BLACKHOLE
Security Review

MARCH, 2025

www.chaindefenders.xyz
https://x.com/ChDefendersEth

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

USDC gets placed in the contract as the prize pool. Once the game starts users can come and bid to win the prize pool. Bids increase by $100 each time. If the Bid expires (after 1 hour) the last bidder can claim the prize pool. On each bid 2% is added to the total prize to bid. 1% of the bid goes to fee recipient and the other 1% goes to the prize pool. The prize pool keeps growing as more bids are placed. The winning bidder (the one whose bid expires) can claim the prize pool. The final bid is sent to fee recipient.

Disclaimer

The ChainDefenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

Id	Files in scope
1	USDC20Vault.sol

Roles

Id	Roles
1	Owner
2	User

Executive Summary

Issues found

Severity	Count	Description
High	0	Critical vulnerabilities
Medium	3	Significant risks
Low	5	Minor issues with low impact
Informational	1	Best practices or suggestions
Gas	1	Optimization opportunities

Findings

Medium

Mid 01 DoS Due To Blacklisted Previous Bidder

Location

USDC20Vault.sol:71

Description

The issue arises in the `placeBid` function, where the contract attempts to refund the previous `highestBidder` using a direct token transfer (`push` model). If the previous `highestBidder` becomes blacklisted (e.g., by the token contract or due to regulatory reasons), the `transfer` call to refund their bid will fail. This failure will revert the entire transaction, preventing new bids from being placed. As a result, the bidding process will be halted, and the contract will become unusable for future bids.

Recommendation

Adopt a `pull` model for refunds instead of the current `push` model. Specifically:

1. Store the refundable amount for the previous `highestBidder` in a mapping (e.g., `pendingRefunds`).

2. Allow users to withdraw their refunds manually by calling a separate `withdrawRefund` function.
3. Ensure the `withdrawRefund` function is non-reentrant and properly handles edge cases, such as zero refunds.
4. Update the `placeBid` function to only update the `pendingRefunds` mapping instead of directly transferring funds to the previous bidder.

This approach decouples the refund process from the bidding process, ensuring that a blacklisted `highestBidder` does not block new bids.

Status

Acknowledged

Mid 02 Use `safeTransfer` And `safeTransferFrom` Instead Of `transfer` And `transferFrom`

Location

USDC20Vault.sol:65

USDC20Vault.sol:68

USDC20Vault.sol:71

USDC20Vault.sol:96

USDC20Vault.sol:108

USDC20Vault.sol:109

Description

The contract currently uses `transfer` and `transferFrom`, which return a boolean value but do not revert on failure. This requires explicit `require` statements to check for success. However, this approach may miss edge cases in some ERC-20 tokens (e.g., USDT) that do not return `true`/`false` and instead rely on low-level success signals. Currently the protocol supports only USDC, but this could change in the future

To improve security and compatibility, it is recommended to use OpenZeppelin's `SafeERC20` library, which ensures safe transfers by handling token implementations that do not return a boolean value.

Recommendation

1. Import SafeERC20

Add the following import statement at the beginning of the contract:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

2. Update IERC20 Declaration

Use `SafeERC20` with the `using` directive:

```
using SafeERC20 for IERC20;
```

3. Replace `transfer` and `transferFrom` with `safeTransfer` and `safeTransferFrom`

Modify all instances where `transfer` and `transferFrom` are used:

Status

Acknowledged

Mid 03 `placeBid` Can Be Called After Claiming

Location

USDC20Vault.sol:44

Description

The `placeBid` function does not check whether the game has already been claimed. This means users can still place bids even after the auction should have ended.

Issue Details:

1. `canClaim()` Logic
    - A bid can be claimed if:
        - The game has started (`init` is `true`).
        - The bid has not been claimed (`!claimed`).
        - The required time has passed (`block.timestamp ⩾ lastBidTime + HOUR_IN_SECONDS`).
        - There is a valid highest bidder (`highestBidder ≠ address(0)`).
    - However, the `placeBid()` function only prevents bids when `canClaim()` returns `true`.
    - This does not account for whether a bid has already been claimed—which means bids can still be placed even after claiming.
2. `placeBid()` Missing Check
    - The function should include a check to ensure that no more bids can be placed after a claim has occurred.

