SHELLBOXES

Crowdswap V3

Smart Contract Security Audit

Prepared by ShellBoxes

March 12th, 2024 - March 14th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Crowdswap
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
`https://github.com/CrowdSwap/swaps`	6f68540ea8c9298d7547332436d0517907b37d29

Re-Audit

Repository	Commit Hash
`https://github.com/CrowdSwap/swaps`	5bd93e90dbc917d0b4b4067ab0afec0559a6bf5d

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Crowdswap engaged ShellBoxes to conduct a security assessment on the Crowdswap V3 beginning on March 12th, 2024 and ending March 14th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Crowdswap

CrowdSwap is a cross-chain opportunity optimization and automation platform. It aims to reach mass adoption in crypto for every human being and overcome actual problems that reside from a fast-growing business space like DeFi.

Issuer	Crowdswap
Website	https://crowdswap.org
Type	Solidity Smart Contract
Documentation	Crowdswap Documentation
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
			Likelihood	

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Crowdswap V3 implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 2 high-severity, 1 medium-severity, 3 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. User Can Bypass Swap Fees by Setting Invalid Affiliate Code	CRITICAL	Fixed
SHB.2. Missing fromToken Address Verification in Middle Swaps	HIGH	Fixed
SHB.3. Incorrect Amount in Encoded Swap Data	HIGH	Partially fixed
SHB.4. Front-Run Attack	MEDIUM	Fixed
SHB.5. Missing Swap Receiver Address Verification	LOW	Fixed
SHB.6. Missing Affiliate Fee Percentage Value Verification	LOW	Fixed
SHB.7. Floating Pragma	LOW	Fixed

3 Finding Details

SHB.1 User Can Bypass Swap Fees by Setting Invalid Affiliate Code

- Severity: CRITICAL
- Status: Fixed
- Likelihood: 3
- Impact: 3

Description:

In the CrowdSwapV3 contract, there is a risk of fee bypass if a user sets an invalid affiliateCode parameter in the _crossDexParams for cross dex swaps or _swapParams for normal swaps. The _deductFee function is responsible for transferring fees to the feeTo address based on the feeCalcDirection. However, the _feePercentageCalculator function calculates the fee amount based on the fee percentage associated with the provided affiliateCode. If theaffiliateCode is set to 0 or not set by the owner, the user will not pay any fee for the swap. Since the affiliateFeePercentage mapping is public, users can know all the affiliate codes and can exploit this by setting an invalid affiliateCode to bypass fees.

Files Affected:

SHB.1.1: CrowdSwapV3.sol

```
    function _deductFee(
        IERC20Upgradeable _token,
        address _onBehalfOfAddress,
        uint256 _amount,
        uint32 _affiliateCode
    ) private returns (uint256, uint256) {
        uint256 _amountFee = _feePercentageCalculator(_amount,
            ↪ _affiliateCode);
        if (_amountFee > 0) {
            _safeTransferTokenTo(_token, payable(feeTo), _amountFee);

            emit FeeDeducted(
                _onBehalfOfAddress,
                address(_token),
                _affiliateCode,
                _amount,
                _amountFee
            );
```

SHB.1.2: CrowdSwapV3.sol

```
    function _feePercentageCalculator(
        uint256 _calculationAmount,
        uint32 _affiliateCode
    ) private view returns (uint256) {
        uint256 _percentage = affiliateFeePercentage[_affiliateCode];
        return (_percentage * _calculationAmount) / (1e20);
    }
```

Recommendation:

To mitigate the risk, consider adding a verification step in the _feePercentageCalculator function to check the _percentage value, If it is 0, use the default fee percentage set in the affiliateFeePercentage[0] mapping. This will help prevent fee bypass by ensuring that fees are always applied, even when an invalid affiliateCode is provided. Additionally, consider making the affiliateFeePercentage mapping private to prevent users from accessing the values directly.

Updates

The Crowdswap team fixed the issue by adding a verification step in the _setAffiliateFeePercentage function. Now, any fee percentage should be between the MIN_FEE and MAX_FEE range. If the fee percentage is not defined, the system will use the default fee percentage associated with the affiliate code 0 that is initialized in the initialize function.

