SHELLBOXES

Crowdswap ETF

Smart Contract Security Audit

Prepared by ShellBoxes

March 4th, 2024 - March 8th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Crowdswap
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
`https://github.com/CrowdSwap/crowd_etf`	6c4a382c16c28703d04273798c1757de08d67908

Re-Audit

Repository	Commit Hash
`https://github.com/CrowdSwap/crowd_etf`	c98427d3056a7130a92f9cdff8cd1d3cf80b4332

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Crowdswap engaged ShellBoxes to conduct a security assessment on the Crowdswap ETF beginning on March 4th, 2024 and ending March 8th, 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	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

2 Findings Overview

2.1 Disclaimer

During the audit, it was noted that the ETFProxy contract performs external calls to another contract called the swapContract. It is important to note that this contract is out of scope for this audit and was not reviewed as part of this assessment.

While the swapContract contract is out of scope, it is assumed that the contract has been thoroughly tested and will always act as intended. However, it is important to keep in mind that any issues or vulnerabilities within this contract could potentially affect the swapping logic behind the ETFProxy Contract.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the Crowdswap ETF 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.

2.3 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 medium-severity, 2 low-severity, 1 informational-severity, 1 undetermined-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Withdrawal Functionality Blocked by Paused State	MEDIUM	Fixed
SHB.2. Potential Duplicate Tokens with Different Percentages Added to Plan Token Percentages	LOW	Fixed

SHB.3. Missing Address Verification	LOW	Fixed
SHB.4. Potential Creation of Duplicate Plans	INFORMATIONAL	Acknowledged
SHB.5. User Can Manipulate the Token Price	UNDETERMINED	Acknowledged

3 Finding Details

SHB.1 Withdrawal Functionality Blocked by Paused State

- Severity : MEDIUM
- Status : Fixed

- Likelihood : 2
- Impact : 2

Description:

The withdrawWithSwap function allows users to withdraw tokens from an investment by executing swaps. However, the function includes the whenNotPaused modifier, preventing users from withdrawing tokens when the contract is paused. This limitation can prevent users from executing swaps and gaining profits at specific times, impacting the expected behavior of the function.

Files Affected:

SHB.1.1: ETFProxy.sol

```
    function withdrawWithSwap(
        uint256 _tokenId,
        IERC20Upgradeable _tokenOut,
        uint16 _percentage,
        SwapInfo[] memory _swaps
    ) external nonReentrant whenNotPaused {
```

Recommendation:

Consider documenting the behavior of this function, indicating that it cannot be executed when the contract is paused. Additionally, clarify in the documentation that the contract owner has the ability to pause the contract, and users should be aware of this possibility when planning their token withdrawals and swaps.

Updates

The Crowdswap team fixed this issue by removing the whenNotPaused modifier from the withdrawWithSwap function.

SHB.2 Potential Duplicate Tokens with Different Percentages Added to Plan Token Percentages

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

Description:

The createPlan function in the ETFReceipt contract allows the contract owner to create a new plan with the specified name and token percentages. However, there is a potential issue where the _tokenPercentages parameter can contain the same token with different percentage values. This can lead to confusion and unexpected behavior in the future, as the function will add both percentages for the same token to the planTokenPercentages mapping. This could result in incorrect calculations or unintended outcomes when managing the plan's token percentages.

Files Affected:

SHB.2.1: ETFReceipt.sol

```
        for (uint256 i = 0; i < _tokenPercentages.length; i++) {
            planTokenPercentages[_planId].push(_tokenPercentages[i]);
        }
        emit PlanCreated(msg.sender, _planId, _tokenPercentages.length);
```

Recommendation:

To mitigate this issue, it is recommended to add a check in the createPlan function to ensure that each token appears only once in the _tokenPercentages array. If a token is found to have multiple percentage values, consider rejecting the input and throwing an error to prevent the creation of a plan with conflicting token percentages.

Updates

The Crowdswap team fixed this issue by checking if a token address already exists using the _existAddressInArray function.

SHB.3 Missing Address Verification

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

Description:

Certain functions within the ETFReceipt and ETFProxy contracts lack address verification, allowing for the possibility of addresses being identical to address(0). This absence of address verification poses a potential security vulnerability that could lead to unintended behaviors or exploitation.

Files Affected:

SHB.3.1: ETFReceipt.sol

```
function setETFProxyAddress(address _ETFProxyAddress) external
    ↪ onlyOwner {
    ETFProxyAddress = _ETFProxyAddress;
}
```

SHB.3.2: ETFProxy.sol

```
    function setETFReceiptAddress(
        address _ETFReceiptAddress
    ) external onlyOwner {
        ETFReceiptAddress = _ETFReceiptAddress;
        emit SetETFReceiptAddress(_ETFReceiptAddress);
    }

    /**
    * @notice Sets the address of the swap contract.
    * @param _swapContract The address of the swap contract.
    * @dev Allows the owner of the contract to update the address of the swap contract.
    */
    function setSwapContract(address _swapContract) external onlyOwner {
        swapContract = _swapContract;
        emit SetSwapContract(_swapContract);
    }
```

Recommendation:

To address this issue, it is recommended to implement robust address verification checks in the relevant functions of the project contracts. Ensure that the provided addresses are distinct from address(0) to enhance security and prevent potential misuse or vulnerabilities.

