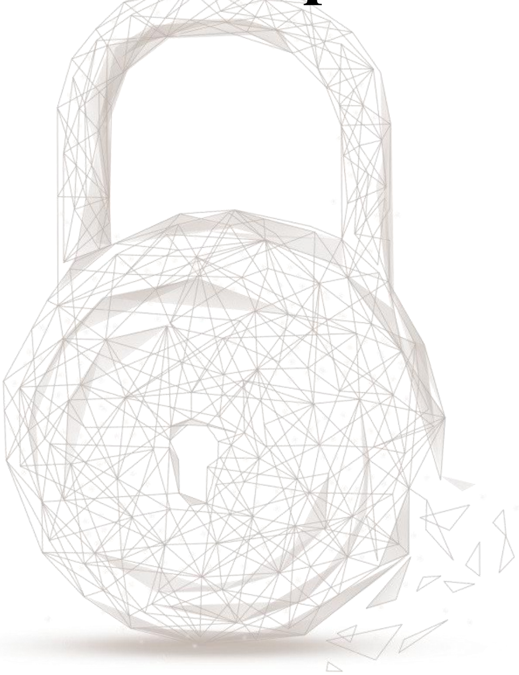
BEOSIN

Blockchain Security

Smart Contract Audit Report
for
CoverCompared



Audit Number: 202201061801

Contract Name: CoverCompared

Deployment Platform: Ethereum

Contract Address:

https://github.com/CoverCompared/smart-contracts

Commit Hash:

ac0a584f04d0da60b1710e079b4a980c81dd320b (old)

85627d944e59b79b79c6d8ede59c794f52390f8d (latest)

Audit Start Date: 2021.12.16

Audit Completion Date: 2022.01.06

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of CoverCompared smart contracts, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the CoverCompared contract was found to have 1 high, 1 medium and 5 info risk items. As of the completion of the audit, all risk items have been fixed or properly handled. The overall result of the CoverCompared smart contract is Pass.** The following is the detailed audit information for this project.

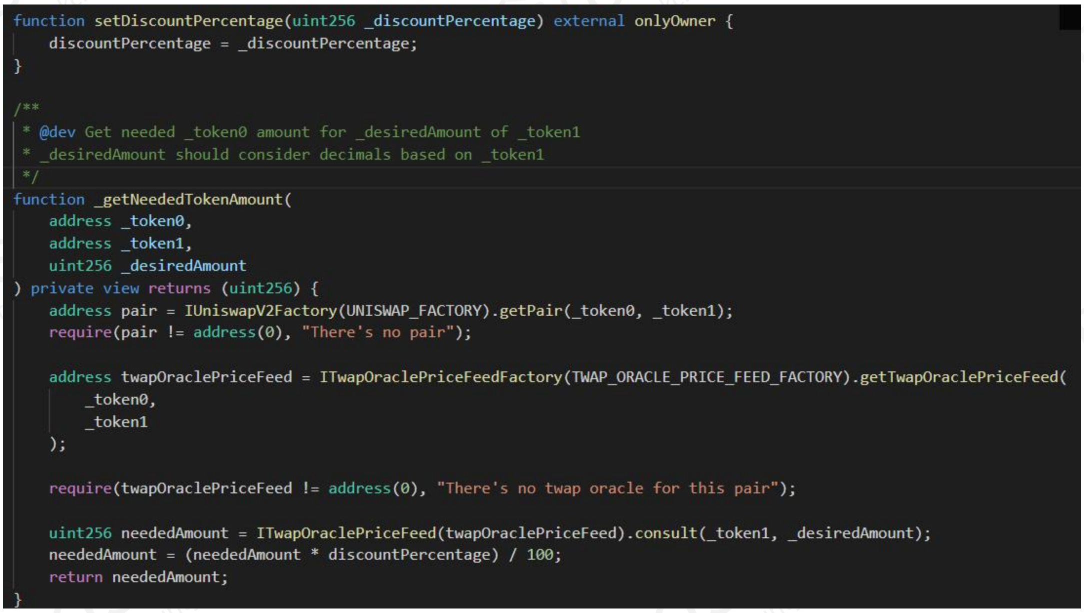
Index	Risk items	Risk level	Fix results status
Cover-1	Uncapped discount rates	High	Fixed
Cover-2	Implementing Exceptions	Medium	Fixed
Cover-3	Non-standard event triggering	Info	Fixed
Cover-4	Owner Privilege issues	Info	Fixed
Cover-5	Redundant codes	Info	Fixed
Cover-6	Pause module	Info	Fixed
Cover-7	Unnecessary payable attributes	Info	Fixed