Proof of Concept

Add the following test and run it:

```
it("Should not allow bid after claim", async () => {
    const player1Before = await usdc20.balanceOf(player1.getAddress())
    ;
    const bid1 = await vault.getNextBidAmount();
    await usdc20.connect(player1).approve(await vault.getAddress(),
    bid1);
    await vault.connect(player1).placeBid(bid1);

    // Fast-forward time
    await ethers.provider.send("evm_increaseTime", [3601]);
    await ethers.provider.send("evm_mine");

    // Claim prize
    await vault.connect(player1).claim();
    const player1After = await usdc20.balanceOf(player1.getAddress());

    expect(player1After).to.be.gt(player1Before);
    expect(player1After).to.be.equal(19899000000);
    expect(await vault.getPrizePool()).to.equal(0n);
```

```
18
19      // Bid should revert now
20      const bid2 = await vault.getNextBidAmount();
21      await usdc20.connect(player2).approve(await vault.getAddress(),
        bid2);
22      await expect(vault.connect(player2).placeBid(bid2)).to.be.reverted
        ;
23    });
```

Recommendation

Add a check in `placeBid()` to prevent new bids if the prize has already been claimed:

Modify `placeBid()` to include a `require(claimed == false, "Game has already been claimed");` check:

```
function placeBid(uint256 expectedBid) external nonReentrant {
      require(init, "Game has not been started");
+     require(!claimed, "Game has already been claimed"); // Prevent
      bids after claiming
      require(!canClaim(), "Game is ready for claiming, no more bids
      allowed");
```

Status

Fixed

Low

Low 01 Blacklisted Highest Bidder Leads To Stuck Funds

Location

USDC20Vault.sol:100

Description

The issue arises in the `claim` function of the `USDC20Vault` contract. If the `highestBidder` becomes blacklisted (e.g., by the token contract or due to regulatory reasons) after they are eligible to claim but before they actually call the `claim` function, the funds in the contract will become stuck. This is because the `require` statement in the `claim` function ensures that only the `highestBidder` can claim the prize pool, and the subsequent `transfer` to the blacklisted address will fail. As a result, neither the `highestBidder` nor the contract owner will be able to access the funds, effectively locking them in the contract indefinitely.

Recommendation

Introduce a claim period mechanism to address this issue. Specifically:

1. Add a configurable claim period (e.g., 7 days) during which the `highestBidder` can claim their prize.
2. After the claim period expires, allow the contract owner to withdraw the prize pool and any unclaimed funds to prevent them from being permanently stuck.
3. Emit an event when the owner withdraws unclaimed funds to ensure transparency.
4. Ensure the logic is implemented securely to avoid abuse by the owner or other parties.

Status

Fixed

Low 02 Missing Zero Address Check

Location

USDC20Vault.sol:31

USDC20Vault.sol:41

Description

The `USDC20Vault` contract does not validate that the `feeDestination` address provided in the constructor or the `changeFeeDestination` setter function is not the zero address (`address(0)`). If the `feeDestination` is set to the zero address, any subsequent transfers to this address (e.g., deployer fees) will fail, causing the contract to malfunction. This could result in funds being locked in the contract and disrupt its intended functionality.

Recommendation

Add a check in both the constructor and the `changeFeeDestination` function to ensure that the `feeDestination` address is not the zero address. For example:

- Use `require(_feeDestination ≠ address(0), "Invalid fee destination address")` in both places.

This will prevent the contract from being deployed or updated with an invalid `feeDestination` address.

Status

Acknowledged

Low 03 Fee On Transfer Or Rebasing Tokens Not Supported

Location

USDC20Vault.sol

Description

The contract assumes that the USDC token being used is a standard ERC20 token without any additional mechanics such as fees on transfer or rebasing. However, USDC is an upgradeable token, meaning its implementation can be modified in the future to include features like fees on transfer. If such a change occurs, the

contract will not function as intended because it does not account for a reduced token amount being received after a transfer. Additionally, the contract does not support rebasing tokens, which dynamically adjust balances, as it relies on static balance calculations for operations like bidding, refunds, and prize distribution.

This limitation could lead to unexpected behavior, such as:

1. Insufficient funds being transferred to the contract or recipients due to transfer fees.
2. Incorrect calculations of balances and amounts when interacting with rebasing tokens.

