



Smart Contract Security Audit

Polkamarkets

2021-11-04

| | |
|---|----|
| 1. Introduction | 3 |
| 2. Disclaimer | 3 |
| 3. Scope..... | 3 |
| 4. Conclusions..... | 4 |
| 5. Issues and Recommendations | 6 |
| PMSC01 – Missing Ranges Check | 6 |
| PMSC02 – Improvable Comparisons | 7 |
| PMSC03 – Lack of Inputs Validation..... | 7 |
| PMSC04 – Reentrancy..... | 9 |
| PMSC05 – Comment Mismatch..... | 10 |
| PMSC06 – Use of Require Statement without Message | 10 |
| PMSC07 – Outdated Compiler Version..... | 11 |
| PMSC08 – GAS Optimization | 11 |
| Dead Code | 12 |
| Storage Optimization | 13 |
| Unnecessary payable modifier | 13 |
| Wrong Visibility | 13 |
| Duplicated Logic..... | 14 |
| PMSC09 – Code Style | 15 |

1. Introduction

Polkamarkets is a DeFi-Powered Prediction Market built for cross-chain information exchange and trading where users can take positions on outcomes of real-world events in a decentralized and interoperable platform based on Polkadot.



Polkamarkets aims to solve the low usage & volume problems by incentivising liquidity providers and traders to facilitate & take large positions, while a system for curation and resolution ensures efficient and trustworthy markets.

As requested by Polkamarkets and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit in order to evaluate the security of the Prediction Market smart contract.

2. Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

3. Scope

The scope of this evaluation includes the following smart contract:

- PredictionMarket.sol
SHA256: D92D7A29F601BD4EAB0452254F6FB59F5DF843EC1AB510E64AB07D0F1678F194

The fixes applied were address in the following repository:

- <https://github.com/bepronetwork/bepro-js/tree/feature/prediction-markets-red4sec-changes>

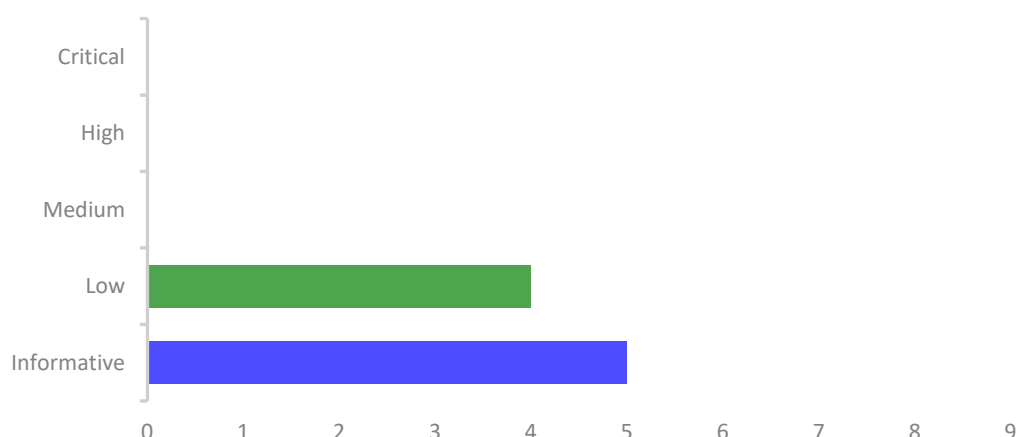
4. Conclusions

To this date, 4th of November 2021, the general conclusion resulting from the conducted audit is that **Polkamarket's smart contract is secure**. Nevertheless, there are a few highlights and possible improvements worth mentioning, these do not pose any risk by themselves, and we have classified such issues as informative only, but they will help Polkamarkets to continue to improve the security and quality of the project.

- Some methods **do not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- The proper functioning of the **PredictionMarket** relies on external contracts (*RealitioERC20*) which are out of scope in the current audit, as well as the responsibility to properly resolve the markets.
- The work and collaboration by the Polkamarkets technical team has been excellent since the beginning of the audit, solving some of the issues in a short period of time.
- A few of the found **issues** in the **audit have been reviewed and corrected** by the Polkamarkets team and subsequently reviewed and verified by the Red4Sec team. As communicated by the client, the **assumed issues** will be applied in version 2 of the protocol

Found issues have been classified in the following levels of risk according to the impact level defined by CVSS v3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST).

VULNERABILITY SUMMARY



Below we have a complete list of the issues detected by Red4Sec, presented and summarized in a way that can be used for risk management and mitigation.

