QuillAudits

# AUDIT REPORT

January 22 2025

For

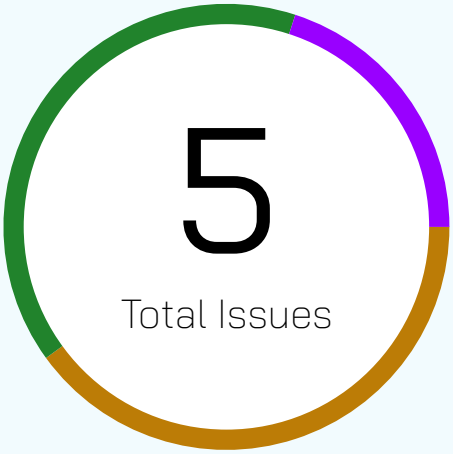
B3X.AI

# Table of Content

# Executive Summary

| | |
|---|---|
| **Project name** | B3X |
| **Overview** | Token sale contract is used for selling projects' own B3X tokens on deposit of usdc token |
| **Project Website** | https://www.b3x.ai/ |
| **Audit Scope** | The scope of this audit was to analyse the B3x Smart Contracts for quality, security, and correctness.<br><br>https://github.com/b-cube-ai/b3x-token-sale-evm-contracts |
| **Branch** | Main |
| **Contracts Under Scope** | contracts/sale<br>contracts/token/b3x.sol |
| **Method** | Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives. |
| **Review 1** | 8th January 2025 - 15th January 2025 |
| **Updated Code received** | 16th January 2025 |
| **Review 2** | 18th - 22nd January 2025 |
| **Fixed In** | https://github.com/b-cube-ai/b3x-token-sale-evm-contracts/commit/6975801e10d1695f0f139366bca37808a11c906f |

# Number of Issues per Severity




| | |
|---|---|
| High | 0 (0.00%) |
| Medium | 2 (40.00%) |
| Low | 2 (40.00%) |
| Informational | 1 (20.00%) |

Severity

| Issues | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open | 0 | 0 | 0 | 0 |
| Resolved | 0 | **2** | **2** | **1** |
| Acknowledged | 0 | 0 | 0 | 0 |
| Partially Resolved | 0 | 0 | 0 | 0 |

# Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception Disorder
- Gasless Send
- Balance Equality
- Compiler Version Not Fixed
- Redundant Fallback Function
- Send Instead of Transfer
- Style Guide Violation
- Unchecked External Call
- Unchecked Math
- Unsafe Type Inference
- Access Management
- Implicit Visibility Level
- Centralization of Control
- Improper or Missing Events
- Logical Issues and Flaws
- Arithmetic Computations Correctness
- Race Conditions/Front Running
- SWC Registry
- Malicious Libraries
- Missing Zero Address Validation
- Upgradeable Safety

# Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

**The following techniques, methods, and tools were used to review all the smart contracts.**

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

### Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

# Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

### High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

### Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

### Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

### Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

# Types of Issues

**Open**

Security vulnerabilities identified that must be resolved and are currently unresolved.

**Resolved**

Security vulnerabilities identified that must be resolved and are currently unresolved.

**Acknowledged**

Vulnerabilities which have been acknowledged but are yet to be resolved.

**Partially Resolved**

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

# Medium Severity Issues

## Sale can be initialized more than once

Resolved

### Path

sale.sol

### Function

Initialize()

### Description

In contract sale.sol, there is Initialize() function where the owner calls the function to set the sale starting and ending time period. The issue here is that Initialize() function can be called many times with different start and end times. There is no direct harm but say if the attacker gets access to the owner's private key and then can access the contract. There won't be any monetary problem but it'll result in users' deposits getting denied if sale is initialized more than once.

```
69    // @audit sale can be initialize more than once
70    /// @param _saleStart time when the token sale starts
71    /// @param _saleClose time when the token sale closes
      ftrace | funcSig
72    function Initialize(uint64 _saleStart, uint64 _saleClose) external onlyOwner {
73        if (_saleStart <= block.timestamp) revert InvalidSaleStart();
74        if (_saleClose <= _saleStart) revert InvalidSaleClose();
75        saleCancelled = false;
76        saleStart = _saleStart;
77        saleClose = _saleClose;
78        isUsdcWithdrawn = false;
79    }
80
```

### Recommendation

To resolve the issue please make sure that Initialize() function is called once.

