Report

v. 1.0

Customer

RealT

Smart Contract Audit

Reg Ccip Core

30th April 2024

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	30.04.24	A. Zveryanskaya	Initial Draft
0.2	30.04.24	A. Zveryanskaya	Minor revision
1.0	30.04.24	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

RealT is a platform that offers simplified investments in real estate. The company mission is to democratize access to real estate investment opportunities, curated by a team of real estate professionals.

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Files:

```
/
        CCIPSenderReceiver.sol        REG.sol
```

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

5 Our findings

We found 2 major, and a few less important issues. All identified Major issues have been fixed.

Fixed 2 out of 2 issues

6 Major Issues

CVF-2 FIXED

- **Category** Unclear behavior
- **Source** CCIPSenderReceiver.sol

Description This error should include the data returned by the failed ether transfer attempt.

Client Comment *We agree with this issue. This is fixed by adding the returned data for more information.*

```
240 revert CCIPErrors.FailedToWithdrawEth(
```

CVF-3 FIXED

- **Category** Suboptimal
- **Source** CCIPSenderReceiver.sol

Recommendation The check for the ”IERC165” interface ID is redundant, as ”super.supportsInterface” would do it anyway.

Client Comment *We agree with this issue since IERC165's interface ID is the same as IERC165Upgradeable's interface ID. We deleted this redundant check.*

```
551 interfaceId == type(IERC165).interfaceId ||
    super.supportsInterface(interfaceId);
```

7 Recommendations

CVF-1 INFO

- **Category** Unclear behavior
- **Source** CCIPSenderReceiver.sol

Description Unchained initializers should be used here.

Client Comment *OpenZeppelin advises against using unchained initializers, as they require manual verification to ensure no initializers are missing. For instance, invoking __ERC20Pausable_init_unchained() alone would have no effect. However, calling __ERC20Pausable_init() triggers __Pausable_init_unchained(), which then initializes the _paused variable. Similarly, __ERC20Permit_init_unchained on its own does nothing, but __ERC20Permit_init() executes __EIP712_init_unchained(name, "1") to set up the name and version.*
Further details and examples can be found in the documentation and source code: OpenZeppelin Documentation on upgradeable contracts:

- *OpenZeppelin Docs*
- *ERC20PausableUpgradeable.sol Code: GitHub - ERC20PausableUpgradeable.sol*
- *ERC20PermitUpgradeable.sol Code: GitHub - ERC20PermitUpgradeable.sol*

Ongoing issues related to this topic can be monitored through the OpenZeppelin GitHub issue tracker: GitHub Issue #4602.

```
71  __Pausable_init();
    __AccessControl_init();
    __UUPSUpgradeable_init();
```

CVF-5 FIXED

- **Category** Suboptimal
- **Source** CCIPSenderReceiver.sol

Description In case ”destinationChainReceiver” is zero, the chain is effectively disallowed, but remains in the ”_allowlistedChainsList”.

Recommendation Consider removing the chain from the list in such a case. An effective way to implement such removal would be to modify the ”isInList” mapping to store the indexes of allowed chains within the ”_allowListedChainsList” instead of just boolean flags.

Client Comment *The _allowedChainsList array maintains a record of chains that have been permitted previously. To verify whether a chain is presently authorized, the isAllowlistedDestinationChain function should be used. To better represent its function, we renamed the array to _chainsListHistory.*

```
180 if (destinationChainReceiver != address(0) && !chainState.isInList)
    ↪ {
     _allowlistedChainsList.push(destinationChainSelector);
     chainState.isInList = true;
}
```

CVF-6 FIXED

- **Category** Suboptimal
- **Source** CCIPSenderReceiver.sol

Description In case ”allowed” is false, the token is effectively disallowed, but remains in the ”_allowlistedTokensList”.

Recommendation Consider removing the token from the list in such a case. An effective way to implement such removal would be to modify the ”isInList” mapping to store the indexes of allowed tokens within the ”_allowListedTokensList” instead of just boolean flags.