Table 1. Key Audit Findings

Risk Descriptions and Fix Results

[Cover-1 High] Uncapped discount rates

Description: In the ExchangeAgent contract, the owner can modify the discountPercentage parameter arbitrarily (default is 75). A value greater than 100 will cause the user to send more tokens to this contract.

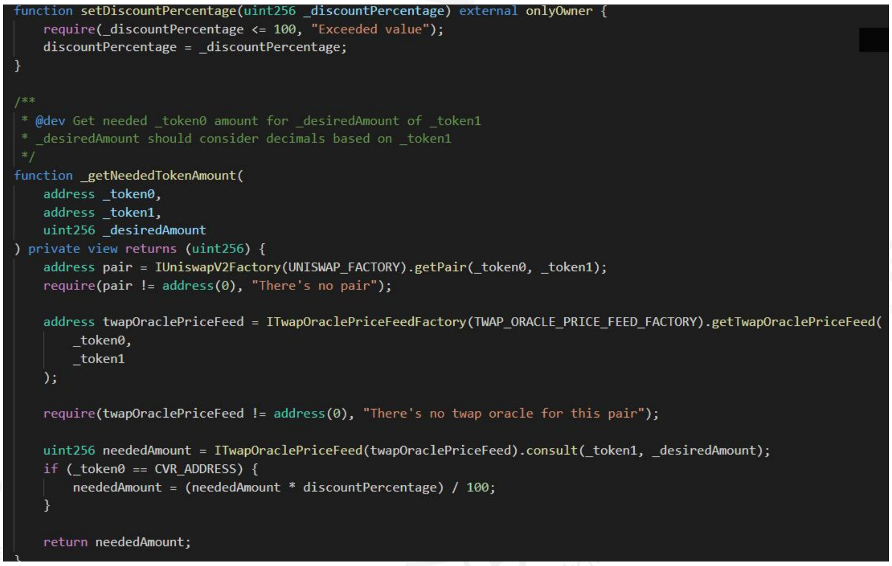


```
function setDiscountPercentage(uint256 _discountPercentage) external onlyOwner {
    discountPercentage = _discountPercentage;
}

/**
 * @dev Get needed _token0 amount for _desiredAmount of _token1
 * _desiredAmount should consider decimals based on _token1
 */
function _getNeededTokenAmount(
    address _token0,
    address _token1,
    uint256 _desiredAmount
) private view returns (uint256) {
    address pair = IUniswapV2Factory(UNISWAP_FACTORY).getPair(_token0, _token1);
    require(pair != address(0), "There's no pair");

    address twapOraclePriceFeed = ITwapOraclePriceFeedFactory(TWAP_ORACLE_PRICE_FEED_FACTORY).getTwapOraclePriceFeed(
        _token0,
        _token1
    );

    require(twapOraclePriceFeed != address(0), "There's no twap oracle for this pair");

    uint256 neededAmount = ITwapOraclePriceFeed(twapOraclePriceFeed).consult(_token1, _desiredAmount);
    neededAmount = (neededAmount * discountPercentage) / 100;
    return neededAmount;
}
```

Figure 1 source code of related functions (old)

Fix recommendations: It is recommended to set the modification limit to 100.

Fix results: Fixed.

