Public

Report

04/05/2023

Prediction Market v2

Content

Introduction

Polkamarkets is a DeFi-Powered **Prediction Market** built for cross-chain information exchange and trading with the purpose of users taking positions on outcomes of real-world events in what aims to be 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 solicited by **Polkamarkets** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **Prediction Market v2** project.

This **Public Report** has been prepared for **Polkamarkets** and presents a summary of the results of the security of the project audited. It is based on the technical report which contains all the details and technical descriptions of the security assessment, this report does not contain all technical details and may not include all the issues identified during the audit.

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 either "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project, nor as guarantee on the operation or viability of the implemented financial product.

Red4Sec makes full effort and applies every resource available for each audit, however it does not warrant the function, nor the safety of the project and it cannot be deemed a sufficient assessment of the code's utility and safety, bug-free status, or any other declarations of the project. Additionally, Red4Sec makes no security assessments or judgments about the underlying business strategy, or the individuals involved in the project.

Blockchain technology and cryptographic assets come with their own new risks and challenges, where the ecosystem, platform, its programming language, and other software related to said technology can have vulnerabilities that could lead to exploits. As a result, the audit cannot guarantee the explicit security of the audited projects.

The audit reports can be used to improve the code quality of smart contracts, to help limit the vectors of attack and to lower the high level of risks associated with utilizing new and continually changing technologies such as cryptographic tokens and blockchain, but they are unable to detect any future security concerns with the related technologies.

Audit Scope

Red4Sec Cybersecurity has made a thorough audit of the **Prediction Market v2** security level against attacks, identifying possible errors in the design, configuration, or programming; therefore, guaranteeing the availability, integrity and confidentiality of the project and the possible assets treated and stored.

The scope of this evaluation includes the following items provided by **Polkamarkets**:

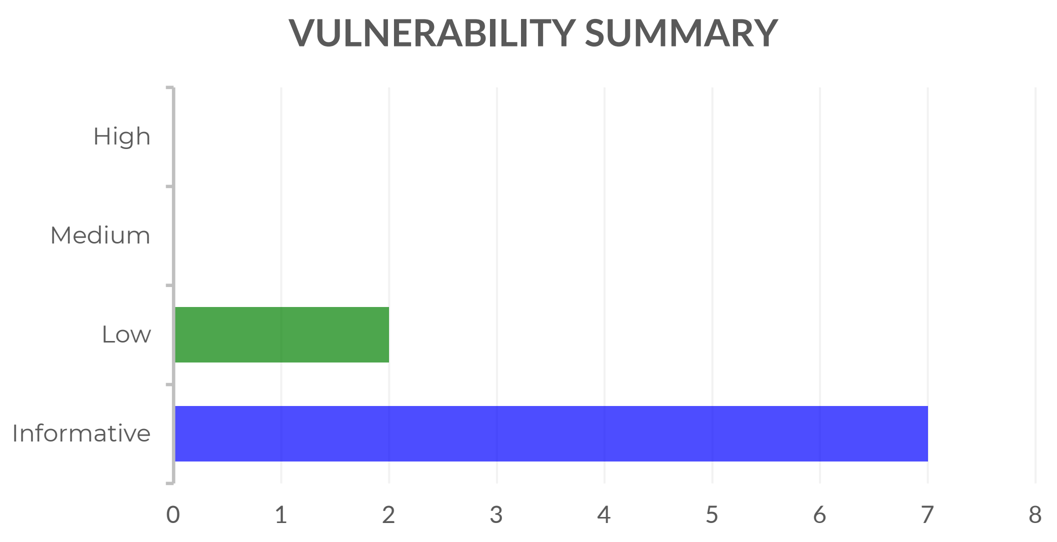
- https://github.com/Polkamarkets/polkamarkets-js/tree/feature/v2/contracts
  - branch: `feature/v2`
  - commit: `25e77512f91e6337dff5fb396941e8734537a5ae`
  - PredictionMarketV2.sol

Executive Summary

The security audit against **Prediction Market v2** has been conducted between the following dates: **03/04/2023** and **21/04/2023**.

Once the analysis of the technical aspects has been completed, the performed analysis shows that the audited source code contained vulnerabilities that that were successfully mitigated by the Polkamarkets team.

