Audit Report - SuperDapp

`Title` : `SuperDapp Audit Report`

`Author` : `DHANANJAY BHAVAR`

`Date` : `February 15, 2024`

Protocol Summary

The `SuperDapp` contract is an ERC20 token with a transfer unlock mechanism, built on the ERC20Permit standard. It inherits from the `ERC20Permit` contract to leverage ERC20 functionality and permit support.

The `ERC20Permit` contract which is an implementation of the ERC20 Permit extension allowing approvals to be made via signatures, as defined in EIP-2612. It adds the `permit` method, which can be used to change an account's ERC20 allowance by presenting a message signed by the account. This allows the token holder account to not need to send a transaction and thus not required to hold Ether at all.

Token transfers are restricted by the `onlyUnlocked` modifier, ensuring execution only when the contract is unlocked or by designated roles. The constructor initializes the contract state with token minting to the admin address, while the `unlock` function allows the admin to unlock transfers for all token holders.

Risk Classification

`High`: Direct impact on the funds or the main functionality of the protocol.

`Medium`: Indirect impact on the funds or the protocol's functionality.

`Low`: Minimal to no impact on the funds or the protocol's main functionality.

Audit Scope

`contract address`: `0xF209C16c3F079ADd2C7b080A3cd64AAA0D325124`

`contracts`: SuperDapp.sol

`Solc Version`: 0.8.13

`Chain(s)`: Ethereum

Findings Summary

Severity	No of issues found
HIGH	1
MEDIUM	2
LOW	11
TOTAL	14

FINDINGS

[H-1] Missing Admin Change Mechanism in ERC20 Token Contract

Vulnerability Type: Access Control

Description: The ERC20 token contract being audited lacks a mechanism to change the `admin` address or `distributor` address. This absence of functionality poses a **critical security risk as the contract relies on admin access for certain crucial functions**, such as unlocking transfers. Consequently, if the private key associated with the admin address is lost or compromised, it becomes impossible to regain control over these critical functions, leading to a potential loss of funds or disruption of token functionality.

Impact: Loss of control over crucial contract functions, such as transfer unlocking.

Recommended Mitigation: Introduce a mechanism within the contract to change the admin address securely. This mechanism should only be accessible by the current admin and should require adequate authentication and authorization to prevent unauthorized changes.

Likelihood & Impact:

- Impact : HIGH
- Likelihood : MEDIUM
- Severity : HIGH

[M-1] Race Condition Exploit in `approve` and `transferFrom` Functions

Vulnerability Type: Race Condition

Description: The ERC20 token contract being audited contains a vulnerability arising from the interaction between the `approve` and `transferFrom` functions. This vulnerability allows an attacker to exploit a race condition, resulting in a discrepancy between the intended and actual token allowance for a spender.

Scenario for Exploitation:

1. **Initial Approval:** Alice calls `approve(Bob, 1000)` to authorize Bob to spend 1000 tokens.
2. **Subsequent Approval:** Alice attempts to reduce the approved allowance for Bob by calling `approve(Bob, 500)` before the initial transaction is mined.
3. **Exploitation:** If Bob's `transferFrom(Alice, X, 1000)` transaction is mined before Alice's subsequent approval, Bob can transfer 1000 tokens despite Alice's attempt to reduce the allowance.
4. **Discrepancy:** After Alice's transaction is mined, Bob can further exploit the situation by calling `transferFrom(Alice, X, 500)`, resulting in a total transfer of 1500 tokens, exceeding the intended allowance.

Impact: The vulnerability enables an attacker to transfer tokens exceeding the approved allowance, leading to potential financial losses for token holders.

Recommended Mitigation :

1. **Reduce Allowance to Zero:** Before setting the desired allowance for a spender, the contract should first reduce the sender's allowance for that spender to zero. This ensures that any previous allowance is revoked before granting a new allowance.
2. **Set Desired Allowance:** After reducing the allowance to zero, the contract should set the desired allowance for the spender to prevent race conditions between multiple `approve` transactions.

```
function setAllowance(address spender, uint256 amount) public
returns (bool) {
// Reduce allowance to zero before setting desired allowance
_approve(msg.sender, spender, 0);
// Set desired allowance for the spender
_approve(msg.sender, spender, amount);

return true;
}
```

Likelihood & Impact :

- Impact : MEDIUM
- Likelihood : HIGH
- Severity : MEDIUM

[M-2] Possible precision loss due to incorrect arithmetic order in `Math.mulDiv` function

Vulnerability Type: Arithmetic Error

Description: The `Math.mulDiv` function in the `SuperDapp.sol` contract performs a multiplication on the result of a division at several occurrences. Solidity's integer

division truncates, which can lead to precision loss when performing division before multiplication. This vulnerability can result in incorrect calculation outcomes and potential financial losses for users.