Client Comment *The _allowedTokensList array maintains a record of tokens that have been permitted previously. To verify whether a token is presently authorized, the isAllowlistedToken function should be used. To better represent its function, we renamed the array to _tokensListHistory.*

```
204 if (allowed && !tokenState.isInList) {
     _allowlistedTokensList.push(token);
     tokenState.isInList = true;
}
```

CVF-7 INFO

- **Category** Procedural
- **Source** CCIPSenderReceiver.sol

Description We didn't review these files.

```
10 import {IRouterClient} from "@chainlink/contracts-ccip/src/v0.8/ccip
      ↪ /interfaces/IRouterClient.sol";
   import {Client} from "@chainlink/contracts-ccip/src/v0.8/ccip/
      ↪ libraries/Client.sol";
   import {IAny2EVMMessageReceiver} from "@chainlink/contracts-ccip/src
      ↪ /v0.8/ccip/interfaces/IAny2EVMMessageReceiver.sol";
   import {IERC165} from "@chainlink/contracts-ccip/src/v0.8/vendor/
      ↪ openzeppelin-solidity/v4.8.0/contracts/utils/introspection/
      ↪ IERC165.sol";
   import {CCIPErrors} from "../libraries/CCIPErrors.sol";
   import {ICCIPSenderReceiver} from "../interfaces/ICCIPSenderReceiver
      ↪ .sol";
```

CVF-8 INFO

- **Category** Procedural
- **Source** CCIPSenderReceiver.sol

Description Solidity compiler is smart enough to precompute constant hash expressions.

Recommendation Consider using expressions instead of hardcoded hashes.

Client Comment *We utilize precomputed constant hashes to reduce gas consumption.*

```
33 0x65d7a28e3265b37a6474929f336521b332c1681b933f6cb9f3376673440d862a;
      ↪ // = keccak256("PAUSER_ROLE")
```

```
36 0x189ab7a9244df0848122154315af71fe140f3db0fe014031783b0946b8c9d2e3;
      ↪ // = keccak256("UPGRADER_ROLE")
```

CVF-9 INFO

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The type for these constants should be more specific.

Client Comment *In the contract, _linkToken and _wrappedNativeToken are not used as IERC20 types, so there is no need to specify their type.*

```
40 address private constant _linkToken =
```

```
43 address private constant _wrappedNativeToken =
```

CVF-10 INFO

- **Category** Suboptimal
- **Source** CCIPSenderReceiver.sol

Description Hardcoding mainnet addresses makes testing harder.

Recommendation Consider passing the addresses as constructor arguments and storing in immutable variables.

Client Comment *Upgradeable contracts should not include a constructor. Here, the constructor is solely used to _disableInitializers, without requiring any arguments. To keep the constructor straightforward, we use constants, ensuring we adjust the correct addresses for each chain.*

```
41 0x514910771AF9Ca656af840dff83E8264EcF986CA; // LINK on Ethereum
```

```
44 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2; // WETH on Ethereum
```

CVF-11 INFO

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The keys type for this mapping should be more specific.

Client Comment *There's no need to treat the token as IERC20 in this context, so specifying the type is unnecessary.*

```
49 mapping(address => AllowlistTokenState) private _allowlistedTokens;
```

CVF-12 INFO

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The element type for this array should be more specific.

Client Comment *There's no need to treat the token as IERC20 in this context, so specifying the type is unnecessary.*

```
53 address[] private _allowlistedTokensList;
```

CVF-13 FIXED

- **Category** Procedural
- **Source** CCIPSenderReceiver.sol

Recommendation It is a good practice to put a comment into an empty block to explain why the block is empty.

Client Comment *The receive() function should be removed since the contract is not intended to receive Ether. Users can pay CCIP fees through the payable functions transferTokens and transferTokensWithPermit.*

```
98 receive() external payable {}
```

CVF-14 FIXED

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The argument type should be ”IERC20”.

Client Comment *We agree with this approach which saves gas by eliminating the need to cast the token as IERC20. We use it whenever it saves gas.*

```
133 modifier onlyAllowlistedToken(address token) {
193     address token,
250     address token
265     address token,
267     address feeToken,
285     address token,
287     address feeToken,
360     address token
382     address token,
384     address feeToken,
479     address token,
481     address feeTokenAddress,
510     address token,
512     address feeToken,
```

CVF-15 FIXED

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The argument type should be ”IRouterClient”.

