SuperDAPP

Smart Contract Audit

DATE: 26th April 2024 | PREPARED BY: Entersoft Australia

Contents

Revision History & Version Control

Start Date	End Date	Author	Comments/Details
16 Apr 2024	26 Apr 2024	Gurkirat	Interim Report for the Client

Reviewed by	Released by
Nishita Palaksha	Nishita Palaksha

Entersoft was commissioned to perform a source code review on SuperDAPP's solidity smart contracts. The review was conducted between April 16, 2024, to April 26, 2024. The report is organized into the following sections.

- Executive Summary: A high-level overview of the security audit findings.
- Technical analysis: Our detailed analysis of the Smart Contract code

The information in this report should be used to understand overall code quality, security, correctness, and meaning that code will work as described in the smart contract.

1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to: (i) smart contract best coding practices and vulnerabilities in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

2.0 Overview

2.1 Project Overview

During the period of **16 Apr 2024 to 26 Apr 2024**, Entersoft performed smart contract security audits for **Superdapp**.

2.2 Scope

The scope of this audit was to analyze and document the smart contract codebase for quality, security, and correctness.

The following files were reviewed as part of the scope:

- SuperDapp.sol
- GroupMembership.sol

Contract Address - 0x00f8Da33734FeB9b946fEC2228C25072D2e2E41f

OUT-OF-SCOPE: External contracts, External Oracles, other smart contracts in the repository, or imported smart contracts.

2.3 Project Summary

Project Name	No. of Smart Contract File(s)	Verified	Vulnerabilities
Superdapp	1	Yes	As per report. Section 2.6

2.4 Audit Summary

Delivery Date	Method of Audit	Consultants Engaged
26 Apr 2024	Manual and Automated approach	3

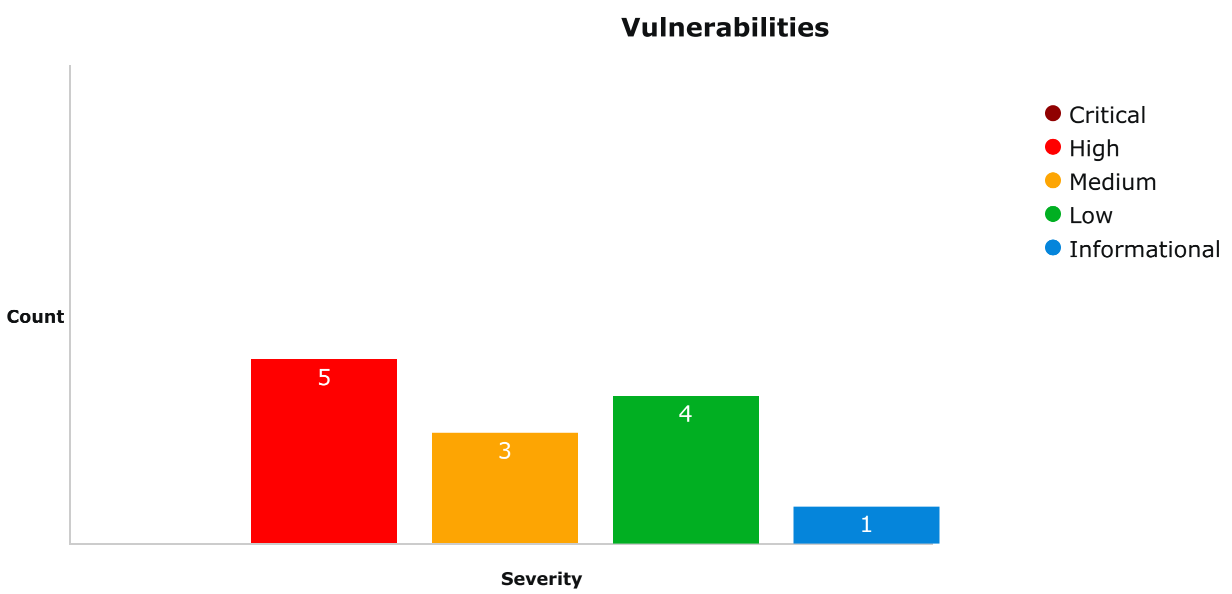
2.5 Security Level References

Every vulnerability in this report was assigned a severity level from the following classification table:

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

2.6 Vulnerability Summary

● Critical	● High	● Medium	● Low	● Informational
0	5	3	4	1



3.0 Executive Summary

Entersoft has conducted a comprehensive technical audit of the SuperDapp smart contract through a comprehensive smart contract audit approach. The primary objective was to identify potential vulnerabilities and security risks within the codebase, ensuring adherence to industry-leading standards while prioritizing security, reliability, and performance.