A continuous process was created during the audit in order to improve the security of the PredictionMarket project. Polkamarkets's team has fixed some issues detailed in this report and the current status of the contract, after the review made by the Red4Sec team, is as follows:

| Table of vulnerabilities | | | |
|--------------------------|--|-------------|-----------|
| Id. | Vulnerability | Risk | State |
| PMSC01 | Missing Ranges Check | Low | Fixed |
| PMSC02 | Improvable Comparisons | Low | Fixed |
| PMSC03 | Lack of Inputs Validation | Low | Partially |
| PMSC04 | Reentrancy Attack | Low | Assumed |
| PMSC05 | Comment Mismatch | Informative | Fixed |
| PMSC06 | Use of Require statement without message | Informative | Fixed |
| PMSC07 | Outdated Compiler Version | Informative | Assumed |
| PMSC08 | GAS Optimization | Informative | Assumed |
| PMSC09 | Code Style | Informative | Assumed |

5. Issues and Recommendations

PMSC01 – Missing Ranges Check (**Low**)

Every time we carry out operations or receive external values, which are expected to be in a certain range of values, it is convenient to verify that the result of the operations is always within the mentioned range.

This is the case of *outcomeId* received from the external call to *realitio* where the result should be verified to be 0 or 1.

```
Market storage market = markets[marketId];

RealitioERC20 realitio = RealitioERC20(realitioAddress);
// will fail if question is not finalized
uint256 outcomeId = uint256(realitio.resultFor(market.resolution.questionId));

market.resolution.outcomeId = outcomeId;

emit MarketResolved(msg.sender, marketId, outcomeId, now);
emitMarketOutcomePriceEvents(marketId);

return market.resolution.outcomeId;
```

The *nextState* function should also verify that the resulting state never exceeds 2 or the function would fail.

```
/// @dev Transitions market to next state
function nextState(uint256 marketId) private {
    Market storage market = markets[marketId];
    market.state = MarketState(uint256(market.state) + 1);
}
```

Source reference

- PredictionMarket.sol: 500, 651

The correction has been applied in the following commit:

- <https://github.com/bepronetwork/bepronetwork/commit/194fcd6e95f7ebfe1ffbafe8885afd67e2ca79c19>

PMSC02 – Improvable Comparisons (Low)

There are two *require* in the *createMarket* method that must be improved, first it should be verified that the *closesAt* variable is greater, not greater or equal, to avoid that the same transaction creates a market and closes, this behaviour could allow a flash loan to perform unexpected actions.

Secondly, the *arbitrator* variable must be checked to be different from *address(0)*, not equal to itself.

```
require(msg.value > 0, "stake needs to be > 0");  
require(closesAt >= now, "market must resolve after the current date");  
require(arbitrator == address(arbitrator), "invalid arbitrator address");
```

Additionally, in the *buy* and *sell* methods there should be verification stating that *shares > 0*, this will prevent unwanted shares, under certain circumstances (*fee* = 0), with accounts that do not have or will not have shares.

Source reference

- PredictionMarket.sol: 192, 193, 293

The correction has been applied in the following commit:

- <https://github.com/bepronetwork/bepro-js/commit/362bcecb41a06a3858647f2ce3eaf48c44a4252d>

PMSC03 – Lack of Inputs Validation (Low)

Some methods of the different contracts in the **Polkamarkets** project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

- During the logic of the constructor in the contract, none of the received arguments are verified. For example, *fee*, *_requiredBalance*, *_realtioTimeout*, ... to be greater than zero.

Source reference

- PredictionMarket.sol: 161
- Additionally, it is recommended to verify the value of the variable *realtioTimeout* in the constructor, since the value must be more than 0 and

less than 365 days, as can be seen in the following *require* within the *_askQuestion* method of the **RealitioERC20** contract.

```
// A timeout of 0 makes no sense, and we will use this to check existence
require(timeout > 0, "timeout must be positive");
require(timeout < 365 days, "timeout must be less than 365 days");
```

Source reference

- PredictionMarket.sol: 168

The correction has been applied in the following commit:

- <https://github.com/bepronetwork/bepro-js/commit/e668338ab4fee9cc2ba87cdda6fc10fcac4cf51a>
- In some methods of the **Polkamarkets** project, the *marketId* value sent through the arguments is not checked to be valid, so it allows the use of non-existent *marketId*, it is recommended to use the *isMarket* modifier provided for this.
- These methods should reverse the execution in case the market does not exist, otherwise the queries by third party contracts will obtain erroneous results if they use non-existent markets or outcomes.
In addition, it is convenient that the revert is carried out with a clear message and not due to errors in the arithmetical operations (safeMath revert).

