Smart Contract Security Audit Report

For

PowerContract

Date Issued: October.22, 2025

Version: v1.0

Confidentiality Level: Public

Contents

1 Abstract

This report was prepared for PowerContract smart contract to identify issues and vulnerabilities in its smart contract source code. A thorough examination of PowerContract smart contracts was conducted through timely communication with PowerContract, static analysis using multiple audit tools and manual auditing of their smart contract source code.

The audit process paid particular attention to the following considerations.

- A thorough review of the smart contract logic flow
- Assessment of the code base to ensure compliance with current best practice and industry standards
- Ensured the contract logic met the client's specifications and intent
- Internal vulnerability scanning tools tested for common risks and writing errors
- Testing smart contracts for common attack vectors
- Test smart contracts for known vulnerability risks
- Conduct a thorough line-by-line manual review of the entire code base

As a result of the security assessment, issues ranging from critical to informational were identified. We recommend that these issues are addressed to ensure a high level of security standards and industry practice. The recommendations we made could have better served the project from a security perspective.

- Enhance general coding practices to improve the structure of the source code.
- Provide more comments for each function to improve readability.
- Provide more transparency of privileged activities once the agreement is in place.

2 Overview

2.1 Project Summary

Project Summary	Project Information
Name	PowerContract
Start date	October.21, 2025
End date	October.22, 2025
Contract type	DEFI
Language	Solidity

2.2 Report HASH

Name	HASH
PowerContract	https://etherscan.io/address/0xbc9B096988Db6d4d39B553A3e4B2C4E1bf36d3F2#code

3 Project contract details

3.1 Contract Overview

PowerContract.sol

The PowerContract contract is a decentralized system for purchasing and distributing computing power. Users can purchase computing power with USDT (optionally binding a referrer). The system records purchase history, referral relationships, and statistical data. Once a referrer's account reaches a certain threshold (MIN_REFERRER_PURCHASE) in computing power purchases, it becomes a valid referrer.The project's multisig wallet receives USDT payments, while the finance wallet distributes and withdraws USDT or WBTC rewards to users who have purchased computing power (with unique IDs to prevent duplicate payments). The contract supports detailed paginated query interfaces (purchase history, withdrawal history, and referral relationships). Overall, this is a computing power sales and revenue distribution contract that supports a referral system and dual-currency reward settlement.

4 Audit results

4.1 Key messages

ID	Title	Severity	Status
01	Fixed computing power price constants that do not distinguish between token precision and value lead to pricing errors	Low	Ignore
02	Risk of repeated withdrawals due to user purchase amount not being reduced	Low	Ignore
03	_safeBindReferrer duplicate verification issue	Low	Ignore

4.2 Audit details

4.2.1 Fixed computing power price constants that do not distinguish between token precision and value lead to pricing errors

ID	Severity	Location	Status
1	Low	PowerContract.sol	Ignore

Description

The definition of COMPUTING_POWER_PRICE_WITH_DECIMALS is as follows:

```
uint256 private constant COMPUTING_POWER_PRICE_WITH_DECIMALS = 12 * 10 ** 6;
```

The same constant represents $12 in USDT and over $8,000 in WBTC. If the original design intent was "computing power price per unit = 12 USDT or the equivalent of 0.00017 WBTC," this implementation is clearly incorrect.

Code location:

```
function paymentUsdt(address _recipient, uint256 _amount, uint256 _id) external nonReentrant whenNotPaused {
    if (msg.sender != financeWallet) revert NotAuthorized();
    if (_recipient == address(0)) revert InvalidAddress();
    if (_amount == 0) revert InvalidAmount();
    if (_id == 0) revert InvalidId();
    if (amount[_recipient] < COMPUTING_POWER_PRICE_WITH_DECIMALS) revert RecipientNotPurchased();

function paymentWbtc(address _recipient, uint256 _amount, uint256 _id) external nonReentrant whenNotPaused {
    if (msg.sender != financeWallet) revert NotAuthorized();
    if (_recipient == address(0)) revert InvalidAddress();
    if (_amount == 0) revert InvalidAmount();
    if (_id == 0) revert InvalidId();
    if (amount[_recipient] < COMPUTING_POWER_PRICE_WITH_DECIMALS) revert RecipientNotPurchased();
```

If the project mistakenly believes this represents "12 USDT," then the logic is that computing power is significantly overvalued. If subsequent calculations of revenue or referral rewards are tied to actual investment amounts, this

could lead to abnormal revenue distribution or unfair distribution of computing power. If multi-coin purchases are supported in the future, this constant will lead to significant pricing inconsistencies between different token paths.

Recommendation

It is recommended to set separate price constants for different tokens.

Status

Ignore.

4.2.2 Risk of repeated withdrawals due to user purchase amount not being reduced

ID	Severity	Location	Status
2	Low	PowerContract.sol	Ignore

Description

amount[_recipient] is the user's cumulative purchase amount (accumulated in _purchaseComputingPower).

This variable is used to determine whether the user has purchased computing power, that is, whether they are eligible to receive payments. However, in the payment functions paymentUsdt / paymentWbtc :

This simply checks if amount[_recipient] >= COMPUTING_POWER_PRICE_WITH_DECIMALS .

This does not reduce the user's paid balance. Therefore, as long as the same user has purchased computing power once (>= 12 USDT), they can receive payments from the finance wallet repeatedly without any consumption or deduction logic.

Although this call is controlled by financeWallet in the current logic, two risks remain:

State logic inconsistency: amount[_recipient] is treated as "eligibility check" rather than "balance"; its meaning is confused, and future logic expansion may misuse it.

