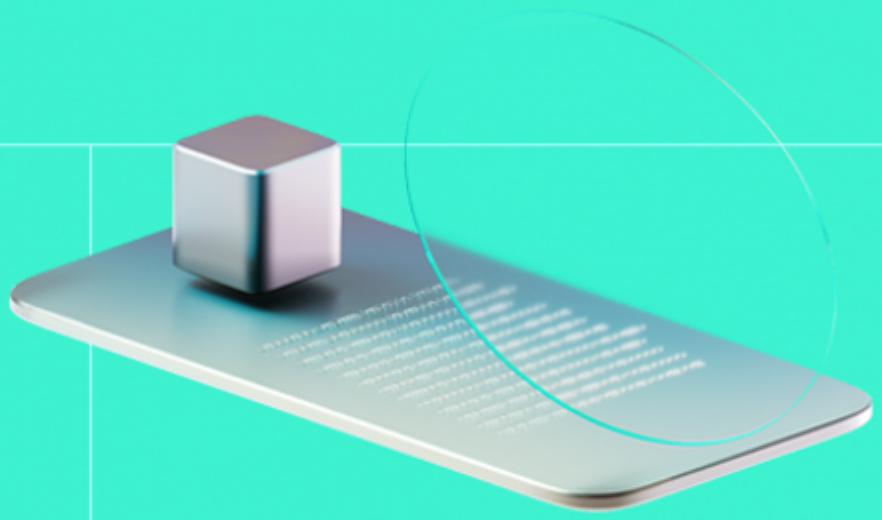
HACKEN

Smart Contract Code Review And Security Analysis Report

Customer: FatBoy

Date: 01/07/2024



We express our gratitude to the FatBoy team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Document

Name	Smart Contract Code Review and Security Analysis Report for FatBoy
Audited By	Carlo Parisi, Viktor Raboshchuk
Approved By	Przemyslaw Swiatowiec
Website	https://fatboygame.io/
Changelog	21/06/2024 - Preliminary Report -02/07/24 - Final Report
Platform	Arbitrum One, Ethereum, BNB Chain
Language	Solidity
Tags	Presale
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://gitlab.cleevio.cz/cleeviox/backend/fatboy-presale-sc
Commit	7a0f4875cf7366846c1f145d48054b9d5c511798

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

10	5	0	5
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	5
Medium	2
Low	1

Vulnerability	Status
F-2024-3962 - Contract Funds Can be Drained due to Wrong Cost Calculations	Mitigated
F-2024-3964 - Incorrect Definition of Stablecoins Decimal	Mitigated
F-2024-4001 - Missing Threshold Leading to Possible Draining of Funds	Mitigated
F-2024-4003 - Potential Loss of Purchased Tokens Due to Insufficient Payment Validation in _getTokensForCost Function	Mitigated
F-2024-4006 - Missing On-Chain Bonus Tokens for Referrer in _processReferralBonus	Mitigated
F-2024-3958 - Missing Checks for Zero Address	Fixed
F-2024-3960 - Missing Old Values Update During Stablecoin Address Set	Fixed
F-2024-3961 - Incorrect Decimal Assumption for ERC20 Tokens Causes Calculation Errors	Fixed
F-2024-3963 - Denial of Service Attack Due to Decimals Calculations	Fixed
F-2024-3997 - Hardcoded Price of Stablecoin	Fixed

Documentation quality

- Functional requirements are provided.
- NatSpec is present.
- Technical description is not provided.
- Development environment is described.

Code quality

- The code has sufficient quality.

Test coverage

Code coverage of the project is **83.59%** (branch coverage).

- Deployment and basic user interactions are covered with tests.

Table of Contents

System Overview

FatBoy is a presale protocol with the following contracts:

PresaleClaiming - is a smart contract designed for managing the presale and claiming process of tokens. It integrates functionalities for buying tokens during a presale period, applying referral and volume bonuses, and facilitating the claiming of purchased tokens after the presale ends. This contract is upgradeable and uses a proxy pattern for deployment.

ReferralUSDTRewards.sol - is a smart contract that allows users to claim USDT rewards based on a Merkle proof, which verifies their eligibility and the amount claimed, with additional functionalities for setting the Merkle root by authorized server addresses and emergency withdrawals by the contract owner.