The following methods return erroneous values if *marketId* does not exist:

- getUserMarketShares
- getMarketResolvedOutcome
- isMarketVoided
- getMarketData
- getMarketAltData
- getMarketOutcomeIds

The following methods return erroneous values if *outcomeId* does not exist:

- getMarketOutcomePrice
- getMarketOutcomeData

In the following methods a revert must be carried out if *marketId* does not exist, otherwise it will fail due to safeMath.

- getUserClaimStatus
- getUserLiquidityPoolShare
- getUserClaimableFees

- `getMarketPrices`
- `getMarketLiquidityPrice`
- `getMarketOutcomePrice`
- `getMarketOutcomeData`

Source reference

- `PredictionMarket.sol`: 131, 138, 143
- All methods that receive *marketId* from arguments.

PMSC04 – Reentrancy (Low)

The Reentrancy attack is an EVM vulnerability that occurs when external contract calls can make new calls to the calling contract before the initial execution is completed. For a function, this means that the state of the contract could change in the middle of its execution as a result of a call to an untrusted contract or the use of a low-level function with an external address.

The *removeLiquidity* method carries out a *transfer* on the *claimFees* call that can invoke a transfer, which could facilitate a reentrancy attack and bypass the require *market.liquidityShares[msg.sender] > shares*.

```
/// @dev Removes liquidity to a market - external
function removeLiquidity(uint256 marketId, uint256 shares)
    external
    payable
    timeTransitions(marketId)
    atState(marketId, MarketState.open)
{
    Market storage market = markets[marketId];

    require(market.liquidityShares[msg.sender] >= shares, "user does not have enough balance");
    // claiming any pending fees
    claimFees(marketId);
}
```

It would be convenient to add the *nonReentrant()*¹ modifier of Open Zeppelin to avoid this type of attack in the functions that perform any type of token transfer.

Source reference

- `PredictionMarket.sol`: 435

¹ <https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard>

PMSC05 – Comment Mismatch (Informative)

During the security audit, it has been detected that one of the comments found in the source code does not correspond to the context of the fee field.

This comment refers to the fact that the fee variable can be updated by the owner of the contract, however there is neither an owner in the contract nor a method to update it.

```
// governance
uint256 public fee; // fee % taken from every transaction, can be updated by contract owner
// realtio configs
address public realtioAddress;
uint256 public realtioTimeout;
```

In order to have a well-documented code it is recommended to check all the code thoroughly to find mistakes of this type.

Source reference

- PredictionMarket.sol: 116

The correction has been applied in the following commit:

- <https://github.com/bepronetwork/bepro-js/commit/7a1f4c9ec333573d67d7831e5526de2db5b22044>

PMSC06 – Use of Require Statement without Message (Informative)

Throughout the audit, it was verified that the reason message is not specified in some require methods, in order to give the user more information, which consequently makes it more user friendly.

```
modifier isMarket(uint256 marketId) {
    require(marketId < marketIndex);
    _;
}
```

This functionality is compatible since version *0.4.22* and the contract's pragma indicates *0.6.0*, this will result in compatibility with this feature.

Source reference

- PredictionMarket.sol: 127, 139, 144

The correction has been applied in the following commit:

- <https://github.com/bepronetwork/bepro-js/commit/f8856f113b7b9bf0185a3acf6cb56a18fe12f03c>

PMSC07 – Outdated Compiler Version (Informative)

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version.

We have detected that the audited contract uses the following version of solidity pragma `^0.6.0`:

```
pragma solidity ^0.6.0;  
pragma experimental ABIEncoderV2;
```

The *0.6.X* version of Solc is affected by different known bugs that have already been fixed in later versions. It is always a good policy to use the most up to date version of the pragma.

Keep in mind that updating the compiler version will require modifying some parts of the code, such as the alias *now* that is deprecated from version *0.7.0*, so it should be changed to *block.timestamp*.

Reference

<https://github.com/ethereum/solidity/blob/develop/Changelog.md>

PMSC08 – GAS Optimization (Informative)

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On EVM based blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to arithmetical operations and comparisons.

Dead Code

In programming, a part of the source code that is never used is known as dead code. The execution of this type of code consumes unnecessary GAS during deployment.