Exploit Scenario:

Consider the following scenario in a contract A:

```
contract A {
    function f(uint n) public {
        coins = (oldSupply / n) * interest;
    }
}
```

If `n` is greater than `oldSupply`, the result of `(oldSupply / n)` will be zero due to integer division truncation, leading to `coins` being zero. However, if `(oldSupply * interest / n)` was used instead, `coins` would have been 1.

Impact: Incorrect calculation outcomes can lead to financial losses for users interacting with the contract.

Recommended Mitigation:

Consider ordering multiplication before division.

Likelihood & Impact:

- Likelihood: HIGH
- Impact: MEDIUM
- Severity: MEDIUM

[L-1] Unsafe Usage of `block.timestamp` in ERC20Permit.permit Function

Timestamp Manipulation

Description: The `ERC20Permit.permit` function in the `SuperDapp.sol` contract utilizes `block.timestamp` for comparisons. However, `block.timestamp` can be manipulated by

miners, rendering it unreliable for critical functionality such as deadline comparisons. This vulnerability introduces the risk of exploitation by malicious actors manipulating `block.timestamp` to bypass intended restrictions or deadlines.

```
require(block.timestamp <= deadline, "ERC20Permit: expired dea
dline");
```

Impact: Relying on `block.timestamp` for critical functionality introduces a security risk due to its susceptibility to manipulation by miners.

Recommended Mitigation: Utilize alternative timestamp sources, such as block numbers or external timestamp oracles, which are less susceptible to manipulation and provide more reliable data for contract functionality.

Likelihood & Impact:

- Likelihood: LOW
- Impact: HIGH
- Severity: LOW

[L-2] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Vulnerability Type: Security Best Practice Violation

Description:

```
return keccak256(abi.encodePacked("\x19Ethereum Signed Messag
e:\n", Strings.toString(s.length), s));
```

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` ⇒ `0x123456` ⇒ `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` ⇒ `0x0...1230...456`).

Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast

to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

Impact: Using `abi.encodePacked()` with dynamic types can lead to hash collisions, which may compromise the integrity and security of the hashing operation

Recommended Mitigation: Modify the code to use `abi.encode()` instead of `abi.encodePacked()` to prevent hash collisions and enhance the security of the hashing operation.

Likelihood & Impact:

- Likelihood: HIGH
- Impact: LOW
- Severity: LOW

[L-3] Unsafe ERC20 Operations without SafeERC20 Library Usage

Vulnerability Type: Security Best Practice Violation

Description:

The ERC20 operations `transfer` and `transferFrom` in the `SuperDapp.sol` contract are implemented without using the SafeERC20 library. Direct usage of ERC20 functions without the SafeERC20 library can introduce vulnerabilities and unexpected behaviours due to non-standard return values and potential reentrancy issues. It's recommended to use **OpenZeppelin's SafeERC20** library to ensure safe and predictable ERC20 operations.

Impact: Lack of safety checks increases the risk of reentrancy attacks and other vulnerabilities associated with ERC20 operations.

Recommended Mitigation: Replace direct usage of ERC20 functions with equivalent functions from the SafeERC20 library provided by OpenZeppelin to ensure safe and predictable ERC20 operations.

Likelihood & Impact:

- Likelihood: HIGH
- Impact: LOW

- Severity: LOW

[L-4] Functions Not Used Internally Should be Marked as External

Vulnerability Type: Code Quality Improvement

Description:

The functions `transfer` and `transferFrom` in the `SuperDapp.sol` contract are defined as public but are not used internally within the contract. It is recommended to mark functions as external if they are not intended to be called internally. Marking such functions as external provides clarity and may lead to optimizations by the compiler.

Impact: Marking functions as external may allow the compiler to optimize the generated bytecode, potentially improving gas efficiency.

Recommended Mitigation: Modify the functions to be marked as external if they are not intended to be called internally within the contract. This provides clarity regarding their intended usage.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-5] Constants should be defined and used instead of literals

Vulnerability Type: Code Quality Improvement

Description:

At several instances, the `SuperDapp.sol` contract uses literals directly instead of defining constants for certain values. It is considered a best practice to define constants for values that are used multiple times throughout the codebase to improve readability, maintainability, and to avoid inconsistencies.

Impact: Usage of literals instead of constants may lead to code duplication and inconsistency if the value needs to be updated or changed in the future.