During the analysis, a total of **10 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

To this date, **05/04/2023**, the general conclusion resulting from the conducted audit is that the **Prediction Market project is secure** and does not present any known vulnerabilities. The development team has solved all the deficiencies detected during the audit to ensure the confidentiality, integrity and availability of the project.

The general conclusions of the performed audit are:

- It is important to mention that PredictionMarket did not support all non-standard token transfers, such as USDT, but the development team applied the necessary measures to correct this behavior.
- 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.
- A **Reentrancy pattern** was detected during the security audit, but with limited conditions for its exploitation. This issue has been correctly solved, consequently improving the security of the contract.
- Certain methods **did not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- The project was developed using an outdated compiler version with major known bugs, this is an absolutely discouraged practice and have been resolved before deployment.
- It is important to highlight that the present document is a public report and does not report all the technical details.

To deal with the detected vulnerabilities, an action plan was elaborated to guarantee their resolution, prioritizing those vulnerabilities with the highest risk. Consequently, **Polkamarkets has satisfactorily solved all the issues** found and applied the necessary recommendations.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

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

Table of vulnerabilities		
Vulnerability	**Risk**	**State**
Reentrancy pattern	**Low**	**Fixed**
Insufficient required balance validation	**Low**	**Assumed**
Insecure token transfers	**Low**	**Fixed**
Incompatible with variable balance tokens	**Informative**	**Assumed**
Limited ether transfer	**Informative**	**Fixed**
Lack of inputs validation	**Informative**	**Fixed**
Optimize error reporting	**Informative**	**Partially Fixed**
Outdated compiler	**Informative**	**Fixed**
Code quality	**Informative**	**Fixed**
GAS optimization	**Informative**	**Fixed**

Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

- Category
- Active
- Risk
- Description
- Recommendations

Reentrancy pattern

Category	Risk	State
Timing and State	Low	Fixed

The Reentrancy attack is a 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.

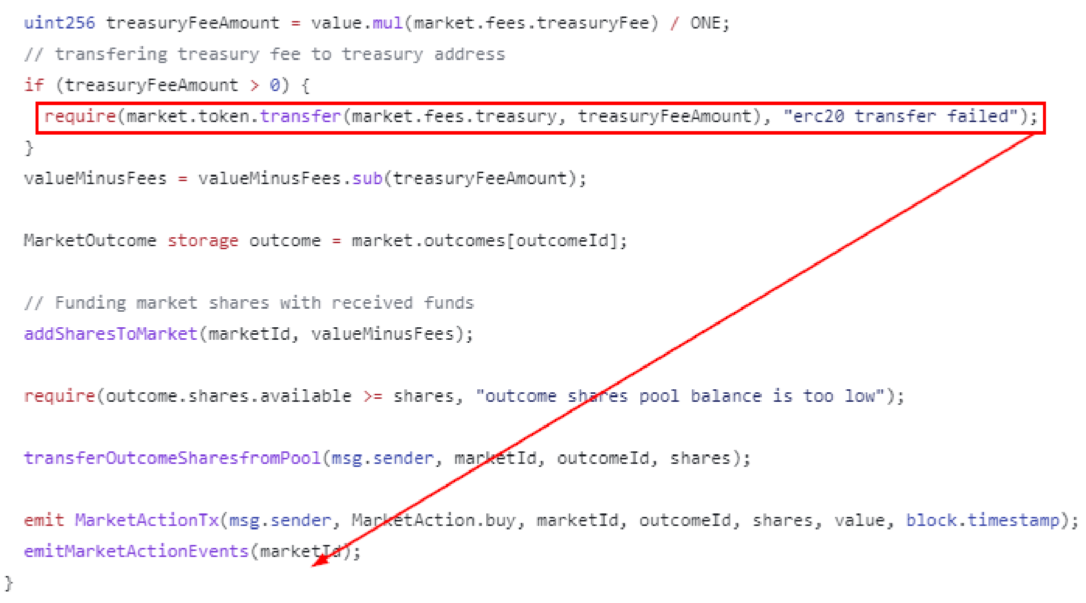
One way to mitigate this vulnerability is to use a checks-effects-interactions pattern in the contract design. In this pattern, the contract first performs all the necessary checks to ensure that the function call is valid and then performs all the state changes before interacting with other contracts or accounts.

