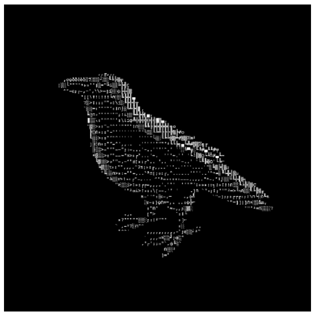

EscrowSwap Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

Escrow swap is a platform, which allows users to create and adjust otc trades. All open trades are listed on their website for the users to navigate easily.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	EscrowSwap
Repository	smartcontract.escrowswap
Commit	04ff3abfe646...
Date	June 2023

Issues Found

Severity	**Count**
Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	1
Informational	0
Gas Optimizations	0
Total Issues	**4**

Summary of Findings

Title	**Status**
[H-1] Unhandled Overpayment in Native Token Transfers	Resolved
[M-1] Hardcoded WETH Address Leads to Cross-chain Incompatibility	Resolved
[M-2] Unchecked ether transfer during ERC20 transfers in $_handleIncomingTransfer$	Resolved
[L-1] Outgoing transfers are vulnerable to returnbombs	Resolved

5 Findings

5.1 High Risk

[H-1] Unhandled Overpayment in Native Token Transfers

Context: EscrowSwap.sol

Impact: Users who accidentally send more native tokens than required will not be refunded the excess amount. This could lead to financial losses for users and potentially lead to a loss of trust in the system.

Description: In the `_handleIncomingTransfer` function, if the `_token` address is `0x0`, indicating that native tokens are being sent, the function requires that the `msg.value` is greater than or equal to the `_amount` specified.

```
function _handleIncomingTransfer(address _sender, uint256 _amount, address _token, address _dest) private {
    if (_token == address(0)) {
        require(msg.value >= _amount, "_handleIncomingTransfer msg value less than expected amount");
    } else {
        // We must check the balance that was actually transferred to this contract,
        // as some tokens impose a transfer fee and would not actually transfer the
        // full amount to the escrowswap, resulting in potentially locked funds
        IERC20 token = IERC20(_token);
        uint256 beforeBalance = token.balanceOf(_dest);
        token.safeTransferFrom(_sender, _dest, _amount);
        uint256 afterBalance = token.balanceOf(_dest);
        require(beforeBalance + _amount == afterBalance, "_handleIncomingTransfer token transfer call did not
        ↪    transfer expected amount");
    }
}
```

However, if the user inadvertently sends more than the required `_amount`, the excess is not refunded or accounted for.

```

    TradeOffer memory newOffer = TradeOffer({
        seller: msg.sender,
        tokenOffered: _tokenOffered,
        tokenRequested: _tokenRequested,
>         amountOffered: _amountOffered,
        amountRequested: _amountRequested
    });
```

The function simply accepts the excess amount without notifying the user or taking any further action. However, given the fact that its mostly expected for users to pass `msg.value` that equals `tokensOffered` this is realistic, yet not that likely scenario.

However, the same vulnerability exists when accepting a trade offer.

```
function acceptTradeOffer(uint256 _id, address _tokenRequested, uint256 _amountRequested)
    payable
    external
    nonReentrant
    nonEmergencyCall
    {
        TradeOffer memory trade = tradeOffers[_id];

        if (trade.tokenRequested != _tokenRequested) { revert MisalignedTradeData(); }
        if (trade.amountRequested != _amountRequested) { revert MisalignedTradeData(); }
        if (trade.amountOffered == 0) { revert EmptyTrade(); }

        _deleteTradeOffer(_id);
        emit TradeOfferAccepted(_id, msg.sender);

        //Transfer from buyer to seller.
        _handleRelayTransfer(
            msg.sender,
            trade.amountRequested,
            trade.tokenRequested,
            address(trade.seller)
        );

        //Fee Payment calculation and exec.
        _handleFeePayout(
            msg.sender,
            trade.amountRequested,
            trade.tokenRequested,
            trade.tokenOffered
        );

        //Transfer from the vault to buyer.
        _handleOutgoingTransfer(msg.sender, trade.amountOffered, trade.tokenOffered);
    }
```

Given it's not really practical for the callee to calculate the **exact** amount to be processed beforehand, it is **very** likely that he or she will just send slightly more so the transaction goes through.

However, if that happens the extra money that the callee sent would be lost, because the `acceptTradeOffer` function doesn't account for this scenario.

Recommendation: To mitigate this vulnerability, both functions should be modified to refund any excess native tokens sent by the user. This can be accomplished by sending back the difference between `msg.value` and `_amount` to the sender.

5.2 Medium Risk

[M-1] Hardcoded WETH Address Leads to Cross-chain Incompatibility

Context: EscrowSwap.sol

Impact: This vulnerability potentially restricts the system's functionality to a single chain, Ethereum, due to the hardcoded Wrapped Ether (WETH) address.

Therefore, the contract cannot be used on other blockchains like Polygon, leading to significant limitations in cross-chain interoperability.