IClaiming.sol - interface for the PresaleClaiming contract.

IPresale.sol - interface for the PresaleClaiming contract.

Privileged roles

- The owner of the PresaleClaiming contract can set referral bonus tiers, enable/disable referral bonuses, toggle volume buy bonuses, configure volume buy bonus tiers, change presale states, activate/deactivate claiming, set the treasury wallet and stable coin addresses (USDC and USDT), establish a minimum purchase amount, set the claimable token and withdraw tokens from the contract.
- Accounts with SERVER_ROLE in the PresaleClaiming contract can set the presale phase.
- The owner of the ReferralUSDTRewards contract can withdraw tokens from the contract.
- Accounts with SERVER_ROLE in ReferralUSDTRewards contract can set the Merkle root.

Risks

- The project utilizes Solidity version 0.8.20 or higher, which includes the introduction of the PUSH0 (0x5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- A potential risk exists with the contract being centralized around the setter for the `currentPresalePhaseIndex`. If the server role does not set this value accurately and in a timely manner, it could affect token distribution.
- A potential risk exists with the `emergencyWithdrawToken` function, which can only withdraw `claimableToken`. In an emergency, no other tokens can be withdrawn using this function, except for native tokens, which can be withdrawn using the `emergencyWithdrawNativeCoin` function. This limitation can hinder access to other tokens during critical situations, affecting liquidity and operational flexibility.
- `PresaleClaiming.sol` imports the `AggregatorV3Interface` from the Chainlink library, which is used to interact with external data feeds. This introduces a risk as the contract is dependent on these external data feeds for its operation. If these data feeds are compromised, manipulated, or become unavailable, it could impact the functionality and security of the contract.
- `buyTokensWert` in the contract `PresaleClaiming.sol` relies on `wert`, a third-party payment processor. This introduces a risk as the contract's functionality is dependent on the reliability and security of this external service. If `wert` is compromised, experiences downtime, or if its API changes unexpectedly, it could disrupt the contract's operations.
- The `setPaymentToken` function in the `PresaleClaiming.sol` contract checks the decimal places of a payment token when it is added as a payment method. If a non-standard token changes its decimal places after being added, the contract does not have a mechanism to re-validate or remove this token. This could lead to potential issues with transactions, as the contract assumes a fixed decimal place for each token.
- The current implementation of MerkleProof.sol in the OpenZeppelin library exhibits a vulnerability when handling leaf data that is exactly 64 bytes in size. This issue, detailed in Issue #278 from the sherlock-audit repository, enables an attacker to bypass the merkle-tree proof, leading to a security risks if the leaves are not built carefully. Additionally, this vulnerability is acknowledged with a warning in the MerkleProof.sol contract(#3091).

Findings

Vulnerability Details

F-2024-3964 - Incorrect Definition of Stablecoins Decimal - Critical

Description:

The contract relies on the assumption that `_STABLE_COIN_DECIMALS` represents the number of decimals for stablecoins like USDT and USDC. This assumption is used in important protocol functions:

```
uint8 private constant _STABLE_COIN_DECIMALS = 6;

function _computeExchangeRate(address paymentToken) private view returns (uint256 exchangeRate
    if (paymentToken != address(0) && _isStableCoin(paymentToken)) {
        exchangeRate = _ONE_ETHER;
        decimals = _STABLE_COIN_DECIMALS;
    } else {
        exchangeRate = _getExchangeRate(paymentToken);
        decimals = _BASE_DECIMALS;
    }
}

function _getReferralBonusTokens(
    uint256 tokensBought,
    uint256 usdAmount
) private view returns (uint256 referrerBonusUsdt, uint256 referredBonusTokens){
    ....
    referrerBonusUsdt = ((usdAmount * referrerPercentage) / _PERCENTAGE_DIVISOR) / (10 ** (_BA
    ....
}
```

This assumption holds true on Ethereum's chain but may not be consistent across other EVM-compatible blockchains. For instance, on Binance Smart Chain (BSC), the most widely used versions of USDT and USDC indeed have 18 decimals.

It should be noted that it is planned to deploy the `PresaleClaiming` contract on the BSC chain. Incorrect stablecoin decimals can lead to miscalculations and disrupt contract functionality, potentially leading to financial losses.

Status: Mitigated

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: To mitigate these risks, it is crucial for the contract to validate the decimal assumptions based on the specific blockchain being used. Implementing robust checks and validations ensures compatibility across different EVM chains maintains contract reliability.

Resolution:

Remediation (revised commit: 3c55e84f): The issue has been mitigated due to the client's comment:

“There will be no usage of USDT and USDC stable coins on BSC main net. The issue is fixed by checking if they stable coin has 6 decimals and other other payment tokens have 18 decimals.”

F-2024-3961 - Incorrect Decimal Assumption for ERC20 Tokens Causes Calculation Errors - High

Description:

The PresaleClaiming smart contract is designed to manage the presale and claiming process of tokens, This contract allows token purchases using native cryptocurrency or ERC20 tokens, applies referral and volume bonuses, and facilitates the claiming of purchased tokens after the presale.

In the `getCost()`, `_buyTokensWithERC20()`, and `_computeExchangeRate()` functions, the contract assumes that non-stablecoin payment tokens have 18 decimals. However, this assumption is not universally applicable to all tokens.

When the payment token's decimal precision deviates from the expected 18 decimals (`_BASE_DECIMALS`), the contract's calculations are significantly impacted. This discrepancy results in inaccurate computations for cost and receivable tokens. In extreme cases, where the token's decimals differ greatly from the expected value, these miscalculations can severely disrupt the contract's operations:

- **Inaccurate Cost Calculations:** The user may be overcharged or undercharged when buying tokens.
- **Incorrect Token Allocation:** Users may receive more or fewer tokens than they are entitled to.
- **Operational Disruption:** Significant deviations in decimal precision can lead to the contract malfunctioning, affecting the overall presale and claiming process.

It should be noted that only whitelisted tokens can be used for purchase operation, whitelisting operation is performed by the contract owner.

The `_computeExchangeRate` the function is returning hardcoded `_BASE_DECIMALS`, then these hardcoded decimals impact the cost of the token calculated by the `_buyTokensWithERC20` function:

```
function buyTokens(
    string calldata _referralCode,
    bytes32[] calldata _merkleProofReferral,
    uint256 _amountToBuy,
    address _paymentTokenAddress
) external payable override withPresaleActive {
    uint256 valueSent = msg.value;
    address user = msg.sender;

    (uint256 exchangeRate, uint8 decimals) = _computeExchangeRate(_paymentTokenAddress);

    ...

    else if (isSupportedPaymentToken[_paymentTokenAddress]) {
        IERC20 paymentToken = IERC20(_paymentTokenAddress);
        _buyTokensWithERC20(paymentToken, _amountToBuy, cost, user, decimals, bonusTokens, usdAm
...

function _computeExchangeRate(address paymentToken) private view returns (uint256 exchangeRate,
if (paymentToken != address(0) && _isStableCoin(paymentToken)) {
    exchangeRate = _ONE_ETHER;
    decimals = _STABLE_COIN_DECIMALS;
} else {
    exchangeRate = _getExchangeRate(paymentToken);
    decimals = _BASE_DECIMALS;
}
}

function _buyTokensWithERC20(
```

```
...
) private {
    tokensSold += tokensBought;
    tokensInfoPerPhase[currentPresalePhaseIndex].tokensSold += tokensBought;
    if (decimals < _BASE_DECIMALS) {
        cost = cost / 10 ** (_BASE_DECIMALS - decimals);
    }
    paymentToken.safeTransferFrom(user, treasuryWallet, cost);
```

Status: Fixed

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: It is recommended either to:

- fix aforementioned functions to include specific token decimals,
- or to enhance the `setPaymentToken()` function by adding a check to ensure that any token being added has exactly 18 decimals.

Resolution: **Remediation (revised commit: 3c55e84f):** Check for exactly 18 decimals is provided in the setPaymentToken function.