The *isMarket* modifier is not used during the execution of the contract, however the [lack of input validations](#) issue mentions that there is not a continuous check of whether the *marketId* exists or not, so it would be convenient to either eliminate the modifier or to use it.

Source reference

- PredictionMarket.sol: 126

Another clear example of dead or unnecessary code is in the *outcomes* argument of the *createMarket* method, since its value is required to be 2, this argument should not be necessary in the function.

Source reference

- PredictionMarket.sol: 184

Storage Optimization

The use of the *immutable*² keyword is recommended to obtain less expensive executions, by having the same behaviour as a constant. However, by defining its value in the constructor we have a significant save of GAS.

```
// governance
uint256 public fee; // fee % taken from every transaction, can be updated by contract owner
// realtio configs
address public realtioAddress;
uint256 public realtioTimeout;
// market creation
IERC20 public token; // token used for rewards / market creation
uint256 public requiredBalance; // required balance for market creation
```

Source reference

- PredictionMarket.sol: 116, 118, 119, 121, 122

Unnecessary payable modifier

During the audit, it has been identified that some methods of the **PredictionMarket** contract have the *payable* modifier, which results unnecessary since they are not destined to receive assets.

It is convenient to consider eliminating the *payable* modifier of these functions, to avoid sending funds during the call.

Source reference

- PredictionMarket.sol: 319, 427, 608

Wrong Visibility

In order to simplify the contract for the users, it is recommended to turn the following variables to private.

```
uint256 public constant MAX_UINT_256 =
    115792089237316195423570985008687907853269984665640564039457584007913129639935;
uint256 public constant ONE = 10**18;
```

Source reference

- PredictionMarket.sol: 49,51

² <https://docs.soliditylang.org/en/v0.6.5/contracts.html#immutable>

Duplicated Logic

The *createMarket* function performs a *require* of *msg.value > 0* on line 191, which will be performed again on the *addLiquidity* function on line 365, it is convenient to avoid these unnecessary GAS expenses, by reducing these verifications to the *addLiquidity* private function.

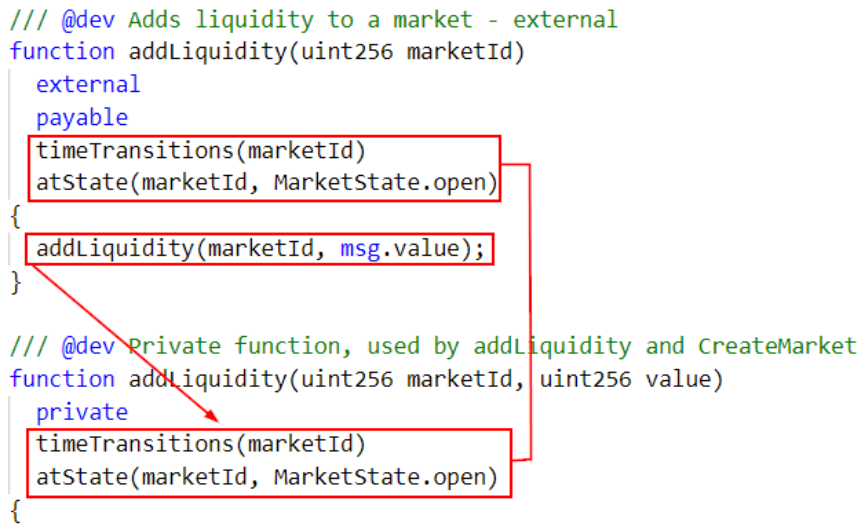
Source reference:

- PredictionMarket.sol: 191

The external method of *addLiquidity* contains the modifiers *timeTransitions* and *atState* which also own the private method that it calls.

```
/// @dev Adds liquidity to a market - external
function addLiquidity(uint256 marketId)
    external
    payable
    timeTransitions(marketId)
    atState(marketId, MarketState.open)
{
    addLiquidity(marketId, msg.value);
}

/// @dev Private function, used by addLiquidity and CreateMarket
function addLiquidity(uint256 marketId, uint256 value)
    private
    timeTransitions(marketId)
    atState(marketId, MarketState.open)
{
    // ...
}
```



Source reference

- PredictionMarket.sol: 351-352

PMSC09 – Code Style (Informative)

It has been possible to verify that, despite the good quality of the code, there are a few improvements to make the comprehension and code reading better.

As a reference, it is always recommendable to apply some coding style/good practices that can be found in multiple standards such as:

- “Solidity Style Guide” (<https://docs.soliditylang.org/en/v0.8.0/style-guide.html>).

These references are very useful to improve smart contract quality. Some of those practices are common and a popular accepted way to develop software.

The detected examples are listed below:

- The `MAX_UINT_256` variable contemplates the maximum possible integer in solidity, however, if someone is reading the contract it would be quite hard to understand what the number really means.

```
uint256 public constant MAX_UINT_256 =  
115792089237316195423570985008687907853269984665640564039457584007913129639935;
```

It is recommended to replace this value with: **`uint256 MAX_INT_256 = 2**256 - 1`**

Another alternative if the compiler version is updated (from solidity 0.6.8), is **`type(uint256).max`**

Source reference

- PredictionMarket.sol: 49
- In the `createMarket` method, it is advisable to upload the `require` at the beginning of the function, since they are independent of the `marketId` value and it will save GAS in some cases.

```

/// @dev Creates a market, initializes the outcome shares pool and submits a question in Realitio
function createMarket(
    string calldata question,
    string calldata image,
    uint256 closesAt,
    address arbitrator,
    uint256 outcomes
) external payable mustHoldRequiredBalance() returns (uint256) {
    ↑uint256 marketId = marketIndex;
    marketIds.push(marketId);

    Market storage market = markets[marketId];

    require(msg.value > 0, "stake needs to be > 0");
    require(closesAt >= now, "market must resolve after the current date");
    require(arbitrator == address(arbitrator), "invalid arbitrator address");
    // v1 - only binary markets
    require(outcomes == 2, "number of outcomes has to be 2");
}

```

Likewise, it is advisable to raise the increase made on line 232 above the function, for example, move it up to line 187.

```

    ↑
    // incrementing market array index
    marketIndex = marketIndex + 1;

```

Source reference

- PredictionMarket.sol: 232
- Since there are currently only two outcomes, it would be better to use two *struct* within the struct itself than to use an array, as it currently is.

```

return (
    market.liquidityShares[user],
    market.outcomes[0].shares.holders[user],
    market.outcomes[1].shares.holders[user]
);

```

Source reference

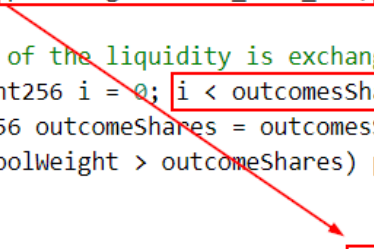
- PredictionMarket.sol: 749
- It is recommended to modify the logic related to the *poolWeight* calculation of *removeLiquidity* so that this variable starts at 0 instead of *type(uint).max*. In the hypothetical case of not having *outcomesShares*, the loop would not establish the value of said variable and a multiplication

would be performed that would produce a denial of service due to integer overflow, it is an impossible case since *outcomesShares* is always destined to have 2 positions, but the logic used is insecure and should not be reused at other times.

```
uint256 poolWeight = MAX_UINT_256;

// part of the liquidity is exchanged for outcome shares if market is not balanced
for (uint256 i = 0; i < outcomesShares.length; i++) {
    uint256 outcomesShares = outcomesShares[i];
    if (poolWeight > outcomesShares) poolWeight = outcomesShares;
}

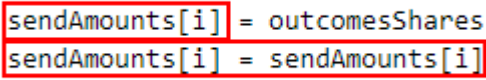
uint256 liquidityAmount = shares.mul(poolWeight).div(market.liquidity);
```



Source reference

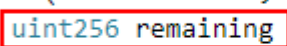
- PredictionMarket.sol: 443
- In the following image you can see how in an existing *for* loop in the *removeLiquidity* method, the *sendAmounts[i]* variable is declared and its value is then replaced with the result of an operation.

```
for (uint256 i = 0; i < outcomesShares.length; i++) {
    sendAmounts[i] = outcomesShares[i].mul(shares) / market.liquidity;
    sendAmounts[i] = sendAmounts[i].sub(liquidityAmount);
}
```



However, in the *addLiquidity* method within the same *for* loop, a new variable called *remaining* is declared, and the result of the final operation is the one assigned to the value of the *sendBackAmounts* array.

```
for (uint256 i = 0; i < outcomesShares.length; i++) {
    uint256 remaining = value.mul(outcomesShares[i]) / poolWeight;
    sendBackAmounts[i] = value.sub(remaining);
}
```



Although both options are valid and functional, it is advisable to use the same method in both cases in order to improve the readability of the code.

Source reference

- PredictionMarket.sol: 382,454



Invest in Security, invest in your future