Super Dapp Audit Report

Author	Gurkirat Singh
Date	17-02-2024

Protocol Introduction

`SuperDapp` is an ERC20 token that incorporates an unlock mechanism for transfers and the ERC20Permit feature for gasless approvals. It is designed with an admin and distributor role, where the admin has the authority to unlock the contract for token transfers and both the admin and distributor can transfer tokens regardless of the lock state.

Imported Base Contracts and Libraries

- **ERC20**: This is a standard interface for fungible tokens, including basic functionalities like transferring tokens and keeping track of balances and allowances.
- **ERC20Permit**: An extension of the ERC20 standard that allows token approvals (allowances) to be made through signatures instead of transactions, as defined in EIP-2612. This means token holders can approve

a spender to transfer tokens on their behalf without executing a transaction, saving gas fees and potentially enhancing user experience.

- **EIP712**: A standard for hashing and signing of typed structured data, enabling the ERC20Permit feature.
- **Counters**: A utility library from OpenZeppelin that provides a counter that can only be incremented or decremented, used here for nonce management in ERC20Permit.

Vulnerability severity ranking as follows:

High: These vulnerabilities entail greater effort for attackers to exploit and may result
in successful protocol compromise.

Medium: These vulnerabilities may not lead to protocol compromise but could be
leveraged by attackers to attack other users using the protocol , chained together with multiple medium findings to constitute a successful compromise.

Info: These vulnerabilities are could be concerned with improper disclosure of information and that should be resolved. They may provide attackers with important information that could lead to additional attack vectors or lower the level of effort necessary to exploit the protocol.

High:	1
Medium	1
Info	6
Total	8

Findings

1. **[H1] Irreversibility of Unlocking Mechanism inside `unLock` function**

   Severity: High

   Type: Access Control

   Description: Once the contract is unlocked, it cannot be locked again. This design decision simplifies the logic but also means that if the contract is unlocked prematurely or maliciously, there's no way to revert back to a locked state to secure the token transfers again.

   Mitigation: Can be solved by following methods

   - Implementing a governance system or requiring multiple signatures to unlock or lock the contract can help mitigate the risks associated with centralized control. This could be a simple multi-sig requirement or a more complex DAO-like governance mechanism where token holders vote on locking or unlocking the contract. Hence securing unlock function.

   - By introducing a lock mechanism function, by utilising above method and introducing multiple signature wallet to lock function and unlock function in smart contract.

2. **[M1] Front-running concerns related to the `approve` function of ERC20 tokens**

   Severity: Medium

Type: Front running

Description:

An owner decides to change the allowance of a spender from one value to another. If the owner sends out a transaction to lower the allowance, a malicious actor could see this pending transaction and quickly send a transaction with a higher gas fee to use the old, higher allowance before the owner's transaction is confirmed. This could result in the malicious actor withdrawing more tokens than the owner intended by the time of the change.

In a decentralized exchange (DEX) environment, a user might approve a DEX to spend tokens and then place an order to sell those tokens. A front-runner could see the transaction in the mempool and execute their own trade first, potentially at a more favorable price, and then immediately sell the tokens after the original user's trade goes through, affecting market prices and potentially disadvantaging the original trader.

Mitigation: Could be resolved by two ways

- **Use of `permit` Function**: As mentioned, the `ERC20Permit` extension allows for a more secure way to approve token spending without the transaction being visible in the mempool until the actual transfer happens. This can mitigate front-running risks associated with the `approve` function.

- **Decreasing Allowance to Zero**: One recommended practice to mitigate the risks associated with changing allowances is to always set the allowance to zero before setting it to a new value. This two-step process (first to zero, then to the new amount) can help prevent the front-running of allowance changes. However, this method increases gas costs.

3. **[L1] Functions not used internally should be marked external**

Severity: Low

Type: Code optimisation

Description: Functions should be marked as external whenever possible, as **external functions are more gas-efficient than public ones**. This is because external functions expect arguments to be passed from the external call, which can save gas.

