Security Audit Report

Prepared for: *DappRadar*
Online report: dappradar-token-airdrop
Date: *12/16/2021*

Token Airdrop Security Audit

We were tasked with performing an audit of the DappRadar codebase and in particular their token and airdrop implementation.

Over the course of the audit, we identified a potentially incorrect execution path in the constructor of the Radar token as well as an improper validation of ECDSA signatures in the `Lib` dependency of the project both of which should be remediated.

Overall, the codebase is of a high standard and we advise the DappRadar team to integrate all our suggestions to ensure the code is production ready.

Files in Scope	Repository	Commit(s)
Airdrop.sol (AIR)	**dapp-radar**	**b2399d6a7d**, **e4d2ffe537**
Lib.sol (LIB)	**dapp-radar**	**b2399d6a7d**, **e4d2ffe537**
RadarToken.sol (RTN)	**dapp-radar**	**b2399d6a7d**, **e4d2ffe537**

During the audit, we filtered and validated a total of **2 findings utilizing static analysis** tools as well as identified a total of **5 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they introduce potential misbehaviours of the system as well as exploits.

The list below covers each segment of the audit in depth and links to the respective chapter of the report:

- 💻 **Compilation**
- 🔍 **Static Analysis**
- 👁 **Manual Review**
- 🖌 **Code Style**

Compilation

The project utilizes `hardhat` as its development pipeline tool, containing an array of tests and scripts coded in JavaScript.

To compile the project, the `compile` command needs to be issued via the `npx` CLI tool to `hardhat`:

```bash
npx hardhat compile
```

The `hardhat` tool automatically selects between Solidity versions `0.8.4` based on the version specified within the `hardhat.config.js` file.

The project contains discrepancies with regards to the Solidity version used, however, they are located in external dependencies of the project and as such can be safely ignored.

The DappRadar team has locked the `pragma` statements to `0.8.4`, the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `hardhat` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Static Analysis

The execution of our static analysis toolkit identified **48 potential issues** within the codebase of which **45 were ruled out to be false positives** or negligible findings.

The remaining **3 issues** were validated and grouped and formalized into the **2 exhibits** that follow:

ID	Severity	Addressed	Title
AIR-01S	Informational	Yes	Data Location Optimization
RTN-01S	Informational	Yes	Variable Shadowing

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in the token airdrop of DappRadar.

As the project at hand implements a cryptographic airdrop, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth within the protocol's specification and that the **usage of cryptography is securely performed**.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed improper sanitization of cryptographic signatures** within the system, however, they were conveyed ahead of time to the DappRadar team to be **promptly remediated**.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to an exemplary extent.

A total of **5 findings** were identified over the course of the manual review of which **3 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
LIB-01M	Medium	Yes	Insecure Elliptic Curve Recovery Mechanism
RTN-01M	Minor	Yes	Improper Mint Execution Flow
RTN-02M	Minor	Yes	Irreversible Minter Status

Code Style

During the manual portion of the audit, we identified **2 optimizations** that can be applied to the codebase that will decrease the gas-cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
AIR-01C	Informational	Yes	Variable Mutability Specifiers
RTN-01C	Informational	Yes	Usage of Value Literal

Airdrop Static Analysis Findings

ON THIS PAGE

AIR-01S: Data Location Optimization

Type	Severity	Location
Gas Optimization	Informational	Airdrop.sol:L57

Description:

The linked variable is a `memory` argument in an `external` function.

Example:

contracts/Airdrop.sol

SOL

Copy

```sol
function claimTokens(
    bytes memory _signature,
    address _recipient,
    uint256 _amount
) external whenNotPaused nonReentrant {
```

Recommendation:

We advise it to be set as `calldata` optimizing its read-access cost.

Alleviation:

The data location specifier was properly changed to `calldata`.

RadarToken Static Analysis Findings

ON THIS PAGE

RTN-01S: Variable Shadowing

Type	Severity	Location
Language Specific	Informational	RadarToken.sol:L32, L33

Description:

The linked variables shadow existing variables in the `ERC20` inheritance chain.

Example:

contracts/RadarToken.sol

SOL

Copy

```sol
/**
 * @dev constructor
 * @param _name token name
 * @param _symbol token symbol
 * @param _decimals decimals
 * @param _reserveAddress address to hold initially minted tokens
 * @param _mintAddresses array of address to hold initial tokens
 * @param _mintAmounts array of initlal token amounts to be minted to mint addres
 */
constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    uint256 _cap,
    address _reserveAddress,
    address[] memory _mintAddresses,
    uint256[] memory _mintAmounts
) ERC20(_name, _symbol) ERC20Capped(_cap) Ownable() {
```

Recommendation:

We advise them to be renamed to avoid the naming collision.

