fuzzland

iotube Audit Report

Signature: 0c1ba9c5be8a9f07252dbbedac838f427b9ee50c9891987ce7abb24125e6867d

Iotube Audit Report

Executive Summary

On Feb 6, 2025, the iotubeproject team engaged Fuzzland to conduct a thorough security audit of their project. The primary objective was identifying and mitigating potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 10 person days, involving 2 engineers who reviewed the code over a span of 5 day. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Fuzzland team identified 9 issues across different severity levels and categories.

Scope

Project Name	ioTube
Repo	ioTube
Commit	`c4797cc243d741cf326222f622f4ce3e1aa1f7b9`
Fixed	`1f5854e5681982ad6c25610e5ba5e476e5e56a61`
Language	Solidity
Scope	contracts/*.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into three distinct levels: high, medium, low. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	0	0
Medium Severity Issues	0	0
Low Severity Issues	6	6
Info Severity Issues	3	3

Findings

[Low] Unrestricted Token Approval in `CrosschainTokenCashierWithPayloadRouter`

The `approveCrosschainToken` function in `CrosschainTokenCashierWithPayloadRouter` contract allows any caller to trigger token approvals for arbitrary addresses. The function approves unlimited amount (`type(uint256).max`) of tokens without any validation of the input parameters.

An attacker could:

- Call `approveCrosschainToken` with a malicious contract address
- The malicious contract could implement `coToken()` to return any token address
- This would result in the router approving the attacker's contract to spend any amount of the specified token
- An attacker can execute a malicious `approve()` function through the `call()` in `safeApprove()`

```
function _approve(address _ctoken, address _coToken) private {
    require(safeApprove(_coToken, _ctoken, type(uint256).max), "failed to
approve allowance to crosschain token");
}

function approveCrosschainToken(address _crosschainToken) public {
    _approve(_crosschainToken, ICrosschainToken(_crosschainToken).coToken());
}

function safeApprove(address _token, address _spender, uint256 _amount)
internal returns (bool) {
    // selector = bytes4(keccak256(bytes('approve(address,uint256)')))
    (bool success, bytes memory data) =
_token.call(abi.encodeWithSelector(0x095ea7b3, _spender, _amount));
    return success && (data.length == 0 || abi.decode(data, (bool)));
}
```

Recommendation:

- Implement a whitelist mechanism for valid crosschain tokens
- Add access control to the approval function

Project Response:

This router is a help contract, with no permissions or assets.

[Low] Unauthorized token withdrawal

The transfer function in `UniswapUnwrapper` contract is marked as `external` without any access control. Any external user can call this function to transfer any ERC20 tokens held by the contract to an arbitrary address.

This vulnerability allows attackers to:

- Withdraw any ERC20 tokens held by the contract
- Disrupt normal swap operations
- Cause loss of user funds

```
// contracts/UniswapUnwrapper.sol
function transfer(IERC20 _token, address _to, uint256 _amount) external {
    require(_token.transfer(_to, _amount), "UniswapUnwrapper: transfer
failed");
    emit Swap(address(_token), address(0), _amount, 0, _to);
}
```

Recommendation:

Change the transfer function to internal to restrict access to contract internals only:

```
function transfer(IERC20 _token, address _to, uint256 _amount) internal {
    require(_token.transfer(_to, _amount), "UniswapUnwrapper: transfer
failed");
    emit Swap(address(_token), address(0), _amount, 0, _to);
}
```

Project Response:

Any assets transferred to this unwrapper will be withdrawn immediately in the same transaction, so there should be no remaining assets in it.

[Low] All functions of the `ShadowTokenListManager` contract cannot be called

All functions of the `ShadowTokenListManager` call functions of the `tokenList` with the `onlyowner` modifier, but since the `ShadowTokenListManager` is not the owner of the `tokenList` contract passed in its constructor, all functions of the `ShadowTokenListManager` cannot be called.

```
// contracts/iotube/ShadowTokenListManager.sol contract ShadowTokenListManager
is Ownable { TokenList public tokenList; constructor(address _addr) Ownable()
public { tokenList = TokenList(_addr); }
```

Recommendation:

Create `tokenList` in the `ShadowTokenListManager` constructor instead of directly passing in the tokenList contract address.

Project Response:

This contract is not in use.