Recommendation

Either document the limitations of the contract in the comments and deployment documentation to inform users about the unsupported token types or add support for such tokens.

Status

Acknowledged

Low 04 Single-step Ownership Transfer Pattern Can Be Dangerous

Location

USDC20Vault.sol:8

Description

The `USDC20Vault` contract inherits from `OpenZeppelin's Ownable` contract, which allows for immediate ownership transfers. This creates a risk where ownership could be accidentally transferred to an incorrect or inaccessible address, resulting in the contract becoming permanently locked without an owner.

The `Ownable2Step` pattern from `OpenZeppelin` provides a more secure ownership transfer mechanism by requiring the new owner to accept the transfer in a separate transaction. This two-step process reduces the risk of accidental transfers to incorrect addresses.

Recommendation

Change the code in the following way:

```
- import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol"
    ;
+ import { Ownable2Step } from "@openzeppelin/contracts/access/
    Ownable2Step.sol";

- contract USDC20Vault is Ownable {
+ contract USDC20Vault is Ownable2Step {
```

Status

Acknowledged

Low 05 Missing Event Emissions In Critical Functions

Location

USDC20Vault.sol:36

USDC20Vault.sol:40

USDC20Vault.sol:95

Description

The following functions modify contract state but do not emit events, which reduces transparency and makes it harder to track changes off-chain:

1. `startGame()` – Sets `init = true`, indicating the game has started, but does not log this change.

2. `changeFeeDestination(address _feeDestination)` – Updates the `feeDestination` address without emitting an event, making it harder to track fee destination changes.
3. `addToPrizePool(uint256 amount)` – Transfers USDC and updates `prizePool`, but does not emit an event, making it difficult to monitor prize pool updates.

Recommendation

Declare events:

```
+ event GameStarted();
+ event FeeDestinationChanged(address newFeeDestination);
+ event PrizePoolUpdated(uint256 amount);
```

Modify functions to emit events:

```
function startGame() public onlyOwner {
      init = true;
+     emit GameStarted();
}

function changeFeeDestination(address _feeDestination) public
    onlyOwner {
      feeDestination = _feeDestination;
+     emit FeeDestinationChanged(_feeDestination);
}

function addToPrizePool(uint256 amount) external onlyOwner {
      require(usdc20.transferFrom(msg.sender, address(this), amount), "
    Transfer failed");
      prizePool += amount;
+     emit PrizePoolUpdated(amount);
}
```

Status

Acknowledged

Informational

Info 01 Missing Restart Functionality

Location

USDC20Vault.sol:8

Description

The contract currently uses the `init` state variable to determine whether the game has started. However, once the game ends (e.g., after a claim is made), there is no mechanism to reset the game state and allow it to be restarted. This limitation means that the contract can only be used for a single game instance. To start a new game, the contract would need to be redeployed, which is inefficient and costly. This design restricts the contract's usability and flexibility, especially if the owner wants to run multiple games over time.

Recommendation

Implement a mechanism to reset the game state after it ends, allowing the contract to be reused for subsequent games. This could involve:

1. Adding a `resetGame` function that resets relevant state variables (`init`, `highestBidder`, `currentBaseAmount`, `highestBid`, `prizePool`, `lastBidTime`, `claimed`) to their initial values.
2. Ensuring that only the owner can call the `resetGame` function to prevent unauthorized resets.
3. Adding checks to ensure the game can only be reset after it has ended (e.g., after a claim or a timeout period).

Status

Acknowledged

Gas

Gas 01 Unnecessary Ternary Operator

Location

USDC20Vault.sol:47

USDC20Vault.sol:79

Description

The ternary operator used in these places is redundant. If `currentBaseAmount` is `0`, the expression still resolves to `BASE_INCREMENT`, making the conditional expression unnecessary. Keeping the ternary operator adds unnecessary bytecode, increasing gas costs.

Recommendation

Refactor both expressions to:

```
uint256 baseAmount = currentBaseAmount + BASE_INCREMENT;
uint256 nextBase = currentBaseAmount + BASE_INCREMENT;
```

This simplifies the code while maintaining the same behavior.

Status

Acknowledged