Updates

The Crowdswap team fixed this issue by adding a zero address check to the affected functions. The address verification is implemented in the _requiredValidAddress function.

SHB.4 Potential Creation of Duplicate Plans

- Severity : INFORMATIONAL
- Status : Acknowledged

- Likelihood : 1
- Impact : 0

Description:

The createPlan function in the ETFReceipt contract allows the contract owner to create a new plan with an existing plan name and token percentages. This could lead to duplicate plans in the plans array, as the function does not check for the uniqueness of plan names. Duplicate plans with the same name may cause confusion and unexpected behavior, potentially leading to inconsistencies in the application's logic or data.

Files Affected:

SHB.4.1: ETFReceipt.sol

```
    function createPlan(
        string memory _name,
        TokenPercentage[] memory _tokenPercentages
    ) external onlyOwner {
        _requiredValidTokenPercentages(_tokenPercentages);

        Plan memory _plan = Plan(_name, true);
        plans.push(_plan);

        uint256 _planId = plans.length - 1;

        for (uint256 i = 0; i < _tokenPercentages.length; i++) {
            planTokenPercentages[_planId].push(_tokenPercentages[i]);
        }
        emit PlanCreated(msg.sender, _planId, _tokenPercentages.length);
    }
```

Recommendation:

It is recommended to add a check in the createPlan function to verify that a plan with the same name does not already exist before creating a new plan. You can use a mapping or another data structure to keep track of existing plan names and prevent duplicates.

Updates

The Crowdswap team acknowledged this issue, stating that they can have similar plans as they might deactivate a plan and create another one with the same name for some business purposes.

SHB.5 User Can Manipulate the Token Price

- Severity : UNDETERMINED
- Status : Acknowledged
- Likelihood : 3
- Impact : -

Description:

The SwapInfo struct within the contract is designed to contain information about a token swap, with the price attribute meant to determine the price of the token swap. However, in the current implementation, users can set the price attribute when providing swap information for the invest function. This deviates from the intended use, as the price attribute is typically meant to store invest-time prices for later comparison with real-time prices to calculate user profits. Allowing users to set this value directly could lead to incorrect pricing, potentially resulting in misleading profit calculations.

Files Affected:

SHB.5.1: ETFProxy.sol

```
/**
* @notice Struct representing information about a token swap.
* @member token The token contract involved in the swap. (tokenOut
   ↪ for invest and tokenIn for withdraw)
* @member price The price of the token swap. (tokenOut for invest
   ↪ and tokenIn for withdraw)
* @member data Additional data related to the swap.
*/
struct SwapInfo {
    IERC20Upgradeable token;
    uint256 price;
    bytes data;
}
```

SHB.5.2: ETFProxy.sol

```
            IETFReceipt.TokenDetail memory _tokenDetail = IETFReceipt
                .TokenDetail({
                    token: address(_swaps[i].token),
                    amount: _amountOut,
                    price: _swaps[i].price
                });

            _tokenDetails[i] = _tokenDetail;
```

Recommendation:

To mitigate the risk of incorrect token pricing, consider implementing a mechanism to determine token prices accurately, such as retrieving real-time prices from a trusted oracle.

Updates

The Crowdswap team has acknowledged the issue, stating that they store the initial token price at the time of investment for comparison with the current price. This data is used to calculate the percentage of profit. They noted that if users deliberately input incorrect prices, it would only lead to a misinterpretation of their profit percentage, which does not affect their actual investment data.

4 Best Practices

BP.1 Remove Dead Code

Description:

The _batchWithdrawSwap function in the ETFProxy contract contains commented-out code, which is considered dead code and should be removed. Commented-out code does not contribute to the functionality of the contract and can clutter the codebase, making it harder to read and maintain.

Files Affected:

BP.1.1: ETFProxy.sol

```
            uint256 _remainsAmount = _invest.tokenDetails[i].amount -
                _amountOut;
            // if (_doubleCheckPrice) {
            // _checkAmountOut(
            // _invest.tokenDetails[i].token,
            // address(_tokenOut),
            // _swaps[i].price,
            // _tokenOutPrice,
            // _slicedAmountIn,
            // _amountOut
            // );
            // }

            if (_percentage != MAX_P) {
```

Status - Fixed

BP.2 Use MAX_FEE Constant In _calculateFee function

Description:

The _calculateFee function currently uses a hardcoded constant (1e20) to represent the maximum fee percentage. While a constant MAX_FEE is declared in the contract, the function does not use it for the calculation. It is recommended to use the MAX_FEE constant for fee calculations instead of the hardcoded value to improve code readability and maintainability.