By following this important security measure, the contract can ensure that it completes all state changes before allowing any external interactions, preventing reentrancy and other types of attacks in smart contracts.

When **PredictionMarketV2** interacts with the ERC-20 token of each market, it delegates the execution flow to an external contract, arbitrarily configured by the user when creating the market.

This `market.token` may contain reentrancy issues, if it reacts in any way to `transfer`, or malicious functions, which can be exploited in the `buy`, `sell`, and `claimFees` methods.

In the case of the `_buy` and `_sell` methods, the values are updated after the external calls are made, so the call can be redirected to any method before updating the values. This would allow the market maker to re-enter the function and buy or sell shares at an old price that does not reflect the actual volume value.



```
  uint256 treasuryFeeAmount = value.mul(market.fees.treasuryFee) / ONE;
  // transfering treasury fee to treasury address
  if (treasuryFeeAmount > 0) {
    require(market.token.transfer(market.fees.treasury, treasuryFeeAmount), "erc20 transfer failed");
  }
  valueMinusFees = valueMinusFees.sub(treasuryFeeAmount);

  MarketOutcome storage outcome = market.outcomes[outcomeId];

  // Funding market shares with received funds
  addSharesToMarket(marketId, valueMinusFees);

  require(outcome.shares.available >= shares, "outcome shares pool balance is too low");

  transferOutcomeSharesfromPool(msg.sender, marketId, outcomeId, shares);

  emit MarketActionTx(msg.sender, MarketAction.buy, marketId, outcomeId, shares, value, block.timestamp);
  emitMarketActionEvents(marketId);
}
```

_buy method

In the case of `claimFees`, used internally by `_removeLiquidity` and `_claimLiquidity`, it also makes interaction with an external contract and is susceptible to lead to reentrancy situations.
Since the project intends to work with third-party tokens, it is recommended to fix all reentrancy patterns by applying the *checks-effects-interactions pattern* and adding the `nonReentrant` modifier.

Recommendations

It is **essential to always make the state changes in the storage before making transfers or calls to external contracts**, in addition to implementing the necessary measures to avoid duplicated calls or chain calls to the methods. A good practice to avoid reentrancy is to organize the code according to the following pattern:

- The first step is conducting all necessary **checks**.
- The next step is the application of all **effects.**
- All external **interactions** take place in the last step.

References

- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
- https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

Source Code References

- PredictionMarketV2.sol#L390
- PredictionMarketV2.sol#L458
- PredictionMarketV2.sol#L893

Fixes Review

This issue has been addressed in the following pull request:

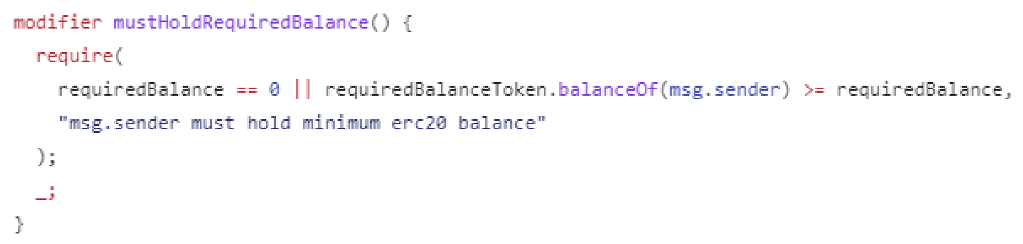
- https://github.com/Polkamarkets/polkamarkets-js/pull/13

Insufficient required balance validation

Category	Risk	State
Design Weaknesses	**Low**	**Assumed**

The requirement of the balance set to create the markets can be eluded with an ephemeral balance product of a flash loan or a chained operation of swaps.

The `mustHoldRequiredBalance` modifier only checks the current balance, but not its antiquity. Additionally, it does not require any type of staking of the tokens, so it is only necessary to fulfill the requirement during the `createMarket` invocation.



```
modifier mustHoldRequiredBalance() {
  require(
    requiredBalance == 0 || requiredBalanceToken.balanceOf(msg.sender) >= requiredBalance,
    "msg.sender must hold minimum erc20 balance"
  );
  _;
}
```

The required balance can be obtained `requiredBalance`, create a market and return the purchased token in the same transaction, without having to be the actual holder of the `requiredBalanceToken` token or demonstrate any commitment to the project.