Our focus was on prompt and efficient identification and resolution of vulnerabilities to enhance the overall robustness of the solidity smart contract. Importantly, our audit process intentionally avoided reliance solely on automated tools, emphasizing a more in-depth and nuanced approach to security analysis.

Conducted from April 16, 2024, to April 26, 2024, our team diligently assessed and validated the security posture of the solidity smart contract, ultimately finding a number of vulnerabilities as per vulnerability summary table.

Testing Methodology:
We have leveraged static analysis techniques extensively to identify potential vulnerabilities automatically with the aid of cutting-edge tools such as Slither and Aderyn. Apart from this, we carried out extensive manual testing to iron out vulnerabilities that could slip through an automated check. This included a variety of attack vectors like reentrancy attacks, overflow and underflow attacks, timestamp dependency attacks, and more.
While going through the due course of this audit, we also ensured to cover edge cases, and built a combination of scenarios to assess the contracts' resilience. Our attempt to leave no stone unturned involved coming up with both negative and positive test cases for the system, and grace handling of stressed scenarios.

Our testing methodology in Solidity adhered to industry standards and best practices, integrating partially implemented OWASP and NIST SP 800 standards for encryption and signatures. Solidity's renowned security practices were complemented by tools such as Solhint for linting, and the Solidity compiler for code optimization. Sol-profiler, Sol-coverage, and Sol-sec were employed to ensure code readability and eliminate unnecessary dependencies.

Findings and Security Posture:
Below is the Attack Vector Coverage

- Reentrancy Vulnerability: Test scenarios are designed to check for reentrancy vulnerabilities, where an attacker can repeatedly call a contract function to exploit its state and potentially drain funds.
- Denial-of-Service DoS Attacks: Dynamic testing aims to detect scenarios that may lead to a contract being stuck in an infinite loop or consuming excessive gas, causing a DoS attack.
- Front-Running Attacks: Dynamic testing explores potential scenarios that could be exploited by front-running attacks, where an attacker leverages timing discrepancies to execute transactions before the intended transaction.
- Logic Flaws and Race Conditions: Testing uncover logic flaws and race conditions that may result from concurrent execution of contract functions or interactions with other contracts.

- Permission and Access Control: Dynamic testing explores different access levels and roles to ensure that permissions are properly enforced, preventing unauthorized access to critical functions or data.

Tools Used for Audit:
In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Slither,aderyn. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. Entersoft takes pride in utilizing these tools, which significantly contribute to the quality, security, and maintainability of our codebase.

Code Review / Manual Analysis:
Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analyzed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

Auditing Approach and Methodologies Applied:
The solidity smart contract was audited in a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of code:

- Code quality and structure: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analyzing the overall architecture of the solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.

- Security vulnerabilities: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, Signatures, and deprecated functions.

3.1 Findings

Vulnerability ID	Contract Name	Severity	Status
1	SuperDapp.sol	High	Pending
2	GroupMembership.sol	High	Pending
3	SuperDapp.sol	High	Pending
4	SuperDapp.sol	High	Pending
5	SuperDapp.sol	High	Pending
6	SuperDapp.sol	Medium	Pending
7	GroupMembership.sol, SuperDapp.sol	Medium	Pending
8	SuperDapp.sol	Medium	Pending
9	SuperDapp.sol	Low	Pending
10	SuperDapp.sol	Low	Pending
11	GroupMembership.sol	Low	Pending
12	SuperDapp.sol	Low	Pending
13	SuperDapp.sol, GroupMembership.sol	Informational	Pending

3.2 Recommendations

Overall, the smart contracts are very well written, and they adhere to best security practices and industry guidelines.

4.0 Technical Analysis

4.1 Non-Compliance with ERC20 Return Values in transfer and transferFrom Functions

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

SuperDapp.sol

Description:

The SuperDAPP smart contract relies on the transfer and transferFrom functions of the ERC20 standard to move tokens between accounts. According to the ERC20 standard, these functions should return a boolean value indicating the success or failure of the operation. However, not all token implementations adhere strictly to this standard. Some tokens, like those mimicking Tether (USDT), might implement these functions without returning any value, which can lead to unexpected reversion of function calls when these return values are assumed.