**POC**

```
function testFail_SaleCanBeInitializeMoreThanOnce() external {
 vm.warp(block.timestamp+1 days);
 vm.prank(owner);
 sale.allocateB3X(100_000_000e18);

 vm.prank(user1);
 sale.depositUsdc(user1, 1000e6);

 vm.prank(owner);
 sale.Initialize(uint64(block.timestamp + 1 days), uint64(block.timestamp + 1 weeks));

 vm.prank(user1);
 sale.depositUsdc(user1, 1000e6);
}
```

## Arbitrary from is used instead of msg.sender in transferFrom

Resolved

### Path

sale.sol

### Function

depositUsdc()

### Description

In contract sale.sol, there is depositUsdc() function in which the user deposits the usdc amount to get the project's B3X tokens in return. Here on L105 there is transferFrom function in which beneficiary variable is used for from parameter. If a userA has approval of tokens from some other userB for some tokens then userA can put userB as beneficiary which will result in draining the funds of userB.

```
ftrace | funcSig
100   function depositUsdc(address beneficiary↑, uint256 _amount↑) external nonReentrant {
101       if (saleCancelled) revert SaleCancelled();
102       if (beneficiary↑ == address(0) || beneficiary↑ == address(this)) {
103           revert InvalidAddress();
104       }
105       if (usdcDeposited + _amount↑ > usdcHardCap()) revert MaxDepositAmountExceededForTGE();
106       if (block.timestamp < saleStart) revert SaleNotStarted();
107       if (block.timestamp > saleClose) revert SaleEnded();
108       if (_amount↑ == 0) revert InvalidValue();
109       if (deposits[beneficiary↑] + _amount↑ > MAX_DEPOSIT) revert MaxDepositAmountExceeded();
110
111       // @audit msg.sender should be used instead of a from value
112       IERC20(usdc).transferFrom(beneficiary↑, address(this), _amount↑);
113
114       _deposit(beneficiary↑, _amount↑);
115   }
```

### Recommendation

To resolve the issue please make sure to change the beneficiary parameter to msg.sender.

### Impact

- userB might be lose funds if tokens are already approved to userA

# Low Severity Issues

## Allocated amount should be more than 100K B3X tokens

Resolved

### Description

In token sale contract currently the user amount deposit calculation is dependent on how much B3X tokens are in the contract therefore, if user is to deposit 2000 USDC which is maximum token allowed to buy B3X tokens then at-least more than 100K B3X tokens should be present for first few users to deposit/buy with full 2000 USDC amount.

### Recommendation

To resolve the issue please make sure that tokens allocated in contract should be according to the tokenomics which is decided so that users will be able to buy them with max deposit amount if feasible.

## Missing zero address check

Resolved

### Path

sale.sol

### Function

constructor()

### Description

In constructor() of the contract sale.sol there should be a check for zero address for b3x and usdc contracts. Even though it does not seem necessary and owner is trusted and puts correct addresses. To actually ensure safety it is recommended to use zero address check.

```
62      // @audit check required for usdc zero contract address
        ftrace
63      constructor(address _b3x1, address _usdc1) Ownable(msg.sender) {
64          b3x = _b3x1;
65          usdc = _usdc1;
66          isUsdcWithdrawn = false;
67      }
68
```

### Recommendation

To resolve the issue add zero address check for b3x and usdc addresses.

# Informational Severity Issues

## Solidity pragma should be specific not wide

Resolved

### Description

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

## Gas Optimizations

Resolved

### Description

- To optimize gas the statement saleCancelled == false can be written as !saleCancelled in both withdrawUsdcAfterCancellelation() and withdrawUsdcAfterCancellationByOwner().
- getDepositers() and isDepositer() functions can be marked external.
- SafeCast for uint256 is not necessary after solidity version > 0.8.0

# Functional Tests

**Some of the tests performed are mentioned below:**

- [PASS] testFail_IfClaimB3XBeforeSaleClose() (gas: 254553)
- [PASS] testFail_InitializeWithSaleCloseLessThanSaleStart() (gas: 13168)
- [PASS] testFail_SaleCanBeInitializeMoreThanOnce() (gas: 261273)
- [PASS] testFail_depositUsdcIfMoreThan2000() (gas: 97358)
- [PASS] testFail_depositUsdcMoreThanB3X() (gas: 644630)
- [PASS] testFail_depositUsdcMoreThanOnceBySameUserGreaterThanMaxDeposit() (gas: 271831)
- [PASS] test_Initialize() (gas: 17665)
- [PASS] test_withdrawFunctionality() (gas: 442680)
- [PASS] test_allocateB3XTokens() (gas: 70442)
- [PASS] test_blacklistedUserFunctionality() (gas: 223951)
- [PASS] test_blacklistedUserFunctionalityCancelAllocation() (gas: 223795)
- [PASS] test_allocateB3XTokensCanBeDoneManyTimes() (gas: 88897)
- [PASS] testFuzz_allocateB3XTokens(uint256) (runs: 1000, μ: 84181, ~: 84262)
- [PASS] testFuzz_depositUsdcMoreThanB3X(uint256) (runs: 1000, μ: 37401265, ~: 37401312)

# Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

# Closing Summary

In this report, we have considered the security of the B3x contracts. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

# Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in B3x smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of B3x smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the B3x Team to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

# About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.


| | |
|---|---|
| **6+**<br>Years of Expertise | **1M+**<br>Lines of Code Audited |
| **$30B+**<br>Secured in Digital Assets | **1K+**<br>Projects Secured |

Follow Our Journey

# AUDIT REPORT

January 22 2025

For



Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com