BetETH Smart Contract Audit

Introduction

The BetETH team asked us to review and audit their Odds contract. We worked with the team over the course of a month to fix any issues that were found during the audit process

The audited contract is in this Github repository. The version used for this report is the commit c87dd0a3e6d05ecb3154be0bab6faefb201ab756.

Good job using OpenZeppelin to write minimal extra code. The team did a poor job of automated tests code coverage, with 0% of the code being covered prior to our assessment.

Here's our assessment and recommendations, in order of importance.

Overview of Issues

Index	Issue Title	Issue Status	Severity
1	A malicious player can game the system and drain the contract of funds	**Open**	High
2	Players can read the private variable `factor` to determine the answer prior to their bet	**Open**	High
3	If no ETH is ever sent to the contract, the first winners of the game will not be rewarded	**Open**	High
4	numWins: A user can cause this value to underflow and produce an undesired state	**Open**	Med
5	Insufficient Testing	**Open**	Med
6	numLosses: This is an unused Variable	Closed	Low
7	flip: Function visibility should be explicitly defined	Closed	Low
8	block.blockhash() was deprecated in Solidity 0.4.22	Closed	Low
9	withdrawFunds: Comments should be clearer	Closed	Low
10	Should not use magic constants	Closed	Low
11	Undefined event `ForwardEth`	Closed	Low

Details of Issues

1. A malicious player can game the system and drain the contract of funds

Severity	Status	Comment
High	**Open**	Under review

Description
If a user wins the coin flip, they win 2x the bet they made (provided the contract has that much ETH in it). A malicious user can take advantage of some false assumptions that were made by the contract author by performing the following steps:

1) Get the value of the private variable, `factor` (see issue #2 below)
2) Look at the blockhash of the current block
3) Make a bet with a calculation based on (1) and (2)

By following these steps, a user can "predict" the outcome of each toss and take all of the house funds.

Recommendation
Use an oracle, such as Oraclize, to get a random number from the random.org APIs, and use this for the outcome of each toss.

2. Players can read the private variable factor to determine the answer prior to their bet

Severity	Status	Comment
High	**Open**	Under review

Description
The author assumes that, because they made the `factor` variable private, users will not be able to read it, thus not allowing them to guess the outcome of the coin toss. Unfortunately, due to the public nature of the Ethereum blockchain, this value can be gathered by a malicious actor.

Recommendation
Do not rely on a private variable to calculate the outcome of the toss. See the recommendation section of issue #1 above for a fix to this issue.

3. If no ETH is ever sent to the contract, the first winners of the game will not be rewarded

Severity	Status	Comment
High	**Open**	Under review

Description
The contract does not get initialized with any ETH. If the first players of the game win, the contract will see that `address(this).balance = 0`, thus never achieving a payout.

Recommendation
Initialize the contract with ETH.

4. numWins: A user can cause this value to underflow and produce an undesired state

Severity	Status	Comment
Med	**Open**	Under review

Description
Solidity does not inherently protect against under/overflows with basic arithmetic. Because of this, undesired results can occur from simple math.

`numWins` counts the number of wins that the contract has seen and keeps a log of this. If the first bet of a contract is a loss, the value for `numWins` will become 1.1579209e+77, which is an unintended result.

Note: The severity of this was ranked as a 2 because, while it is a quite severe issue, there are no funds at stake, as this variable is simply a counter used informatively.

Recommendation
Always use OpenZeppelin's SafeMath library for any arithmetic operations.

5. Insufficient Testing

Severity	Status	Comment
Med	**Open**	Under review

Description
There was 0% test coverage for this contract. CaliberDLT recommends close to 100% test coverage prior to our analysis of the code.

Recommendation
Write programmatic tests for each aspect of the contract.

6. numLosses: This is an unused Variable

Severity	Status	Comment
Low	Closed	Under review

Description
`numLosses` is defined as a variable at the beginning of the codebase, but is never again used throughout the entirety of the contract.