Locations:

SuperDapp.sol (Functions: deposit, withdraw, transfer)

Remediation:

To mitigate this risk and enhance contract robustness, it is recommended to integrate OpenZeppelin's SafeERC20 library. This library provides safeTransfer and safeTransferFrom functions that handle ERC20 token transfers without assuming boolean return values, thus ensuring compatibility with both compliant and non-compliant tokens.

Impact:

The functions in question do not utilize OpenZeppelin's SafeERC20 library, which is designed to safely interact with ERC20 tokens, especially those not fully compliant with the standard. This poses a risk, particularly when interfacing with non-standard tokens, as the lack of return value handling can cause transactions to fail, leading to state inconsistencies or denial of service, and potentially allowing malicious activities through deliberate transaction failures.

Code Snippet:

NA

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.2 Potential Denial of Service (DoS) Vulnerability in calculateTimeBonus Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

GroupMembership.sol

Description:

The calculateTimeBonus function iterates over an array (bonusesKeys) derived from groupTimeBonusesKeys[groupId] to calculate a bonus based on the avgGroupAge. This function, if invoked with an unusually large array, could consume an excessive amount of gas due to the iteration across all array elements. If an attacker can manipulate or influence the size of bonusesKeys, they might cause the function to exceed the block gas limit. Such a scenario would result in the transaction not being processed, potentially leading to a denial of service and loss of gas fees for the user.

Locations:

GroupMembership.sol, Line: 264,286

Remediation:

To mitigate this risk, consider implementing a maximum limit on the length of the bonusesKeys array that can be processed in a single transaction. Alternatively, restructuring the logic to avoid iterating through potentially large arrays and utilizing more efficient data access patterns could help. Another approach might involve caching the result of expensive computations or optimizing the way groupTimeBonuses and groupTimeBonusesKeys are structured to reduce the need for iterative processes.

Impact:

The function lacks safeguards against large array sizes for bonusesKeys, which is problematic as it can be controlled by the groupId parameter indirectly influencing the array length. The loop processing this array can lead to high gas consumption when the array is large, providing a vector for DoS attacks

Code Snippet:

NA

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-21-unbounded-loops

Proof of Vulnerability:

N.A.

4.3 Potential Denial of Service (DoS) Vulnerability in sellMultipleShares Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

SuperDapp.sol

Description:

The sellMultipleShares function in the smart contract processes an array of token IDs to execute the sellShares function for each ID. This approach can lead to high gas consumption when the input array contains a large number of token IDs. In scenarios where the sellShares operation itself consumes a significant amount of gas, the cumulative gas cost for the entire sellMultipleShares transaction might exceed the block gas limit. This would result in the transaction failing to be mined, leading to a denial of service for the user and potential loss of gas fees incurred during the transaction attempt.Functions not used internally could be marked external

Locations:

SuperDapp.sol, Line: 243,259

Remediation:

To address this vulnerability, it is recommended to introduce limitations on the size of the input array. This can be achieved by setting a maximum allowable array size that balances usability with gas cost considerations. Additionally, implementing a mechanism to process the token IDs in batches—allowing the function to handle larger arrays over multiple transactions—could further mitigate the risk of exceeding gas limits. Implementing checks to ensure the sellShares function itself is optimized for gas efficiency would also be beneficial.

Impact:

The function does not impose checks on the size of the input array tokenId, which leaves it susceptible to DoS attacks when processing excessively large arrays. This could be exploited by an attacker by deliberately triggering the function with a large number of token IDs, thereby forcing excessive gas consumption.

Code Snippet:

NA

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-21-unbounded-loops

Proof of Vulnerability:

N.A.

4.4 Premature Balance Update Vulnerability in deposit Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

SuperDapp.sol

Description:

The deposit function in the smart contract updates the depositor's balance in the mapDeposits mapping before confirming the successful transfer of tokens from the depositor's address to the contract's address. This premature updating of the balance can lead to inconsistencies in the recorded state if the subsequent transferFrom call fails for any reason (e.g., due to insufficient allowance or balance in the depositor's account). Such failures would revert the transaction but not before the state has been optimistically modified.

Locations:

SuperDapp.sol, Line: 176,185

Remediation:

To mitigate this risk, the deposit function should be restructured to follow the checks-effects-interactions pattern strictly. Specifically, the function should first call transferFrom and only update the mapDeposits mapping after this call returns successfully. This change ensures that the contract's state only reflects received funds

Impact:

The function first increases the user's balance in the mapDeposits and then attempts to transfer tokens using the ERC20 transferFrom method. This sequence of operations violates the checks-effects-interactions pattern, where state changes should only occur after all external interactions (and their associated effects) have been successfully completed. The failure to adhere to this pattern can lead to misleading state information if the transaction fails and reverts after the balance update.