Files Affected:

BP.2.1: ETFProxy.sol

```
    uint256 public constant MAX_FEE = 1e20; //100%
```

BP.2.2: ETFProxy.sol

```
    function _calculateFee(
        uint256 _amount,
        uint256 _percentage
    ) internal pure returns (uint256) {
        return (_amount * _percentage) / 1e20;
    }
```

Status - Fixed

BP.3 Ensure Correct Event Parameter Values

Description:

In the WithdrawWithoutSwap function, the Withdrawn event is emitted with parameters (msg.sender, _tokenId, _invest.id).

However, according to the event's declaration in the contract, the third parameter should represent the plan ID. To maintain consistency and clarity, emit the Withdrawn event with parameters (msg.sender, _tokenId, _invest.planId).

Files Affected:

BP.3.1: ETFProxy.sol

```
    event Withdrawn(
        address indexed user,
        uint256 indexed investId,
        uint256 indexed planId
    );
```

BP.3.2: ETFProxy.sol

```
        emit Withdrawn(msg.sender, _tokenId, _invest.id);
```

Status - Fixed

BP.4 Correct Event Parameter Names

Description:

The SetFee event's parameter names do not match the FeeInfo struct's member names, leading to potential confusion. To maintain consistency and clarity, ensure that the event parameter names match the struct's member names. Specifically, change stakeFee to investFee and unstakeFee to withdrawFee. This adjustment will align the event parameter names with the struct's member names, improving the readability and maintainability of the code.

Files Affected:

BP.4.1: ETFProxy.sol

```
    /**
     * @dev A struct containing parameters needed to calculate fees
     * @member feeTo The address of feeTo
     * @member investFee The fee of invest step
     * @member withdrawFee The fee of withdraw step
     */
    struct FeeInfo {
        address payable feeTo;
        uint256 investFee;
        uint256 withdrawFee;
    }
```

BP.4.2: ETFProxy.sol

```
    event SetFee(
        address indexed user,
        address feeTo,
        uint256 stakeFee,
        uint256 unstakeFee
    );
```

Status - Fixed

BP.5 Remove Hardhat Console Import

Description:

The ETFProxy contract imports hardhat/console.sol library, which is used for debugging purposes in development environments. However, including this import in production code is unnecessary and could potentially lead to security risks. It is recommended to remove the hardhat/console.sol import to ensure that only necessary and safe code is included in the production environment.

Files Affected:

BP.5.1: ETFProxy.sol

```
import "hardhat/console.sol";
```

Status - Fixed

BP.6 Remove Unused InvestDetails Struct

Description:

The InvestDetails struct is defined in the ETFReceipt contract but is not used anywhere in the contract. This unused struct adds unnecessary complexity to the contract and consumes storage space. It is recommended to remove the InvestDetails struct to simplify the contract and reduce gas costs and storage overhead.

Files Affected:

BP.6.1: ETFReceipt.sol

```
    struct InvestDetails {
        uint256 amount;
        uint256 price;
    }
```

Status - Fixed

BP.7 Simplify Token Modification in burnAndModify Function

Description:

The burnAndMint function, which burns an existing token and mints new tokens for a specified address according to a given investment plan, can be changed to modifyInvestDetails. This function would modify the existing token's investment details, such as the creation time and token details, instead of burning and minting new tokens. Since the token owner remains the same as the recipient address _to, this approach simplifies the process by eliminating the need to consume tokens and mint new ones, reducing gas costs and storage overhead.

Files Affected:

BP.7.1: ETFReceipt.sol

```
    function burnAndMint(
        uint256 _tokenId,
        address _to,
        uint256 _planId,
        TokenDetail[] memory _tokenDetails
    ) public onlyETF {
        address _owner = ERC721Upgradeable.ownerOf(_tokenId);
        require(
            isApprovedForAll(_owner, ETFProxyAddress) ||
                getApproved(_tokenId) == ETFProxyAddress,
            "ETFReceipt: approve needed"
        );
        _burn(_tokenId);

        uint256 _newId = receipts.length;
        Invest memory _newInvest = Invest({
            id: _newId,
            planId: _planId,
            createTime: block.timestamp
        });

        receipts.push(_newInvest);
        for (uint256 i = 0; i < _tokenDetails.length; i++) {
            tokenDetails[_newId].push(_tokenDetails[i]);
        }

        _mint(_to, _newId);
        emit BurnedAndMinted(_tokenId, _to, _newId);
    }
```

Status - Acknowledged

BP.8 Mismatch Between Code and Specification in changePlanActiveStatus Function

Description:

The changePlanActiveStatus function allows the contract owner to activate or deactivate an existing plan. However, the function signature includes a parameter _name for the new name of the plan, which contradicts the function's Natspec to only modify the plan's active status. This mismatch between the code and the specification could lead to confusion and unexpected behavior. To align the code with the documentation and avoid confusion, consider either modifying the function to allow changing both the plan's name and active status, or remove the _name parameter if only the active status should be modified. Clearly document the intended behavior of the function to ensure that users understand its purpose and usage.