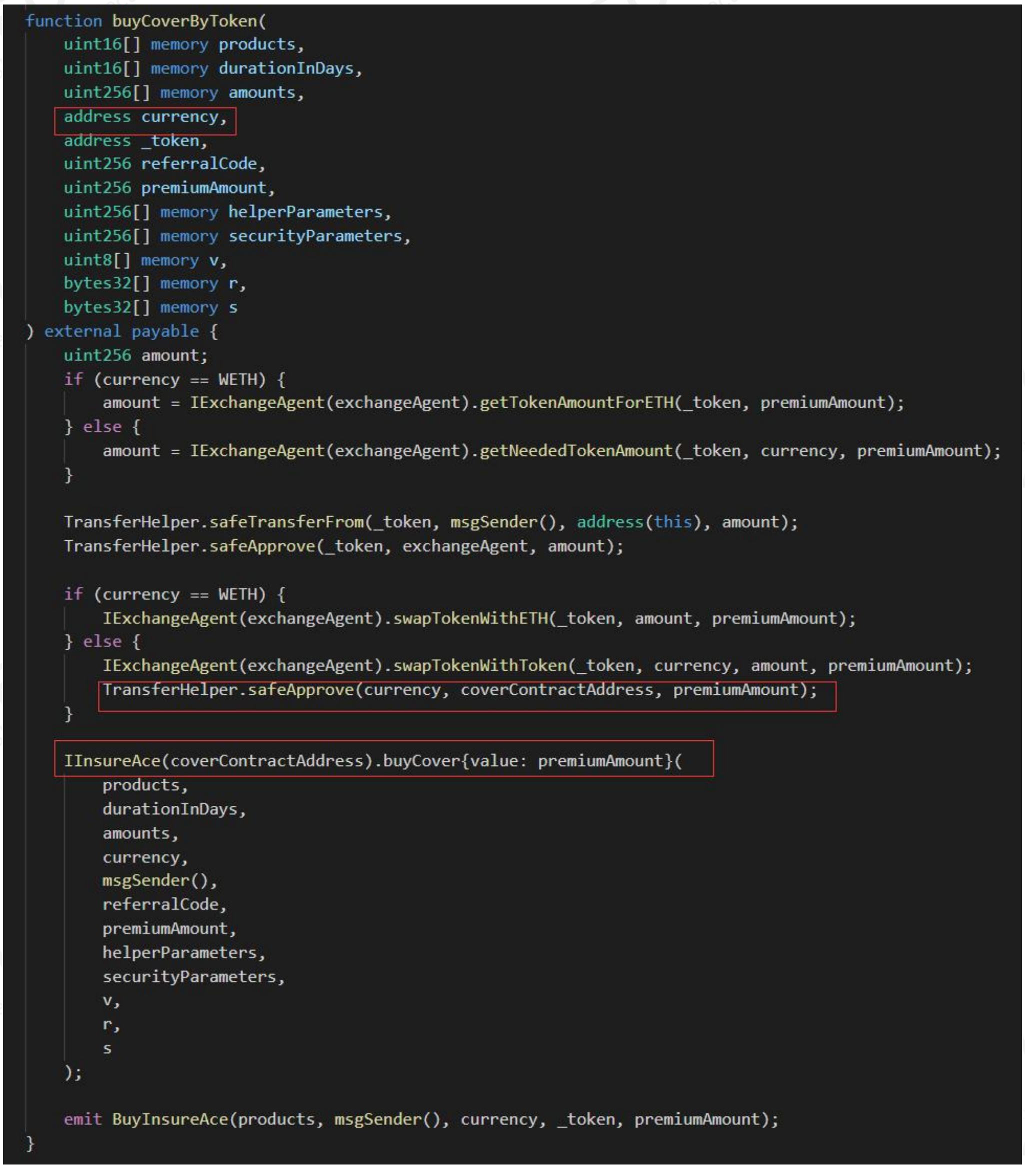


```
function setDiscountPercentage(uint256 _discountPercentage) external onlyOwner {
    require(_discountPercentage <= 100, "Exceeded value");
    discountPercentage = _discountPercentage;
}

/**
 * @dev Get needed _token0 amount for _desiredAmount of _token1
 * _desiredAmount should consider decimals based on _token1
 */
function _getNeededTokenAmount(
    address _token0,
    address _token1,
    uint256 _desiredAmount
) private view returns (uint256) {
    address pair = IUniswapV2Factory(UNISWAP_FACTORY).getPair(_token0, _token1);
    require(pair != address(0), "There's no pair");

    address twapOraclePriceFeed = ITwapOraclePriceFeedFactory(TWAP_ORACLE_PRICE_FEED_FACTORY).getTwapOraclePriceFeed(
        _token0,
        _token1
    );

    require(twapOraclePriceFeed != address(0), "There's no twap oracle for this pair");

    uint256 neededAmount = ITwapOraclePriceFeed(twapOraclePriceFeed).consult(_token1, _desiredAmount);
    if (_token0 == CVR_ADDRESS) {
        neededAmount = (neededAmount * discountPercentage) / 100;
    }

    return neededAmount;
}
```

Figure 2 source code of related functions (latest)

[Cover-2 Medium] Implementing Exceptions

Description: In the InsureAceCover contract, when using the *buyCoverByToken* function, if the currency is not weth, send the coverContractAddress to the premiumAmount quantity of eth instead of currency, so that the currency obtained and authorized by the exchange is meaningless.

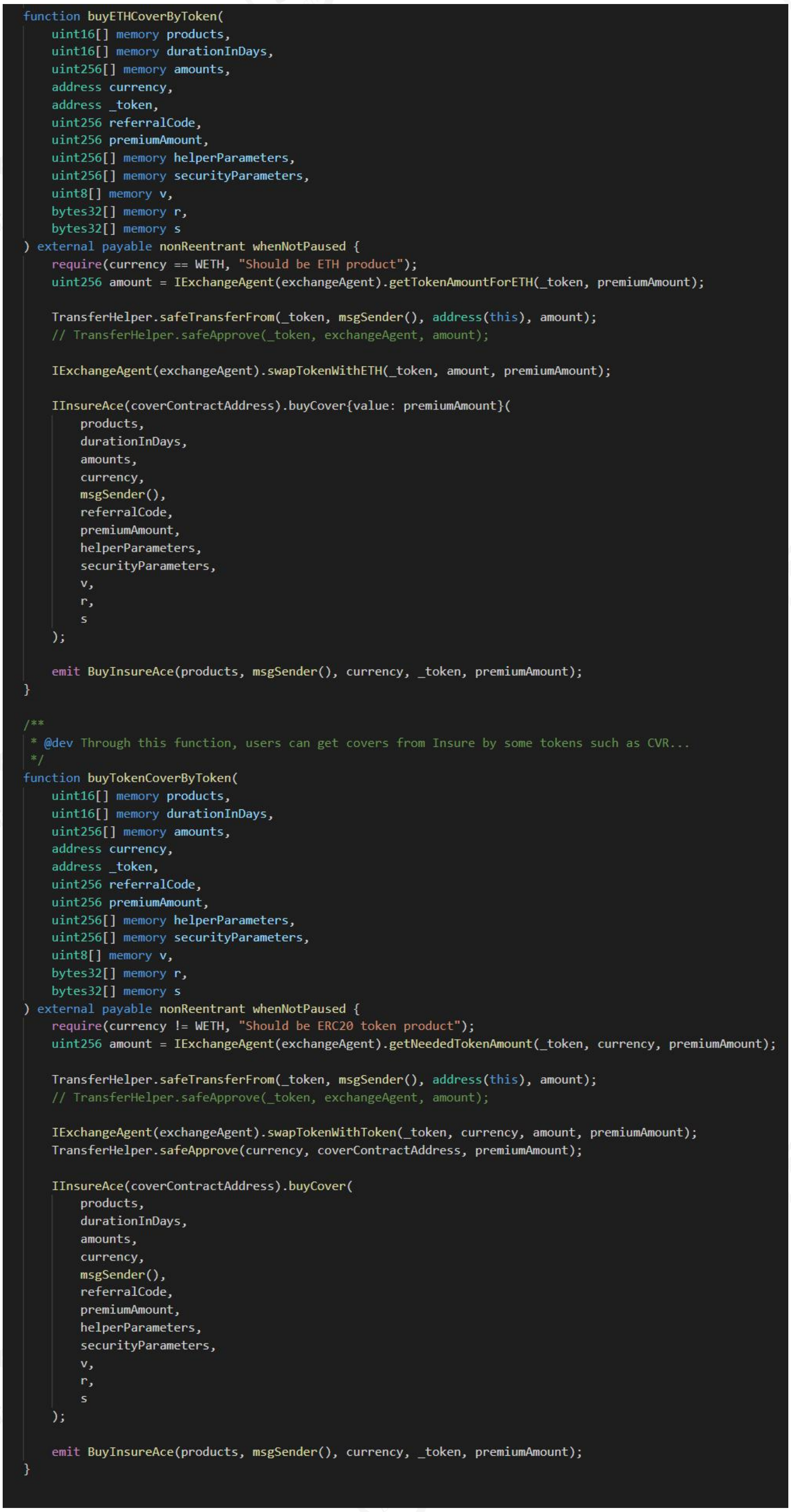


```
function buyCoverByToken(
    uint16[] memory products,
    uint16[] memory durationInDays,
    uint256[] memory amounts,
    address currency,
    address _token,
    uint256 referralCode,
    uint256 premiumAmount,
    uint256[] memory helperParameters,
    uint256[] memory securityParameters,
    uint8[] memory v,
    bytes32[] memory r,
    bytes32[] memory s
) external payable {
    uint256 amount;
    if (currency == WETH) {
        amount = IExchangeAgent(exchangeAgent).getTokenAmountForETH(_token, premiumAmount);
    } else {
        amount = IExchangeAgent(exchangeAgent).getNeededTokenAmount(_token, currency, premiumAmount);
    }

    TransferHelper.safeTransferFrom(_token, msgSender(), address(this), amount);
    TransferHelper.safeApprove(_token, exchangeAgent, amount);

    if (currency == WETH) {
        IExchangeAgent(exchangeAgent).swapTokenWithETH(_token, amount, premiumAmount);
    } else {
        IExchangeAgent(exchangeAgent).swapTokenWithToken(_token, currency, amount, premiumAmount);
        TransferHelper.safeApprove(currency, coverContractAddress, premiumAmount);
    }

    IInsureAce(coverContractAddress).buyCover{value: premiumAmount}(
        products,
        durationInDays,
        amounts,
        currency,
        msgSender(),
        referralCode,
        premiumAmount,
        helperParameters,
        securityParameters,
        v,
        r,
        s
    );

    emit BuyInsureAce(products, msgSender(), currency, _token, premiumAmount);
}
```