Code Snippet:

```
        function deposit(uint256 amount) public {

    require(amount > 0, "Deposit amount must be greater than 0");

    require(

        suprToken.transferFrom(msg.sender, address(this), amount),

        "Deposit failed"

    );

    mapDeposits[msg.sender] = mapDeposits[msg.sender].add(amount);

    emit Deposit(msg.sender, amount);

}
```

Reference:

Proof of Vulnerability:

N.A.

4.5 Reentrancy Vulnerabilities

Severity	Status	Type of Analysis
● High	Identified	Static

Contract Name:

SuperDapp.sol

Description:

The functions buyShares(uint256, uint256) and sellShares(uint256), deposit(uint256), and withdraw() in the SuperDapp contract as contract contain reentrancy vulnerabilities.

Locations:

SuperDapp.sol

Remediation:

To mitigate the reentrancy vulnerabilities, perform state changes after external calls have been made. This ensures that no external calls can interfere with the state changes in progress. Use libraries like openzeppelin reentrancy guard etc.

Impact:

In SuperDapp.buyShares(uint256, uint256), the external call groupMembership.joinGroup(groupId, amount, amount.sub(treasuryFee).sub(subjectFee), msg.sender) is made before state variables tokenInitialPrices[tokenId] and totalBuyAmount[msg.sender] are written. This can lead to reentrancy attacks if the joinGroup function or the _safeMint function inside it calls back into the SuperDapp contract before these state changes are finalized.

Similarly, in SuperDapp.sellShares(uint256), the external call groupMembership.leaveGroup(groupId, tokenId, msg.sender) is made before state variables mapDeposits[deadAddress], mapDeposits[msg.sender], and totalSellAmount[msg.sender] are written. This can also lead to reentrancy attacks if the leaveGroup function calls back into the SuperDapp contract before these state changes are finalized.
deposit(uint256), and withdraw() also expose same vulnerabilities.

Code Snippet:

NA

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1

Proof of Vulnerability:

N.A.

4.6 Address Validation Missing in SuperDapp Contract

Severity	Status	Type of Analysis
Medium	Identified	Static

Contract Name:

SuperDapp.sol

Description:

Assigning values to address state variables without validating the input address may introduce vulnerabilities, especially if the address can be address(0) (zero address).

Locations:

SuperDapp.sol, Line: 59

Remediation:

Implement checks to ensure that the input address is not address(0) before assigning it to state variables. You can add a require statement to ensure this condition is met.

Impact:

The assignment operations suprToken = _suprToken and groupMembershipAddress = _groupMembershipAddress; directly assign the input address _suprToken and _groupMembershipAddress to state variables without any validation.

Code Snippet:

NA

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Proof of Vulnerability:

N.A.

4.7 Functions not used internally could be marked external

Severity	Status	Type of Analysis
Medium	Identified	Static

Contract Name:

GroupMembership.sol, SuperDapp.sol

Description:

Functions that are not called internally within the contract and are intended to be called externally by users or other contracts should be marked as external instead of public for clarity and potentially a slight optimization.

Locations:

- **File:** GroupMembership.sol, **Line:** 82, 87, 92, 112, 183, 233, 237, 241, 245, 251, 257, 288, 312, 329, 355
- **File:** SuperDapp.sol, **Line:** 52, 69, 73, 77, 123, 147, 153, 177, 188, 198, 206, 244, 291

Remediation:

Change the visibility of these functions to external if they are intended to be called from outside the contract. This improves readability and provides a hint to developers about the intended usage.

Impact:

The mentioned functions are declared as public but are not called internally within the contract. They are likely meant to be called externally.