Recommendation
Remove `numLosses`.

Remediation
The BetETH team has removed `numLosses` in this commit.

7. flip: Function visibility should be explicitly defined

Severity	Status	Comment
Low	Closed	Under review

Description
`flip` is the main function of the contract but it's visibility is not explicitly defined. Although functions are public by default, we recommend explicitly defining this for each function.

Recommendation
Add the `public` keyword to the `flip` function.

Remediation

The BetETH team has explicitly added the `public` keyword to the `flip` function in this commit.

8. `block.blockhash()` was deprecated in Solidity 0.4.22

Severity	Status	Comment
Low	Closed	Under review

Description
CaliberDLT always recommends using the latest stable release available. The Odds contract uses `block.blockhash()` to get the blockhash of a block, but this method was deprecated in Solidity version 0.4.22 in favor of a more simple `blockhash()`.

Recommendation
Change `block.blockhash()` to `blockhash()`.

Remediation
The BetETH team has changed `block.blockhash()` to `blockhash()` in this commit.

9. withdrawFunds: Comments should be clearer

Severity	Status	Comment
Low	Closed	Under review

Description
We recommend that code is written so that anybody can pick it up and understand what is going on after just a few minutes of reading. This is aided by clear comments. The `withdrawFunds` function should be documented so that new users can understand the desired coal of the function.

Recommendation
Add comments to `withdrawFunds`.

Remediation
The BetETH team has added comments to all functions in this commit.

10. Should not use magic constants

Severity	Status	Comment
Low	Closed	Under review

Description
CaliberDLT recommends avoiding magic constants throughout the entirety of the contract. This is to help readability, avoid confusion, and help future viewers of the contract. The Odds contract uses an arbitrary winning multiplier that can be made into a variable.

Recommendation
Remove the value `2` in these two lines, and replace them with a constant variable that is defined that the beginning of the contract.

Remediation
The BetETH team has added completed the recommendation in this commit.

11. Undefined event `ForwardEth`

Severity	Status	Comment
Low	Closed	Under review

Description
The `withdrawFunds` function has an event titled `ForwardETH` that was never defined.

Recommendation
Add a definition for `ForwardETH`. Additionally, it is recommended that the BetETH team adds the `emit` keyword before the event.

Remediation
The BetETH team has added completed the recommendation in this commit.

Threat Model

The BetETH contract allows users to participate in a game of luck with the potential of winning and earning rewards. It is a simple flip of the coin, using randomness calculated on the blockchain.

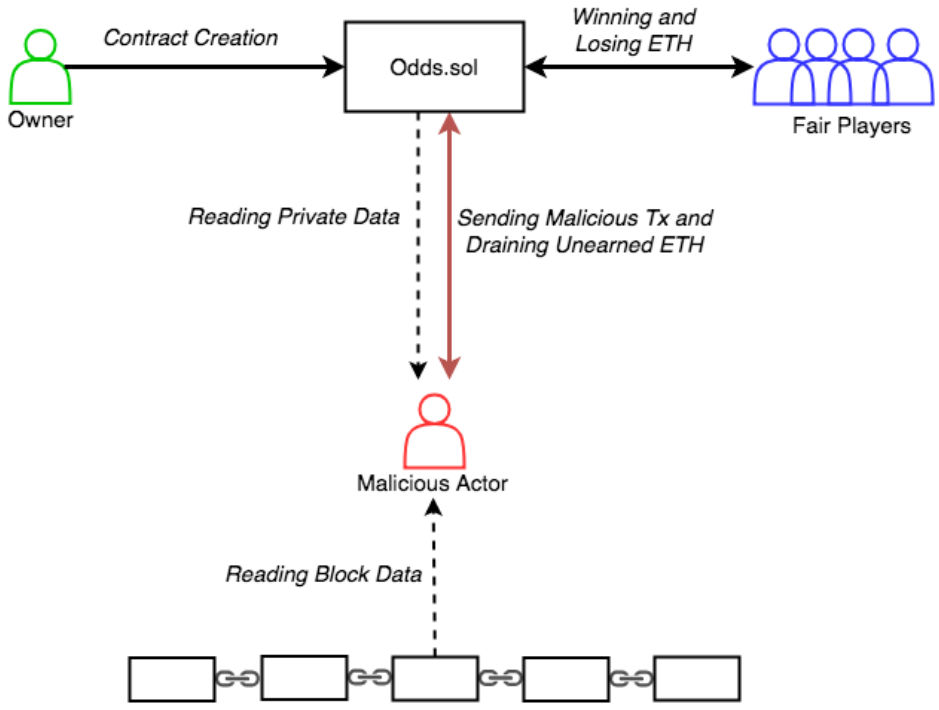
With all contracts that deal with the transfer of value, there is inherent risk to the actions being performed on this contract.