Recommendations

- Make a stake of the tokens to be able to create the market in the next block.

Source Code References

- PredictionMarketV2.sol#L193-L199
- PredictionMarketV2.sol#L233
- PredictionMarketV2.sol#L219-L220

Fixes Review

Polkamarkets team explanation:

"While it is true that the mustHoldRequiredBalance modifier checks only the current balance and not its age, exploiting this loophole via flash loans or chained swaps would require significant effort and technical expertise. Furthermore, the incentive to do so is relatively low, as creating a market does not inherently provide any immediate financial gain for the exploiter."

Insecure token transfers

Category	Risk	State
Design Weaknesses	Low	Fixed

The ERC-20 standard specifies that the `transfer` and `transferFrom` methods must return a boolean with the result of this operation. However, it must be considered that certain tokens, such as **USDT**, do not strictly implement the standard, since they do not return the *boolean* type returning *void* instead. For this reason, if these cases are not covered, this call will fail whenever the token does not return the expected *boolean*, which could deny the use of third-party tokens.

```
// transferring funds
require(desc.token.transferFrom(msg.sender, address(this), desc.value), "erc20 transfer failed");
```

The **PredictionMarketV2** contract does not cover these cases, so it is recommended to use `safeTransfer` and `safeTransferFrom` from openzeppelin's **SafeERC20** library when interacting with third-party tokens.

Recommendations

- It is recommended to use `SafeERC20` from *OpenZeppelin* contracts, which already makes the verification after the execution of the transfers.

References

- https://eips.ethereum.org/EIPS/eip-20
- https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#SafeERC20

Source Code References

- PredictionMarketV2.sol#L296
- PredictionMarketV2.sol#L390
- PredictionMarketV2.sol#L415
- PredictionMarketV2.sol#L458
- PredictionMarketV2.sol#L481
- PredictionMarketV2.sol#L606
- PredictionMarketV2.sol#L694
- PredictionMarketV2.sol#L760
- PredictionMarketV2.sol#L809
- PredictionMarketV2.sol#L856
- PredictionMarketV2.sol#L893

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/14

Incompatible with variable balance tokens

Category	Risk	State
Undefined Behavior	**Informative**	**Assumed**

It is important to acknowledge that the logic of the contract does not contemplate the use of tokens with variable balance, such as aDAI or Ampleforth. If the user deposited an aDAI, interest would be generated in the contract and could never be claimed by the owner or the project, being locked in the contract forever.

Additionally, it does not contemplate ERC20 tokens with fee during the `transferFrom`, therefore, the amount received by the `PredictionMarketV2` will be less than the expected to conduct the transfer.

Moreover, some tokens may implement a fee during transfers, this is the case of USDT, even though the project has currently set it to *0*. So, the `transferFrom` function would return `true` despite receiving less than expected.

As shares are generated without taking said fee into consideration, the records do not reflect the real balance and other market users will end up assuming the cost of the fee.

References

- https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code

Source Code References

- PredictionMarketV2.sol#L296
- PredictionMarketV2.sol#L415
- PredictionMarketV2.sol#L606
- PredictionMarketV2.sol#L694
- PredictionMarketV2.sol#L390
- PredictionMarketV2.sol#L458
- PredictionMarketV2.sol#L481
- PredictionMarketV2.sol#L694
- PredictionMarketV2.sol#L760
- PredictionMarketV2.sol#L809
- PredictionMarketV2.sol#L856
- PredictionMarketV2.sol#L893

Fixes Review

Polkamarkets team explanation:

"We have deliberately designed our system to be compatible with standard ERC20 tokens, which are our users' primary tokens of choice. By doing this, we avoid the complications of handling variable balance tokens and can focus on providing a seamless user experience with stable tokens."

Limited ether transfer

Category	Risk	State
Undefined Behavior	**Informative**	**Fixed**

Because the method for transferring ether used is `transfer` (which is limited to 2300 gas) and `call` is not used, which would be limited to the gas provided by the user in the transaction. If a contract implements a `fallback` method that consumes more than 2300 gas, it may not work properly with the `PredictionMarketV2` contract.