Description: In the provided Solidity code snippet, the WETH address is hardcoded to the address `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2`. This means that the system is implicitly tied to the Ethereum network where this WETH contract resides.

https://github.com/escrowswap/escrowswap-sol-only/blob/cc2a1577c5212a146f21ed3f03f6ffa5d778aad1/escrowswap_v1.0.sol#L59-L60

```
        weth = IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
        // @audit won't work for multiple chains
```

Cross-chain functionality is crucial for many decentralized applications (dApps) to tap into various liquidity sources or to leverage different advantages of different chains. However, due to this hardcoded WETH address, the system will not function correctly on other chains. For example, on the Polygon network, this hardcoded address will not refer to a WETH contract, and consequently, any functionality reliant on WETH will fail.

Recommendation: It is highly recommended to remove the hardcoded WETH address and replace it with an immutable variable contract that is passed as an argument in the constructor during contract deployment.

This approach will allow for different WETH addresses to be used, depending on the chain where the contract is being deployed, ensuring cross-chain compatibility.

[M-2] Unchecked ether transfer during ERC20 transfers in _handleIncomingTransfer

Context: EscrowSwap.sol

Impact: This vulnerability could have a high impact on users as it can potentially lead to loss of funds. Users might accidentally send additional ether when interacting with the contract, particularly when transferring ERC20 tokens. As the contract currently stands, the unprocessed ether gets trapped in the contract without a way to retrieve it.

Description: The `_handleIncomingTransfer` function is responsible for processing incoming transfers within the trade offer creation process. It distinguishes between Ethereum (0x address) and ERC20 token transfers.

```
    function _handleIncomingTransfer(address _sender, uint256 _amount, address _token, address _dest) private {
        if (_token == address(0)) {
            require(msg.value >= _amount, "_handleIncomingTransfer msg value less than expected amount");

        } else {
            // We must check the balance that was actually transferred to this contract,
            // as some tokens impose a transfer fee and would not actually transfer the
            // full amount to the escrowswap, resulting in potentially locked funds

            IERC20 token = IERC20(_token);
            uint256 beforeBalance = token.balanceOf(_dest);
            token.safeTransferFrom(_sender, _dest, _amount);
            uint256 afterBalance = token.balanceOf(_dest);
            require(beforeBalance + _amount == afterBalance, "_handleIncomingTransfer token transfer call did not
            ↪   transfer expected amount");
        }
    }
```

In the case of Ethereum transfers, the function ensures that the message value (`msg.value`) is greater than or equal to the expected amount.

For ERC20 token transfers, the function transfers the tokens from the sender to the destination address and checks if the transferred amount is equal to the expected amount.

However, the function lacks safeguards to prevent accidental loss of funds. When the function is called to process an ERC20 token transfer, it does not verify whether extra ether was also sent in the transaction (`msg.value > 0`). As a result, if a user inadvertently includes additional ether in the transaction, these funds will become permanently stuck in the contract.

Recommendation: To protect users against accidental loss of funds, the `_handleIncomingTransfer` function should include a check to confirm that `msg.value` equals zero when handling ERC20 token transfers. If `msg.value > 0`, the contract should revert the transaction and emit an appropriate error message. This modification will prevent users from accidentally sending extra ether in these transactions.

Here's how you can add this safeguard:

```
else {
    require(msg.value == 0, "_handleIncomingTransfer: Unexpected Ether transfer");
    // The rest of your code
}
```

With this check in place, any transactions that mistakenly include ether alongside an ERC20 token transfer will be automatically reverted, protecting users from unintentional loss of funds.

5.3 Low Risk

[L-1] Outgoing transfers are vulnerable to returnbombs

Context: EscrowSwap.sol

Impact: A successful exploitation of this vulnerability could allow an attacker to perform a Denial of Service (DoS) attack against the EscrowSwap callers. By forcing the contract to exhaust all its gas through a 'returnbomb'.

Description

The vulnerability arises from the use of a low-level call in the handling of ETH transfers in EscrowSwap:

```
(bool success, ) = _dest.call{value: _amount, gas: gas}("");`
```

A standard Solidity call such as the one used above automatically copies bytes to memory without considering gas costs. This means that the call will copy any amount of bytes to local memory. When bytes are copied from returndata to memory, the memory expansion cost is paid.

The issue lies in the fact that this gas is paid by the caller and in the caller's context. Thus, the callee can impose an arbitrary gas cost on the caller by 'returnbombing', causing the caller to exhaust its gas and halt execution.

Recommendation

To prevent the 'returnbombing' issue, it is recommended to utilize `ExcessivelySafeCall` available at:

https://github.com/nomad-xyz/ExcessivelySafeCall

As an alternative, you could adopt Chainlink's `_callWithExactGas` approach which has implemented gas management in a more controlled way:

https://github.com/code-423n4/2023-05-chainlink/blob/1d84bf6f19f3200eea6dbaf99e378ea8a772d8c6/contracts/Router.sol#L190-L233

Both of these methods provide improved control over the gas consumption during contract execution, preventing the potential for a DoS attack via the 'returnbomb' mechanism.