[Low] `TetherToken::transfer` has no return value, which causes `UniswapUnwrapper::transfer` to revert

`TetherToken::transfer` has no return value, but `UniswapUnwrapper::transfer` checks the return value of the token transfer, which causes users to revert when transferring TetherToken through `UniswapUnwrapper::transfer` because of the sentence `require(_token.transfer(_to, _amount), "UniswapUnwrapper: transfer failed")` .

```
// contracts/token/TetherToken.sol function transfer(address _to, uint _value)
public whenNotPaused { require(!isBlackListed[msg.sender]); if (deprecated) {
return UpgradedNonStandardToken(upgradedAddress).transferByLegacy(msg.sender,
_to, _value); } else { return super.transfer(_to, _value); } }
```

```
// contracts/UniswapUnwrapper.sol function transfer(IERC20 _token, address _t
o, uint256 _amount) external { require(_token.transfer(_to, _amount), "Uniswap
Unwrapper: transfer failed");//@audit emit Swap(address(_token), address(0), _
amount, 0, _to); }
```

Recommendation:
Stop checking the `transfer` return value directly and use `safeTransfer/safeTransferFrom` instead.

Status:

Fixed

[Low] Any user can call `UniswapUnwrapper::onReceive` to steal any amount

Since `UniswapUnwrapper::onReceive` is external and has no permission restrictions, any user can call `UniswapUnwrapper::onReceive` to steal any amount by simply setting the parameter in `_payload` to `swapData.deadline < block.timestamp`.

- POC
- Output

`SolanaHub::onReceive` and `EthereumHub::onReceive` also have the same problem.

Recommendation:

`UniswapUnwrapper::onReceive` , `SolanaHub::onReceive` and `EthereumHub::onReceive` should have permission restrictions.

Project Response:

Any assets transferred to this unwrapper will be withdrawn immediately in the same transaction. So, there should be no remaining assets in it.

[Low] Use `increaseAllowance` / `decreaseAllowance` instead of `approve`

Many contracts in the project use `approve` instead of `increaseAllowance` / `decreaseAllowance` .Changing an allowance with `approve` carries the risk that an attacker may front-run the transaction and use both the old and the new allowance. It is recommended to use `increaseAllowance` or `decreaseAllowance` to avoid this issue.

1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM
https://docs.google.com

The specific scenario is as follows:

1. Owner approves 10 ether to spender
2. Owner is going to change the 10 ether approved to spender to 1 ether
3. Spender transfers the 10 ether approved by owner to his own account before the transaction is executed
4. Owner's authorization (change 10 ether to 1 ether) transaction is executed, spender can transfer 1 ether to his own account
5. At this time, owner only wants to approve 1 ether to spender, but spender finally gets 10 ether + 1 ether, which exceeds the owner's authorization amount

Recommendation:

Use `increaseAllowance/decreaseAllowance` instead of `approve` .

Status:

Acknowledged.

[Info] Many contracts did not correctly inherit the `ownable` of openzeppelin

`TokenConfigList`, `CashierConfig`, `EthereumHubPrepaid`, `SolanaHubPrepaid` and many others inherit openzeppelin's `ownable`, but have no constructor or do not pass in parameters in the constructor.

```
// contracts/CashierConfig.sol import "@openzeppelin/contracts/access/Ownable.
sol"; //... contract CashierConfig is Ownable { //... constructor( string memo
ry _id, bool _withPayload, uint256 _startBlockHeight, address _cashier, addres
s _validator, address _tokenSafe, address _tokenList ) { id = _id; withPayload
= _withPayload; startBlockHeight = _startBlockHeight; cashier = _cashier; vali
dator = _validator; tokenSafe = _tokenSafe; tokenList = _tokenList; } }
```

Recommendation:

Pass a parameter as owner in the constructor, or inherit from `contracts/ownership/Ownable.sol` in the project instead.

Status:

Fixed

[Info] `TetherToken::deprecate` should only be called once

`TetherToken::deprecate` can set `_upgradedAddress`, but there is no limit on the number of calls. Suppose the first call sets address A, and then the second call sets address B. At this time, if the contract is upgraded to address B after the user interacts with address A, the funds in address A may not be processed correctly.
`Deprecate` should be limited to one call in the function to avoid confusion in fund management caused by address changes.

```
// contracts/token/TetherToken.sol
function deprecate(address _upgradedAddress) public onlyOwner {
    deprecated = true;
    upgradedAddress = _upgradedAddress;
    emit Deprecate(_upgradedAddress);
}
```

Recommendation:

`TetherToken::deprecate` should only be called once.

Status:

Fixed

[Info] Centralization Risk

Contract single node account permission owner, can directly withdraw contract funds.

```
    function withdraw() external onlyOwner {
        msg.sender.transfer(address(this).balance);
    }

    function withdrawToken(address _token) public onlyOwner {
        // selector = bytes4(keccak256(bytes('balanceOf(address)')))
        (bool success, bytes memory balance) =
_token.call(abi.encodeWithSelector(0x70a08231, address(this)));
        require(success, "failed to call balanceOf");
        uint256 bal = abi.decode(balance, (uint256));
        if (bal > 0) {
            require(safeTransfer(_token, msg.sender, bal), "failed to withdraw
token");
        }
    }
```

Recommendation:

Consider implementing multi-signature mechanism for critical operations.

Status:

Fixed