F-2024-3962 - Contract Funds Can be Drained due to Wrong Cost Calculations - High

Description:

The security vulnerability was observed in a situation when the calculated `cost` becomes very low due to adjustments for decimals, especially when `decimals` of `paymentToken` are less than `_BASE_DECIMALS`. In such cases, the following formula is used:

```
_BASE_DECIMALS: cost / 10 ** (_BASE_DECIMALS - decimals).
```

If the initial cost calculation results in a value below a certain threshold, dividing it by a large number can reduce the cost to zero or near-zero, enabling users to acquire tokens without proper payment.

```
    function getCost(
        .....
    ) external view returns (uint256 cost, uint256 tokensBought, uint256 usdAmount, uint256 volu
        uint256 exchangeRate;
        uint256 decimals = _BASE_DECIMALS;

        if (decimals < _BASE_DECIMALS) {
            cost = cost / 10 ** (_BASE_DECIMALS - decimals);
        }
    }

    function _buyTokensWithERC20(
        ....
    ) private {
        .....
        if (decimals < _BASE_DECIMALS) {
            cost = cost / 10 ** (_BASE_DECIMALS - decimals);
        }
        paymentToken.safeTransferFrom(user, treasuryWallet, cost);

    }
```

For example, when a payment token has 18 decimals and a stablecoin has 6 decimals, if the calculated cost falls below `10**12` in the token's native decimals, then the cost would be equal `0` due to Solidity division truncation. In a such scenario, users could potentially acquire tokens without transferring any ERC20 tokens.

This vulnerability undermines the integrity of the presale process by allowing users to drain the contract's funds without genuine payment, posing significant financial risks.

Status: Mitigated

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: Ensure that the adjusted `cost` remains above a minimum acceptable threshold after division. This can be achieved using a `require` statement to validate the adjusted `cost` is greater than zero.

Resolution: **Remediation (revised commit: 3c55e84f):** The issue has been mitigated due to the client's comment:

"There will be minimum payment by a user, which is set to 48 $. Cost of 1 token is 0.016-0.030 $. The minimum threshold is calculated before the transfer of the funds but for extra insurance we added a check to ensure the result cost is above 0."

F-2024-3997 - Hardcoded Price of Stablecoin - High

Description: The `_computeExchangeRate()` function currently hardcodes the exchange rate of stablecoins to `1 ether`, assuming a fixed value that does not necessarily match the real-time market price of these stablecoins. This means that regardless of the actual market value of the stablecoin (e.g., USDT or USDC), the contract always treats 1 dollar as equivalent to 1 stablecoin. When the stablecoin's actual value deviates from 1 dollar (depeg scenario), transactions involving stablecoins could be severely mispriced.

```
uint256 private constant _ONE_ETHER = 1 ether;
if (paymentToken != address(0) && _isStableCoin(paymentToken)) {
    exchangeRate = _ONE_ETHER;
    decimals = _STABLE_COIN_DECIMALS;
}
```

Status: Fixed

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: To mitigate this issue, the contract should dynamically fetch exchange rates from reliable oracles and parameterize decimals based on the token's specifications. This approach ensures that the contract accurately reflects real-world values and remains resilient to price fluctuations in stablecoins and other tokens.

Resolution: **Remediation (revised commit: 3c55e84f):** The price of the stablecoins is not hardcoded anymore and is retrieved using an oracle.

F-2024-4001 - Missing Threshold Leading to Possible Draining of Funds - High

Description:

There is a critical vulnerability in the `_getCostForTokens` function of the smart contract. If the product of `tokensToBuy` and `tokensPrice` is less than the `exchangeRate`, the calculated cost becomes zero, which could allow an attacker to drain the contract's funds.

The vulnerability arises when the calculated `tokensToBuy * tokensPrice` falls below the `exchangeRate`, causing the cost to be incorrectly set to zero due to integer division in Solidity. Exploiting this flaw could enable an attacker to drain the contract's funds by making minimal token purchases, potentially leading to substantial financial losses.

```
function _getCostForTokens(
    uint256 tokensToBuy,
    uint256 exchangeRate
) private view returns (uint256 cost) {
    cost = tokensToBuy * tokensInfoPerPhase[currentPresalePhaseIndex].tokensPrice / exchangeRate
}
```

Status: Mitigated

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Consider implementing a minimum purchase requirement threshold to ensure that the calculated cost is always non-zero. This can prevent the contract from being drained by small token purchases that exploit this vulnerability.

Resolution: **Remediation (revised commit: 3c55e84f):** The issue has been mitigated due to the client's comment:

"There will be minimum payment by a user, which is set to 48 $. Cost of 1 token is 0.016-0.030 $. The minimum threshold is calculated before the transfer of the funds but for extra insurance we added a check to ensure the result cost is above 0."