Client Comment *Agreed, this avoids the need to cast IRouterClient repeatedly. Corrected.*

```
69 address router
```

```
213 address router
```

CVF-16 FIXED

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The return type should be ”IRouterClient”.

Client Comment *Agreed, consistency achieved as above. Corrected.*

```
315 function getRouter() external view override returns (address) {
```

CVF-17 INFO

- **Category** Bad datatype
- **Source** CCIPSenderReceiver.sol

Recommendation The return type should be more specific.

Client Comment *In the contract, _linkToken and _wrappedNativeToken are not used as IERC20 types, so there is no need to specify their type.*

```
320 function getLinkToken() external pure override returns (address) {
```

```
325 function getWrappedNativeToken() external pure override returns (
    ↪ address) {
```

```
344     returns (address[] memory)
```

CVF-18 INFO

- **Category** Procedural
- **Source** REG.sol

Description Solidity compiler is smart enough to precompute hash expressions.

Recommendation Consider using expressions instead of hardcoded hashes.

Client Comment *We utilize precomputed constant hashes to reduce gas consumption.*

```
31 0x65d7a28e3265b37a6474929f336521b332c1681b933f6cb9f3376673440d862a;
      ↪ // = keccak256("PAUSER_ROLE")

33 0x189ab7a9244df0848122154315af71fe140f3db0fe014031783b0946b8c9d2e3;
      ↪ // = keccak256("UPGRADER_ROLE")

35 0x54fd3c12c8b3fc99211f3f953b8d8233c4cdca02cfeedbead260add43c0f1bd5;
      ↪ // = keccak256("MINTER_GOVERNANCE_ROLE")

37 0x0dc18c621ac7c12ef6a5f7771b48b18abf4dd7238e67a277b031c58b2c7b7c09;
      ↪ // = keccak256("MINTER_BRIDGE_ROLE")
```

CVF-19 INFO

- **Category** Suboptimal
- **Source** REG.sol

Recommendation Unchained initializers should be used here.

Client Comment *OpenZeppelin advises against using unchained initializers, as they require manual verification to ensure no initializers are missing. For instance, invoking __ERC20Pausable_init_unchained() alone would have no effect. However, calling __ERC20Pausable_init() triggers __Pausable_init_unchained(), which then initializes the _paused variable. Similarly, __ERC20Permit_init_unchained on its own does nothing, but __ERC20Permit_init() executes __EIP712_init_unchained(name, "1") to set up the name and version.*
Further details and examples can be found in the documentation and source code: OpenZeppelin Documentation on upgradeable contracts:

- *OpenZeppelin Docs*
- *ERC20PausableUpgradeable.sol Code: GitHub - ERC20PausableUpgradeable.sol*
- *ERC20PermitUpgradeable.sol Code: GitHub - ERC20PermitUpgradeable.sol*

Ongoing issues related to this topic can be monitored through the OpenZeppelin GitHub issue tracker: GitHub Issue #4602.

```
57  __ERC20_init("RealToken Ecosystem Governance", "REG");
    __ERC20Pausable_init();
    __AccessControl_init();
60  __ERC20Permit_init("RealToken Ecosystem Governance");
    __UUPSUpgradeable_init();
```

CVF-20 INFO

- **Category** Suboptimal
- **Source** REG.sol

Recommendation It would be more efficient to pass a single array of structs with two fields, rather than two parallel arrays. This would also make the length check unnecessary.

Client Comment *We prefer to maintain the style of using two arrays, "accounts/amounts" for mint/burn functions, and "recipients/amounts" for the transfer function.*

```
125 address[] calldata accounts,
    uint256[] calldata amounts
```

```
156 address[] calldata recipients,
    uint256[] calldata amounts
```

CVF-21 FIXED

- **Category** Bad datatype
- **Source** REG.sol

Recommendation The type for this argument should be ”IERC20”.

Client Comment *We agree with this approach which saves gas by eliminating the need to cast the token as IERC20.*

```
178 address tokenAddress,
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting