BAIL security

BAILSEC.IO

EMAIL : OFFICE@BAILSEC.IO

TWITTER : @BAILSECURITY

TELEGRAM : @HELLOATBAILSEC

FINAL REPORT:

Trustswap MultiSender

December 2023

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Project	Trustswap
Website	trustswap.com
Type	MultiSender
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/trustswap/multi-sender-contracts/blob/46e92fe7964ab3b507ef90c4009488b4745b58ee/src/MultiSender.sol
Resolution	https://github.com/trustswap/multi-sender-contracts/blob/2045ed247e2f83757d075d3ffe7ce75d79ed0a1a/src/MultiSender.sol
Resolution 2	https://github.com/trustswap/teamfinance-contract-multisender/blob/f8da91856cde123a9e0607f7e7792071fb916915/src/MultiSender.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	3	3		
Medium	3	3		
Low	2		1	1
Informational	1	1		
Configurational	0			
Governance	0			
Quality assurance	0			
Total	9	7	1	1

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Configurational	Issues which may arise due to different configurational settings
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior
Quality assurance	Aggregated minor issues, ensuring a high quality codebase.

3. Detection

MultiSender

The Multisender contract is a versatile smart contract designed to simplify the distribution of ERC20, ERC721, ERC1155 tokens, and Ether to multiple recipients from a single source, respectively the msg.sender.

This contract offers a range of essential features and customizable parameters to enhance its usability, it allows its users to send various tokens, including ERC20, ERC721, ERC1155, and Ether, to multiple recipients in the same transaction. This streamlined approach to token distribution significantly improves efficiency across a wide array of use cases.

Additionally, the contract provides the flexibility of applying optional fees for token transfers. These fees are initially denominated in USD and will be the same nominal value for each transaction during the distribution process, it is expected that the fee is provided as msg.value.

It's important to note that certain wallets and tokens may be exempt from these fees, enhancing user flexibility.

To convert the fee amounts from USD to ETH, the contract utilizes a UNIV2 pool for the USD to ETH conversion.

The contract owner enjoys extensive control over various critical parameters, including the selection of the UniV2Router and UniV2Pair contracts, the destination wallet for collected fees (companyWallet), the initial fee amount in USD (feesInUSD), whitelisted wallets exempt from fees, tokens excluded from fees, transfer limits for multi-send transactions, and whitelisted admins with authority over addresses and token exclusions.

Furthermore, the contract owner has the capability to reclaim any ERC20, ERC721, ERC1155, or Ether residing within the contract.

Resolution:

During the resolution, the business logic has been shifted to use a reliable oracle instead of the UniswapV2 pool.

Issue	multisendETH will not work with corresponding fee logic
Severity	High

Description	The multisendETH function facilitates the batch transfer of ETH to various different recipient addresses. The crux with this function is that the fee is taken in the same denomination and relies on the msg.value, similar to the function itself. This will have multiple impacts which make this function flawed:
	
	1. Scenario of msg.value < feeInEth will transfer the full msg.value out but the function still distributes any ether balance in the contract to recipients. This will either transfer unauthorized funds out or revert.
	
	2. The refund scenario allows users to circumvent the fee completely:
	
	`uint256 remaining = msg.value - totalSent;`
	`if (remaining > 0) {`
	`safeTransferETH(msg.sender, remaining);`
	`}`
	
	Whenever this function is called, the fee is taken directly from the msg.value. This means that the msg.value is expected to cover the fee AND all transfers, resulting in the msg.value forced to be larger than the totalSent variable, to cover the fee. Once the fee is withdrawn to the companyWallet, no refund is happening for the multisendETH scenario:
	
	`if (!isEthTransfer && msg.value > feeInEth) {`
	`safeTransferETH(msg.sender, msg.value - feeInEth);`
	`}`
	
	which is great, since the leftover amount is in fact meant to be distributed towards the recipients. However, at the end of the function, the leftover amount is then distributed back to the user:

```
        uint256 remaining = msg.value - totalSent;
        if (remaining > 0) {
        safeTransferETH(msg.sender, remaining);
        }
```

This will result in a revert if the contract has insufficient tokens or will drain the contract and refund the fee to the user (if sufficient funds reside in the contract).
It is of course also possible to provide a msg.value which is not larger than the sum of all amounts, this will result in the same issue, since the loop still attempts to transfer the all amounts out, even if insufficient msg.value is existent due to the fee deduction.