Figure 3 source code of related functions (old)

Fix recommendations: It is recommended to add judgment currency to make *buyCover* execute correctly.

Fix results: Fixed. The original *buyCoverByToken* function is split into *buyETHCoverByToken* function and *buyTokenCoverByToken* function for different cases.

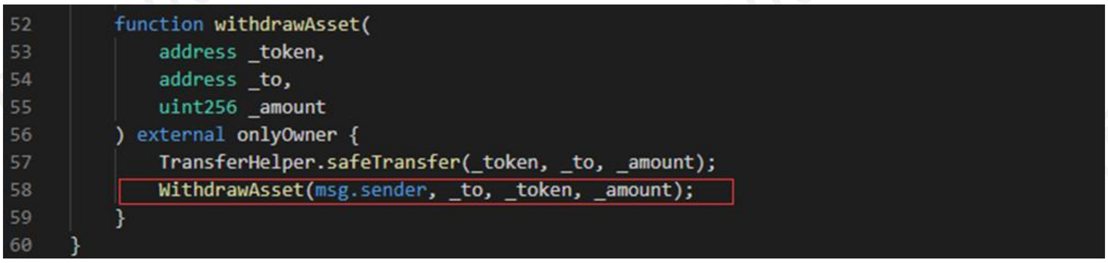


```
function buyETHCoverByToken(
    uint16[] memory products,
    uint16[] memory durationInDays,
    uint256[] memory amounts,
    address currency,
    address _token,
    uint256 referralCode,
    uint256 premiumAmount,
    uint256[] memory helperParameters,
    uint256[] memory securityParameters,
    uint8[] memory v,
    bytes32[] memory r,
    bytes32[] memory s
) external payable nonReentrant whenNotPaused {
    require(currency == WETH, "Should be ETH product");
    uint256 amount = IExchangeAgent(exchangeAgent).getTokenAmountForETH(_token, premiumAmount);

    TransferHelper.safeTransferFrom(_token, msgSender(), address(this), amount);
    // TransferHelper.safeApprove(_token, exchangeAgent, amount);

    IExchangeAgent(exchangeAgent).swapTokenWithETH(_token, amount, premiumAmount);

    IInsureAce(coverContractAddress).buyCover{value: premiumAmount}(
        products,
        durationInDays,
        amounts,
        currency,
        msgSender(),
        referralCode,
        premiumAmount,
        helperParameters,
        securityParameters,
        v,
        r,
        s
    );

    emit BuyInsureAce(products, msgSender(), currency, _token, premiumAmount);
}

/**
 * @dev Through this function, users can get covers from Insure by some tokens such as CVR...
 */
function buyTokenCoverByToken(
    uint16[] memory products,
    uint16[] memory durationInDays,
    uint256[] memory amounts,
    address currency,
    address _token,
    uint256 referralCode,
    uint256 premiumAmount,
    uint256[] memory helperParameters,
    uint256[] memory securityParameters,
    uint8[] memory v,
    bytes32[] memory r,
    bytes32[] memory s
) external payable nonReentrant whenNotPaused {
    require(currency != WETH, "Should be ERC20 token product");
    uint256 amount = IExchangeAgent(exchangeAgent).getNeededTokenAmount(_token, currency, premiumAmount);

    TransferHelper.safeTransferFrom(_token, msgSender(), address(this), amount);
    // TransferHelper.safeApprove(_token, exchangeAgent, amount);

    IExchangeAgent(exchangeAgent).swapTokenWithToken(_token, currency, amount, premiumAmount);
    TransferHelper.safeApprove(currency, coverContractAddress, premiumAmount);

    IInsureAce(coverContractAddress).buyCover(
        products,
        durationInDays,
        amounts,
        currency,
        msgSender(),
        referralCode,
        premiumAmount,
        helperParameters,
        securityParameters,
        v,
        r,
        s
    );

    emit BuyInsureAce(products, msgSender(), currency, _token, premiumAmount);
}
```

