DetectBox
Powered by UNSNARL

#DECENTRALISED PROACTIVE AUDIT WAR-ROOMS

Smart Contract Protocol Security Assessment

DROPZONE KOMET

AUGUST 22, 2023

TABLE OF CONTENTS

SUMMARY

This report has been prepared for Komet Technology, Inc to discover issues and vulnerabilities in the source code of DropZone as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live

CERTIFICATION

This is to Certify that the **DropZone Security Report** was prepared using gold standard web3 security standards with all standard and advanced checks deployed.

The **Main Auditor** Involved in the Audit was -
Zuhaib Mohammed(@0xzoobi)

The **Detect-Warden** involved in the Audit was -
JMariadlcs(@devScrooge)

The Auditor mentioned hereby was chosen after self due-dilligence from the project's end.

All reported issues were either acknowledged or fixed by the project.

DetectBox's audit mitigation & residual vulnerability check did not find any major vulnerability to report within the Scope and requirements of the audit.

From
Team DetectBox

DROPZONE
Security Assessment Report

ABOUT DETECTBOX

DetectBox by **UNSNARL** brings to you world's first Decentralised Audit War-Rooms. An intense collaborative Environment where chosen auditors/audit team, project's decision makers, Projects developers and Detect-Wardens join hands to conduct rigorous security audits with absolute transparency followed by a **Proof of Audit (POA).**

We at DetectBox not only find bugs in smart-contracts but also take a deep look into the project's **tokenomics**, **white-paper**, **business logic**, **functional flows** and overall development health.

At DetectBox, you get to choose from a pool of Independent Security researchers which is curated by strong due-diligence. List your project requirements, verify the auditor's past work from their profiles and choose the auditor that rightly fits your project's needs and your budget.

Our auditors have successfully completed **200+ audits**, found over **2200+ vulnerabilities** and secured over **$102M+ worth of TVL.**

To know more about us visit - detectbox.io
To see our docs - https://unsnarl.gitbook.io/detectbox/

Yours Securely
UNSNARL

OVERVIEW

Project Summary

DropZone by Komet is a smart contract protocol which helps in sending ETH, ERC20 , ERC712 and ERC1155 tokens in batch. In exchange for the service there is service the user needs to pay the fee.

Audit Summary

A time-boxed independent security assessment of the Komet DropZone contract was done by Zuhaib Mohammed(@zuhaib44), JMariadIcs(@devScrooge) and Team DetectBox with a focus on the security aspects of the application's implementation.
We performed the security assessment based on the agreed scope, following our approach and methodology. Based on our scope and our performed activities, our security assessment revealed 1 Critical, 3 High severity, 2 Medium severity and 6 Low severity security issues. Additionally, 3 Informational and 1 Gas suggestion was also made which, if resolved appropriately, may improve the quality of the Project's Smart contract.

Audit Timeline: 1st August'23 - 8th August'23

Code Repository:
https://github.com/vedant77/dropzone

Review commit hash:
c71608c1ba1537b2e6249b351e1f3b26caf6d6a6

OVERVIEW

Audit Methodology

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough functional testing and meticulous manual code reviews. To ensure comprehensive issue coverage, we employ checklists derived from industry best practices and widely recognized concerns, specifically tailored to Solidity smart contract assessment.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. Code Quality: We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. Best Practices: Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. Documentation and Comments: We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behaviour of the contract.

Auditing smart contracts involves a comprehensive analysis of the code to identify potential vulnerabilities and security risks. To achieve comprehensive coverage, we employ a series of security checklist tables, each addressing specific areas of concern. These include:

- System / Platform
- Access Control
- Storage

OVERVIEW

- Gas Issues and Efficiency
- Code Issues
- Error Handling and Exception Handling:
- Transaction Handling
- Entrypoint Validation
- Administration and Operator Functions
- Additional Topics and Test Cases

Vulnerability Summary

Severity classification

Severity	Impact : High	Impact : Medium	Impact : Low
Likelihood : High	Critical	High	Medium
Likelihood : Medium	High	Medium	Low
Likelihood : Low	Medium	Low	Low

Findings Summary

Critical	1 issue
High	3 issues
Medium	2 issues
Low	6 issues
Informational	3 issues
Gas Optimisations	1 issue

OVERVIEW

Audit Scope

The code under review is two smart contracts written in the Solidity language and includes 463 LOC- lines of code.