This issue will also result in the companyWallet being able to drain all ETH in the contract with the following two PoCs:

PoC 1:

1. companyWallet calls

multisendETH([companyWallet], [0.99e18)

current fee = 1e18;

msg.value = 0.99e18

2. handleFees transfers the full msg.value to the company wallet since the msg.value is below the fee amount and within the 95% range:

```
        if (msg.value < feeInEth) {
        // allow 5% less fee due to price change
        if (((feeInEth - msg.value) * 100) / feeInEth > 5) revert
FeeNotMet(feeInEth, msg.value);

        safeTransferETH(companyWallet, msg.value);

        return msg.value;
```

	} 3. the multisendETH function will continue as usual, transferring 1e18 ETH to the company wallet (expected the contract has any funds inside) This attack can be arbitrarily executed to drain all ETH in the contract. **PoC 2:** 1. companyWallet calls multisendETH([2e18], [companyWallet]) msg.value = 2e18 current fee = 2e18 2. Fee is being transferred out as expected 3. The multisendETH function will continue to transfer out 2 ETH to the company wallet as part of the standard distribution (expected the contract holds sufficient funds to be drained).
Recommendations	Whenever functions handle native token logic, one needs to be very careful. This function exposes different issues which need to be addressed on its own. First of all, it needs to be ensured that the msg.value must in fact be larger than the fee, otherwise it will shift in the msg.value < _feeInEth scenario where the full msg.value is being transferred out. Secondly, the totalSent variable should be increased by the fee which is sent out in an effort to properly address the refund at the end of the function, ensuring no unauthorized refunds take place. Refunds should only apply if totalSent + _feeInEth < msg.value.

	Once this function is fixed, extensive tests are expected and mandatory before the resolution round is validated. Additional charges might apply upon logical changes.
Comments / Resolution	Resolved, an explicit revert has been implemented if the msg.value is below the fee for the ETH distribution scenario. Additionally, the initial balance before the distribution is determined by msg.value deducted the fees, which paves the way for the correct distribution.

Issue	UniV2 pool can be manipulated to save fee
Severity	High

Description	During the handleFee function, the getFee function is invoked, which calculates the amount of eth for the corresponding feesInUSD value via a UniswapV2 pool:
	
	isWethFirst
	? uniV2Router.getAmountIn(_feesInUSD, reserve0, reserve1)
	: uniV2Router.getAmountIn(_feesInUSD, reserve1, reserve0);
	
	The issue with this calculation is that any user can manipulate the pool ratio in the same transaction with a large swap:
	
	function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure returns (uint amountIn) {
	require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
	require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
	uint numerator = reserveIn.mul(amountOut).mul(1000);
	uint denominator = reserveOut.sub(amountOut).mul(997);
	amountIn = (numerator / denominator).add(1);
	}
	
	Such that the denominator becomes a very large value (increase reserveOut; decrease reserveIn). This can then result in a fee of 1 wei.

	This logic will also be vulnerable to slippage if an illiquid pool is used and/or the fee is very large.
Recommendations	Consider using a reliable oracle for this conversion, such as Chainlink. The ETH/USD price feed could be an option, or to make it simple the USDT/ETH price feed.
Comments / Resolution	Resolved, the ETH/USD price is now fetched from the Chainlink oracle, which is then used to convert the nominal USD fee to the corresponding ETH fee.

