OdinTrade

Security Audit Report

May 24, 2025

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Code and architecture review:

- Review the source code.
- Read the project's documentation.
- Review the project’s architecture.

Stage goals

- Build a good understanding of how the application works.

1.2.2 Check code against vulnerability checklist:

- Understand the project’s critical assets, resources, and security requirements.
- Manual check for vulnerabilities based on the auditor's checklist.
- Run static analyzers.

Stage goals

- Identify and eliminate typical vulnerabilities (gas limit, replay, flash-loans attack, etc.)

1.2.3 Threat & Attack Simulation:

- Analyze the project's critical assets, resources and security requirements.
- Exploit the found weaknesses in a safe local environment.
- Document the performed work.

Stage goals

- Identify vulnerabilities that are not listed in static analyzers that would likely be exploited by hackers.
- Develop Proof of Concept (PoC).

1.2.4 Report found vulnerabilities:

- Discuss the found issues with developers
- Show remediations.
- Write an initial audit report.

Stage goals

- Verify that all found issues are relevant.

1.2.5 Fix bugs & re-audit code:

- Help developers fix bugs.
- Re-audit the updated code again.

Stage goals

- Double-check that the found vulnerabilities or bugs were fixed and were fixed correctly.

1.2.4. Deliver final report:

- The Client deploys the re-audited version of the code

- The final report is issued for the Client.

Stage goals

- Provide the Client with the final report that serves as security proof.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

OdinTrade is a trustless cross-chain settlement protocol that leverages an extended Hashed Time-Locked Contract (HTLC) design to enable secure, decentralized atomic swaps between Bitcoin and EVM-compatible blockchains. By utilizing pre-signed claim transactions and a dual-timelock mechanism, OdinTrade allows users to swap assets across chains without intermediaries while maintaining claimability. The protocol sources liquidity from centralized exchanges to provide competitive rates and aggregates it in a decentralized manner, making atomic swaps more efficient for both end-users and integrators.

1.4 Project Dashboard

Project Summary

Title	Odin Trade Security Audit
Client name	OdinX
Project name	Odin Trade
Timeline	7.05.2025 - 21.05.2025
Number of Auditors	3

Project Last Log

Date	Commit Hash	Note
23.04.2025	bac81164541c62c7bf0d30978f51e0763f3de3a9	fixed secret creation reducing amountOut decimals

Project Scope

The audit covered the following smart contract files.

