# Bonsai3 IDO
# Security Audit Report

**PREPARED FOR:**

**Bonsai3 Crypto**

**ARCADIA CONTACT INFO**
**Email:** audits@arcadiamgroup.com

**Telegram:** https://t.me/thearcadiagroup

**Revision history**

| Date | Reason | Commit |
|---|---|---|
| 11/16/2023 | Initial Audit Scope | #ea30e43e474d51e526d9bfbb2c16e4ab56b64243 |
| | Review Of Remediations | |

# Table of Contents

# Executive Summary

## Introduction

Bonsai3 engaged Arcadia to perform a security audit of their main smart contracts version 2. Our review of their codebase occurred on the commit hash #ea30e43e474d51e526d9bfbb2c16e4ab56b64243

## Review Team

1. Tuan “Anhnt” Nguyen - Security Researcher and Engineer
2. Joel Farris - Project Manager

## Project Background

**Bonsai3** offers a pioneering no-code toolkit designed by developers, aiming to be the WordPress equivalent for Web3. It empowers teams to deploy projects swiftly, affordably, and securely at any scale. The platform ensures community ownership and decentralized asset management, featuring unique tool sets for token creation, pre-sales, smart contract management, and market-making strategies. The roadmap encompasses phases focused on mechanics, community-driven features, UI/UX enhancements, and a comprehensive suite of developer tools, promising a robust evolution in line with Web3's future.

## Coverage

For this audit, we performed research, test coverage, investigation, and review of Bonsai3’s main contract followed by issue reporting, along with mitigation and remediation instructions as outlined in this report. The following code repositories, files, and/or libraries are considered in scope for the review.

| Contracts | Lines | nLines | nSLOC | Comment Lines | Complex. Score |
|---|---|---|---|---|---|
| bonsai3-smart-contracts/src/FeeManager.sol | 18 | 12 | 2 | 13 | |
| bonsai3-smart-contracts/src/TokenVault.sol | 36 | 36 | 24 | 5 | 43 |

| | | | | | |
|---|---|---|---|---|---|
| bonsai3-smart-contracts/src/Bonsai3.sol | 254 | 240 | 199 | 16 | 91 |
| bonsai3-smart-contracts/src/TokenFactory.sol | 22 | 22 | 15 | 2 | 15 |
| bonsai3-smart-contracts/src/Interfaces/IFeeManager.sol | 224 | 194 | 143 | 14 | 83 |

## Methodology

Arcadia completed this security review using various methods, primarily consisting of dynamic and static analysis. This process included a line-by-line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

The followings are the steps we have performed while auditing the smart contracts:

- Investigating the project and its technical architecture overview through its documentation
- Understanding the overview of the smart contracts, the functions of the contracts, the inheritance, and how the contracts interface with each others thanks to the graph created by Solidity Visual Developer
- Manual smart contract audit:
    - Review the code to find any issue that could be exploited by known attacks listed by Consensys
    - Identifying which existing projects the smart contracts are built upon and what are the known vulnerabilities and remediations to the existing projects
    - Line-by-line manual review of the code to find any algorithmic and arithmetic related vulnerabilities compared to what should be done based on the project's documentation
    - Find any potential code that could be refactored to save gas
    - Run through the unit-tests and test-coverage if exists
- Static Analysis:
    - Scanning for vulnerabilities in the smart contracts using Static Code Analysis Software
    - Making a static analysis of the smart contracts using Slither
- Additional review: a follow-up review is done when the smart contracts have any new update. The follow-up is done by reviewing all changes compared to the audited commit revision and its impact to the existing source code and found issues.


## Summary

There were **0** issues found, **0** of which were deemed to be ‘critical’, and **0** of which were rated as ‘high’. At the end of these issues were found throughout the review of a rapidly changing codebase and not a final static point in time.

| Severity Rating | Number of Original Occurrences | Number of Remaining Occurrences |
|---|---|---|
| CRITICAL | 8 | 0 |
| HIGH | 6 | 0 |
| MEDIUM | 2 | 0 |
| LOW | 0 | 0 |
| INFORMATIONAL | 0 | 0 |

# Findings in Manual Audit

## (BI3-1) Could not create new Token by using **memory** keyword.

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High.