TypeFile	Lines
contracts/KometBundlingUpgradable.sol	409
contracts/StructHelper.sol	54

Auditors Involved :

Main Auditor -

Zuhaib Mohammed- https://app.detectbox.io/profile/0xzoobi

Detect Warden -

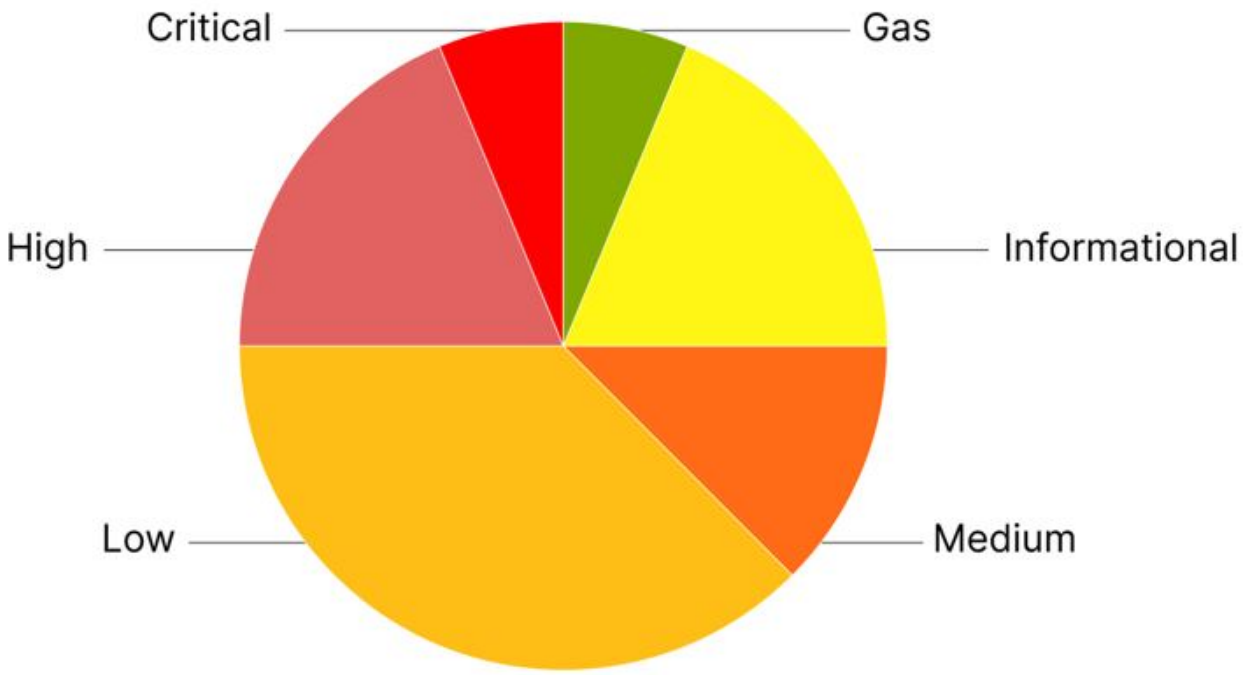
JMariadlcs - https://app.detectbox.io/profile/devScrooge