Recommended Mitigation: Identify values that are used multiple times throughout the codebase and define them as constants with descriptive names to improve code maintainability and readability.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-6] Inefficient Usage of Indexed Event Fields

Vulnerability Type: Gas Efficiency Improvement

Description:

The `SuperDapp.sol` contract defines events with indexed fields without considering the optimal indexing strategy based on the number of fields in the event and the gas usage. While indexing event fields can make them more quickly accessible to off-chain tools, it also incurs additional gas costs during emission. It is recommended to carefully select which event fields should be indexed based on the number of fields and the importance of gas usage.

Impact: Inefficient indexing of event fields can lead to unnecessary gas consumption during event emission, potentially impacting the overall gas usage and transaction costs.

Recommended Mitigation: Consider the gas usage implications of indexing event fields and optimize the indexing strategy accordingly to balance gas efficiency and off-chain tool accessibility.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-7] Lack of Descriptive Reason Strings in require() Statement

Vulnerability Type: Code Readability Improvement

Description:

The `SuperDapp.sol` contract utilizes the `require()` statement without providing a descriptive reason string or custom error message. Including descriptive reason strings in `require()` statements enhances code readability and provides clarity about the conditions being checked.

Impact: Lack of descriptive reason strings in `require()` statements can make it challenging for developers to understand the purpose and conditions being checked, potentially leading to confusion during code maintenance and debugging.

Recommended Mitigation: Modify the `require()` statement to include a descriptive reason string or custom error message that explains the condition being checked and the consequences of failure.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-8] Dubious Typecasting and Legacy Practice in tryRecover Function

Vulnerability Type: Code Quality Improvement

Description:

The `tryRecover` function in the provided smart contract code performs dubious typecasting and utilizes a legacy practice for recovering the v component of a signature. This practice involves shifting a `bytes32` value 255 bits to the right and may lead to unexpected behavior if the input is not formatted correctly. Additionally, adding 27 to the `v` value is a legacy practice to normalize the recovery id and may not be necessary or correct for all Ethereum chains.

Impact: Dubious typecasting from `uint256` to `uint8` may lead to unexpected behavior if the input is not formatted correctly.

Recommended Mitigation: Ensure that the `vs` input is provided in a specific format where the most significant bit is strictly the recovery id bit, and no other bits are set.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-9] Shadowing of Inherited Function in ERC20Permit Constructor

Vulnerability Type: Code Quality Improvement

Description:

The `SuperDapp.sol` smart contract extends 'ERC20Permit', which inherits from 'ERC20'. Both 'ERC20' and 'ERC20Permit' have a function named 'name()'. However, the 'ERC20Permit' constructor is meant to initialize the 'EIP712' domain with the name of the token. This can lead to confusion and potential errors due to shadowing of the 'name()' function from 'ERC20' and 'IERC20Metadata'.

Impact: Shadowing the 'name()' function from 'ERC20' and 'IERC20Metadata' in the 'ERC20Permit' constructor can cause confusion and potentially lead to errors if the functions are not called carefully.

Recommended Mitigation: Rename the constructor of the 'ERC20Permit' contract to avoid shadowing the 'name()' function from 'ERC20' and 'IERC20Metadata'. Choose a name that clearly describes its purpose without overlapping with existing function names.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW

- Severity: LOW

[L-10] Usage of EVM Assembly

Vulnerability Type: Code Quality Improvement

Description:

The smart contract utilizes EVM assembly, which is error-prone and can lead to vulnerabilities if not implemented correctly. EVM assembly is low-level and complex, making it difficult to maintain and prone to errors. As a result, its usage should be avoided whenever possible to reduce the risk of introducing vulnerabilities.

Impact: EVM assembly is error-prone due to its low-level and complex nature, increasing the likelihood of introducing vulnerabilities or bugs into the smart contract codebase.

Recommended Mitigation: Refrain from using EVM assembly in the smart contract codebase whenever possible. Instead, utilize higher-level Solidity constructs to achieve the desired functionality.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

[L-11] `ERC20Permit._PERMIT_TYPEHASH_DEPRECATED_SLOT` should be constant

Vulnerability Type: State Variable Optimization

Description:

The state variable `ERC20Permit._PERMIT_TYPEHASH_DEPRECATED_SLOT` in the `SuperDapp.sol` contract should be marked as constant if its value never changes. Marking it as constant allows the compiler to optimize its storage and reduces gas costs during contract deployment and function calls.

Impact: Failing to mark state variables as constant when their value never changes can lead to suboptimal gas usage and increased contract deployment and function call costs.

Recommended Mitigation: Make the state variable as constant.

Likelihood & Impact:

- Likelihood: NONE
- Impact: LOW
- Severity: LOW

Conclusion

The audit of the `SuperDapp` smart contract has revealed opportunities for optimization and strengthening in various areas, including code structure, gas usage, and security protocols. By implementing the suggested improvements, the contract can be enhanced to meet industry standards and provide a more secure and efficient platform for decentralized applications.