[M-1] TITLE (Root + Impact)

The `constructor` in the `Proxy` contract doesn't have a zero check. This causes the contract to behave unpredictably, as it might reference an invalid or nonexistent contract, making it vulnerable to exploits

Description

The `constructor` in the `Proxy` contract lacks a zero address check. This oversight can lead to unpredictable behavior, as the contract may attempt to reference a contract at an invalid or nonexistent address. Without the check, a zero address could be assigned, which might cause the contract to malfunction or become vulnerable to attacks. Properly validating the contract address in the constructor ensures that the `proxy` contract only interacts with a valid contract, improving security and reliability.

Impact

Unpredictable Behavior: If the contract references a zero address, it may fail to interact with the intended contract, causing the proxy to malfunction or not function at all.

Proof of Concepts

To demonstrate this, I wrote a similar proxy contract in Remix IDE and initialized the address as address(0) in the constructor. After deploying the contract, I retrieved the implementation contract address, but it showed the address as not valid (0x0000000000000000000000000000000000000000)

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

contract Proxy {
    // implementation address
    address private immutable _implementation;
    //don't have zero check

    constructor() {
        _implementation = address(0);
    }


    function getimple() external view returns (address) {

        return _implementation;
    }

}
```

Output :

```
address: 0x0000000000000000000000000000000000000000
```

Recommended mitigation

Add checking condition of the address

```
constructor(address implementation) {
------->   require(implementation != address(0));
    _implementation = implementation;
}
```

[I-1] TITLE (Root + Impact)

In `Distributor.sol`, the `Distributed` event doesn't have any indexed parameters, which makes it very hard to filter for specific events.

Description

In `Distributor.sol`, the `Distributed` event lacks `indexed` parameters, making it difficult to efficiently filter and search for specific events

Impact : difficult to efficiently filter and search for specific events

Proof of Concepts

event Distributed(address token, address[] winners, uint256[] percentages, bytes data);

Recommended mitigation

you can add indexed like this : event Distributed(address indexed token, address[] indexed winners, uint256[] percentages, bytes data);

[I-1] TITLE (Root + Impact)

The `Distributor__InvalidCommissionFee` error in the `Distributor` contract is not used anywhere. It reduces code readability and maintainability

Description

The custom error `Distributor__InvalidCommissionFee` is declared in the contract but not used anywhere in the code. Keeping unused declarations can create confusion for developers and auditors, as it may imply missing or incomplete validation logic. Additionally, it slightly increases the contract's bytecode size and reduces overall code clarity

Impact

Affects code readability and maintainability.

Recommended mitigation

Remove if not using anywhere in the contract

[H-1] TITLE (Root + Impact)

The same signature can be used to distribute the unauthorized implementation. This happens because whoever owns the signature can distribute the unauthorized implementation.

Description

The same signature can be used in different `distribute` implementations causing that the caller who owns the signature, to distribute on unauthorized implementations

Impact : Anyone who possesses the signature can distribute an implementation without the organizer's consent

Proof of Concepts

I wrote a test code to explain this issue. If the owner sets the same contestId for an implementation, it causes a problem. You can see in the test code I wrote: first, I distribute the implementation once it's deployed, and then distribute it again with the same signature. After that, I wrote another implementation and distributed it — it passed the test case without throwing an error

```solidity

function testSignatureCanBeUsedToNewImplementation() public {

   address organizer = TEST_SIGNER;
   bytes32 contestId = keccak256(abi.encode("Jason", "001"));
   //
   Distributor first_distrubute = new Distributor(address(proxyFactory),
stadiumAddress);
   vm.startPrank(factoryAdmin);
   proxyFactory.setContest(organizer, contestId, block.timestamp + 8
days, address(first_distrubute));
   vm.stopPrank();

   bytes32 salt = keccak256(abi.encode(organizer, contestId,
address(first_distrubute)));
   address proxyAddress = proxyFactory.getProxyAddress(salt,
address(first_distrubute));

   vm.startPrank(sponsor);
   MockERC20(jpycv2Address).transfer(proxyAddress, 10000 ether);
   vm.stopPrank();


   assertEq(MockERC20(jpycv2Address).balanceOf(proxyAddress),
10000 ether);
   assertEq(MockERC20(jpycv2Address).balanceOf(user1), 0 ether);
   assertEq(MockERC20(jpycv2Address).balanceOf(stadiumAddress),
0 ether);
   //
   // 2. Organizer creates a signature
   (bytes32 digest, bytes memory sendingData, bytes memory
signature) = createSignatureByASigner(TEST_SIGNER_KEY);
   assertEq(ECDSA.recover(digest, signature), TEST_SIGNER);