FINDINGS

Detailed Summary of Findings

Sl. No.	Name	Severity
C-01	Protocol fees are not correctly implemented	Critical
H-01	Usage of an incorrect version of Ownable library can potentially malfunction all onlyOwner functions	High
H-02	Signature malleability of EVM's ecrecover	High
H-03	Decimals value can be manipulated	High
M-01	No Storage Gap for Upgradeable Contracts	Medium
M-02	Possible DOS (out-of-gas) on for loops	Medium
L-01	call() should be used instead of transfer() on an address payable	Low
L-02	Use SafeTransfer instead of transfer	Low
L-03	Usage of an incorrect version of SafeERC20 library can potentially malfunction all ERC20 functions	Low
L-04	Front-runnable Initializers	Low
L-05	Floating Pragma Solidity Version	Low
L-06	Missing event for important parameter change	Low
I-01	Missing Revert Message in the withdraw function	Informational
I-02	Unused internal function	Informational
I-03	The require check in executeTx needs to be updated	Informational
G-01	bytes4 conversion can be directly done	Gas

FINDINGS



Static Analysis

No major issues were found. Some false positive errors were reported by the tool. All the other issues have been categorized above according to their level of severity.

Manual Review

Critical Severity Issues

C-01. Protocol fees are not correctly implemented

FINDINGS

Description :

The getPlatformFee function is supposed to be used for the protocol to get fees on each transaction. However the function is now working as intended.
As it can be seen, the function increases the value of user_fees_nativetoken adding the whole msg.value and not a percentage of the tx value. This function neither discounts the protocol fees from the tx value so that when the withdraw transaction is performed, the balance of the contract is likely to be 0.

Code Snippets :

```
function getPlatformFee(
    uint256 _amount⬆,
    bool _isNative⬆,
    uint256 decimals⬆
) internal {
    uint256 decimal_amount = _amount⬆ / 10**decimals⬆;
    if (decimal_amount >= fee_limit) {
        if (_isNative⬆) {
            require(
                msg.value >= _amount⬆ + platform_fee,
                "platform fee required"
            );
        } else {
            require(msg.value >= platform_fee, "platform fee required");
        }

        user_fees_nativetoken[msg.sender] += msg.value;
        emit platformFee(msg.sender, msg.value);
    }
}
```

FINDINGS

```
function withdraw() public {
    _onlyOwner();
    balance = 0;
    (bool success, ) = msg.sender.call{value: address(this).balance}("");
    require(success);
}
```

Recommendations:

Change the behaviour of the function to take a percentage of the fee and discount it from the tx value transferred so that when the withdrawal is executed there is actual balance.

High Severity Issues

H-01. Usage of an incorrect version of Ownable library can potentially malfunction all onlyOwner functions

Description:

The **KometBundlingUpgradable.sol** contract is designed to be deployed as an upgradeable proxy contract.

However, the current implementation is using an non-upgradeable version of the Ownable library: **@openzeppelin/contracts/access/Ownable.sol** instead of the upgradeable version: **@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol**

FINDINGS

A regular, non-upgradeable Ownable library will make the deployer the default owner in the constructor. Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts. Therefore, there will be no owner when the contract is deployed as a proxy contract.
As a result, all the **onlyOwner** functions will be inaccessible.

Code Snippets :

```
contract KometBundlingUpgradable is
    StructHelper,
    UUPSUpgradeable,
    Initializable,
    Ownable
```

Recommendations:

Use **@openzeppelin/contracts-upgradeable/accessOwnableUpgradeable.sol** and **@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol** instead.
And modify the initialize() function as shown below :

```
    function initialize() public virtual initializer {
        __Ownable_init();
        _initialize();
}
```


FINDINGS

H-02. Signature malleability of EVM's ecrecover

Description:

The built-in EVM precompile ecrecover is susceptible to signature malleability which could lead to replay attacks (references: https://swcregistry.io/docs/SWC-117,https://swcregistry.io/docs/SWC-121 &https://medium.com/cryptronics/signature-replay-vulnerabilities-in-smart-contracts-3b6f7596df57).

Code Snippets :

```
);
require(msg.sender != address(0), "invalid-address");
require(msg.sender == ecrecover(digest, v↑, r↑, s↑), "invalid-signature");
bytes4 selector = bytes4(data↑);
```

Recommendations:

Consider using OpenZeppelin's ECDSA library: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol

H-03. Decimals value can be manipulated

Description:

In getPlatformFee the third parameter is decimals which is used to calculate decimal_amount and then later used to inside the if statement or not. The problem is that this decimal variable is directly set up by the user and can be manipulated to not represent the correct decimal of the token.