The recommended option to send Ether depends on each specific scenario and the needs of the contract, so the project must study which option best suits its needs.
Following the different ways to send ether:

- **transfer:** If a fallback function is not provided in the receiving smart contract, the call to transfer will fail. 2300 gas is the maximum allowed, which is sufficient to finish the transfer process. To guard against reentry assaults, and it is hardcoded.
- **send**: It functions similarly to a transfer call and has a 2300 gas cap as well. The status is given back as a boolean.
- **call:** It is advised to use this method when sending ETH to a smart contract. The fallback feature of the receiving addresses is activated by the empty parameter.

Recommendations

- Use the call `call` to send ether.

Source Code References

- PredictionMarketV2.sol#L498
- PredictionMarketV2.sol#L701
- PredictionMarketV2.sol#L767
- PredictionMarketV2.sol#L816
- PredictionMarketV2.sol#L863
- PredictionMarketV2.sol#L900

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/15

Lack of inputs validation

Category	Risk	State
Data Validation	**Informative**	**Fixed**

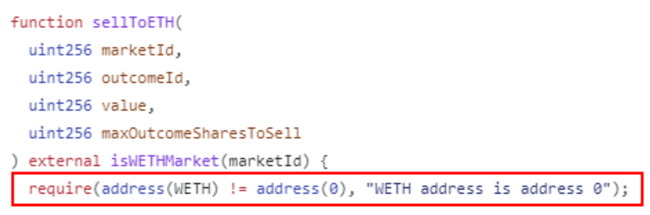
The arguments of the constructor in the **PredictionMarketV2** contract do not properly check the arguments, `_requiredBalanceToken`, `_WETH`, `_realitioTimeout and _requiredBalance`, which can lead to major errors.

In the case of the addresses for external smart contracts, it is recommended to use the `supportsInterface` method of the EIP-165 if possible, to verify that the established contracts contain the expected functionalities.

Source Code References

- PredictionMarketV2.sol#L210-L214

In the case of WETH, the only method that checks its value is `sellToETH`, but it is not checked in the constructor, nor in the rest of the methods that use `WETH`. Since the contract cannot work with `WETH = address(0)` this verification should be done in the constructor.



```
function sellToETH(
  uint256 marketId,
  uint256 outcomeId,
  uint256 value,
  uint256 maxOutcomeSharesToSell
) external isWETHMarket(marketId) {
  require(address(WETH) != address(0), "WETH address is address 0");
```

Source Code References

- PredictionMarketV2.sol#L302
- PredictionMarketV2.sol#L320
- PredictionMarketV2.sol#L426
- PredictionMarketV2.sol#L490
- PredictionMarketV2.sol#L614
- PredictionMarketV2.sol#L700
- PredictionMarketV2.sol#L766
- PredictionMarketV2.sol#L815
- PredictionMarketV2.sol#L862
- PredictionMarketV2.sol#L899

Recommendations

- It is recommended to check that the inputs of the user are as expected in order to guarantee the correct functioning of the contract.

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/16

Optimize error reporting

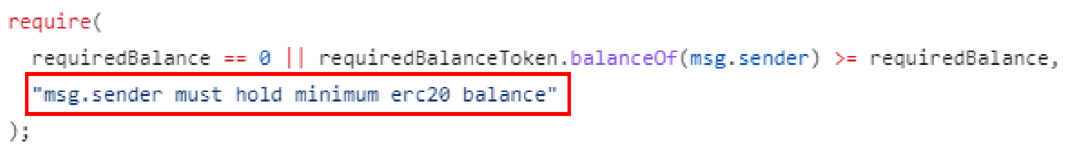
Category	Risk	State
Auditing and Logging	**Informative**	**Partially Fixed**

The project has adequate error handling and clear error messages, however, the system for reporting errors can be optimized by using the improvements introduced in the latest solidity versions for a more efficient form of notifying users about a failure in the operation of the project and lower GAS consumption.

Reduce require messages length

Ethereum Virtual Machine operates under a 32-byte word memory model where an additional gas cost is paid by any operation that expands the memory that is in use.