Issue	Claim of ERC20 will not work
Severity	**High**
Description	Within the claimERC20 function, the transfer is being made as follows: token.transferFrom(address(this), owner(), amount); This will however not work, since no approval was granted. The transferFrom methodology should never be used if the transfer is from inside.
Recommendations	Consider using the standard transfer method for outgoing transfers instead of transferFrom. Extensive testing is required before the validation of the resolution round.
Comments / Resolution	Resolved.

Issue	Denomination of arithmetic operation within handleFees can be abused to pay less fee than expected.
Severity	Medium
Description	The following arithmetic operation within the handleFees function is being executed: if (((feeInEth - msg.value) * 100) / feeInEth > 5) revert FeeNotMet(feeInEth, msg.value); For 100%, the denomination is 100, while usually for smart contract operations, 100% is denominated in 10_000 BPS. This low denomination can be exploited by a user to provide a lower fee than expected. Consider the following scenario: _feeInETH = 100e18 provided msg.value = 94.1e18 follows the operation: (100e18 - 94.1e18) * 100 / 100e18 = 5.9 Since solidity rounds down, this will result in 5. Allowing the user to provide 94.1% instead of the desired minimum fee of 95%. This example is obviously extreme, as we doubt the fee will be 100 ETH. However, on other blockchains, this scenario can be amplified.
Recommendations	Consider increasing the denomination to 10_000.
Comments / Resolution	Resolved

Issue	Lack of msg.value refund if fee is zero
Severity	Medium
Description	Given the handleFees function, it is possible that the fee is zero. In such a scenario, the caller will not receive a msg.value refund, essentially resulting in a loss of funds.
Recommendations	Consider refunding any msg.value in that scenario. This should be only done for multisend functions which NOT distributed ETH, as this would break the whole function flow.
Comments / Resolution	Resolved

Issue	All multisend transactions can be DoS'ed
Severity	Medium
Description	For all four multisend functions, the handleFee function is called, which allows the user to provide a fee up to a minimum of 95% of the expected fee value. However, as within the "UniV2 pool can be manipulated to save fee" issue described, users can have an impact on the ETH price. A malicious user can therefore manipulate the pool to increase the ETH price, this will then result in a revert of the multisend calls, as the provided msg.value will always be <95% of the expected fee, if a malicious user artificially increases the ETH price.
Recommendations	Consider using a reliable oracle for the price determination, such as the Chainlink USDT/ETH oracle.
Comments / Resolution	Resolved

Issue	Lack of safe usage for ERC721
Severity	Low

Description	The contract uses the standard transfer pattern for ERC721 transfers:
	
	L 94:
	
	token.transferFrom(msg.sender, recipients[i], tokenIds[i]);
	
	L 155:
	
	token.transferFrom(address(this), owner(), tokenId);
	
	Which does not execute a safe check if the recipient is a smart contract. In such a scenario, the possibility exists that the smart contract cannot handle the ERC721, which would result in stuck funds.

Recommendations	Consider using the safe method: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L152 The safeERC20 method should also be used for ERC20 transfers.
Comments / Resolution	Partially resolved, for the claimERC20 function, transfer is used. Resolution 2: safeTransfer is now used for the claimERC20 function.

Issue	Unnecessary freedom for users
Severity	Low
Description	Throughout the codebase, users can interact with the following state manipulating functions: multisendERC20; multisendERC721; multisendERC1155; multisendETH These functions allow users to provide parameters which are not necessary for the regular business logic, such as a value of 0 or a malicious token parameter. Generally speaking, we always highly recommend the limitation of user flexibility to counter any possible exploits due to flexible input parameters.
Recommendations	Consider whether it makes sense to add a whitelist functionality for token addresses. Consider validating that the provided amount parameter is non-zero.
Comments / Resolution	Acknowledged.

Issue	Unused declarations
Severity	Informational
Description	The contract uses the following unused declarations: L 6: import "openzeppelin-contracts/token/ERC20/utils/SafeERC20.sol"; L 16: address public usdTokenAddress;

Recommendations	Consider using or removing these lines.
Comments / Resolution	Resolved.