FINDINGS

Code Snippets :

```
ftrace | funcSig
function getPlatformFee(
    uint256 _amount⬆,
    bool _isNative⬆,
    uint256 decimals⬆
) internal {
    uint256 decimal_amount = _amount⬆ / 10 ** decimals⬆;
```

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract collectFeePoC {


    uint256 fee_limit = 100;
    //Imagine the user wants to send 10000 USDT
    //USDT has 8 decimals places
    uint256 public amount = 10000 * 10 ** 8;
    //malicious user decides to pass decimals as 18 instead of 8
    uint256 public decimals = 18;
    uint256 public decimal_amount = amount / 10 ** decimals;

    // Since the decimal_amount is set to zero, the if block to collect the fee is skipped
    // if(decimal_amount >= fee_limit) {
    //     //code to collect fee
    // }
}
```

Recommendations:

Instead of allowing user to pass the decimal you can use the below code snippet to get the decimal place of the contract.

```
uint decimal_places = IERC20(contract_address).decimals()
//use this and pass it on to the getPlatformFee() function
```

FINDINGS

Medium Severity Issues

M-01. No Storage Gap for Upgradeable Contracts

Description:

For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments". Otherwise it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts.

Code Snippets :

```
uint256 public platform_fee = 10000000000000000; //0.01 ether
uint256 internal fee_limit = 100;

ftrace | funcSig
function _onlyOwner() internal view {
```

Recommendations:

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

FINDINGS

M-02. Possible DOS (out-of-gas) on for loops

Description:

The for loop in the functions **batchTransferNFTForUser**, **batchTransferERC1155**,**transferNativeCurrency**,**batchTransferERC20TokensSingle** is unbounded.
This basically means there is no check for the length of the array of the for loop. Missing this check poses a risk of out-of-gas resulting in reverting of the transaction..

Code Snippets :

```
TransferItemSingle memory item;
for (uint256 i = 0; i < length; ) {
    item = items[i];
    ierc20.safeTransferFrom(msg.sender, item.to, uint256(item.value));
    unchecked {
        ++i;
    }
}
```

```
for (uint256 i = 0; i < length; i++) {
    item = items[i];
    ierc1155.safeBatchTransferFrom(
        msg.sender,
        item.to,
        item.ids,
        item.amounts,
        item.data
    );
}
```

```
TransferItemSingle memory item;
for (uint256 i = 0; i < length; ) {
    item = items[i];
    ierc20.safeTransferFrom(msg.sender, item.to, uint256(item.value));
    unchecked {
        ++i;
    }
}
```

FINDINGS

```
for (uint256 i = 0; i < items⬆.length; i++) {
    item = items⬆[i];
    user_balances[msg.sender] = SafeMath.sub(
        user_balances[msg.sender],
        item.value
    );
    payable(item.to).transfer(item.value);
}
```

Recommendations:

A good check to prevent this from happening is to define a MAX_LENGTH variable and check if the current array length exceeds that. The dev can do proper unit testing for this, and find the optimum value that can be set for MAX_LENGTH.

Low Severity Issues

L-01. call() should be used instead of transfer() on an address payable

Description:

The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

FINDINGS

The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.
- Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Code Snippets :

```
for (uint256 i = 0; i < items⬆.length; i++) {
    item = items⬆[i];
    user_balances[msg.sender] = SafeMath.sub(
        user_balances[msg.sender],
        item.value
    );
    payable(item.to).transfer(item.value);
}
```

Recommendations:

Use call() instead of transfer()

```
( bool success, ) = payable(item.to).call{value: item.value}("");
require(success, "Transfer Failed");
```

FINDINGS

L-02. Use SafeTransfer instead of transfer

Description:

The SafeERC20 library is already being used in the contract but missing as part of the withdrawTokens.
Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked.

Code Snippets :

```
IERC20(_token).transfer(
    msg.sender,
    getTokenBalance(address(this), _token)
);
```

Recommendations:

Use the safeTransfer when you want to withdraw tokens from the contract.

```
IERC20(_token).safeTransfer(
    msg.sender,
    getTokenBalance(address(this), _token)
);
```

FINDINGS

L-03. Usage of an incorrect version of SafeERC20 library can potentially malfunction all ERC20 functions

Description:

Based on the context and comments in the code, the **KometBundlingUpgradable.sol** contract is designed to be deployed as an upgradeable proxy contract.
However, the current implementation is using an non-upgradeable version of the SafeERC20 library: @openzeppelin/contracts/access/SafeERC20.sol instead of the upgradeable version: @openzeppelin/contracts-upgradeable/access/SafeERC20Upgradeable.sol.
Similarly, other imports also belong to the @openzeppelin/contracts instead of **@openzeppelin/contracts-upgradeable**

Code Snippets :

```
import "./StructHelper.sol";
import "@openzeppelin/contracts/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts/proxy/utils/UUPSUpgradeable.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
```

Recommendations:

Import the libraries from @openzeppelin/contracts-upgradeable/

FINDINGS

L-04. Front-runnable Initializers

Description:

The contract's initializer is missing access controls, allowing any user to initialize the contract. By front-running the contract deployers to initialize the contract, the user can pass his/her address and get gain the ownership of the contract.

Code Snippets :

```
function initialize(address anOwner) public virtual initializer {
    _initialize(anOwner);
}
```

Recommendations:

While the code that can be run in contract constructors is limited, setting the owner in the contract's constructor to the msg.sender or tx.origin and adding the onlyOwner modifier to the initializer would be a sufficient level of access control.

L-05. Floating Pragma Solidity Version

Description:

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

FINDINGS

Code Snippets :

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;
```

Recommendations:

Consider upgrading all contracts to Solidity version 0.8.16 at a minimum, but ideally to the latest version.

L-06. Missing event for important parameter change

Description:

Important parameter or configuration changes should trigger an event to allow being tracked off-chain.

Code Snippets :

```
function withdraw() public {
    _onlyOwner();
    balance = 0;
    (bool success, ) = msg.sender.call{value: address(this).balance}("");
    require(success);
}
```

```
function setFees(bool _approval⬆, uint256 _fee⬆, uint256 _feeLimit⬆) public {
    _onlyOwner();
    paid_using_native_currency = _approval⬆;
    platform_fee = _fee⬆;
    fee_limit = _feeLimit⬆;
}
```

Recommendations:

Emit Events for important parameter change

FINDINGS

Informational Issues

I-01. Missing Revert Message in the withdraw function

Description:

The require statement in the withdraw function lacks a revert message. As a result, when a tx fails it will be easily to analyse to add one.

Code Snippets :

```
function withdraw() public {
    _onlyOwner();
    balance = 0;
    (bool success, ) = msg.sender.call{value: address(this).balance}("");
    require(success);
}
```

Recommendations:

Add a require statement to the withdraw function

```
(bool success, ) = msg.sender.call{value: address(this).balance}("");
require(success, "Transfer Failed");
```

I-02. Unused internal function

Description:

The pushItem function is an internal function that is not used in any part of the code.

FINDINGS

Code Snippets :

```
function pushItem(
    TransferItemSingle[] memory array↑,
    TransferItemSingle memory item↑
) internal pure returns (TransferItemSingle[] memory) {
    uint256 length = array↑.length;
    TransferItemSingle[] memory newArray = new TransferItemSingle[](
        length + 1
    );
    for (uint256 i = 0; i < length; i++) {
        newArray[i] = array↑[i];
    }
    newArray[length] = item↑;
    return newArray;
}
```

Recommendations:

Remove the function.

I-03. The require check in executeTx needs to be updated

Description:

The check require(msg.sender != address(0), "invalid-address"); is not a required check as it basically checks if the msg.sender is address(0). There is no security risk associated with the risk. I suggest making the changes suggested in Recommendation section.

