Assure DeFi®

THE VERIFICATION GOLD STANDARD

Security Assessment

ViFoxCoin

Date: 19/11/2025

Audit Status: PASS

Audit Edition: Standard

ASSURE DEFI®

THE VERIFICATION GOLD STANDARD

Risk Analysis

Vulnerability summary

Classification	Description
High	High-level vulnerabilities can result in the loss of assets or manipulation of data.
Medium	Medium-level vulnerabilities can be challenging to exploit, but they still have a considerable impact on smart contract execution, such as allowing public access to critical functions.
Low	Low-level vulnerabilities are primarily associated with outdated or unused code snippets that generally do not significantly impact execution, sometimes they can be ignored.
Informational	Informational vulnerabilities, code style violations, and informational statements do not affect smart contract execution and can typically be disregarded.

Executive Summary

According to the Assure assessment, the Customer's smart contract is **Well Secured.**

Insecure	Poorly Secured	Secured	Well Secured

Scope

Target Code And Revision

For this audit, we performed research, investigation, and review of the ViFoxCoin contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

Target Code And Revision

Project	Assure
Language	Solidity
Codebase	https://bscscan.com/token/0xec69f1351b66902cd5e79f0924a5ad049f682540
Audit Methodology	Static, Manual

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

Category	Item
Code review & Functional Review	• Compiler warnings. • Race conditions and Reentrancy. Cross-function race conditions. • Possible delays in data delivery. • Oracle calls. • Front running. • Timestamp dependence. • Integer Overflow and Underflow. • DoS with Revert. • DoS with block gas limit. • Methods execution permissions. • Economy model. • Private user data leaks. • Malicious Event log. • Scoping and Declarations. • Uninitialized storage pointers. • Arithmetic accuracy. • Design Logic. • Cross-function race conditions. • Safe Zeppelin module. • Fallback function security. • Overpowered functions / Owner privileges

AUDIT OVERVIEW

HIGH

No high issues were found.

MEDIUM

1. Incomplete Ownership Transfer / Hidden Super-Admin [Fixed ✅]

Issue:

The contract exposes a function called transferOwnership and emits the standard OwnershipTransferred event, implying that control is transferred from previousOwner to newOwner.

However, the contract uses AccessControl with a DEFAULT_ADMIN_ROLE, and that role is never revoked or transferred from the deployer. The DEFAULT_ADMIN_ROLE remains with the original deployer forever unless they manually renounce it (which the contract never enforces).

Relevant code:

```
// In constructor
_grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
_grantRole(ADMIN_ROLE, msg.sender);
_grantRole(MINTER_ROLE, msg.sender);
_setRoleAdmin(MINTER_ROLE, ADMIN_ROLE);

function transferOwnership(address newOwner) external onlyRole(ADMIN_ROLE) {
    require(newOwner != address(0), "New owner is the zero address");
    grantRole(ADMIN_ROLE, newOwner);
    revokeRole(ADMIN_ROLE, msg.sender);
    emit OwnershipTransferred(msg.sender, newOwner);
}
```

Recommendation: Proper Ownership Transfer of Default Admin

Add a function (only callable by DEFAULT_ADMIN_ROLE) to transfer that role, and use it in transferOwnership:

```
function transferOwnership(address newOwner) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(newOwner != address(0), "New owner is the zero address");
```

```
    // Give new owner both DEFAULT_ADMIN_ROLE and ADMIN_ROLE
    _grantRole(DEFAULT_ADMIN_ROLE, newOwner);
    _grantRole(ADMIN_ROLE, newOwner);

    // Remove roles from current owner
    _revokeRole(ADMIN_ROLE, _msgSender());
    _revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());

    emit OwnershipTransferred(_msgSender(), newOwner);
}
```

Fix: Governance and ownership semantics now correctly match expectations: the multisig is the true and only super-admin. The original high-severity concern is fully addressed on-chain.

2. Vesting Logic Can Permanently Lock Non-Allocated Tokens on Vested Accounts [Acknowledged/Mitigated ✅]

Issue:

Accounts that have an entry in vestings[account] (for example, those provided via the allocations array in the constructor) are subject to vesting constraints on all transfers from that address.

