Pepe's Game

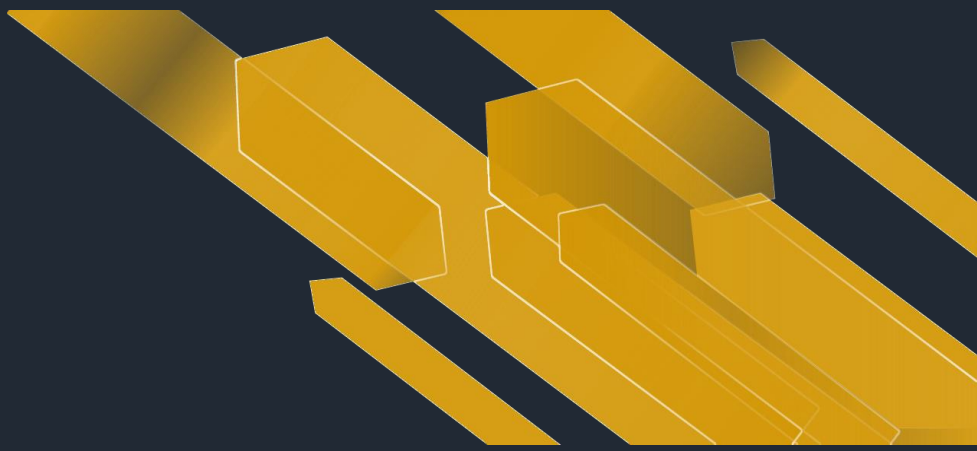
SMART CONTRACT SECURITY AUDIT

Date:

March 10th 2023

Written by:

Pim Hakkert



DISCLAIMER

The following disclaimer outlines important information related to a Solidity smart contract audit conducted by Pim Hakkert.

Pim performed the audit as a single individual, and his report does not express any opinion or judgment about the client or their project. It is crucial to understand that this audit should not be taken as an endorsement or criticism of the client or their project.

It is important to acknowledge that no smart contract can be declared as "safe" due to the absence of regulations regarding their quality. Pim's responsibility was limited to reviewing the Solidity smart contracts delivered by the client within the scope of the audit.

Despite his best efforts to review the code to the best of his abilities, it cannot be assumed that the smart contracts are entirely safe. Therefore, the client must assess whether additional audits by third parties are necessary to identify any issues with the smart contracts.

Solidity is a continuously evolving language, and Ethereum (hard)forks past the front cover’s date can render parts of the audit review obsolete. As such, it is essential to keep in mind that the audit report reflects the state of the reviewed contracts as of the audit's date.

To mitigate the risks associated with using smart contracts, it is recommended that the client stay up-to-date with any changes to the Solidity language and Ethereum network, conduct regular audits, and engage with professionals experienced in smart contract development and security.

PROCESS

The smart contracts are audited according to a 7-step plan:

Step 1. The client forks the Github repository that contains the smart contracts. This repository contains a valid installation of a framework such as Hardhat or Foundry.

Step 2. The client modifies the repository to prepare it for an audit and creates a readme file that outlines instructions for installing dependencies and running tests. The client also invites Pim's Github account (pimhakkert) as a collaborator.

Step 3. The tests are reviewed and either approved to run at an acceptable level or flagged for fixes before the audit can commence.

Step 4. Pim conducts a multi-day audit of the code and produces an audit report that is shared with the client.

Step 5. The client responds to all questions and findings raised in the audit report, making any necessary fixes, and pushes the updated code to the Github fork.

Step 6. The updated code is reviewed, and a final audit report is prepared and sent to the client.

Step 7. The client approves the final report, and a public version of the audit report is published.

SUMMARY

Details

Project name: Pepe's Game

Description: Pepe's Game contracts hold the betting information and functionality for the main game. These contracts are supplied with information from an oracle for off-chain calculations and logic.

Audit details

Methods used: Static analysis, manual review, fuzzing

Audit timeframe: March 7th 2023 – March 10th 2023

Findings

Total findings	9
High risk	0
Medium risk	0
Low risk	6
Informational / Gas	3

Team Interaction

"Working with the developer in charge of the in-scope smart contracts was a nice experience. He proved to be knowledgeable, readily answering all queries and offering constructive feedback on the findings. Some of the identified issues were initially included due to insufficient information but were later eliminated following the first audit round. The developer promptly furnished the necessary context for these findings. "- Pim

Project & team score

Preparation

Communication

Processing speed

Contracts audited

PepeBet.sol	contracts/pepeBet.sol
PepeOracle.sol	contracts/pepeOracle.sol
PepePool.sol	contracts/pepePool.sol

FINDINGS

Audit findings are organized by their severity, and any affected contracts are identified. Each finding features a title and a unique ID for easy reference. Within the finding, the issue is clearly explained, relevant lines of code are highlighted, a recommendation is presented, and any associated links to support the suggestion are provided. Msaenee sms esja erjer aje dndmdmmmd emasdja aernenda aern

High risk

There were no high-severity findings found within any of the in-scope smart contracts.

High severity findings are critical issues that pose a significant threat to the smart contract's security, functionality, or user assets.

These issues may result in the loss of funds, unauthorized access, or a complete breakdown of the contract's intended operations. Addressing high severity findings is of utmost importance, and they should be resolved before the contract is deployed or used in production.

Medium risk

There were no medium-severity found within any of the in-scope smart contracts.

Medium-severity findings are issues that may not immediately compromise the smart contract's security but can still have a considerable impact on its proper functioning or user experience.