The scope of this threat model is to analyze the contract from the perspective of a malicious entity attempting to drain the contract of funds. This analysis inspected access control exploits, arithmetic issues, and false assumptions.

Number Prediction Exploit

The diagram below attempts to visualize an attack vector that was found during the audit. Over time, fair players will be interacting with the smart contract, thus leaving a number of ETH sitting in Odds.sol. A malicious actor can predict the outcome of each transaction, and is able to effectively drain the contract of all available ETH.

By getting the current block number with the `blocknumber()` function, as well as reading the private variable `factor` in the contract, the malicious actor is able to calculate the next value that will be used, thus allowing him to win every time. He will be able to continue this for as long as the contract lives.



Test Coverage

The BetETH team did not provide any tests for their repository at the time of submission, stating “They will work on it during the audit”. While CaliberDLT does not approve of this approach, we worked closely with the team to ensure a positive result.

CaliberDLT analyzed the tests *after* the completion of the audit, and the following section will be a reflection of this analysis.

It has to be noted that test coverage is only about 33% for the contract. The graphical representation below shows the amount of code covered (red) compared to the total desired coverage recommended by CaliberDLT.

The BetETH team did not test the constructor nor did they test the `withdrawFunds` function. Both functions are very simple, but are also critical. CaliberDLT recommends testing those functions as well. Additionally, the tests we received from there were very standard—they tested for no edge cases. We recommend writing a wholesome suite of tests to cover every possible outcome.

CaliberDLT checked the codebase against Mythril, an existing smart contract auditing tool. The tool returned no issues that were not pointed out in the issues section above. The results of the software tool and the CaliberDLT matched exactly, thus confirming the recommendations made by the CaliberDLT team.

Deployment Strategy

CaliberDLT provides a recommended deployment strategy to ensure a successful outcome of the code that was audited.

We recommend adding the following files to your project:

2_deploy_contracts.js
Truffle.js

With these two files, you will be able to successfully deploy the contract as we would have done it ourselves. Set up a local node, or simply use Infura as your provider, and the contract will be successfully deployed to the main network without any issues.

While we suggest a specific deployment method, it is up to BetETH to choose a final deployment strategy. Many exist with various levels of complexity.

Disclosure

CaliberDLT (“CDLT”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via CDLT blog posts and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. Blockchain implementations are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third-party in any respect, including regarding the bug free nature of code, fitness, merchantability, operability or compatibility of any code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CDLT owes no duty to any third-party by virtue of publishing these Reports.

PURPOSE OF REPORTS. The Reports and the analysis described therein are created solely for Clients and published with their consent. Unless explicitly stated otherwise, the scope of our review is limited to a review of blockchain code and only the code we note as being within the scope of our review within this Report. The code language itself remains under development and is subject to unknown risks and flaws. Unless explicitly stated otherwise, the review does not extend to the compiler layer, or any other areas beyond blockchain code that could present security risks. Blockchain implementations are emergent technologies and carry with them high levels of technical risk and uncertainty.

TIMELINESS OF CONTENT. The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by CDLT.