Key functions:

```
struct Vesting {
    uint256 totalAllocation;
    uint256 startTimestamp;
    uint256 cliff;
    uint256 duration;
    uint256 released;
}

function vestedAmount(address account) public view returns (uint256) {
    Vesting memory vest = vestings[account];
    if (vest.totalAllocation == 0) return balanceOf(account);

    // ... otherwise vesting formula based on totalAllocation ...
}

function transferableAmount(address account) public view returns (uint256) {
    Vesting memory vest = vestings[account];
    if (vest.totalAllocation == 0) return balanceOf(account);

    uint256 vested = vestedAmount(account);
    uint256 alreadyReleased = vest.released;
    if (vested <= alreadyReleased) return 0;
    uint256 allowed = vested - alreadyReleased;
    uint256 bal = balanceOf(account);
    return allowed > bal ? bal : allowed;
}

function _update(address from, address to, uint256 amount)
```

```
    internal
    virtual
    override(ERC20Pausable)
{
    if (from != address(0) && to != address(0)) {
        Vesting storage vest = vestings[from];
        if (vest.totalAllocation > 0) {
            uint256 allowed = transferableAmount(from);
            require(amount <= allowed, "Vesting: amount exceeds vested tokens");
            vest.released += amount;
        }
    }
    super._update(from, to, amount);
}
```

Recommendation:

Track “locked amount” instead of “released”:

Conceptually:

locked = max(totalAllocation - vestedAmount, 0).

transferable = balance - locked (not exceeding balance).

Enforce amount <= transferable on _update.

This way, extra tokens acquired later (outside vesting) are always part of transferable as long as balance > locked.

Track vesting in a separate “vesting balance” pool:

Store allocated vesting amounts separately from the main ERC20 balance.

On each vesting epoch, move some vesting balance into the transferable balance.

Then _update doesn’t need to know about vesting at all.

Fix: The underlying code behavior remains unchanged, but strict wallet segregation + blacklisting + monitoring effectively prevent the problematic scenario. As long as these controls are maintained, the practical risk is near-zero.

3. Daily Mint Limit Is Not Applied to Staking Rewards [N/A ✅]

Issue:

There is a daily mint limit mechanism (dailyMintLimit, dailyMinted, lastMintReset) that is correctly applied to transferVFX (conversion of off-chain “VFX points” into tokens) but not applied at all to tokens minted as staking rewards.

Minting-related state:

```
// Minting control
uint256 public dailyMintLimit;
uint256 public dailyMinted;
uint256 public lastMintReset;
uint256 public penaltyPool;
```

Check of the daily limit (only in canMintTokens to transferVFX):

```
if (dailyMinted + totalTokens > dailyMintLimit) {
    return (false, totalTokens, false, true, "Exceeds daily mint limit");
}
```

but in staking:

```
// In unstake()
if (totalRewards > 0) {
    uint256 remain = maxSupply - totalSupply();
    uint256 mintable = totalRewards <= remain ? totalRewards : remain;

    if (mintable > 0) {
        _mint(msg.sender, mintable);
        mintedRewards = mintable;
    }

    if (mintable < totalRewards) {
        emit RewardsCapped(msg.sender, totalRewards, mintable, totalRewards -
mintable);
    }
}
```

Notably:

unstake() does not:

- Call canMintTokens.
- Check dailyMintLimit.
- Update dailyMinted.

The only cap on staking rewards is maxSupply, not any per-day limit.

Recommendation:

If yes (global emission cap desired):

Add daily mint accounting inside unstake():

```
// Before minting in unstake()
if (dailyMinted + mintable > dailyMintLimit && block.timestamp < lastMintReset + 1
days) {
    // Either cap or revert; design-dependent
    revert("Exceeds daily mint limit from staking");
}

// Optionally reset daily if needed
if (block.timestamp >= lastMintReset + 1 days) {
    dailyMinted = 0;
    lastMintReset = block.timestamp;
}

dailyMinted += mintable;
_mint(msg.sender, mintable);
```

If no (deliberate design):

Clearly document that dailyMintLimit applies only to transferVFX mints and not to staking rewards.