   vm.warp(8.01 days);
   //
   // 3. Caller distributes prizes using the signature
   proxyFactory.deployProxyAndDistributeBySignature(
      TEST_SIGNER, contestId, address(first_distrubute), signature,
sendingData
   );
   // after
   assertEq(MockERC20(jpycv2Address).balanceOf(user1), 9500
ether);
   assertEq(MockERC20(jpycv2Address).balanceOf(stadiumAddress),
500 ether);
   //
   // 4. For some reason there is a new distributor implementation.
   // The Owner set the new distributor for the same contestId
   Distributor new_distributor = new Distributor(address(proxyFactory),
stadiumAddress);
   vm.startPrank(factoryAdmin);
   proxyFactory.setContest(organizer, contestId, block.timestamp + 8
days, address(new_distributor));
   vm.stopPrank();
   bytes32 newDistributorSalt = keccak256(abi.encode(organizer,
contestId, address(new_distributor)));
   address proxyNewDistributorAddress =
proxyFactory.getProxyAddress(newDistributorSalt,
address(new_distributor));
   vm.startPrank(sponsor);
   MockERC20(jpycv2Address).transfer(proxyNewDistributorAddress,
10000 ether);
   vm.stopPrank();
   //
   // 5. The caller can distribute prizes using the same signature in
different distributor implementation
   vm.warp(20 days);
   proxyFactory.deployProxyAndDistributeBySignature(
      TEST_SIGNER, contestId, address(new_distributor), signature,
sendingData
   );
}


```

Result :

```solidity

Ran 1 test for test/integration/ProxyFactoryTest.t.sol:ProxyFactoryTest
[PASS] testSignatureCanBeUsedToNewImplementation() (gas:
2614853)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 22.18ms
(7.55ms CPU time)

Ran 1 test suite in 26.25ms (22.18ms CPU time): 1 tests passed, 0
failed, 0 skipped (1 total tests)

```

Recommended mitigation : Add implementation to avoid distribute any other unauthorized implementation.

```solidity

function deployProxyAndDistributeBySignature(
      address organizer,
      bytes32 contestId,
      address implementation,
      bytes calldata signature,
      bytes calldata data
   ) public returns (address) {

      //Once the data is signed, the signature is tied to that specific data.
So, no one can reuse it for something else unless the exact same data
is signed again by the approver
   --------->>>>>>    bytes32 digest =
_hashTypedDataV4(keccak256(abi.encode(contestId,implementation,d
ata)));

      //The contract will use the ECDSA.recover() function to check if the
signature matches the expected sender (the organizer)
      if (ECDSA.recover(digest, signature) != organizer) revert
ProxyFactory__InvalidSignature();
      bytes32 salt = _calculateSalt(organizer, contestId,
implementation);
      if (saltToCloseTime[salt] == 0) revert
ProxyFactory__ContestIsNotRegistered();
      if (saltToCloseTime[salt] > block.timestamp) revert
ProxyFactory__ContestIsNotClosed();
      address proxy = _deployProxy(organizer, contestId,
implementation);
      _distribute(proxy, data);
      return proxy;
   }
```

[M-1] TITLE (Root + Impact)

The digest in the ProxyFactory contract is not used according to the EIP-712 standard specification. This leads to signature failure and allows for data manipulation

Description

The digest in the ProxyFactory contract does not follow the https://eips.ethereum.org/EIPS/eip-712, resulting in signature validation failures and potential data manipulation.

Impact

1) Since the signature is not being calculated according to the EIP-712 standard, wallets and external tools that are EIP-712-compliant will fail to verify the signature correctly. This means the signature might be rejected by these wallets, causing integration failures
2) Without the correct data hashing and TypeHash inclusion, malicious actors could potentially manipulate the data or contestId fields before the signature is generated. This could lead to someone signing tampered data that wasn't intended

Recommended mitigation

Define and use the `typeHash` of the function.