File name
packages/evm-contracts/OdinTradeEvm.sol
sdk

1.5 Summary of findings

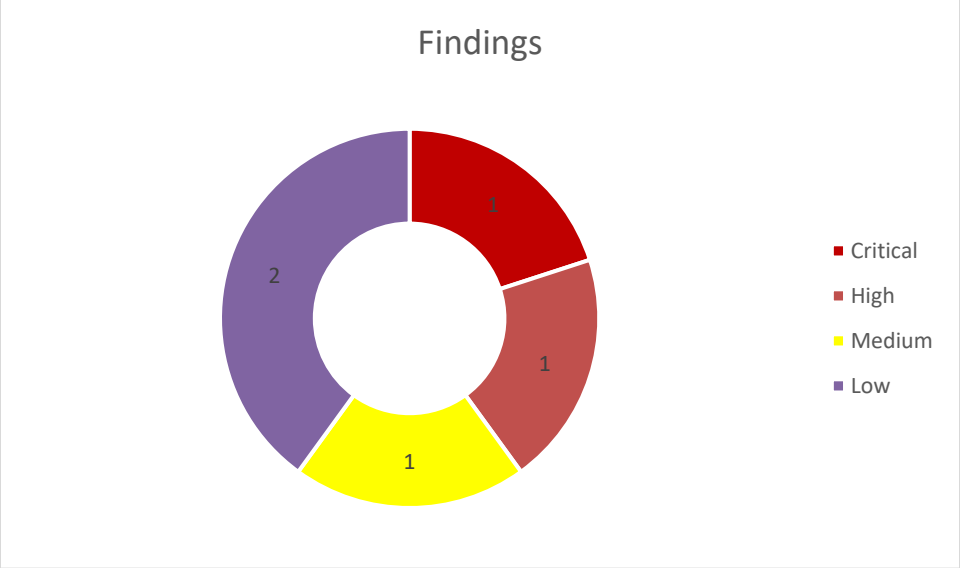


Figure 1. Findings chart

Severity	# of Findings
CRITICAL	1
HIGH	1
MEDIUM	1
LOW	2

2. FINDINGS REPORT

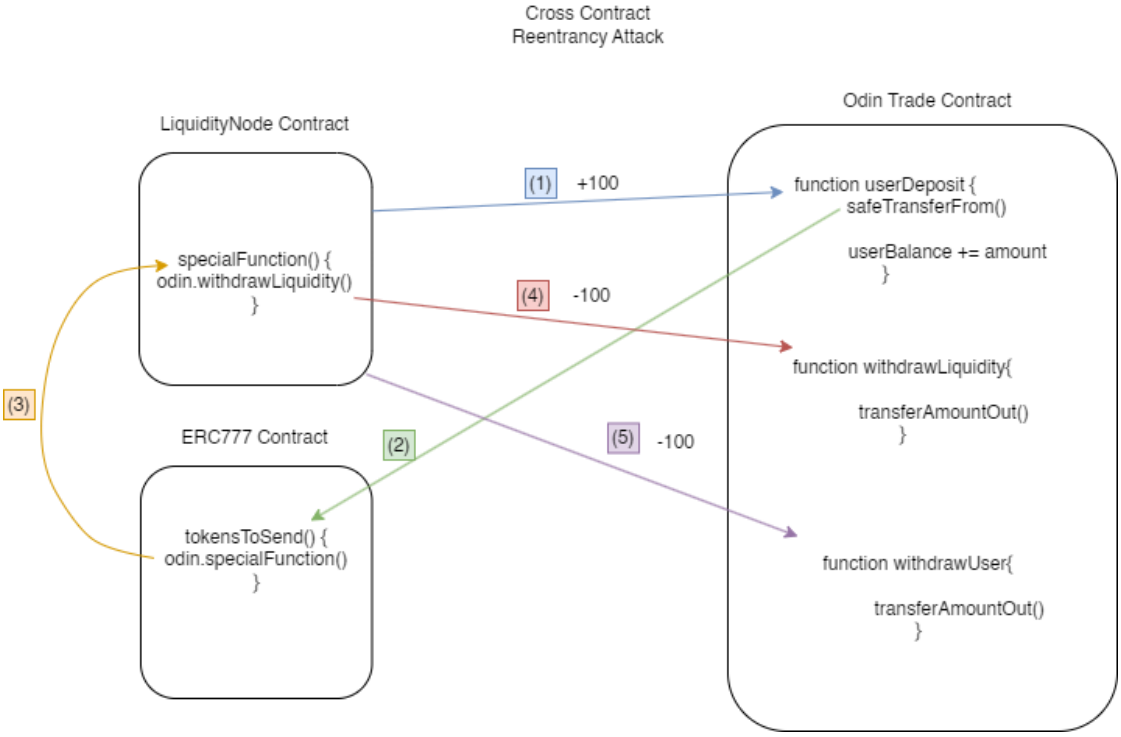
2.1 Critical

2.1.1 Double spending via ERC777 Reentrancy in userDeposit

Status: Fixed

Description

The OdinTrade contract is vulnerable to a double-spending attack by the Liquidity Node. Although the Liquidity Node is a relatively trusted actor, the protocol is defined as trustless. Due to functions vulnerable to cross function reentrancy and the fact that LN can deposit tokens as user, it is possible to withdraw funds 2 times instead of 1, draining user's sent funds. This issue is different from the common one, where scam tokens are used, because the ERC777 token's malicious hooks can be difficult to detect, while its behavior can look benign. function and missing reentrancy protections on withdrawLiquidity and userWithdraw.