SHB.1.3: CrowdSwapV3.sol

```
    function _setAffiliateFeePercentage(
        uint32 _code,
        uint256 _feePercentage
    ) private {
        // 1e18 is 1%
        require(
            MIN_FEE <= _feePercentage && _feePercentage <= MAX_FEE,
            "CrowdSwapV3: feePercentage is not in the range"
        );
```

SHB.1.4: CrowdSwapV3.sol

```
    function _deductFee(
        IERC20Upgradeable _token,
        address _onBehalfOfAddress,
        uint256 _amount,
        uint32 _affiliateCode
    ) private returns (uint256) {
        uint256 _percentage = _affiliateFeePercentage[_affiliateCode];
        //default affliate code is 0
        if (_percentage == 0) {
            _percentage = _affiliateFeePercentage[0];
        }
```

SHB.2 Missing fromToken Address Verification in Middle Swaps

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

In the crossDexSwap function, middle swaps are executed in a loop starting with the first fromToken. However, the function does not verify that the fromToken for each middle swap is the same as the toToken of the previous swap. This verification is crucial to ensure that the amountIn for each subsequent swap is the amountOut of the previous swap. Without this verification, a malicious user could manipulate the swapList array to include tokens that do not follow this pattern, potentially draining the contract's funds and executing unauthorized swaps.

Exploit Scenario:

1. A malicious user checks the contract's balance for specific tokens.
2. This user then calls the crossDexSwap function and inserts a swapList containing swaps that do not follow the expected sequence.
3. For example, the user inserts a swap sequence like T1 -> T2, T3 -> T4, where T3 already has a balance in the contract.
4. The contract executes these swaps sequentially, starting with the first TokenIn (T1) in the list.
5. Since there is no verification that the swaps are sequential and that the input token should be the last output token of the previous swap, the second swap (T3 -> T4) will be executed based on the contract's balance of T3, and not the output of the first swap (T2).
6. This results in the user receiving the output swapped amount of T4, while the contract loses the amount of T3. This loss could be significant if there is a large price difference between T2 and T3, potentially leading to financial losses for the contract.

Files Affected:

SHB.2.1: CrowdSwapV3.sol

```
        for (uint256 i = 0; i < _crossDexParams.swapList.length; i++) {
            fromToken = ERC20Upgradeable(_crossDexParams.swapList[i].
                ↪ fromToken);
```

```
toToken = ERC20Upgradeable(_crossDexParams.swapList[i].toToken);
dexAddress = _extractDexAddress(
    _crossDexParams.swapList[i].dexFlag
);

// Handle token replacement
if (_crossDexParams.swapList[i].isReplace) {
    _crossDexParams.swapList[i].params[
        _crossDexParams.swapList[i].index
    ] = abi.encode(amountIn);
}

// Perform the swap
bytes memory swapData = _assembleCallData(
    _crossDexParams.swapList[i]
);
amountOut = _swap(
    dexAddress,
    swapData,
    fromToken,
    toToken,
    amountIn
);

emit MiddleSwapEvent(
    address(fromToken),
    address(toToken),
    amountIn,
    amountOut,
    _crossDexParams.swapList[i].dexFlag
);

amountIn = amountOut;
```

```
        }
```

Recommendation:

Consider adding a requirement in the crossDexSwap function to verify that the fromToken of each swap in the swapList is the same as the toToken of the previous swap. This verification ensures that swaps are executed in the correct sequence, preventing potential fund draining attacks and unauthorized swaps.

Updates

The Crowdswap team has fixed this issue by ensuring that the fromToken of each swap in the swapList is the same as the toToken of the previous swap.

SHB.2.2: CrowdSwapV3.sol

```
        for (uint256 i = 0; i < _crossDexParams.swapList.length; i++) {
            toToken = ERC20Upgradeable(_crossDexParams.swapList[i].
                ↪ toToken);
            dexAddress = _extractDexAddress(
                _crossDexParams.swapList[i].dexFlag
            );

            // amount replacement
            _crossDexParams.swapList[i].params[
                _crossDexParams.swapList[i].index
            ] = abi.encode(amountIn);

            // Perform the swap
            bytes memory swapData = _assembleCallData(
                _crossDexParams.swapList[i]
            );
            amountOut = _swap(
                dexAddress,
                swapData,
                fromToken,
                toToken,
                amountIn
            );

            emit MiddleSwapEvent(
                address(fromToken),
                address(toToken),
                amountIn,
                amountOut,
                _crossDexParams.swapList[i].dexFlag
            );

            amountIn = amountOut;
            fromToken = toToken;
```