Code location:

```
function paymentUsdt(address _recipient, uint256 _amount, uint256 _id) external nonReentrant whenNotPaused {
    if (msg.sender != financeWallet) revert NotAuthorized();
    if (_recipient == address(0)) revert InvalidAddress();
    if (_amount == 0) revert InvalidAmount();
    if (_id == 0) revert InvalidId();
    if (amount[_recipient] < COMPUTING_POWER_PRICE_WITH_DECIMALS) revert RecipientNotPurchased();
    if (_isIdAlreadyUsed(_id, TokenType.USDT)) revert DuplicateId();

    // Check finance wallet balance and allowance
    uint256 financeBalance = IERC20(usdtToken).balanceOf(financeWallet);
    uint256 financeAllowance = IERC20(usdtToken).allowance(financeWallet, address(this));

    if (financeBalance < _amount) revert InsufficientFinanceWalletBalance();
    if (financeAllowance < _amount) revert InsufficientFinanceWalletAllowance();

    // Mark ID as used
    usedUsdtIds[_id] = true;

    // Record USDT withdrawal history
    uint256 recordId = withdrawalUsdtRecords.length;
    withdrawalUsdtRecords.push(WithdrawalRecord({
        user: _recipient,
        amount: _amount,
        timestamp: block.timestamp,
        success: true,
        tokenType: TokenType.USDT,
        id: _id
    }));
    userUsdtWithdrawalHistory[_recipient].push(recordId);

    // Update user and global USDT withdrawal amounts
    userUsdtWithdrawalAmount[_recipient] += _amount;
    totalUsdtWithdrawalAmount += _amount;

    // Perform external call last
    IERC20(usdtToken).safeTransferFrom(financeWallet, _recipient, _amount);

    emit PaymentProcessed(_recipient, _amount, block.timestamp, _id);
}
```

Recommendation

It is recommended to use independent mapping to record the amount that users can receive.

Status

Ignore.

4.2.3 _safeBindReferrer duplicate verification issue

ID	Severity	Location	Status
3	Low	PowerContract.sol	Ignore

Description

_purchaseComputingPower() calls _validateReferrer() , which already completes the _isValidDirectReferrer() check and the _wouldCreateCircularReferral() check.

```
function _purchaseComputingPower(uint256 _amount, address _referrerAddress) internal {
    if (_amount == 0) revert InvalidAmount();

    uint256 computingPowerPrice = COMPUTING_POWER_PRICE_WITH_DECIMALS;

    if (_amount % computingPowerPrice != 0) revert OnlyIntegerAmount();

    uint256 maxPurchaseAmount = 10_000_000 * 10**6;
    if (_amount > maxPurchaseAmount) revert InvalidAmount();

    uint256 computingPower = _amount / computingPowerPrice;
    if (computingPower == 0) revert InvalidAmount();

    if (_referrerAddress != address(0)) {
        if (!_validateReferrer(msg.sender, _referrerAddress)) {
            revert ReferralValidationFailed();
        }
    }

    if (IERC20(usdtToken).balanceOf(msg.sender) < _amount) revert InsufficientBalance();
    if (IERC20(usdtToken).allowance(msg.sender, address(this)) < _amount) revert InsufficientAllowance();

    if (_referrerAddress != address(0)) {
        _safeBindReferrer(msg.sender, _referrerAddress);
    }
```

```
function _validateReferrer(address _user, address _referrerAddress) internal view returns (bool) {
    if (_referrerAddress == address(0)) return false;
    if (_referrerAddress == _user) return false;
    if (referrer[_user] != address(0)) return false;

    if (!_isValidDirectReferrer(_referrerAddress)) {
        return false;
    }

    if (_wouldCreateCircularReferral(_user, _referrerAddress)) {
        return false;
    }

    return true;
}
```

_safeBindReferrer() calls the same check again, duplicating the logic.

```
function _safeBindReferrer(address _user, address _referrerAddress) internal {
    if (_referrerAddress == address(0)) revert InvalidAddress();
    if (_referrerAddress == _user) revert SelfReferral();
    if (referrer[_user] != address(0)) revert ReferralAlreadyExists();

    if (!_isValidDirectReferrer(_referrerAddress)) {   // Already judged in _validateReferrer
        revert InvalidReferrer();
    }

    if (_wouldCreateCircularReferral(_user, _referrerAddress)) {  // Already judged in _validateReferrer
        revert CircularReferral();
    }

    referrer[_user] = _referrerAddress;
    referrals[_referrerAddress].push(_user);

    emit ReferralBound(_user, _referrerAddress);
}
```

Recommendation

Keep _validateReferrer() as the only entry point for validation; simplify _safeBindReferrer() to only perform binding logic and event triggering, and no longer repeat validation.

Status

Ignore.

5 Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Disclaimer

This report is issued in response to facts that occurred or existed prior to the issuance of this report, and liability is assumed only on that basis. Shield Security cannot determine the security status of this program and assumes no responsibility for facts occurring or existing after the date of this report. The security audit analysis and other content in this report is based on documents and materials provided to Shield Security by the information provider through the date of the insurance report. in Shield Security's opinion. The information provided is not missing, falsified, deleted or concealed. If the information provided is missing, altered, deleted, concealed or not in accordance with the actual circumstances, Shield Security shall not be liable for any loss or adverse effect resulting therefrom. shield Security will only carry out the agreed security audit of the security status of the project and issue this report. shield Security is not responsible for the background and other circumstances of the project. Shield Security is not responsible for the background and other circumstances of the project.