These issues could potentially be exploited by an attacker in certain scenarios or lead to unexpected outcomes. Addressing medium-severity findings is important, and they should be carefully reviewed and fixed as necessary.

Low risk

Six (6) low-severity findings were found while auditing the in-scope smart contracts.

Low-severity findings are minor issues that are unlikely to have a significant impact on the smart contract's overall security or functionality.

These issues may still be worth addressing to improve the contract's robustness, reliability, or user experience, but they are generally not considered urgent.

L1: Withdrawal function pausability

Pausing the `withdraw()` function prevents users from removing their USDC from the smart contract. A malicious actor with access to the owning smart contract may use this to lock funds and then take their time to siphon these funds away using an exploit.

End users may raise questions on the pausability of `withdraw()`.

Affected Contracts

- PepeBet.sol

Recommendation

Remove the `whenNotPaused` modifier in the `withdraw()` function
OR
Implement a static time locked period in which users cannot withdraw
OR
Add a comment for why the modifier is added

Result

The team removed the `whenNotPaused` modifier after realizing it was accidentally placed there.

L2: Withdrawal function pausability

The for loops within the functions `constructor()`, `addNewAssets()`, `unapproveAssets()` (PepeBet.sol) and `settleBets` (PepeOracle.sol) could hit the block gas limit and revert.

Affected Contracts

- PepeBet.sol
- PepeOracle.sol

Recommendation

Avoid using arrays in Solidity to add new data. Refactor functions to add assets one by one, and loop call these functions from off-chain. Remove the for loop within `constructor()` (PepeBet.sol).

Result

Acknowledged. Loops called from external sources are now batched when possible.

L3: Contract layout

The layout of all smart contracts does not follow the Solidity style guide. Other developers and auditors may have more difficulty reading a contract if not laid out in a common order. Private functions are found between external functions, reducing the overall readability of the smart contract.

Affected Contracts

- All

Recommendation

Follow the official Solidity style guide for smart contract layout. It is recommended to group & order functions by visibility.

Related links

- https://docs.soliditylang.org/en/v0.8.17/style-guide.html#order-of-layout

Result

Resolved.

L4: Contract inheritance

Several contracts have interfaces located in `/contracts/interfaces`, but do not extend from these interfaces. This could lead to a mis-match in required functionality by the interfaces.

Affected Contracts

- PepeBet.sol
- PepePool.sol

Recommendation

Extend `PepeBet.sol` from `IpepeBet.sol` and `PepePool.sol` from `IpepePool.sol` to require appropriate use of the interfaces.

Result

Resolved.

L5: Lack of documentation

All contracts lack comments for state variables, structs, functions, errors, events, and modifiers. The code is hard to decipher by other developers and auditors.

Affected Contracts

- All

Recommendation

Add documentation for all the aforementioned items.

Result

Resolved.

L6: Usage of custom errors

Throughout the contracts some errors are written using strings and some are written using custom errors.

Affected Contracts

- All

Recommendation

Use custom errors in all exceptions for consistency.

Result

Acknowledged.

Informational/Gas

Three (3) info/gas severity findings were found while auditing the in-scope smart contracts.

Informational severity findings, often associated with gas savings, are observations that do not pose a direct risk to the smart contract's security or functionality but may provide valuable insights for optimization, code readability, or adherence to best practices.

Addressing these issues can lead to a more efficient contract execution and reduced gas costs, improving the overall user experience.

IG1: Definition of IUSDC at compile time

In all contracts, the address of `IUDSC` is hardcoded within the smart contract. Using these contracts in devnets, testnets, and livenets will most likely not work.

Affected Contracts

- All

Recommendation

Initialize `IUSDC` in the contract constructor, allowing for different USDC addresses (and mock ones) to be used.

Result

Resolved. `IUSDC` is now initialized in the constructor.

IG2: betId() function name

It is unclear that function `betId()` is a function that creates a new betId.

Affected Contracts

- PepeBet.sol

Recommendation

Rename `betId()` to `createBetId()` or `generateBetId()`.

Result

Resolved. The function `betId()` was removed in favor for a more simpler approach to create an identifying number for bets.

IG3: Variable naming

Several variables are defined vaguely, causing confusing about their use.

Affected Contracts

- All

Recommendation

Consider following the following renaming recommendations.

PepeBet:

`*amount/*price`: Any variable containing the word "amount" or "price" should be renamed to indicate the unit of amount. "amount" is USDC, and should reflect that. In some contexts such as the `BetDetails` struct, "amount" can be seen as the amount of assets to be betted on instead of the amount of USDC placed as a wager.

`::32,33`: A comment is added to indicate this unit is a basis point. Postfixing these variables with `Bps` removes the requirement for that comment and clarifies its usage anywhere in the contract. Consider also postfixing `Usdc` to `::32`.

`::54`: `payout` should be renamed to `payoutUsdc`.

`::66`: `requested` and `available` should be postfixed with `Usdc`.

PepeOracle.sol:

`*price`: Any variable containing the word "amount" or "price" should be renamed to indicate the unit of amount. "amount" is USDC and should reflect that.

`::12`: `closePrice` should be renamed to `closePriceUsdc`.

PepePool.sol:

`pepeBet` is not a clear indicator of an address holding a `PepeBet.sol` smart contract. Change the name of this variable to something like `pepeBetContract`.

Result

Acknowledged and partially resolved.

CLOSING WORDS

”Pepe Game’s contracts are logically sound and should perform as they are intended. Resolving the findings within these smart contracts has made it easier for any future auditors to focus on security vulnerabilities rather than semantics. It was a pleasure auditing your contracts.”**- Pim**