F-2024-4003 - Potential Loss of Purchased Tokens Due to Insufficient Payment Validation in _getTokensForCost Function - High

Description:

The `_getTokensForCost` function has an issue that can result in users receiving `0` tokens if the product of `exchangeRate * amountToPay` is less than `tokensPrice`. This bug could lead to significant losses for users who make purchases and receive no tokens in return, while the contract itself does not suffer any loss of funds.

```
function _getTokensForCost(
    uint256 amountToPay,
    uint256 exchangeRate
) private view returns (uint256 tokensBought) {
    tokensBought = exchangeRate * amountToPay / tokensInfoPerPhase[currentPresalePhaseIndex].tok
}
```

The vulnerability occurs when the product of `exchangeRate * amountToPay` is less than `tokensPrice`. Due to integer division truncation in Solidity, the division `exchangeRate * amountToPay / tokensPrice` results in zero, causing `tokensBought` to be `0`. As a result, no tokens are allocated to the user for their payment. If exploited, this vulnerability can lead to financial losses for users who make payments expecting to receive tokens in return.

Status: Mitigated

Classification

Impact: 4/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: To fix the issue in the `_getTokensForCost` function, ensure that the calculation results in a meaningful number of tokens for the user's payment.

Resolution: **Remediation (revised commit: 3c55e84f):** The issue has been mitigated due to the client's comment:

"The provided input exchange rate is always with 18 decimals and there is a check in the _getExchangeRate() function to be above 0. More checks in the _getExchangeRate were added to ensure the call reverts if we don't get exchangeRate. Token Price will always be between 16e15 - 30e15 inclusively, therefore the tokensBought can't fall to 0 if the user provides funds"

F-2024-3963 - Denial of Service Attack Due to Decimals Calculations - Medium

Description:

In the `_getExchangeRate()` function, there is a critical vulnerability due to comparing two constants: `BASE_DECIMALS` and the `decimals` value from the ERC20 contract.

The `_getExchangeRate` function is designed to return a value called `exchangeRateDecimals`, which represents the exchange rate of a specified payment token or the native coin. This `exchangeRateDecimals` calculation adjusts the result according to the difference between the base decimals and the token-specific decimals:

```
exchangeRateDecimals = uint256(exchangeRate) * 10 ** (_BASE_DECIMALS - decimals);
```

This poses a significant risk of a Denial of Service (DOS) attack, particularly if the payment token's `decimals` value exceeds the standard limit of 18. If the token's `decimals` value is higher than `BASE_DECIMALS`, it will cause an underflow when subtracting from 18, leading to a DOS.

Status: Fixed

Classification

Impact: 4/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Medium

Severity: Medium

Recommendations

Remediation:

To safeguard against such attacks, implement a compatibility check during the token registration process to ensure that only tokens with a `decimals` value less than or equal to `BASE_DECIMALS` are allowed. This ensures that no incompatible tokens can cause underflow issues during exchange rate calculations. Or adjust the exchange rate logic to handle different `decimals` values dynamically. Instead of relying on a constant subtraction, use a more flexible approach that can adapt to various token decimal configurations.

Resolution:

Remediation (revised commit: 3c55e84f): A new check is present in the `setPaymentToken()` function that does not allow to enable tokens that have less or more than `BASE_DECIMALS`.

```
function setPaymentToken(address _paymentToken, bool enabled) external onlyRole(DEFAULT_ADMIN_RC
        if (ERC20(_paymentToken).decimals() != _BASE_DECIMALS) {
            revert InvalidPaymentTokenError(_paymentToken);
        }
        isSupportedPaymentToken[_paymentToken] = enabled;
        emit PaymentTokenSetEvent(_paymentToken, enabled);
    }
```

F-2024-4006 - Missing On-Chain Bonus Tokens for Referrer in _processReferralBonus - Medium

Description:

There is a critical issue in the `_processReferralBonus` function where referrer bonuses are not distributed on-chain, unlike the bonuses for referred users. The relevant code segment is:

```
if (referrerBonusUsdt > 0) {
    emit ReferrerBonusEvent(referrerBonusUsdt, block.timestamp, currentPresalePhaseIndex, referr
}
if (referredBonusTokens > 0) {
    bonusTokens = referredBonusTokens;
    _awardBonusTokens(referredBonusTokens, user);
}
```

The vulnerability arises from discrepancies in how bonus tokens are handled between referrers and referred users. While referred users receive their bonus tokens directly through the `_awardBonusTokens` function, referrers only get an event emission (`ReferrerBonusEvent`) without actual token distribution on-chain.