Consider renaming variables to something like dailyPointsMintLimit to avoid operator/user confusion.

Fix: The behavior is intentional and now documented: daily limits govern off-chain point conversion only, while staking rewards are constrained by maxSupply. With clear communication in the transparency portal and docs, this is no longer a vulnerability but a documented design decision.

1. User Registration Mapping Inconsistencies (ID Reuse)

Issue:

registerUser maps a numeric userId to a wallet address and also tracks reverse mapping and a registeredUsers flag.

```
mapping(uint256 => address) public userIdentifiers;  // userId -> wallet
mapping(address => bool) public registeredUsers;     // wallet -> registered?
mapping(address => uint256) public userIds;          // wallet -> userId
```

The registerUser function:

```
function registerUser(uint256 userId, address userIdentifier)
    external
    onlyRole(MINTER_ROLE)
    whenNotPaused
{
    emit RegisterUserAttempt(userId, userIdentifier);
    require(!registeredUsers[userIdentifier], "User already registered");
    require(userIdentifier != address(0), "Invalid user identifier address");

    userIdentifiers[userId] = userIdentifier;
    userIds[userIdentifier] = userId;
    registeredUsers[userIdentifier] = true;

    emit UserRegistered(userId, userIdentifier);
}
```

The function does not check whether userId is already assigned to some other address.

A MINTER_ROLE operator can call registerUser again with the same userId but a different userIdentifier:

- userIdentifiers[userId] will be overwritten to the new address.
- registeredUsers[oldAddress] remains true.
- userIds[oldAddress] remains mapped to userId.

getUserIdByWalletAddress and getWalletAddressByUserId can then return inconsistent and conflicting information.

Security-wise, transferVFX checks:

```
require(registeredUsers[userIdentifiers[userId]], "User not registered");
```

So it only cares that userIdentifiers[userId] points to a registered address. The old address doesn't lose its

registeredUsers flag and old userIds mapping but is not used for new transfers by ID.

Recommendation: Disallow userId reuse or explicitly support changing the wallet for an existing userId with a dedicated function

INFORMATIONAL

No informational issues were found.

Technical Findings Summary

Findings

Vulnerability Level	Total	Pending	Not Apply	Acknowledged	Partially Fixed	Fixed
High	0					
Medium	3		1	1		1
Low	1					
Informational	0					

Assessment Results

Score Results

Review	Score
Global Score	**90/100**
Assure KYC	https://projects.assuredefi.com/project/vifoxcoin
Audit Score	90/100

The Following Score System Has been Added to this page to help understand the value of the audit, the maximum score is 100, however to attain that value the project must pass and provide all the data needed for the assessment. Our Passing Score has been changed to 84 Points for a higher standard, if a project does not attain 85% is an automatic failure. Read our notes and final assessment below. The Global Score is a combination of the evaluations obtained between having or not having KYC and the type of contract audited together with its manual audit.

Audit PASS

Following our comprehensive security audit of the token contract for the ViFoxCoin project, the project did meet the necessary criteria required to pass the security audit.

Disclaimer

Assure Defi has conducted an independent security assessment to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the reviewed code for the scope of this assessment. This report does not constitute agreement, acceptance, or advocating for the Project, and users relying on this report should not consider this as having any merit for financial ViFoxCoin in any shape, form, or nature. The contracts audited do not account for any economic developments that the Project in question may pursue, and the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude, and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are entirely free of exploits, bugs, vulnerabilities or deprecation of technologies.

All information provided in this report does not constitute financial or investment ViFoxCoin, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence, regardless of the findings presented. Information is provided ‘as is, and Assure Defi is under no covenant to audit completeness, accuracy, or solidity of the contracts. In no event will Assure Defi or its partners, employees, agents, or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions or actions with regards to the information provided in this audit report.

The assessment ViFoxCoins provided by Assure Defi are subject to dependencies and are under continuing development. You agree that your access or use, including but not limited to any ViFoxCoins, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies with high levels of technical risk and uncertainty. The assessment reports could include false positives, negatives, and unpredictable results. The ViFoxCoins may access, and depend upon, multiple layers of third parties.