```solidity
bytes32 public constant DISTRIBUTE_TYPEHASH = keccak256(
   "Distribute(address organizer,bytes32 contestId,bytes data)"
);
```

And then use it in the digest calculation

```solidity
bytes32 digest =
_hashTypedDataV4(keccak256(abi.encode(DISTRIBUTE_TYPEHASH,
contestId, keccak256(data))));
```

[H-2] TITLE (Root + Impact)

If the `STADIUM_ADDRESS` gets blacklisted, it can cause funds to be permanently stuck in the contract, and transactions will fail

Description

The `STADIUM_ADDRESS` is set as an `immutable` address used for transferring reward tokens. If this address becomes blacklisted by the token contract (e.g., due to centralized blacklist control like USDT or other tokens), it will no longer be able to send or receive tokens. Since the address is immutable and cannot be changed after deployment, this could result in funds being permanently stuck in the contract and the reward distribution mechanism breaking, leading to a denial-of-service for users

Impact

1) Funds could be permanently stuck in the contract if the `STADIUM_ADDRESS` gets blacklisted
2) Transactions will fail, leading to a denial-of-service for users
3) The contract may become unusable for its intended purpose,

causing frustration and loss of trust among users

Proof of Concepts

This test function simulates the process of minting, transferring, and blocking tokens to verify that an address that is blocklisted cannot interact with the contract or perform transfers.

Explanation:
Minting Tokens:

The contract owner starts the test and mints 1000 tokens to the STADIUM_ADDRESS.

The balance of STADIUM_ADDRESS is then checked to ensure that the minting was successful.

Approving Tokens:

The STADIUM_ADDRESS approves the contract to spend 500 tokens on its behalf.

The test then simulates a transfer of 500 tokens from STADIUM_ADDRESS to user1 using the transfertoone function. This checks if the transfer works as expected when the allowance is set.

Blocking the Address:

The contract owner blocklists the STADIUM_ADDRESS using the tomakeblock function.

This ensures that STADIUM_ADDRESS can no longer perform any transactions or transfers.

Attempted Transfer After Blocklisting:

The test checks that after STADIUM_ADDRESS is blocklisted, the transfer attempt from user1 to STADIUM_ADDRESS should fail.

The balance of both STADIUM_ADDRESS and user1 is checked to confirm that the transaction didn’t go through.

Assertions:

The test asserts that the STADIUM_ADDRESS still holds the original balance of 1000 tokens, and user1 has 0 tokens since the transfer didn’t happen due to the address being blocklisted.

Purpose:
This test ensures that once an address is added to the blocklist, it cannot perform any transfers or participate in token interactions

```solidity

function testmintsomeamounttoaddress() public {
   address user1 = address(2);
   uint256 mintAmount = 1000;

   // Start as the contract owner
   vm.startPrank(owner);

   // Mint some amount to the STADIUM_ADDRESS
   fakeusdc.mint(STADIUM_ADDRESS, mintAmount);

   // Check the balance of STADIUM_ADDRESS
   assertEq(fakeusdc.balanceoftoken(STADIUM_ADDRESS),
mintAmount);

   vm.stopPrank();

   // Start as the STADIUM_ADDRESS
   vm.startPrank(STADIUM_ADDRESS);

   // Approve the contract (this address) to spend 500 tokens
   fakeusdc.approve(STADIUM_ADDRESS, 500);
   // Transfer 500 tokens from STADIUM_ADDRESS to user1 using
transferFrom
   fakeusdc.transfertoone(STADIUM_ADDRESS, user1, 500);



   vm.stopPrank();


   // Check the balances after the transfer
   assertEq(fakeusdc.balanceoftoken(STADIUM_ADDRESS),
mintAmount - 500); // 500 should be transferred
   assertEq(fakeusdc.balanceoftoken(user1), 500); // user1 should have
500 tokens



   //now token block the address
   vm.startPrank(owner);
   fakeusdc.tomakeblock(STADIUM_ADDRESS);
   vm.stopPrank();



   //now it should not be able to transfer
   vm.startPrank(user1);
   fakeusdc.approve(user1, 500);
   fakeusdc.transfertoone(user1,STADIUM_ADDRESS , 500);
   vm.stopPrank();


   assertEq(fakeusdc.balanceoftoken(STADIUM_ADDRESS),
mintAmount);
   assertEq(fakeusdc.balanceoftoken(user1), 0);




}

```

Recommended mitigation :

It is recommended to allow `STADIUM_ADDRESS` to be updatable by a dedicated admin role to avoid token transfer blacklisting. Moreover, since `STADIUM_ADDRESS` is no longer `immutable`, `storage` collision should be taken into account