### Code Segment

```
function createNewToken(
       uint256 _id,
       uint256 _totalSupply,
       string calldata _tokenName,
       string calldata _tokenTicker,
       uint8 _template,
       address userWallet
   ) public payable contractTurnedOn returns (address) {
       require(
           feeManager.chargeFeeByType{value: msg.value}(
               IFeeManager.FeeTypes.TokenCreationFee,
               userWallet
           ),
           "please pay fee"
       );
       ChildToken memory newChildToken = tokenDetails[_id];
       newChildToken.id = _id;
       newChildToken.tokenName = _tokenName;
```

### Description

**`newChildToken`** is declared as a **`memory`** variable of type **`ChildToken`**. Therefore, any modifications made to **`newChildToken`** won't affect the state variables or data stored in

**tokenDetails** outside the function. Users won't be able to create new tokens through our platform.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Proof of concept

–

### Recommendation

Use **`storage`** will enable modifications to persist beyond the function's execution.

## (BI3-2) Contract code heavily relies on ***tx.origin*** rather than ***msg.sender*** for authorization.

### Status

Unresolved

### Risk Level

Severity: High, likelihood: High.

### Description

The entire contract code heavily relies on tx.origin rather than msg.sender for authorization, potentially exposing vulnerabilities when using call  (sending ETH or calling other contracts).

Besides the issue with authorization, there is a chance that tx.origin will be removed from the Ethereum protocol in the future, so code that uses tx.origin won't be compatible with future releases Vitalik: 'Do NOT assume that tx.origin will continue to be usable or meaningful.'

It's also worth mentioning that by using tx.origin you're limiting interoperability between contracts because the contract that uses tx.origin cannot be used by another contract as a contract can't be the tx.origin (ie proxies ..)

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Proof of concept

–

### Recommendation

Using **`msg.sender`** provides better security by referring to the direct caller of a function, which helps prevent certain types of attacks or unauthorized access compared to using **`tx.origin`**.

## (BI3-3) Absence of validating input parameters.

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High

### Code Segment

```
function createNewToken(
      uint256 _id,
      uint256 _totalSupply,
      string calldata _tokenName,
      string calldata _tokenTicker,
      uint8 _template,
      address userWallet
  ) public payable contractTurnedOn returns (address) {
      require(
          feeManager.chargeFeeByType{value: msg.value}(
              IFeeManager.FeeTypes.TokenCreationFee,
              userWallet
          ),
          "please pay fee"
```

```
        );
.....
function createNewSale(
        uint256 _id,
        uint256 _totalSaleSupply,
        uint256 _hardcap, // change to token price
        uint256 _softCap,
        uint256 _startTime,
        uint256 _endTime,
        address _saleToken,
        address _purchaseToken,
        address _userWallet
    ) public payable returns (address) {
        // Payment
        require(

            feeManager.chargeFeeByType{value: msg.value}(
                IFeeManager.FeeTypes.TokenSaleFee,
                _userWallet
            ),
            "please pay fee"
        );
```

### Description

The functions **`createNewSale`** and **`createNewToken`** do not validate the **`_id`** input parameter, enabling anyone to overwrite the owner, startTime, endTime, and totalSupply of an existing sale. This vulnerability allows potential attackers to manipulate any sale, potentially resulting in locked funds and unauthorized fund withdrawals from the sale.

Additionally, it's essential to validate the other input parameters as well.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Proof of concept

_

### Recommendation

Ensure the validation of input parameters for both the **`createNewToken`** and **`createNewSale`** functions.