SHB.3 Incorrect Amount in Encoded Swap Data

- Severity : HIGH
- Status : Partially fixed
- Likelihood : 2
- Impact : 3

Description:

The swap and crossDexSwap functions in the CrowdSwapV3 contract are working with the amountIn for the swapped amount of tokenIn. However, in cases where the feeCalcDirection is TokenIn, the amountIn is recalculated, and the fees are deducted from this amount. This can lead to a mismatch between the new amountIn passed to the _swap function and the encoded swap data passed to the exchange contract. As a result, the swap actions may not be executed correctly, leading to failures due to insufficient allowance or ETH sent.

- In the swap function, when calling the _swap function to send encoded swap data to the exchange address, the encoded data is not updated with the new amountIn after fees are deducted, leading to potential swap failures.

- In the crossDexSwap function, when calling _assembleCallData to pack the exchange function selector and the swap params, this array is only updated if the isReplace flag is true. However, the amountIn should be always updated for subsequent swaps, ensuring that it reflects the output of the previous swap.

Files Affected:

SHB.3.1: CrowdSwapV3.sol

```
        if (_swapParams.feeCalcDirection == FeeCalcDirection.TokenIn) {
            (_amountIn, ) = _deductFee(
                _fromToken,
                msg.sender,
                _swapParams.amountIn,
                _swapParams.affiliateCode
            );
        }

        address _dexAddress = _extractDexAddress(_swapParams.dexFlag);

        uint256 _amountOut = _swap(
            _dexAddress,
            _swapParams.data,
            _fromToken,
            _toToken,
            _amountIn
        );
```

SHB.3.2: CrowdSwapV3.sol

```
        if (_crossDexParams.feeCalcDirection == FeeCalcDirection.TokenIn) {
            (amountIn, ) = _deductFee(
                fromToken,
                msg.sender,
                amountIn,
                _crossDexParams.affiliateCode
            );
        }

        // Perform middle swaps
        for (uint256 i = 0; i < _crossDexParams.swapList.length; i++) {
            fromToken = ERC20Upgradeable(_crossDexParams.swapList[i].
                ↪ fromToken);
            toToken = ERC20Upgradeable(_crossDexParams.swapList[i].
                ↪ toToken);
            dexAddress = _extractDexAddress(
                _crossDexParams.swapList[i].dexFlag
            );

            // Handle token replacement
            if (_crossDexParams.swapList[i].isReplace) {
                _crossDexParams.swapList[i].params[
                    _crossDexParams.swapList[i].index
                ] = abi.encode(amountIn);
            }

            // Perform the swap
            bytes memory swapData = _assembleCallData(
                _crossDexParams.swapList[i]
            );
            amountOut = _swap(
                dexAddress,
                swapData,
                fromToken,
                toToken,
                amountIn
            );
```

Recommendation:

- For the swap function, encode the swap data _swapParams.data with the new amountIn on-chain using the _assembleCallData function.
- For the crossDexSwap function, remove the isReplace condition and ensure that the params array is updated with the correct amountIn for subsequent swaps, reflecting the output of the previous swap.

Updates

The Crowdswap team addressed the issue in the crossDexSwap function by removing the isReplace variable and always updating the amountIn. However, for the swap function, the team assumes that the responsibility of preparing accurate swap data, especially when feeCalcDirection is set to TokenIn, lies with the user. Callers must ensure the swap data is correctly formulated to account for this specification.

SHB.3.3: CrowdSwapV3.sol

```
            // amount replacement
            _crossDexParams.swapList[i].params[
                _crossDexParams.swapList[i].index
            ] = abi.encode(amountIn);
```

SHB.4 Front-Run Attack

- Severity : MEDIUM
- Status : Fixed

- Likelihood : 1
- Impact : 3

Description:

The setAffiliateFeePercentage function in the CrowdSwapV3 contract allows the contract owner to front-run any swap action and modify the percentage related to the affiliate code provided by the user. This could result in the owner setting higher fees for total swapped amounts, including setting the percentage to 100% to claim the entire swap amount.

Files Affected:

SHB.4.1: CrowdSwapV3.sol

```
    function setAffiliateFeePercentage(
        uint32 _affiliateCode,
        uint256 _feePercentage
    ) external onlyOwner {
        emit setAffiliateFeePercent(
            _affiliateCode,
            affiliateFeePercentage[_affiliateCode],
            _feePercentage
        );
        affiliateFeePercentage[_affiliateCode] = uint256(_feePercentage);
    }
```

Recommendation:

To mitigate the risk, consider implementing a maximum limit percentage value (limitPrc) that the contract owner can set to prevent excessively high percentages for an affiliate code. Additionally, add the whenPaused modifier to the setAffiliateFeePercentage function to ensure that fee percentage changes are only applied when the contract is paused. These measures will enhance the security and fairness of the contract by ensuring that fee adjustments are controlled and reasonable, with changes applied only when the contract is in a paused state.

Updates

The Crowdswap team fixed the issue by allowing the owner to set the affiliate fee percentage only if the contract is paused, adding the whenPaused modifier to the setAffiliateFeePercentage function.

SHB.5 Missing Swap Receiver Address Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The swap and crossDexSwap functions in the CrowdSwapV3 contract lack address verification for the _swapParams.receiverAddress and _crossDexParams.receiverAddress parameters. These addresses should be different from address(0), as the output swapped tokens will be transferred to these addresses. Without this verification, there is a risk of sending tokens to a zero address, which could result in the loss of tokens.

Files Affected:

SHB.5.1: CrowdSwapV3.sol

```
        _safeTransferTokenTo(
            _toToken,
            payable(_swapParams.receiverAddress),
            _amountOut
        );
```

SHB.5.2: CrowdSwapV3.sol

```
        _safeTransferTokenTo(
            toToken,
            payable(_crossDexParams.receiverAddress),
            amountOut
        );
```

SHB.5.3: CrowdSwapV3.sol

```
    function _safeTransferTokenTo(
        IERC20Upgradeable _toToken,
        address payable _receiverAddress,
        uint256 _amountOut
    ) private {
        uint256 _balanceBefore = UniERC20Upgradeable.uniBalanceOf(
            _toToken,
            _receiverAddress
        );
        UniERC20Upgradeable.uniTransfer(_toToken, _receiverAddress,
            ↪ _amountOut);
        uint256 _balanceAfter = UniERC20Upgradeable.uniBalanceOf(
            _toToken,
            _receiverAddress
        );
        require(
            _balanceAfter - _balanceBefore == _amountOut,
            "CrowdSwapV3: tokenOut has not transferred to receiver"
        );
    }
```

Recommendation:

Consider adding a check in the _safeTransferTokenTo function to verify that the receiverAddress is not equal to address(0).

Updates

The Crowdswap team fixed the issue by adding a require statement to verify that the receiverAddress is not equal to address(0).

SHB.5.4: CrowdSwapV3.sol

```
        require(
            _swapParams.receiverAddress != address(0),
            "CrowdSwapV3: receiverAddress is 0"
        );
```

SHB.5.5: CrowdSwapV3.sol

```
        require(
            _crossDexParams.receiverAddress != address(0),
            "CrowdSwapV3: receiverAddress is 0"
        );
```

SHB.6 Missing Affiliate Fee Percentage Value Verification

- Severity : LOW
- Status : Fixed

- Likelihood : 1
- Impact : 2

Description:

The initialize and setAffiliateFeePercentage functions in the CrowdSwapV3 contract lack verification for the _defaultFeePercentage and _feePercentage parameters, which represent the affiliate fee percentage associated with a specific _affiliateCode. This value should be within a specific range because it represents a percentage, with a maximum value of 1e20. Without this verification, there is a risk of setting the affiliateFeePercentage to an invalid or excessively high value, which could lead to unexpected behavior or vulnerabilities in the contract.

