verichains

SECURITY AUDIT OF

TOKEN VAULT

Public Report

Jan 13, 2025

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jan 13, 2025. We would like to thank the Tagger Team for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Token Vault. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified no vulnerable issue in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Token Vault

$TAG is the native currency that powers Tagger, generated fairly through the proof of work model by data workers' labor. $TAG can be used for posting data tasks, staking, purchasing datasets, utilizing datasets, subscribing to software services, AI model customization services, and more.

The vault is designed to be a secure location where $TAG token is held and protected before distributing to participants.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the Token Vault.

The audit was conducted on-chain with the following contract addresses:

NAME	ADDRESS
Token Vault	0x1b30abec4cfa2afffedeb7f10bdf19357365df21
Proxy Contract	0x95E7B14bAA3bC9123D045d324D47Ad76d166eBcb
Proxy's Admin	0x5d234d06c467d99c3e3402a57f8a15dc5d3a6c3b

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert

- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Tagger Team acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Tagger Team understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Tagger Team agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Tagger Team will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Tagger Team, the final report will be considered fully accepted by the Tagger Team without the signature.

2. AUDIT RESULT

2.1. Overview

Note: The `TokenVault` contract is **upgradeable**, so the contract owner can change the contract logic at any time in the future.

The vault is the location where `$TAG` is held. At block 45377985, this vault holds 3,000,000,000 `$TAG` (see the transaction). At the time of auditing, this vault holds 2,396,091,049.7 `$TAG` (after this transaction).

The vault owner holds the **UPGRADER_ROLE** - which allows them to upgrade the vault's logic in the future - and the **PAUSER_ROLE** - which enables them to pause the vault and prevent anyone from claiming tokens. The owner also has **ADMIN_ROLE**, which allow him to change the vault's operator - which worked as a signature verifier for every claim requests.

```
function initialize(
    address taggerToken_,
    address owner_,
    address operator_
)
    validAddress(taggerToken_)
    validAddress(owner_)
    validAddress(operator_)
{
    // ...
    _grantRole(PAUSER_ROLE, owner_);
    _grantRole(UPGRADER_ROLE, owner_);
    _grantRole(ADMIN_ROLE, owner_);

    taggerToken = IERC20(taggerToken_);
    operator = operator_;
    // ...
}
```

At the time of auditing, the owner address is `0x981C1Cf62b8F9542650Da0663f0135eB61ee2CDb` and the operator is set to `0xe6632F9F4368a17361600d8250e7d37b02Aec83c`. Both of them are traditional wallets (Externally Owned Accounts - EOAs). The address of `$TAG` token is `0x208bF3E7dA9639f1Eaefa2DE78c23396B0682025`.

To withdraw `$TAG` from the vault, users must provide the desired claim amount along with a valid, unexpired signature:

```
function claimTokens(
    uint256 amount,
    SignatureWithExpiry memory claimerSignatureAndExpiry
) external nonReentrant whenNotPaused {
```

```
    require(amount != 0, "Zero claimed amount");

    // check the signature expiry
    require(
        claimerSignatureAndExpiry.expiry >= block.timestamp,
        "Claimer signature expired"
    );

    uint256 tokenBalance = IERC20(taggerToken).balanceOf(address(this));
    require(
        tokenBalance != 0 && tokenBalance > amount,
        "Token vault contract is empty"
    );

    // calculate the digest hash, then increment `claimer`'s nonce
    uint256 currentClaimerNonce = claimerNonce[msg.sender];
    bytes32 claimerDigestHash = calculateClaimTokensDigestHash(
        operator,
        msg.sender,
        amount,
        currentClaimerNonce,
        claimerSignatureAndExpiry.expiry
    );

    unchecked {
        claimerNonce[msg.sender] = currentClaimerNonce + 1;
    }

    require(
        checkSignature(
            operator,
            claimerDigestHash,
            claimerSignatureAndExpiry.signature
        ),
        "Invalid operator signature"
    );

    IERC20(taggerToken).safeTransfer(msg.sender, amount);

    emit ClaimTokens(msg.sender, amount, currentClaimerNonce);
}
```

To prevent replay attacks, the vault maintains a unique nonce for each claimer, ensuring that each nonce is used only once.

The `checkSignature()` function utilizes the current vault's operator to validate claimer's signature. The signature is checked by ECDSA if the operator is an Externally Owned Account (as in the current scenario). Otherwise, it is verified through the IERC1271 interface.

```
function checkSignature(
    address signer,
    bytes32 digestHash,
    bytes memory signature
) internal view returns (bool result) {
    if (signer.code.length > 0) {
        result =
            IERC1271(signer).isValidSignature(digestHash, signature) ==
            EIP1271_MAGICVALUE;
    } else {
        address recovered = ECDSA.recover(digestHash, signature);
        result = recovered == signer ? true : false;
    }
}
```

2.2. Findings

During the audit process, the audit team found no vulnerability in the given version of Token Vault.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jan 3, 2025*	Public Report	Verichains Lab
1.1	*Jan 6, 2025*	Public Report	Verichains Lab
1.2	*Jan 13, 2025*	Public Report	Verichains Lab

Table 2. Report versions history