## (BI3-4) Lack of validation for token amount received during token transfers into the contract.

### Status

Unresolved

### Risk Level

Severity: High, likelihood: Medium

### Code Segment

```
function _fundSaleContract(
        address _saleToken,
        uint256 _transferQuantity,
        address origin
    ) internal requireConfirmation returns (bool) {
        IERC20 saleToken = IERC20(_saleToken);

        // feeManager.chargeFeeByType(IFeeManager.FeeTypes.SaleFee, origin);
this fee should apply to the transferred tokens?
        saleToken.transferFrom(origin, address(this), _transferQuantity);
        return true;
    }


……
function _handleTokenPayment(
        uint256 _id,
        uint256 _purchaseAmount,
        address user
    ) internal {
        if (saleDetails[_id].purchaseToken == address(0)) {
            // Using address(0) to represent native currency
```

```
            require(msg.value == _purchaseAmount, "Incorrect ETH amount sent");
        } else {
            IERC20 saleToken = IERC20(saleDetails[_id].purchaseToken);
            require(
                saleToken.transferFrom(user, address(this), _purchaseAmount),
                "Token transfer failed"
            );
        }
    }

function depositTokenForVest(
        uint256 _vestId,
        address _tokenAddress,
        address[] calldata _claimableAddresses,
        uint256[] calldata _claimableAmount,
        uint256 _vestingTimeCliff,
        uint256 _vestingTimePeriod,
        uint256 _totalTokens
    ) external {
        address sender = msg.sender;
        IERC20 token = IERC20(_tokenAddress);
        token.transferFrom(sender, address(this), _totalTokens);
```

### Description

Many tokens apply taxes to user transfers, resulting in the received amount not matching the transfer parameter. This emphasizes the critical need to validate the received token amounts, especially within the sales platform.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
bonsai3-smart-contracts/src/FeeManager.sol
bonsai3-smart-contracts/src/TokenFault.sol
```

### Proof of concept

–

### Recommendation

Perform a double check on the received token amount following token transfers.

## (BI3-5) Inconsistency in soft cap application logic causing permanent fund lock for users.

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: Medium

### Code Segment

```
function endSale(uint256 _id) public returns (bool) {
     require(
         block.timestamp >= saleDetails[_id].endTime,
         "Sale has not ended"
     );

     require(
         saleDetails[_id].owner == tx.origin,
         "Only the sale owner can end the sale"
     );
     saleDetails[_id].saleEnded = true;
     emit SaleEnded(_id);
     return true;
 }

 function userClaimTokens(uint256 _id) public returns (uint256) {
     require(
         didThisUserParticipate(msg.sender, _id),
         "user did not participate"
     );
     require(saleDetails[_id].saleEnded == true, "Sale has not ended");
     require(
```

```
            saleDetails[_id].participatedAmount >= saleDetails[_id].softCap,
            "Sale has not reached softcap"
        );
```

### Description

The sale's owner can conclude a sale without the soft cap condition, yet when a user withdraws, the soft cap condition applies. This situation could potentially lock the user's funds forever if the sale ends without reaching the soft cap

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Proof of concept

–

### Recommendation

Assess the logic for concluding the sale.

## (BI3-6) transferFrom doesn't use msg.sender as parameter.

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: Medium

### Code Segment

```
function chargeFeeByType(
      IFeeManager.FeeTypes key,
      address user
  ) external payable returns (bool) {
      uint256 balance = fees[key].amount;

      if (msg.value >= balance) {
          FeeObject storage invoices = fees[key];
```

```
            invoices.amount = balance;
            invoices.currency = fees[key].currency;

            FeesOccurred storage userPurchase = chargedFees[user];
            userPurchase.feeCodes.push(key);
            userPurchase.payingWallet = user;
            userPurchase.amount += fees[key].amount;
            userPurchase.currency = fees[key].currency;
            _handleTokenPayment(key, msg.value, user);
            userPurchase.paid = true;
            return true;
        } else {
            FeeObject storage invoices = fees[key];

            invoices.amount = balance;
            invoices.currency = fees[key].currency;