Figure 4 source code of related functions (latest)

[Cover-3 Info] Non-standard event triggering

Description: BaseCoverOnChain The withdrawalAsset function event trigger keyword is missing from the contract.

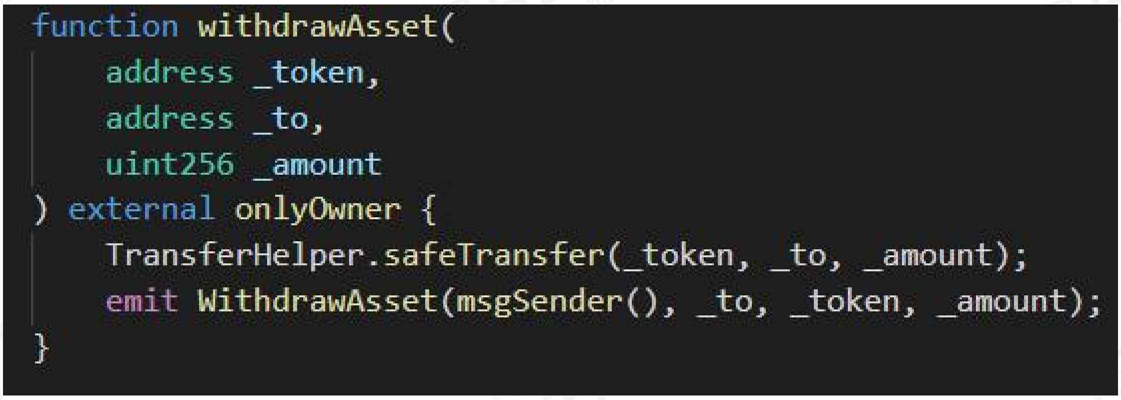


```
    function withdrawAsset(
        address _token,
        address _to,
        uint256 _amount
    ) external onlyOwner {
        TransferHelper.safeTransfer(_token, _to, _amount);
        WithdrawAsset(msg.sender, _to, _token, _amount);
    }
}
```

Figure 5 source code of related functions (old)

Fix recommendations: Add the emit keyword.