Files Affected:

SHB.6.1: CrowdSwapV3.sol

```
    function initialize(
        address payable _feeTo,
        uint256 _defaultFeePercentage,
        DexAddress[] calldata _dexAddresses
    ) public initializer {
        OwnableUpgradeable.initialize();
        PausableUpgradeable.__Pausable_init();
        setFeeTo(_feeTo);
        addDexchangesList(_dexAddresses);
        affiliateFeePercentage[0] = uint256(_defaultFeePercentage);
```

SHB.6.2: CrowdSwapV3.sol

```
    function setAffiliateFeePercentage(
        uint32 _affiliateCode,
        uint256 _feePercentage
    ) external onlyOwner {
        emit setAffiliateFeePercent(
            _affiliateCode,
            affiliateFeePercentage[_affiliateCode],
            _feePercentage
        );
        affiliateFeePercentage[_affiliateCode] = uint256(_feePercentage);
    }
```

Recommendation:

It is recommended to add validation checks in both the initialize and setAffiliateFeePercentage functions to ensure that the _defaultFeePercentage and _feePercentage values are within the valid range (e.g., less than or equal to 1e20).

Updates

The Crowdswap team has fixed this issue by ensuring that each _feePercentage is within the MIN_FEE and MAX_FEE range in the _setAffiliateFeePercentage function.

SHB.6.3: CrowdSwapV3.sol

```
    function _setAffiliateFeePercentage(
        uint32 _code,
        uint256 _feePercentage
    ) private {
        // 1e18 is 1%
        require(
            MIN_FEE <= _feePercentage && _feePercentage <= MAX_FEE,
            "CrowdSwapV3: feePercentage is not in the range"
        );
```

SHB.7 Floating Pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The CrowdSwapV3 contract uses a floating Solidity pragma of 0.8.10, indicating compatibility with any compiler version from 0.8.10 (inclusive) up to, but not including, version 0.9.0. This flexibility could potentially introduce unexpected behavior if the contracts are compiled with a newer compiler version that includes breaking changes.

Files Affected:

SHB.7.1: CrowdSwapV3.sol

```
pragma solidity ^0.8.10;
```

Recommendation:

It is generally recommended to lock the pragma statement to a specific Solidity compiler version to ensure consistent behavior across different compiler versions. To achieve this, consider removing the caret (^) from the pragma statement and specifying a fixed version, such as pragma solidity 0.8.10.

Updates

The Crowdswap team has resolved this issue by fixing the pragma version and locking it to 0.8.10.

4 Best Practices

BP.1 Optimize DexAddress Struct for Storage Efficiency

Description:

To optimize storage usage in the CrowdSwapV3 contract, consider changing the flag variable in the DexAddress struct from uint256 to uint8 to ensure the struct fits within a single storage slot. Since the flag is a binary indicator, a uint8 can efficiently represent its range (0-255), reducing each DexAddress struct's storage footprint. This optimization enhances gas efficiency and overall contract performance.

Files Affected:

BP.1.1: CrowdSwapV3.sol

```
    struct DexAddress {
        uint256 flag;
        address adr;
    }
```

Status - Fixed

BP.2 Optimize _deductFee Function for Gas Efficiency

Description:

To optimize the _deductFee function in the CrowdSwapV3 contract for gas efficiency, consider the following improvements:

1. Remove the second return value: Since all swap functions in the contract only need the _netAmount value and do not require the _amountFee value, consider removing

the second return value from the _deductFee function. This can simplify the function and reduce gas costs.

2. Remove unnecessary parameter: The _onBehalfOfAddress parameter is not needed in the _deductFee function since the fee is deducted from the transaction sender (msg.sender). Removing this parameter and working directly with msg.sender can streamline the function and make it clearer.

Files Affected:

BP.2.1: CrowdSwapV3.sol

```
    function _deductFee(
        IERC20Upgradeable _token,
        address _onBehalfOfAddress,
        uint256 _amount,
        uint32 _affiliateCode
    ) private returns (uint256, uint256) {
        uint256 _amountFee = _feePercentageCalculator(_amount,
            ↪ _affiliateCode);
        if (_amountFee > 0) {
            _safeTransferTokenTo(_token, payable(feeTo), _amountFee);

            emit FeeDeducted(
                _onBehalfOfAddress,
                address(_token),
                _affiliateCode,
                _amount,
                _amountFee
            );
        }
        uint256 _netAmount = _amount - _amountFee;
        return (_netAmount, _amountFee);
    }
```

Status - Partially fixed

BP.3 Use External Visibility for Gas Efficiency in crossDexSwap Function

Description:

The crossDexSwap function in the CrowdSwapV3 contract can be declared as external instead of public to save gas. Declaring the function as external allows calling it only from outside the contract, reducing gas costs compared to public functions, which can be called internally as well.