            FeesOccurred storage userPurchase = chargedFees[user];
            userPurchase.feeCodes.push(key);
            userPurchase.payingWallet = user;
            userPurchase.amount += fees[key].amount;
            userPurchase.currency = fees[key].currency;
            _handleTokenPayment(key, balance, user);
            userPurchase.paid = true;
            return true;
        }
    }

function _handleTokenPayment(
        IFeeManager.FeeTypes key,
        uint256 _purchaseAmount,
        address user
    ) internal {
        if (fees[key].currency == address(0)) {
            // Using address(0) to represent native currency
            require(msg.value == _purchaseAmount, "Incorrect ETH amount sent");
            // Transfer Ether to the recipient address
            (bool sent, ) = payable(__vault).call{value: _purchaseAmount}("");
            require(sent, "Failed to send Ether");
        } else {
```

```
            IERC20 saleToken = IERC20(fees[key].currency);
            require(
                saleToken.transferFrom(user, __vault, _purchaseAmount),
                "Token transfer failed"
            );
        }
}
```

### Description

Any user can invoke the **chargeFeeByType** function, triggering the execution of **saleToken.transferFrom(user, __vault, _purchaseAmount)**. If user A has approved a significant token amount for this contract, any user could transfer an equivalent fee from user A's funds to the **_vault**. If the cost of each transaction is minimal, it could result in complete drainage of the user's funds.

### Code location

```
bonsai3-smart-contracts/src/FeeManager.sol
```

### Proof of concept

- User A approves max uint256 token payment to **FeeManager**
- User B calls **chargeFeeByType** with msg.value zero with user A's address resulting in a transaction that transfers an amount equal to the fee from user A's funds to the _vault.

### Recommendation

Limit access to the Fee Manager to specific actors, such as the **owner** and **bonsai3.**

## (BI3-7) No refund mechanism in FeeManager.

### Status

Unresolved

### Risk Level

Severity: Medium, likelihood: High.

### Code Segment

```
function chargeFeeByType(
        IFeeManager.FeeTypes key,
        address user
    ) external payable returns (bool) {
        uint256 balance = fees[key].amount;

        if (msg.value >= balance) {
            FeeObject storage invoices = fees[key];

            invoices.amount = balance;
            invoices.currency = fees[key].currency;

            FeesOccurred storage userPurchase = chargedFees[user];
            userPurchase.feeCodes.push(key);
            userPurchase.payingWallet = user;
            userPurchase.amount += fees[key].amount;
            userPurchase.currency = fees[key].currency;
            _handleTokenPayment(key, msg.value, user);
            userPurchase.paid = true;
            return true;
        }
```

### Description

If a user sends an amount larger than the required fee, the contract does not refund the excess.

### Code location

```
bonsai3-smart-contracts/src/FeeManager.sol
```

### Recommendation

Features a mechanism designed to reimburse any surplus amount.

## (BI3-8) Flawed payment logic using varied payment tokens

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High.

### Code Segment

```
function addFeeToUser(
        IFeeManager.FeeTypes feeId,
        address user
    ) internal feeMustExist(feeId) {
        FeesOccurred storage userPurchase = chargedFees[user];
        userPurchase.feeCodes.push(feeId);
        userPurchase.payingWallet = user;
        userPurchase.amount += fees[(feeId)].amount;
        userPurchase.currency = fees[feeId].currency;
    }

function chargeFeeByType(
        IFeeManager.FeeTypes key,
        address user
    ) external payable returns (bool) {
        uint256 balance = fees[key].amount;

        if (msg.value >= balance) {
            …
            userPurchase.payingWallet = user;
            userPurchase.amount += fees[key].amount;
            userPurchase.currency = fees[key].currency;
            _handleTokenPayment(key, msg.value, user);
            userPurchase.paid = true;
            return true;
        } else {
            …
            userPurchase.amount += fees[key].amount;
```

```
            userPurchase.currency = fees[key].currency;
            _handleTokenPayment(key, balance, user);
            userPurchase.paid = true;
            return true;
        }
    }
```

### Description

The contract aims to enable the use of multiple tokens for various fees, but the implementation falls short. It aggregates **fees[key].amount** without factoring in token decimals (e.g., USDT: 6, WBTC: 8, WETH: 18). Additionally, **userPurchase.currency/userPurchase.payingWallet** consistently reflects only the latest payment information, disregarding previous transactions.

### Code location

```
bonsai3-smart-contracts/src/FeeManager.sol
```

### Recommendation

- Revise the logic to consider token decimals for accurate fee aggregation.
- Use arrays to retain historical payment details alongside the latest transaction data for **userPurchase.currency/userPurchase.payingWallet**."

## (BI3-10) RefundSale failed due to token.transferFrom usage

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High.

### Code Segment

```
function refundSale(uint256 _id) public {
```

```
        …
        token.transferFrom(
            address(this),
            tx.origin,
            saleDetails[_id].userPurchase[tx.origin]
        );
        emit SaleRefunded(_id, saleDetails[_id].userPurchase[tx.origin]);
}
```

### Description

Use **transferFrom** from contract will result in a failed transaction.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Recommendation

Use **transfer** instead of **transferFrom**

## (BI3-11) The recording of the user's purchased amount is incorrect

### Status

Unresolved

### Risk Level

Severity: High, likelihood: High.

### Code Segment

```
function _calculateUserTokens(
        uint256 _id,
        address userAddress,
        uint256 userAmount
    ) internal {
        saleDetails[_id].participatedAmount += userAmount;
```

```
        saleDetails[_id].userPurchase[userAddress] = userAmount;
        …
    }
```

### Description

This function registers only the most recent participation rather than aggregating the user's overall involvement. When a user participates multiple times and seeks a refund, they'll receive an amount equal to their latest participation, lacking the summation of their total contributions.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Recommendation

Change the code to **saleDetails[_id].userPurchase[userAddress] += userAmount.**

## (BI3-12) Purchases below 100 in input value won't be recorded for the userToken.

### Status

Unresolved

### Risk Level

Severity: High, likelihood: High.

### Code Segment

```
function _calculateUserTokens(
        uint256 _id,
        address userAddress,
        uint256 userAmount
    ) internal {
        …
        saleDetails[_id].userTokens[userAddress] +=
            ((((userAmount * 1e18) / saleDetails[_id].hardcap) / 100) *
```

```
                saleDetails[_id].hardcap) /
            1e18;
        emit UserTokenAmount(saleDetails[_id].userTokens[userAddress]);

    }
```

### Description

The userTokens formula could be shortened to:

```
        saleDetails[_id].userTokens[userAddress] += userAmount / 100;
```

Following this formula, if any participated amount is lesser than 100 (equivalent to 100 * 1e18 in value) would not be recorded.

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Recommendation

Reevaluate the formula and logic for recorded values.

## (BI3-13) Insufficient balance for users to claim.

### Status

Unresolved

### Risk Level

Severity: High, likelihood: High.

### Code Segment

```
modifier youNeedToSpendMore(uint256 _id, uint256 _purchaseAmount) {
        require(
```

```
            ((_purchaseAmount * 1e18) / saleDetails[_id].hardcap) / 100 > 0,
            "User amount is too small"
        );
        _;
}
```

### Description

There are two cases this issue could happen

1. If **_totalSaleSupply** for a sale is less than or equal to the sale's **hardcap**, and there are more than one hundred participants
2. The **userToken** is calculated based on the deposited amount, not as a percentage of the total sale supply. Consequently, if the total deposit amount exceeds a certain threshold, the required tokens for claiming could surpass the **_totalSaleSupply.**

### Code location

```
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Recommendation

1. The userToken calculation should occur at the sale's conclusion, determined by the deposited pool ratio.
2. Review the logic of using hardcap.

## (BI3-14) Failed to create VestToken: non-existent storage array element

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High.

### Code Segment

```
function depositTokenForVest(
        uint256 _vestId,
        address _tokenAddress,
```

```
        address[] calldata _claimableAddresses,
        uint256[] calldata _claimableAmount,
        uint256 _vestingTimeCliff,
        uint256 _vestingTimePeriod,
        uint256 _totalTokens
    ) external {
        …
        VestToken memory newVest;
        newVest.vestId = _vestId;
        newVest.tokenAddress = _tokenAddress;
        newVest.claimableAddress = _claimableAddresses;
        newVest.claimableAmount = _claimableAmount;
        newVest.vestingTimeCliff = _vestingTimeCliff;
        newVest.vestingTimePeriod = _vestingTimePeriod;
        vestingList[_vestId] = newVest;
    }
```

### Description

In the function **depositTokenForVest**, the code uses **vestingList[_vestId]** to set the vesting, but **vestingList** is a storage array variable. This code will result in an out-of-bounds error and revert**.**

### Code location

```
bonsai3-smart-contracts/src/TokenVault.sol
```

### Recommendation

Consider using push or changing the variable to a mapping type.

## (BI3-15) Absence of validating input parameters in creating VestToken.

### Status

Unresolved

### Risk Level

Severity: Critical, likelihood: High

### Code Segment

```
function depositTokenForVest(
      uint256 _vestId,
      address _tokenAddress,
      address[] calldata _claimableAddresses,
      uint256[] calldata _claimableAmount,
      uint256 _vestingTimeCliff,
      uint256 _vestingTimePeriod,
      uint256 _totalTokens
  ) external {
      address sender = msg.sender;
      IERC20 token = IERC20(_tokenAddress);
      token.transferFrom(sender, address(this), _totalTokens);
      VestToken memory newVest;
      newVest.vestId = _vestId;
      newVest.tokenAddress = _tokenAddress;
      newVest.claimableAddress = _claimableAddresses;
      newVest.claimableAmount = _claimableAmount;
      newVest.vestingTimeCliff = _vestingTimeCliff;
      newVest.vestingTimePeriod = _vestingTimePeriod;
      vestingList[_vestId] = newVest;
  }
```

### Description

The functions **`depositTokenForVest`** does not validate the **`_vestId`** input parameter, enabling anyone to overwrite the claimableAddress, vestingTimeCliff, vestingTimePeriod  of an existing vesting. This vulnerability allows potential attackers

to manipulate any vesting, potentially resulting in locked funds and unauthorized fund withdrawals from the vesting.

Additionally, it's essential to validate the other input parameters as well.

### Code location

```
bonsai3-smart-contracts/src/TokenVault.sol
```

### Proof of concept

–

### Recommendation

Ensure the validation of input parameters for both the **`depositTokenForVest`**.

## (BI3-16) Unable to purchase with excess received eth

### Status

Unresolved

### Risk Level

Severity: Medium, likelihood: High.

### Code Segment

```
function _handleTokenPayment(
        IFeeManager.FeeTypes key,
        uint256 _purchaseAmount,
        address user
    ) internal {
        if (fees[key].currency == address(0)) {
            // Using address(0) to represent native currency
            require(msg.value == _purchaseAmount, "Incorrect ETH amount sent");
            // Transfer Ether to the recipient address
            (bool sent, ) = payable(__vault).call{value: _purchaseAmount}("");
            require(sent, "Failed to send Ether");
        } else {
            IERC20 saleToken = IERC20(fees[key].currency);
```

```
            require(
                saleToken.transferFrom(user, __vault, _purchaseAmount),
                "Token transfer failed"
            );
        }
    }
```

### Description

When using ETH as payment, the contract solely accepts the received amount that matches **_purchaseAmount.**

### Code location

```
bonsai3-smart-contracts/src/FeeManager.sol
bonsai3-smart-contracts/src/Bonsai3.sol
```

### Recommendation

Accept the payment if **msg.value >= _purchaseAmount.**

# Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.