Files Affected:

BP.8.1: ETFReceipt.sol

```
    /**
     * @notice To activate/deactivate an existing plan
     * @param _planId The id of specific plan
     * @param _name The new name of the plan.
     * @param _isActive The new status of the plan
     */
    function changePlanActiveStatus(
        uint256 _planId,
        string memory _name,
        bool _isActive
    ) external onlyOwner {
        require(_planId < plans.length, "ETFReceipt: Invalid plan ID");

        // Update the plan attributes
        plans[_planId].active = _isActive;
        plans[_planId].name = _name;

        emit PlanUpdated(msg.sender, _planId, _isActive, _name);
    }
```

Status - Fixed

BP.9 Streamlining Struct Usage for Efficient Data Storage

Description:

To improve efficiency and reduce duplication, optimize the usage of the Invest and InvestDetail structs, as well as the Plan and PlanDetail structs. Instead of storing duplicate values in arrays and mappings, consider directly using the attributes of the InvestDetail and PlanDetail structs as the main structs for the plans and receipts arrays. Remove the planTokenPercentages and tokenDetails mappings to simplify the storage structure and improve efficiency. This approach eliminates the need for separate mappings and arrays, streamlining data access and manipulation.

BP.9.1: IETFReceipt

```
struct Invest {
uint256 id;
uint256 planId;
uint256 createTime;
TokenDetail[] tokenDetails;
}
struct Plan {
uint256 id;
string name;
bool active;
TokenPercentage[] tokenPercentages;
}
```

Files Affected:

BP.9.2: IETFReceipt.sol

```
    struct Plan {
        string name;
        bool active;
    }

    struct PlanDetail {
        uint256 id;
        string name;
        bool active;
        TokenPercentage[] tokenPercentages;
    }
```

BP.9.3: IETFReceipt.sol

```
    struct Invest {
        uint256 id;
        uint256 planId;
        uint256 createTime;
    }

    struct InvestDetail {
        uint256 id;
        uint256 planId;
        uint256 createTime;
        TokenDetail[] tokenDetails;
    }
```

Status - Partially fixed

5 Tests

Results:

→ ETFProxy Contract

✓ Should deploy the contracts

✓ Should create a new plan

✓ Should revert because of miscalculation in percentages when creating a new plan

✓ Should revert because of wrong address when creating a new plan

✓ Should revert because of calling by no owner when creating a new plan

✓ Should changePlanActiveStatus

✓ Should revert because of calling by no owner when changePlanActiveStatus

✓ Should revert because of invalid plan ID when changePlanActiveStatus

✓ Should mint a new NFT receipt

✓ Should burn a NFT receipt

✓ Should revert if burning a NFT for the second time

✓ Should revert because of approve with wrong owner

✓ Should transfer NFT

✓ Should safeTransfer NFT

✓ Should revert transfer because wrong owner NFT

✓ Should transfer account2 to account1 with approved token NFT

✓ Get all plans

✓ Get a user tokens

✓ Should revert if a user wants to burn his NFT

✓ Get user tokens after transfer

✓ Should revert transfer a token to two users

✓ Should revert transfer a token after burn

✓ Get an invest with a tokenId

✓ New tokenId must increment after burning a token

✓ Should burnAndMint

✓ Should withdraw without swap 100

✓ Should withdraw without swap 40

✓ Should withdraw with swap 100

✓ Should withdraw with swap 10

✓ Should invest

Coverage:

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

- Statements Coverage : 82.13%
- Branches Coverage : 40.22%
- Functions Coverage : 62.86%
- Lines Coverage : 80.07%

6 Conclusion

In this audit, we examined the design and implementation of Crowdswap ETF contract and discovered several issues of varying severity. Crowdswap team addressed 3 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
etf/ETFProxy.sol	b10239d893d843160f6a90493c277e28
etf/ETFReceipt.sol	1eded5619f4c153a78a93ee6e3410d86
etf/IETFReceipt.sol	5b97ef36e03f255b0210d1cf55019818

7.2 Re-Audit

Files	MD5 Hash
etf/ETFProxy.sol	f9a5dd26833d314ffe492aeacded5239
etf/ETFReceipt.sol	6c23d040eca543af90252c5e8989fedb
etf/IETFReceipt.sol	f144e75132d9b810f5e76471082304eb

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