Alleviation:

The variables were renamed properly avoiding the naming collision.

Lib Manual Review Findings

ON THIS PAGE

LIB-01M: Insecure Elliptic Curve Recovery Mechanism

Type	Severity	Location
Language Specific	Medium	Lib.sol:L19

Description:

The `ecrecover` function is a low-level cryptographic function that should be utilized after appropriate sanitizations have been enforced on its arguments, namely on the `s` and `v` values. This is due to the inherent trait of the curve to be symmetrical on the x-axis and thus permitting signatures to be replayed with the same `x` value (`r`) but a different `y` value (`s`).

Example:

contracts/lib/Lib.sol

SOL Copy

```sol
function recoverSigner(bytes32 _hash, bytes memory _signature) internal pure retu
    (bytes32 r, bytes32 s, uint8 v) = splitSignature(_signature);

    return ecrecover(_hash, v, r, s);
}
```

Recommendation:

We advise them to be sanitized by ensuring that `v` is equal to either `27` or `28` (`v ∈ {27, 28}`) and to ensure that `s` is existent in the lower half order of the elliptic curve (`0 < s < secp256k1n ÷ 2 + 1`) by ensuring it is less than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A1`. A reference implementation of those checks can be observed in the **ECDSA** library of OpenZeppelin and

the rationale behind those restrictions exists within **Appendix F of the Yellow Paper**.

Alleviation:

The signature recovery mechanism was replaced by the standardized `ECDSA` implementation wherever it was used and the contract has been subsequently removed from the codebase thereby alleviating this exhibit in full.

RadarToken Manual Review Findings

ON THIS PAGE

RTN-01M: Improper Mint Execution Flow

Type	Severity	Location
Logical Fault	Minor •	RadarToken.sol:L42, L49, L54

Description:

The `require` checks pertaining the `_cap` appear to incorrectly function as the first `require` check mandates that the `_cap` is greater-than-or-equal-to (`>=`) the value of one billion yet the code performs a subtraction of the `totalMint` amount from a one billion hard-coded literal which can cause an underflow if the `_cap` exceeds it.

Example:

contracts/RadarToken.sol

SOL

Copy

```sol
/**
 * @dev constructor
 * @param _name token name
 * @param _symbol token symbol
 * @param _decimals decimals
 * @param _reserveAddress address to hold initially minted tokens
 * @param _mintAddresses array of address to hold initial tokens
 * @param _mintAmounts array of initlal token amounts to be minted to mint addres
 */
constructor(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    uint256 _cap,
```

```
36      address _reserveAddress,
37      address[] memory _mintAddresses,
38      uint256[] memory _mintAmounts
39  ) ERC20(_name, _symbol) ERC20Capped(_cap) Ownable() {
40      uint256 totalMint;
41
42      require(_cap >= 10 ** 9, "RadarToken: Cap needs to be greater than a billion"
43      require(_mintAddresses.length == _mintAmounts.length, "RadarToken: must have
44
45      customDecimals = _decimals;
46
47      for (uint i; i < _mintAddresses.length; i++) {
48          require(_mintAddresses[i] != address(0), "RadarToken: cannot have a non-a
49          require(totalSupply() + _mintAmounts[i] <= _cap, "total supply of tokens
50          ERC20._mint(_mintAddresses[i], _mintAmounts[i]);
51          totalMint += _mintAmounts[i];
52      }
53
54      ERC20._mint(_reserveAddress, 10 ** 9 - totalMint);
55  }
```

Recommendation:

We advise the `_cap` to either be statically set to the value of one billion or the last statement to subtract `totalMint` from `_cap` instead of the one billion literal, the latter of which we anticipate to be the expected behaviour.

Alleviation:

The cap notion has been removed from the codebase thereby rendering this exhibit null.

RTN-02M: Irreversible Minter Status

Type	Severity	Location
Logical Fault	Minor	RadarToken.sol:L74-L81

Description:

The `approveMinter` function does not have a negation equivalent, meaning that any minter that has been sent will be perpetually able to mint tokens.