Therefore, exceeding error messages of this length means increasing the number of slots necessary to process the require, reducing the error messages to 32 bytes or less would lead to saving gas.



```
require(
  requiredBalance == 0 || requiredBalanceToken.balanceOf(msg.sender) >= requiredBalance,
  "msg.sender must hold minimum erc20 balance"
);
```

Source Code References

- PredictionMarketV2.sol#L194
- PredictionMarketV2.sol#L240
- PredictionMarketV2.sol#L242
- PredictionMarketV2.sol#L303
- PredictionMarketV2.sol#L399
- PredictionMarketV2.sol#L444
- PredictionMarketV2.sol#L463
- PredictionMarketV2.sol#L518
- PredictionMarketV2.sol#L538
- PredictionMarketV2.sol#L626
- PredictionMarketV2.sol#L731
- PredictionMarketV2.sol#L732
- PredictionMarketV2.sol#L738
- PredictionMarketV2.sol#L779
- PredictionMarketV2.sol#L780
- PredictionMarketV2.sol#L787
- PredictionMarketV2.sol#L826
- PredictionMarketV2.sol#L827
- PredictionMarketV2.sol#L834

Use custom errors instead of require

Custom errors are more gas efficient than revert strings in terms of deployment and runtime cost when the revert condition is met. The `require` statement is more expensive than the use of custom errors because it requires placing the error message on the stack, whether or not the transaction is reverted and regardless of the result of the condition.

Solidity **0.8.4** introduced custom errors, a more efficient way to notify users of an operation failure, The new custom errors are defined through the `error` statement, and they can also receive arguments. As indicated in the following illustrative example:

```
error Unauthorized();
error InsufficientPrice(uint256 available, uint256 required);
```

Afterwards, it is enough to just call them when needed using `if` conditionals:

```
if (msg.sender != owner) revert Unauthorized();
if (amount > balance[msg.sender]) {
            revert InsufficientPrice({
                available: balance[msg.sender],
                required: amount
            });
}
```

This behavior has been observed throughout the entire contract.

Source Code References

- PredictionMarketV2.sol

References

- https://blog.soliditylang.org/2021/04/21/custom-errors

Fixes Review

This issue has been partially addressed in the following pull request:

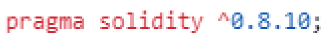
- https://github.com/Polkamarkets/polkamarkets-js/pull/20

Although the require messages length has been fixed, it would be more optimal to use solidity custom errors.

Outdated compiler

Category	Risk	State
Outdated Software	**Informative**	**Fixed**

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. The project has set the `^0.8.10` compiler version in the config.

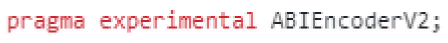


```
pragma solidity ^0.8.10;
```

Solidity branches up to version `0.8.18` has important bug fixes, such as an [issue] that leads to larger builds with unnecessary code, so it is recommended to use the most up to date version of the compiler.

Additionally, as can be seen in the following image, the `ABIEncoderV2` is being used. This Solidity functionality is used to encode and decode complex data structures.

Previously, it was necessary to specify the use of this function in older Solidity branches, however, from version 0.8.0 ABI coder v2 is activated by default. So, the pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect.



```
pragma experimental ABIEncoderV2;
```

Recommendations

- It is always a good policy to use the most up to date version of the pragma.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md
- https://docs.soliditylang.org/en/v0.8.0/080-breaking-changes.html

Source Code References

- PredictionMarketV2.sol#L1
- PredictionMarketV2.sol#L2

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/17

Code quality

Category	Risk	State
Codebase Quality	**Informative**	**Fixed**

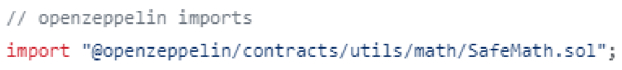
During the audit of the Smart Contract certain bad practices have been detected throughout the code that should be improved, it is always recommended to apply various coding styles and good practices. This is a very common bad practice, especially in these types of projects that are continually changing and improving. This is not a vulnerability in itself, but it helps to improve the code and to reduce the appearance of new vulnerabilities.

Following, we detail a few points that could be improved in terms of style, quality, and readability of the code throughout the audited contract.

Unnecessary SafeMath

The smart contracts analyzed inherit functionalities from *OpenZeppelin* contracts that are unnecessary. This does not imply a vulnerability by itself, but it makes it difficult to understand and maintain the code, as well as unnecessarily increasing its execution cost.