Files Affected:

BP.3.1: CrowdSwapV3.sol

```
    /**
     * @dev Swap the input tokens to output tokens by doing two or more
        ↪ separate swaps between dexes
     **/
    function crossDexSwap(
        CrossDexParams memory _crossDexParams
    ) public payable whenNotPaused returns (uint256) {
```

Status - Fixed

BP.4 Include Constructor in UUPS Upgradeable Contracts with Disable Initializers Comment

Description:

The CrowdSwapV3 contract follows the UUPS (Universal Upgradeable Proxy Standard) pattern for upgradability. It is advisable to include a constructor with the following comment to disable initializers:

BP.4.1: CrowdSwapV3

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
\_disableInitializers();
}
```

This practice helps prevent accidental execution of initializers during contract deployment, ensuring the correct behavior of upgradeable contracts.

Files Affected:

BP.4.2: CrowdSwapV3.sol

```
contract CrowdSwapV3 is
    Initializable,
    UUPSUpgradeable,
    OwnableUpgradeable,
    PausableUpgradeable
{
```

Status - Fixed

5 Tests

Results:

→ AggregatorV3 - Ownable

✓ Call unknown function over Crowdswap contract should be reverted

✓ Send ETH to the contract should be accepted

✓ Change feePercentage

→ AggregatorV3 - swap

✓ Swap TOKEN/TOKEN should be successful

✓ Swap TOKEN/ETH should be successful

✓ Swap ETH/TOKEN should be successful

✓ When the non zero ETh is sent, Swap TOKEN/TOKEN should be failed

✓ When the sent ETh is not equal to amount in, Swap ETH/TOKEN should be failed

✓ When amount out is 0, Swap should be failed

✓ When amount out is lower than min amount, Swap should be failed

✓ When token in is the same token out, Swap should be failed

✓ When dex flag is wrong, Swap should be failed

→ deductFee - From token in

✓ When TOKEN/ETH swap, should transfer from token to feeTo address

✓ When ETH/TOKEN swap, should transfer ETH to feeTo address

→ deductFee - From token out

✓ When TOKEN/ETH swap, should transfer from token to feeTo address

✓ When ETH/TOKEN swap, should transfer ETH to feeTo address

→ AggregatorV3 - crossDexSwap

→ single path

✓ Swap TOKEN/TOKEN should be successful

✓ Swap ETH/TOKEN should be successful

✓ Swap TOKEN/ETH should be successful

→ double path

✓ Swap TOKEN/TOKEN/TOKEN should be successful

✓ Swap ETH/TOKEN/TOKEN should be successful

✓ Swap TOKEN/WETH/TOKEN should be successful

✓ Swap TOKEN/TOKEN/ETH should be successful

✓ When the non zero ETh is sent, Swap TOKEN/*/* should be failed

✓ When the sent ETh is not equal to amount in, Swap ETH/TOKEN/TOKEN should be failed

✓ When amount out is 0, Swap should be failed

✓ When amount out is lower than min amount, Swap should be failed

✓ When swap list is empty, swap should be failed

✓ When dex flag is wrong, Swap should be failed

→ deductFee - From token in

✓ When ETH/TOKEN/TOKEN swap, should transfer from token to feeTo address

✓ When TOKEN/TOKEN/ETH swap, should transfer from token to feeTo address

→ deductFee - From token out

✓ When ETH/TOKEN/TOKEN swap, should transfer from token to feeTo address

✓ When TOKEN/TOKEN/ETH swap, should transfer from token to feeTo address

Coverage:

The code coverage results were obtained by running npx hardhat coverage in the Crowdswap V3 project. We found the following results :

→ CrowdSwapV3.sol

- Statements Coverage : 96.05%
- Branches Coverage : 73.44%
- Functions Coverage : 77.78%
- Lines Coverage : 96.81%

6 Conclusion

In this audit, we examined the design and implementation of Crowdswap V3 contract and discovered several issues of varying severity. Crowdswap team addressed the issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Crowdswap Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
swap/CrowdSwapV3.sol	fda1497029b5597bba0907038717c7f6

7.2 Re-Audit

Files	MD5 Hash
swap/CrowdSwapV3.sol	181f897b4925dfda7de6c62eac87b5e4

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com