= market.closesAtTimestamp, "market not ended yet");
147
148        // resolve market from resolver (reality eth)
149        // returns answer if finalised
150        // else returns error
151        uint256 outcomeId = uint256(marketResolver.resultFor(market.resolution.questionId));
152        market.resolution.outcomeId = outcomeId;
153        market.state = MarketState.RESOLVED;
154
155        _calculateRewards(_marketId);
156
157        emit MarketResolved(_marketId, outcomeId, market.question);
158    }
```

Listing 2.30: AbstractRT.sol

Feedback from the Project If there is no result for market, we will close it and provide refund, instead of resolving it.