The **PredictionMarketV2** contract imports the `SafeMath` library, however, it is not necessary since the release of the pragma version `0.8`, Therefore, it is advisable to eliminate its use from the contract in order to make it more understandable and maintainable, and its elimination would save Gas both in the execution time and during the deployment of the contract.



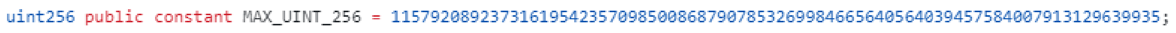
```
// openzeppelin imports
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
```

Source Code References

- PredictionMarketV2.sol#L7

Avoid hardcoded values

In order to improve the readability of the code and make it more user-friendly, maintainable, and auditable, it is recommended to modify the value of the `MAX_UINT_256` variable using constants that are already defined in the compiler itself, as in this case it would be `type(uint256). max`.



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

Source Code References

- PredictionMarketV2.sol#L68

Recommendations

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
- These references are very useful to improve the quality of the smart contract. A few of these practices are generally known and accepted forms to develop software.

References

- https://docs.soliditylang.org/en/v0.8.19/style-guide.html

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/18

GAS optimization

Category	Risk	State
Codebase Quality	**Informative**	**Fixed**

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.

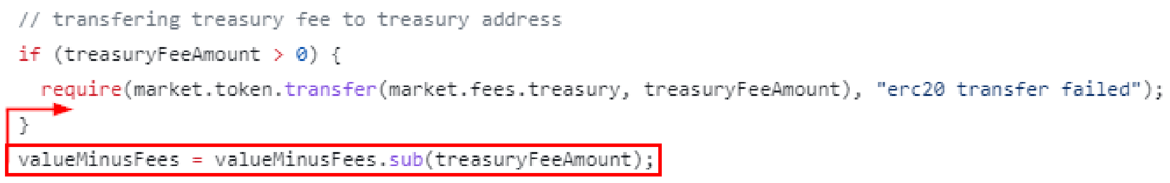
On EVM blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

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.

Logic optimizations

A possible optimization has been identified that would save Gas whenever the `_buy` method is called. As can be seen in the following image, a subtraction is performed between the value of `valueMinusFees` and `treasuryFeeAmount`. However, this logic only makes sense if the value of `treasuryFeeAmount` is greater than *0*. Therefore, in order to avoid performing unnecessary arithmetic operations, it is recommended to perform the operation inside the conditional: `if (treasuryFeeAmount > 0)`.



```
// transfering treasury fee to treasury address
if (treasuryFeeAmount > 0) {
  require(market.token.transfer(market.fees.treasury, treasuryFeeAmount), "erc20 transfer failed");
}
valueMinusFees = valueMinusFees.sub(treasuryFeeAmount);
```

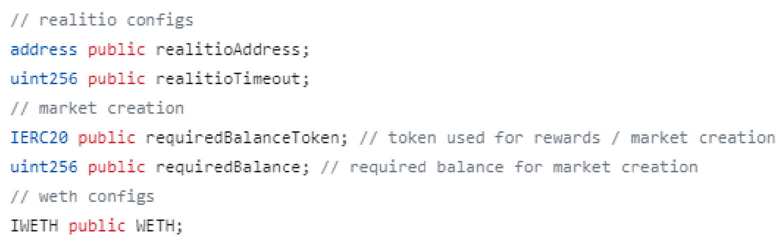
Therefore, by optimizing this function, the cost of GAS in each transaction will be lower, saving the users costs in GAS.

Source Code References

- PredictionMarketV2.sol#L392

Storage optimization

The use of the `immutable` 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 saving of GAS.



```
// realitio configs
address public realitioAddress;
uint256 public realitioTimeout;
// market creation
IERC20 public requiredBalanceToken; // token used for rewards / market creation
uint256 public requiredBalance; // required balance for market creation
// weth configs
IWETH public WETH;
```