Fix results: Fixed.



```
function withdrawAsset(
    address _token,
    address _to,
    uint256 _amount
) external onlyOwner {
    TransferHelper.safeTransfer(_token, _to, _amount);
    emit WithdrawAsset(msgSender(), _to, _token, _amount);
}
```

Figure 6 source code of related functions (latest)

[Cover-4 Info] Owner Privilege issues

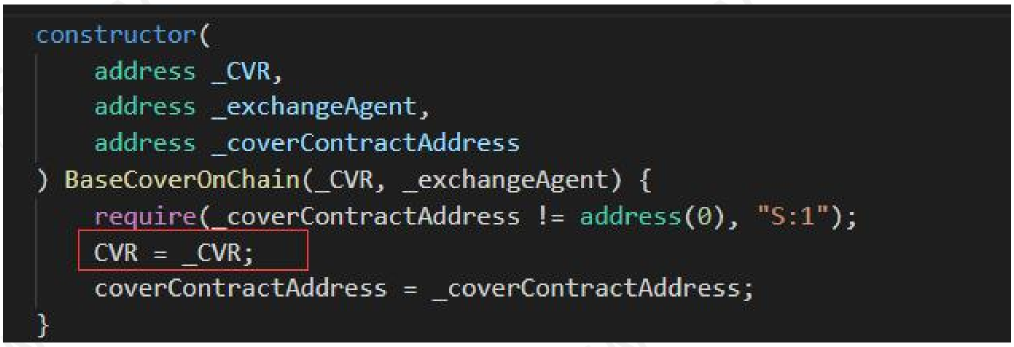
Description: The owner of the project has high privileges, such as adding whitelist and modifying available tokens, etc.

Fix recommendations: It is recommended that the project owner keep the private key or use TimeLock or DAO to manage the owner's privileges.

Fix results: Acknowledged.

[Cover-5 Info] Redundant codes

Description: The variable CVR in InsureAceCover has no effect and is redundant code.

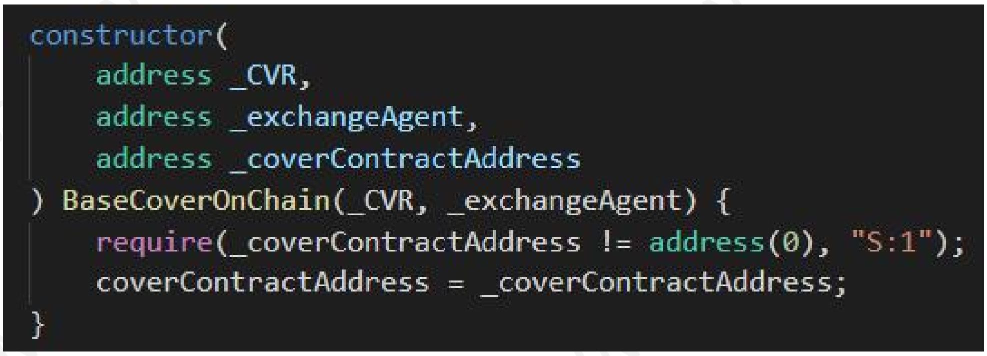


```
constructor(
    address _CVR,
    address _exchangeAgent,
    address _coverContractAddress
) BaseCoverOnChain(_CVR, _exchangeAgent) {
    require(_coverContractAddress != address(0), "S:1");
    CVR = _CVR;
    coverContractAddress = _coverContractAddress;
}
```

Figure 7 source code of related functions (old)

Fix recommendations: It is recommended to delete.

Fix results: Fixed.



```
constructor(
    address _CVR,
    address _exchangeAgent,
    address _coverContractAddress
) BaseCoverOnChain(_CVR, _exchangeAgent) {
    require(_coverContractAddress != address(0), "S:1");
    coverContractAddress = _coverContractAddress;
}
```

Figure 8 source code of related functions (latest)

[Cover-6 Info] Pause module

Description: The project uses many external contract interfaces that are not in the scope of this audit, and there is no pause mechanism.

Fix recommendations: It is recommended to introduce a pause module to avoid further losses when the contract runs abnormally.

Fix results: Fixed.

[Cover-7 Info] Unnecessary payable attributes

Description: The following functions do not require payable attributes: the *buyETHCoverByToken* function and *buyTokenCoverByToken* function in the InsureAceCover contract, and the *buyCoverByToken* token function in the NexusMutualCover contract. The BaseCoverOffChain contract implements *receive*, but does not have a function to extract eth.

Fix recommendations: To avoid user misuse, it is recommended to delete.

Fix results: Fixed.

Appendix 1 Vulnerability Severity Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

BEOSIN

Blockchain Security

Official Website

https://beosin.com

Twitter

https://twitter.com/Beosin_com