verichains

SECURITY AUDIT OF

TIOSWAP SMART CONTRACT

Public Report

Apr 18, 2023

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.
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Apr 18, 2023. We would like to thank the TioSwap 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 TioSwap Smart Contract. 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 found a vulnerability in the application. TioSwap team has resolved and updated the issue following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About TioSwap Smart Contract

What is TioSwap? Alice wants to sell NFTs to Paul, but they are not sure if they can trust each other, so they can't decide if Alice has to send the NFTs first or Paul has to send money first.

That's the problem Tioswap is trying to solve, to make sure NFTs and payment change hands successfully in an atomic transaction.

Tioswap provides a simple interface so sellers can sell their NFTs directly to buyers. The tool provides a safe way for both parties to execute the transaction. Think of it as a decentralized escrow service.

Tioswap acts as an intermediate step to make sure the transaction happens successfully. Tioswap uses TioSwap smartcontract to swap assets between users, and it's 100% trustless.

You can sell one single NFT or a bundle of many tokens from many ERC721, ERC1155 collections just in one go.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of TioSwap Smart Contract.

It was conducted on commit `57440bd5962996e161350c18a791ada30a54a305` from git repository link: *https://github.com/tioswap/tioswap*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`ea79bfbed284694ee10a22266f2e468819089699d14223089245600373aef38a`	`TioSwap.sol`

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

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The TioSwap Smart Contract was written in `Solidity` language, with the required version to be `>=0.8.0 <0.9.0`. The source code was written based on OpenZeppelin's library.

`Tioswap` contract is a marketplace where sellers can sell their tokens (`ERC721` and `ERC1155`) directly to buyers. Seller can sell one single NFT or a bundle of many tokens from many `ERC721`, `ERC1155` collections just in one go.

2.2. Findings

During the audit process, the audit team found a vulnerability in the given version of TioSwap Smart Contract.

TioSwap team fixed the code, according to Verichains's draft report in commit `9590409b624acfb6f0567cd8c95bfcf90f7f7f84`.

#	Issue	Severity	Status
1	Front running `finalizeTrade` function	CRITICAL	Fixed

2.2.1. Front running `finalizeTrade` function CRITICAL

By using the `createTrade` function to list a collection at a low price, attackers can wait for a user to purchase it using the `finalizeTrade` function. Meanwhile, attackers monitor pending transactions in the mempool and, upon a user's purchase, they submit a `changeTradePrice` transaction with a higher gas price (as higher gas price transactions are generally mined first) to increase the item's price. As a result, the user ends up buying the item at a higher price than they originally saw in the marketplace, despite the initial low price.

```
function changeTradePrice(bytes32 tradeId, uint newPrice)
    external
    isWorkingContract
    onlySeller(tradeId)
    validateTradePrice(newPrice)
    isListedTrade(tradeId)
{
    Trade storage trade = trades[tradeId];
    trade.price = newPrice;
    trade.fee = calculateFee(newPrice);
    emit LogTradePriceChanged(tradeId, trade.buyer, newPrice);
}

function finalizeTrade(bytes32 tradeId)
```

```
    external
    isWorkingContract
    isListedTrade(tradeId)
    nonReentrant
{
    ...
    // maximum discount is the calculated fee
    if (feeDiscount > trade.fee)
        feeDiscount = trade.fee;

    // calculate the actual fee after discount
    uint feeAfterDiscount = trade.fee - feeDiscount;

    // transfer amount of (WETH - (fee - discount)) to seller
    if (IERC20(PaymentTokenAddress).transferFrom(msg.sender, seller, (trade.price -
feeAfterDiscount)) == false)
        revert('Payment-Fail');
    ...
}
```

RECOMMENDATION

To prevent the purchase of an item at a price higher than the user intends to pay, a `price` parameter can be added to the `finalizeTrade` function. This would allow for the transaction to be reverted if the item's price exceeds the user's desired purchase price.

```
function finalizeTrade(bytes32 tradeId, uint price) // add price
    external
    isWorkingContract
    isListedTrade(tradeId)
    nonReentrant
{
    Trade storage trade = trades[tradeId];
    address seller = trade.seller;
    address buyer = trade.buyer;
    address referrer = userReferrer[seller];

    require(trade.price == price, 'Invalid price'); // FIX here

    ...
    // maximum discount is the calculated fee
    if (feeDiscount > trade.fee)
    feeDiscount = trade.fee;

    // calculate the actual fee after discount
    uint feeAfterDiscount = trade.fee - feeDiscount;

    // transfer amount of (WETH - (fee - discount)) to seller
    if (IERC20(PaymentTokenAddress).transferFrom(msg.sender, seller, (trade.price -
```


```
feeAfterDiscount)) == false)
    revert('Payment-Fail');
    ...
}
```

UPDATES

- *Apr 18, 2023*: This issue has been acknowledged and fixed by the TioSwap team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 18, 2023*	Public Report	Verichains Lab

Table 2. Report versions history