Mitigation: Change the visibility if the function to external for the following functions

`Line: 244`

```
    function name() public view virtual override retu
rns (string memory) {
```

`Line: 252`

```
    function symbol() public view virtual override re
turns (string memory) {
```

`Line: 269`

```
    function decimals() public view virtual override
returns (uint8) {
```

`Line: 276`

```
    function totalSupply() public view virtual overri
de returns (uint256)
```

Line: 283

```
    function balanceOf(address account) public view v
irtual override returns (uint256)
```

Line: 295

```
    function transfer(address to, uint256 amount) pub
lic virtual override returns (bool) {
```

Line: 318

```
    function approve(address spender, uint256 amount)
public virtual override returns (bool) {
```

Line: 340

```
    function transferFrom(
```

Line: 363

```
    function increaseAllowance(address spender, uint2
56 addedValue) public virtual returns (bool)
```

Line: 383

```
    function decreaseAllowance(address spender, uint2
56 subtractedValue) public virtual returns (bool) {
```

Line: 1369

```
    function permit(
```

Line: 1393

```
    function nonces(address owner) public view virtua
l override returns (uint256)
```

Line: 1474

```
    function transfer(address recipient, uint256 amou
nt) public override onlyUnlocked returns (bool) {
```

`Line: 1486`

```
    function transferFrom(address sender, address rec
ipient, uint256 amount) public override onlyUnlocked
returns (bool) {
```

4. [L2] Unsafe ERC20 Operations should not be used

Severity: Low

Type: Code optimisation

Description: ERC20 functions may not behave as expected. For example: return values are not always meaningful.

`Line: 1475`

```
        return super.transfer(recipient, amount);
```

`Line: 1487`

```
        return super.transferFrom(sender, recipient, amo
unt);
```

Mitigation: It is recommended to use OpenZeppelin's SafeERC20 library.

5. [L3] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Severity: Low

Type: Code optimisation

Description: Consider the example given in your message: `abi.encodePacked(0x123,0x456)` results in `0x123456`, which is ambiguous because it could also be the result of `abi.encodePacked(0x1,0x23456)`. Both sets of inputs produce the same output, which could be exploited in certain contexts to cause hash collisions. These collisions can, under specific conditions, be used to forge or manipulate data in a way that the original contract logic did not intend.

Mitigation: Use `abi.encode()` for Hashing: When encoding multiple arguments to pass to a hash function, use `abi.encode()` instead of `abi.encodePacked()` to ensure that arguments are padded to 32 bytes, preventing hash collisions.

6. **[L4] `require()` / `revert()` statements should have descriptive reason strings or custom errors**

Severity: Low

Type: Code optimisation

Description: The `require()` statement is used to ensure that the `denominator` is greater than `prod1`. If this condition is not met, the transaction will be reverted, but without a descriptive reason, it might not be clear why the transaction failed, especially to someone who is not deeply familiar with the contract's logic.

`Line: 651`

```
require(denominator > prod1);
```

Mitigation: Solidity allows for a string message to be included with `require()` statements. This message is returned if the condition fails, providing context for the failure. The modified code with a descriptive reason string might look like this:

`Line: 651`

```
            require(denominator > prod1, "Denominator mu
st be greater than prod1 to prevent overflow");
```

7. **[L5] Block timestamp in function `permit` inside `ERC20Permit` contract**

   Severity: Low

   Type: Code optimisation

   Description: `block.timestamp` can be manipulated by miners.

   Mitigation: Use third party oracles like Chainlink Time Oracles.

8. **[L6] Constants should be defined and used instead of literals**

   Severity: Low

   Type: Code optimisation

   Description: The use of constants instead of hard-coded literals makes the code more readable, maintainable, and less error-prone. It also provides a single point of reference for values that have a specific meaning and are used multiple times throughout the code.

Line: 270

```
function decimals() public view virtual override returns
(uint8) {
    return 18;
}
```

Mitigation:

Line: 270

```
uint8 public constant decimals = 18;

function decimals() public view virtual override returns
(uint8) {
    return decimals;
}
```