Code Snippet:

```
        function initialize() public initializer {

__ERC721_init("GroupMembership", "GM");

dappAddress = address(0);

}

function setDappAddress(address _dappAddress) public {

require(dappAddress == address(0), "Dapp address already set");

dappAddress = _dappAddress;

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external

Proof of Vulnerability:

N.A.

4.8 Ownership Privileges in SuperDapp Contract

Severity	Status	Type of Analysis
● Medium	Identified	Static

Contract Name:

SuperDapp.sol

Description:

The contracts have owners endowed with privileged rights to execute administrative tasks. However, these owners must be trusted not to engage in malicious activities or deplete funds.

Locations:

SuperDapp.sol

Remediation:

Ensure that owners with privileged rights are trustworthy entities or implement mechanisms such as multi-signature schemes or time-locked contracts to mitigate the risk of centralization. Additionally, consider decentralization strategies to distribute control and reduce reliance on single entities.

Impact:

The contract's owners hold privileged rights for admin tasks, requiring trust to avoid malicious actions or fund depletion. Notable functions include setProtocolFeePercent(uint256), setSubjectFeePercent(uint256), and setTreasuryAddress(address), allowing owner manipulation of fee percentages and treasury address.

Code Snippet:

NA

Reference:

https://www.certik.com/resources/blog/What-is-centralization-risk

Proof of Vulnerability:

N.A.

4.9 External Calls Within Loop in sellShares Function

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

SuperDapp.sol

Description:

The function sellShares(uint256) in the SuperDapp contract contains external calls within a loop. These external calls involve checking for the existence of a group and a group membership using the functions groupMembership.groupExists(groupId) and groupMembership.groupMembershipExists(tokenId), and calling groupMembership.leaveGroup(groupId, tokenId, msg.sender) to handle the selling of shares. Performing external calls within loops can be dangerous as it can lead to unexpected gas costs and potential reentrancy vulnerabilities.

Locations:

SuperDapp.sol

Remediation:

Restructure the function to avoid external calls within loops. You can first perform necessary checks and computations outside the loop, and then iterate over the necessary operations. Ensure that the loop does not depend on state changes that occur as a result of the external calls.

Impact:

The function sellShares(uint256) contains external calls within a loop:

1. It checks for the existence of a group and a group membership using groupMembership.groupExists(groupId) and groupMembership.groupMembershipExists(tokenId) respectively.

2. It calls groupMembership.leaveGroup(groupId, tokenId, msg.sender) to handle the selling of shares.

Code Snippet:

```
        function sellShares(

uint256 tokenId

) public returns (uint256, uint256, uint256) {

uint256 groupId = tokenId.div(10 ** 12);

require(groupMembership.groupExists(groupId), "Group does not exist");

require(

groupMembership.groupMembershipExists(tokenId),

"Group membership does not exist"

);

(

uint256 returnAmount,

uint256 burnAmount,

uint256 maxPossibleReturnAmount

) = groupMembership.leaveGroup(groupId, tokenId, msg.sender);

if (burnAmount > 0) {

address deadAddress = 0x000000000000000000000000000000000000dEaD;

mapDeposits[deadAddress] = mapDeposits[deadAddress].add(burnAmount);

}

mapDeposits[msg.sender] = mapDeposits[msg.sender].add(

maxPossibleReturnAmount

);

totalSellAmount[msg.sender] = totalSellAmount[msg.sender].add(

maxPossibleReturnAmount

);

emit SoldShares(msg.sender, groupId, tokenId, maxPossibleReturnAmount);

return (returnAmount, burnAmount, maxPossibleReturnAmount);

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop

Proof of Vulnerability:

N.A.

ENTERSOFT

4.10 Lack of Event Emission in State Variable Updates within SuperDapp.sol

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

SuperDapp.sol

Description:

The setProtocolFeePercent(uint256) and setSubjectFeePercent(uint256) functions in the SuperDapp contract modify the state variables protocolFeePercent and subjectFeePercent based on the input _feePercent. However, they do not emit any events to notify external listeners about these changes, which can make it difficult for off-chain systems to track the changes in these values.

Locations:

SuperDapp.sol, Lines: 160-164, 166-170

Remediation:

Add events to emit the new values of protocolFeePercent and subjectFeePercent after they are updated in the functions.

Impact:

The functions setProtocolFeePercent(uint256) and setSubjectFeePercent(uint256) modify the state variables protocolFeePercent and subjectFeePercent respectively, but they do not emit any events to signal these modifications.