Step-by-Step Exploit

Actors:

- Liquidity Node (LN): The attacker, who controls an ERC777 token with a malicious tokensToSend hook that is triggered by safeTransferFrom. LN controls both its own contract and the ERC777 contract.

1. LN initiates a deposit:

- Calls userDeposit(token, 100).
- The contract receives 100 tokens. However, mappings (userBalance and usersDepositedBalance) are not yet updated.

2. The ERC777 contract's tokensToSend hook is triggered *before* the mappings are updated

3. The ERC777's tokensToSend hook calls the LN's dedicated function

4. Inside this dedicated function, the LN contract immediately calls withdrawLiquidity(token, 100).
Bypassing the onlyLiquidityNode check (only msg.sender is checked, not tx.origin).

- The withdrawLiquidity calls _availableLiquidity(token):
    - contractBalance = IERC20(token).balanceOf(address(this)) → 100
    - usersDepositedBalance[token] = 0 (not updated yet)
    - usersLockedBalance[token] = 0
    - liquidityNodeLockedBalance[token] = 0
- Result: available liquidity appears as 100, so withdrawal is allowed.
- LN receives 100 tokens back (their deposit, withdrawn prematurely).
- The cross contract call is finished, returning to the original userDeposit function
- userBalance[LN][token] += 100;
- usersDepositedBalance[token] += 100;

5. LN calls userWithdraw:

- LN has a user balance of 100 tokens according to the contract state.
- Calls userWithdraw(token, 100).
- Succeeds, transferring another 100 tokens (which no longer exist in the contract).

Net Effect:

- LN deposited 100, but withdrew 200 (100 during reentrancy + 100 as a user).
- Contract is drained by 100 tokens.

Remediation:

Use ReentrancyGuard (from OpenZeppelin) to prevent nested/external reentrant calls in functions that transfer funds or interact externally. This is relevant for the following functions as well: withdrawLiquidity, userWithdraw, executeSwap, withdrawSwap, userDeposit.

2.2 High

2.2.1 Reentrancy possible in withdraw and execute functions.

Status: Fixed

Description

The withdrawSwap and executeSwap are also vulnerable to cross function reentrancy. They update the HTLC status at the end, after external transfer. They are also not protected by a reentrancy guard. Although draining funds in this case is *not* possible, the withdrawSwap can be called after executeSwap because during the reentrancy call executeSwap->withdrawSwap the HTLC status is not yet changed.

```
function executeSwap(…) external checkHtlcStatus(…) {


    ...
    _decreaseLockedBalance(isUserHtlc_, token, amount);

    ...
    [then transfer]
    _transferOutAmount(token, receiver_, amount); // This is where cross-function reentrancy is pos-
sible

    ...
    [then finally:]
    htlc_.status = HTLC_STATUS.EXECUTED or WITHDRAWN;
}
```

```
function withdrawSwap(
    bytes32 _swapId
) external checkHtlcStatus(_swapId, HTLC_STATUS.CREATED) {

... // reentrancy here
}
```

```
contract AttackerReceiver {
    OdinTrade public trade;
    address public token;

    constructor(address _trade, address _token) {
        trade = OdinTrade(_trade);
        token = _token;
    }

    // This hook gets called during the ERC777 transfer or receive() during ETH transfer
    function tokensReceived( // or receive() for Eth
        address /*operator*/,
        address /*from*/,
        address /*to*/,
        uint256 /*amount*/,
        bytes calldata /*userData*/,
```

```
        bytes calldata /*operatorData*/
    ) external {
        // Reenter the original contract
        trade.withdrawLiquidity(token, 100); // Or some other function
    }

    function attack() external {
        // Initiate first withdrawal to start the process
        trade.withdrawLiquidity(token, 100);
    }
}
```