Status: Mitigated

Classification

Impact: 3/5

Likelihood: 4/5

Exploitability: Semi-Dependent

Complexity: Medium

Severity: Medium

Recommendations

Remediation: To mitigate this vulnerability, ensure both `referrerBonusUsdt` and `referredBonusTokens` are consistently awarded on-chain according to the intended logic.

Resolution: **Remediation (revised commit: 3c55e84f):** This issue is mitigated due to the client's comment:

"The distribution of the referrer rewards is in USDT stable coin on the Arbitrum One chain, which aggregates all referrer bonus tokens from other deployments. Another application monitors these referrer bonus events and updates merkleroot in the ReferralUSDTRewards contract, which is deployed on ARB chain. To implement distribution of USDT rewards on chain is a cross chain operation which cannot be solved easily"

F-2024-3960 - Missing Old Values Update During Stablecoin Address Set - Low

Description: The `PresaleClaiming.sol` contract has functions `setUsdcStableCoinAddress()` and `setUsdtStableCoinAddress()` to set the addresses of USDC and USDT stablecoins respectively. These addresses are marked as supported payment tokens in the `isSupportedPaymentToken[]` mapping. However, if these functions are called more than once with different addresses, the old addresses are not deactivated in the mapping. This means that the old addresses are still marked as supported payment tokens, potentially leading to unexpected behavior or security issues.

```
function setUsdcStableCoinAddress(address _usdcStableCoinAddress) external onlyOwner {
    usdcStableCoinAddress = _usdcStableCoinAddress;
    isSupportedPaymentToken[_usdcStableCoinAddress] = true;
    emit PaymentTokenSetEvent(_usdcStableCoinAddress, true);
}

function setUsdtStableCoinAddress(address _usdtStableCoinAddress) external onlyOwner {
    usdtStableCoinAddress = _usdtStableCoinAddress;
    isSupportedPaymentToken[_usdtStableCoinAddress] = true;
    emit PaymentTokenSetEvent(_usdtStableCoinAddress, true);
}
```

Status: Fixed

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: To ensure that these previous addresses are no longer recognized as supported payment tokens, it is recommended to set the old address values to `false` in the `isSupportedPaymentToken` mapping. This step will effectively remove the old addresses from the list of valid payment tokens, maintaining the integrity and accuracy of the supported payment tokens list.

Resolution: **Remediation (revised commit: 3c55e84f):** The old value for `isSupportedPaymentToken` is now set to false.

F-2024-3958 - Missing Checks for Zero Address - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

The following methods should introduce zero address checks:

- PresaleClaiming.sol: updateTreasuryWallet(), setHotWalletAddress(), setEstimationAddress(), initialize(), _initializeTreasuryWallet(), _initializePayment()
- ReferralUSDTRewards.sol: constructor,

Status: Fixed

Classification

Impact: 1/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Observation Details

F-2024-3968 - Incorrect Total USD Amount Raised Calculation - Info

Description: In lines 196 and 246 of the `PresaleClaiming.sol` contract, there is an issue where the bonus referral amount paid to the referrer is incorrectly added to the `totalUsdAmountRaised`. The amount raised should not increase when paying a bonus, as this is not new USD being raised but USD being paid out by the system.

```
(uint256 bonusTokens, bool isValidProof) = _processBonuses(_referralCode, _merkleProofReferral,
usdAmount += _computeUsdAmount(bonusTokens);
totalUsdAmountRaised += usdAmount;
```

Status: Accepted

Recommendations

Remediation: Consider removing the bonus referral amount from `totalUsdAmountRaised` by updating it with only the actual USD amount paid by the user, ensuring accurate tracking of funds.

F-2024-4000 - Missing valueSent Check Before Buying Tokens - Info

Description:

In `buyTokensWert()` before calling `_buyTokensWithNativeCoin()` there is no check to ensure that `valueSent` higher/equal `cost` of tokens.