Code Snippet:

```
        function setProtocolFeePercent(uint256 _feePercent) external onlyOwner {

require(_feePercent >= 0 && _feePercent <= 100, "Invalid fee percent");

protocolFeePercent = _feePercent;

}

function setSubjectFeePercent(uint256 _feePercent) external onlyOwner {

require(_feePercent >= 0 && _feePercent <= 100, "Invalid fee percent");

subjectFeePercent = _feePercent;

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic

Proof of Vulnerability:

N.A.

4.11 Precision Loss Due to Order of Arithmetic Operations in GroupMembership.sol

Severity	Status	Type of Analysis
Low	Identified	Static

Contract Name:

GroupMembership.sol

Description:

Performing multiplication on the result of a division may result in loss of precision, especially in Solidity where integer division truncates the decimal part. It's recommended to perform multiplication before division to ensure accuracy, especially when dealing with financial calculations.

Locations:

GroupMembership.sol, Lines: 141-181

Remediation:

To avoid precision loss, rearrange the order of operations so that multiplication is performed before division.

Impact:

In the function calculateProfit(uint256,uint256,uint256,uint256), multiplication is performed on the result of a division, which might lead to precision loss.

Code Snippet:

NA

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

Proof of Vulnerability:

N.A.

4.12 Redundant Range Validation in setProtocolFeePercent and setSubjectFeePercent Functions of SuperDapp.sol

Severity	Status	Type of Analysis
Low	Identified	Static

Contract Name:

SuperDapp.sol

Description:

The setProtocolFeePercent(uint256) and setSubjectFeePercent(uint256) functions in the SuperDapp contract contain redundant checks for the range of _feePercent. These checks verify whether _feePercent is both greater than or equal to 0 and less than or equal to 100. However, since _feePercent is an unsigned integer (uint256), it cannot be less than 0, making the check redundant.

Locations:

SuperDapp.sol, Lines: 160-164, 166-170

Remediation:

Remove the redundant part of the condition from the require statement, as checking for _feePercent to be greater than or equal to 0 is unnecessary.

Impact:

The require statement in both functions checks whether _feePercent is greater than or equal to 0 and less than or equal to 100. However, since _feePercent is an unsigned integer (uint256), it cannot be less than 0, making the first part of the condition redundant.

Code Snippet:

NA

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

Proof of Vulnerability:

N.A.

4.13 Solidity pragma should be specific, not wide

Severity	Status	Type of Analysis
● Informational	Identified	Static

Contract Name:

SuperDapp.sol, GroupMembership.sol

Description:

Using a wide version in the Solidity pragma statement (`^0.8.20`) is discouraged. It's recommended to specify a particular version to ensure compatibility and avoid unexpected behavior due to potential breaking changes in future compiler versions.

Locations:

SuperDapp.sol, GroupMembership.sol

Remediation:

Update the pragma statements in the contracts to specify a particular version of Solidity. For example, replace `pragma solidity ^0.8.20;` with `pragma solidity 0.8.20;`.

Impact:

Failure to specify a specific version may lead to compatibility issues or unexpected behavior in future compiler versions. It's important to follow best practices to ensure the stability and security of the contracts.

Code Snippet:

```
pragma solidity ^0.8.20;
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Proof of Vulnerability:

N.A.

5.0 Static Analysis

Static analysis is carried out with the following tools:

6.0 Dynamic Analysis

The following results are the efforts of manual analysis.

Note: The following values for "Result" mean:

- **Positive** indicates that there is no security risk.
- **Negative** indicates that there is a security risk that needs to be remediated.
- **Informational** findings should be followed as a best practice, and they are not visible from the smart contract.
- **Not Applicable** means the attack vector is Not applicable or Not available means the attack vector is Not applicable or Not available means the attack vector is Not applicable or Not available

7.0 Auditing Approach and Methodologies applied

Throughout the audit of the smart contract, care was taken to ensure:

- Overall quality of code
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Mathematical calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of token standards.
- Efficient use of gas.
- Code is safe from Re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

7.1 Structural Analysis

In this step we have analysed the design patterns and structure of all smart contracts. A thorough check was completed to ensure all Smart contracts are structured in a way that will not result in future problems.

7.2 Static Analysis

Static Analysis of smart contracts was undertaken to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

7.3 Code Review / Manual Analysis

Manual Analysis or review of done to identify new vulnerabilities or to verify the vulnerabilities found during the Static Analysis. The contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. It should also be noted that the results of the automated analysis were verified manually.

7.4 Gas Consumption

In this step, we checked the behaviour of all smart contracts in production. Checks were completed to understand how much gas gets consumed, along with the possibilities of optimisation of code to reduce gas consumption.

7.5 Tools & Platforms Used For Audit

Slither, Aderyn

7.6 Checked Vulnerabilities

We have scanned Superdapp smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC-20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

8.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of Superdapp and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Smart Contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of Superdapp and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and Superdapp governs the disclosure of this report to all other parties including product vendors and suppliers.