FINDINGS

Code Snippets :

```
require(msg.sender != address(0), "invalid-address");
require(msg.sender == ecrecover(digest, v↑, r↑, s↑), "invalid-signature");
```

Recommendations:

Update the require check as shown below.

```
require(ECDSA.recover(digest, v, r, s) != address(0), "invalid-signature");

require(ECDSA.recover(digest, v, r, s) == msg.sender, "invalid-signature");
```

Gas Optimization

G-01. bytes4 conversion can be directly done

Description:

There are several constant internal variables that performs a bytes4(keccak256(string)) operation that can be directly computed offchain and set as bytes to save gas.

FINDINGS

Code Snippets :

```
bytes4 internal constant BATCH_TRANSFER_NFT_FOR_USER =
    bytes4(
        keccak256(
            "batchTransferNFTForUser((address,address,address,uint256)[])"
        )
    );
bytes4 internal constant BATCH_TRANSFER_ERC20_SINGLE =
    bytes4(
        keccak256(
            "batchTransferERC20TokensSingle((address,uint96)[],uint256,address,uint256)"
        )
    );
bytes4 internal constant BATCH_TRANSFER_NATIVE_CURRENCY =
    bytes4(keccak256("transferNativeCurrency((address,uint96)[],uint256)"));
bytes32 internal constant EIP712_DOMAIN_TYPEHASH =
    keccak256(
        bytes(
            "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
        )
    );
bytes32 internal constant META_TRANSACTION_TYPEHASH =
    keccak256(bytes("MetaTransaction(uint256 nonce,address from)"));
bytes32 internal DOMAIN_SEPARATOR;
bytes4 internal constant BATCH_TRANSFER_ERC1155 =
    bytes4(
        keccak256(
            "batchTransferERC1155((address,uint256[],uint256[],bytes)[],address)"
        )
    );
```

Recommendations:

Compute the bytes off-chain and directly set them as the internal constant variable to save gas.

POST AUDIT CONCLUSION

Fixing the Findings

Sl. No.	Name	Status
C-01	Protocol fees are not correctly implemented	Fixed
H-01	Usage of an incorrect version of Ownable library can potentially malfunction all onlyOwner functions	Fixed
H-02	Signature malleability of EVM's ecrecover	Fixed
H-03	Decimals value can be manipulated	Fixed
M-01	No Storage Gap for Upgradeable Contracts	Fixed
M-02	Possible DOS (out-of-gas) on for loops	Fixed
L-01	call() should be used instead of transfer() on an address payable	Fixed
L-02	Use SafeTransfer instead of transfer	Fixed
L-03	Usage of an incorrect version of SafeERC20 library can potentially malfunction all ERC20 functions	Acknowledged
L-04	Front-runnable Initializers	Acknowledged
L-05	Floating Pragma Solidity Version	Fixed
L-06	Missing event for important parameter change	Fixed
I-01	Missing Revert Message in the withdraw function	Fixed
I-02	Unused internal function	Fixed
I-03	The require check in executeTx needs to be updated	Fixed
G-01	bytes4 conversion can be directly done	Fixed

APPENDIX

Finding Categories

Finding Categories

Centralization/Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

DISCLAIMER

DetectBox (by UNSNARL) has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. UNSNARL and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

DISCLAIMER

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. UNSNARL is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. UNSNARL's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

UNSNARL in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will UNSNARL, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

DISCLAIMER

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

UNSNARL will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

UNSNARL will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.

GLORY OF AUDITORS AT DETECTBOX

DetectBox has created a pool of best auditors across the globe with significant experience in the web3 security industry auditing multiple protocols across multiple chains. Making audits better through **Proof of Talent**.

217+

Projects Audited

$100M+

Amount Secured

2200+

Vulnerabilities Detected

Follow Our Journey on

@unsnarl_secure

UNSNARL

founders@unsnarl.io

www.detectbox.io