Example:

contracts/RadarToken.sol

SOL

Copy

```sol
74  /**
75   * @dev owner approves user to mint
76   * @param _minter address of minter to approve
77   */
78  function approveMinter(address _minter) external onlyOwner {
79      minters[_minter] = true;
80      emit MinterApproved(msg.sender, _minter);
81  }
```

Recommendation:

We advise a `disableMinter` function to be introduced to the codebase to ensure that a minter can be stripped of their status should an emergency occur.

Alleviation:

The minting notion has been removed from the codebase rendering this exhibit null.

Airdrop Code Style Findings

ON THIS PAGE

AIR-01C: Variable Mutability Specifiers

Type	Severity	Location
Gas Optimization	Informational	Airdrop.sol:L15, L19, L22, L45, L46, L47

Description:

The linked variables are assigned to only once during the contract's `constructor`.

Example:

contracts/Airdrop.sol

SOL Copy

```sol
/**
 * @dev constructor
 * @param _reserveAddress reserve address
 * @param _claimSigner address of message signer
 * @param _token Radar token address
 */
constructor(
    address _reserveAddress,
    address _claimSigner,
    address _token
) Pausable() Ownable() ReentrancyGuard() {
    require(_reserveAddress != address(0), "RadarAirdrop: invalid reserve address
    require(_claimSigner != address(0), "RadarAirdrop: invalid claim signer addre
    require(_token != address(0), "RadarAirdrop: invalid token address");

    reserveAddress = _reserveAddress;
    claimSigner = _claimSigner;
    token = IERC20(_token);
}
```

Recommendation:

We advise them to be set as `immutable` greatly optimizing the codebase's gas cost.

Alleviation:

All three linked variables were properly set to `immutable`.

RadarToken Code Style Findings

ON THIS PAGE

RTN-01C: Usage of Value Literal

Type	Severity	Location
Code Style	Informational	RadarToken.sol:L42, L54

Description:

The linked instances of the one billion value literal can be replaced by a contract-level `constant` instead.

Example:

contracts/RadarToken.sol

SOL

Copy

```sol
require(_cap >= 10 ** 9, "RadarToken: Cap needs to be greater than a billion");
require(_mintAddresses.length == _mintAmounts.length, "RadarToken: must have same

customDecimals = _decimals;

for (uint i; i < _mintAddresses.length; i++) {
    require(_mintAddresses[i] != address(0), "RadarToken: cannot have a non-addre
    require(totalSupply() + _mintAmounts[i] <= _cap, "total supply of tokens cann
    ERC20._mint(_mintAddresses[i], _mintAmounts[i]);
    totalMint += _mintAmounts[i];
}

ERC20._mint(_reserveAddress, 10 ** 9 - totalMint);
```

Recommendation:

We advise this to be done so to better illustrate what the comparisons perform and to increase the legibility of the codebase.

Alleviation:

The value literal is no longer in use rendering this exhibit null.

Finding Types

ON THIS PAGE

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

External Call Validation

Many contracts that interact with DeFi contain a set of complex external call executions that need to happen in a particular sequence and whose execution is usually taken for granted whereby it is not always the case. External calls should always be validated, either in the form of `require` checks imposed at the contract-level or via more intricate mechanisms such as invoking an external getter-variable and ensuring that it has been properly updated.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted `if` blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Solidity language boasts that discerns it from other conventional programming languages. For example, the EVM is a 256-bit machine meaning that operations on less-than-256-bit types are more costly for the EVM in terms of gas costs, meaning that loops utilizing a `uint8` variable because their limit will never exceed the 8-bit range actually cost more than redundantly using a `uint256` variable.

Code Style

An official Solidity style guide exists that is constantly under development and is adjusted on each new Solidity release, designating how the overall look and feel of a codebase should be. In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a contract-level variable that is present in the inheritance chain of the local execution level's context.

Gas Optimization

Gas optimization findings relate to ways the codebase can be optimized to reduce the gas cost involved with interacting with it to various degrees. These types of findings are completely optional and are pointed out for the benefit of the project's developers.

Standard Conformity

These types of findings relate to incompatibility between a particular standard's implementation and the project's implementation, oftentimes causing significant issues in the usability of the contracts.

Mathematical Operations

In Solidity, math generally behaves differently than other programming languages due to the constraints of the EVM. A prime example of this difference is the truncation of values during a division which in turn leads to loss of precision and can cause systems to behave incorrectly when dealing with percentages and proportion calculations.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.