Here is the corresponding code:

```
function buyTokensWert(
    string calldata _referralCode,
    bytes32[] calldata _merkleProofReferral,
    address _userToBuyFor
) external payable withPresaleActive {
    uint256 valueSent = msg.value;
    if (valueSent == 0) {
        revert ZeroValueError();
    }
    ...
    _buyTokensWithNativeCoin(tokensBought, valueSent, user, bonusTokens, usdAmount, isValidProof

    emit TokensBoughtWertEvent(user, valueSent, tokensBought, currentPresalePhaseIndex, block.ti
}

function _buyTokensWithNativeCoin(
    ...
) withNativeCoinEnabled private {
    uint256 amountReceived = valueSent;
    tokensSold += tokensBought;
    tokensInfoPerPhase[currentPresalePhaseIndex].tokensSold += tokensBought;

    uint256 amountToSend = (amountReceived - cost > _RETURN_AMOUNT_THRESHOLD)
        ? cost
        : amountReceived;

    (bool sent,) = payable(treasuryWallet).call{value: amountToSend}("");
    ...
}
```

If the `valueSend` is less than a `cost`, then the transaction would fail in the following line due to the underflow whereas it should return error that user send less value than required:

```
uint256 amountToSend = (amountReceived - cost > _RETURN_AMOUNT_THRESHOLD)
        ? cost
        : amountReceived;
```

Status: Accepted

Recommendations

Remediation:

Consider adding a check to ensure that the `buyTokensWert` function validates the sufficiency of funds before proceeding with token purchases.

```
if (valueSent > 0 && _paymentTokenAddress == address(0)) {
    if (valueSent < cost) {
        revert InsufficientFundsError(cost, valueSent);
    }
       _buyTokensWithNativeCoin()
}
```

F-2024-4013 - Usage of Both Ownable and AccessControl Implementations - Info

Description: The `PresaleClaiming.sol` and `ReferralUSDTRewards.sol` contracts currently implement access control through mechanisms such as `AccessControl` and `Ownable`. An optimization strategy worth considering involves potentially removing the `Ownable` implementation to decrease gas consumption. This approach is viable because both contracts already utilize `AccessControl` to manage roles such as `DEFAULT_ADMIN_ROLE` and `SERVER_ROLE`. By consolidating access control under `AccessControl`, redundant functionality provided by `Ownable` can be eliminated, streamlining the contracts' architecture and enhancing efficiency in gas usage.

Status: Fixed

Recommendations

Remediation: Consider removing the `Ownable` implementation. It may involve replacing the `onlyOwner` modifier in protocol functions with role-based modifiers like `onlyRole(SERVER_ROLE)` or `onlyRole(DEFAULT_ADMIN_ROLE)`.

F-2024-4014 - Missing UUPS Init Function Invocation - Info

Description:

`__UUPSUpgradeable_init()` is missed in the `initialize` function of the PresaleClaiming.sol contract.

Initializing every upgradeable contract ensures proper functioning, security, and enables future upgrades by setting the initial state and preventing re-initialization risks. This is particularly important for UUPS contracts, where initialization establishes necessary configurations and mechanisms for upgradeability.

Despite not being a security vulnerability in this specific scenario, including the `__UUPSUpgradeable_init()` call in the `initialize` function is considered best practice. It ensures consistency and clarity, showing that the contract adheres to the expected upgradeable pattern.

Status: Fixed

Recommendations

Remediation:

It is recommended to include the `__UUPSUpgradeable_init()` call in the initialize function of the PresaleClaiming.sol contract to adhere to best practices for upgradeable contracts, ensuring clarity and future compatibility with OpenZeppelin's libraries.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://gitlab.cleevio.cz/cleeviox/backend/fatboy-presale-sc
Commit	7a0f4875cf7366846c1f145d48054b9d5c511798
Whitepaper	https://whitepaper.fatboygame.io/
Requirements	https://gitlab.cleevio.cz/cleeviox/backend/fatboy-presale-sc/-/blob/main/README.md
Technical Requirements	https://gitlab.cleevio.cz/cleeviox/backend/fatboy-presale-sc/-/blob/main/README.md

Contracts in Scope
./src/PresaleClaiming.sol
./src/ReferralUSDTRewards.sol
./src/interfaces/IClaiming.sol
./src/interfaces/IPresale.sol