References

- https://docs.soliditylang.org/en/v0.8.0/contracts.html#immutable

Source Code References

- PredictionMarketV2.sol#L156-L162

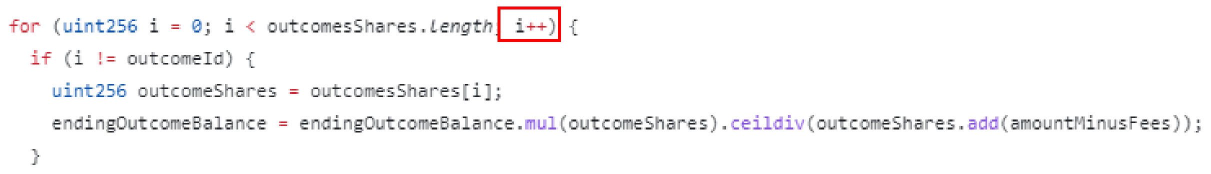
Increase operation optimization

During the audit, it was found that it is possible to optimize all the *for* loops in the project. There are two main ways to increment a variable in solidity:

```
++i will increment the value of i, and then return the incremented value.
i++ will increment the value of i, but return the original value that i held before
being incremented.
```

Since the return of the increment operation of the *for* loop is indifferent and discarded, both `i++` or `++i` instructions are completely valid instructions, however, the `++i` instruction has a considerably lower cost compared to incrementing a variable using `i++`, for this reason it is convenient to use `++i` for the increments of the *for* loops.

An example of this behaviour can be seen below:



```
for (uint256 i = 0; i < outcomesShares.length; i++) {
  if (i != outcomeId) {
    uint256 outcomeShares = outcomesShares[i];
    endingOutcomeBalance = endingOutcomeBalance.mul(outcomeShares).ceildiv(outcomeShares.add(amountMinusFees));
  }
```

References

- https://docs.soliditylang.org/en/latest/types.html#compound-and-increment-decrement-operators

Source Code References

- PredictionMarketV2.sol#L336
- PredictionMarketV2.sol#L358
- PredictionMarketV2.sol#L521
- PredictionMarketV2.sol#L526

- PredictionMarketV2.sol#L541
- PredictionMarketV2.sol#L546
- PredictionMarketV2.sol#L565
- PredictionMarketV2.sol#L574
- PredictionMarketV2.sol#L638
- PredictionMarketV2.sol#L645
- PredictionMarketV2.sol#L656
- PredictionMarketV2.sol#L665
- PredictionMarketV2.sol#L933
- PredictionMarketV2.sol#L944
- PredictionMarketV2.sol#L961
- PredictionMarketV2.sol#L1014
- PredictionMarketV2.sol#L1129
- PredictionMarketV2.sol#L1140
- PredictionMarketV2.sol#L1159
- PredictionMarketV2.sol#L1203
- PredictionMarketV2.sol#L1220
- PredictionMarketV2.sol#L1248

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/Polkamarkets/polkamarkets-js/pull/19

Annexes

Methodology

A code audit is a thorough examination of the source code of a project with the objective of identifying errors, discovering security breaches, or contraventions of programming standards. It is an essential component to the defense in programming, which seeks to minimize errors prior to the deployment of the product.

Red4Sec adopts a set of cybersecurity tools and best security practices to audit the source code of the smart contract by conducting a search for vulnerabilities and flaws.

The audit team performs an analysis on the functionality of the code, a manual audit, and automated verifications, considering the following crucial features of the code:

- The implementation conforms to protocol standards and adheres to best coding practices.
- The code is secure against common and uncommon vectors of attack.
- The logic of the contract complies with the specifications and intentions of the client.
- The business logic and the interactions with similar industry protocols do not contain errors or lead to dangerous situations for the integrity of the system.

In order to standardize the evaluation, the audit is executed by industry experts, in accordance with the following procedures:

Manual Analysis

- Manual review of the code, line-by-line, to discover errors or unexpected conditions.
- Assess the overall structure, complexity, and quality of the project.
- Search for issues based on the SWC Registry and known attacks.
- Review known vulnerabilities in the third-party libraries used.
- Analysis of the business logic and algorithms of the protocol to identify potential risk exposures.
- Manual testing to verify the operation, optimization, and stability of the code.

Automatic Analysis

- Scan the source code with static and dynamic security tools to search for known vulnerabilities.
- Manual verification of all the issues found by the tools and analyzes their possible impact.
- Perform unit tests and verify the coverage.

Vulnerabilities Severity

Red4Sec determines the severity of vulnerabilities found in risk levels according to the impact level defined by CVSSv3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST), classifying the risk of vulnerabilities on the following scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could compromise severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviours that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

RED4SEC

Invest in Security, invest in your future