Remediation:

Please use the OpenZeppeling nonReentrant guard for the following functions:
- withdrawLiquidity

- userWithdraw

- executeSwap

- withdrawSwap

- userDeposit

2.3 Medium

2.3.1 Signature Replay

Status: Acknowledged

Description

If the createUserHtlc (OdinTradeEvm.sol) fails, the nonce is not incremented on chain and the signature becomes available to liquidity node to submit later, although the initial swap failed. This may be the protocol intended design, but we think it must be documented in the protocol, that when a user signs a swap, he/she commits to the swap to be executed even after deadline or failure.

Remediation:

Make signature being able to expire or document this intended design.

2.4 Low

2.4.1 Nested Conditional Complexity

Status: Acknowledged

Description

Complex if-else structures in *executeSwap* and *withdrawSwap* functions reduce code readability and make maintenance error-prone. Although not a security issue now, refactoring to clear, modular helper functions (with descriptive names) is recommended.

```
if (
    (!isUserHtlc_ && !usersAutoDeposit[receiver_]) ||
    (isUserHtlc_ && !liquidityNodeAutoDeposit)
) {
    _transferOutAmount(token, receiver_, amount);
} else {
    if (!isUserHtlc_) {
        userBalance[receiver_][token] += amount;
        usersDepositedBalance[token] += amount;
    }
}
```

One of the cases is unhandled on purpose:

```
isUserHtls_ == true && liquidityNodeAutoDeposit == true
```

We recommend handling this case explicitly to reduce the risk of errors in the future.

The same issue for the withdrawSwap function.

Remediation

Add modular functions to handle these complex conditions.

```
// executeSwap()
bool isUserHtlc_= htlc_.isUserHtlc;

if (isUserHtlc) {
  handleUserHtlcTransfer(liquidityNodeAutoDeposit, htlc_.receiver)
} else { // LN HTLC
  handleLiquidityNodeHtlcTransfer(usersAutoDeposit[receiver_], htlc_.receiver)
}


function handleUserHtlcTransfer(
    bool liquidityNodeAutoDeposit,
    address liquidityNodeAddress,
    address token,
    uint256 amount
) internal {
```

```
    if (liquidityNodeAutoDeposit) {
        // Funds are kept in the contract for the liquidity node. No action needed.
        return;
    } else {
        _transferOutAmount(token, liquidityNodeAddress, amount);
    }
}


function handleLiquidityNodeHtlcTransfer(
    bool userAutoDeposit,
    address userAddress,
    address token,
    uint256 amount
) internal {
    if (userAutoDeposit) {
        // Funds are kept in the contract, update user balance mappings.
        userBalance[userAddress][token] += amount;
        usersDepositedBalance[token] += amount;
    } else {
        _transferOutAmount(token, userAddress, amount);
    }
}
```

2.4.2 No Event for Liquidity Withdrawals

Status: Acknowledged

Description

Functions like userWithdraw and withdrawLiquidity do not emit events for successful withdrawals. This reduces transparency and makes it harder to track withdrawals off-chain.

Remediation

Emit an event when liquidity is withdrawn.

CONCLUSION

This security audit found the OdinTrade protocol to be well-architected. The identified serious issues—most notably double-spending via ERC777 reentrancy and cross-function reentrancy risks—have been fixed after the audit, through the adoption of standard protections like reentrancy guards. Remaining items, such as signature replay design and event emission for withdrawals, are acknowledged but do not pose an immediate risk to user funds or protocol

integrity. Overall, OdinTrade demonstrates security, responsive development, and a commitment to best practices, providing users and liquidity providers with a robust and reliable cross-chain swap solution.

REFERENCES

[1] https://eips.ethereum.org/EIPS/eip-777

[2] https://solidity-by-example.org/hacks/re-entrancy/

[3] https://docs.soliditylang.org/en/v0.8.21/security-considerations.html#use-the-checks-effects-interactions-pattern