RexMLS Smart Contracts Audit

Table-of-Contents

1 - Introduction

1.1 - Review Goals

ConsenSys Diligence has performed an audit of the RexMLS smart contracts. The focus of this review was to ensure the following properties:

Security: Identifying security related issues within each contract and within the system of contracts.

Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:

- Correctness (does it do what it is supposed to do)
- Readability (How easily it can be read and understood)
- Sections of code with high complexity
- Improving scalability
- Quantity and quality of test coverage

1.2 - Summary

1.2.1 - General Overview of Contract System

REX is a global decentralized real estate data exchange. Participants in the system run independent REX nodes to gain access to the ecosystem. The REX node is a combination of the RexIndexer (a decentralized database), Ethereum and IPFS nodes.

A key component of REX is its ratings-based curation system. REX members can submit new real estate listings to the `ListorData` contract for a fee paid in REXToken. Once submitted, a flagging period commences, during which other users have the opportunity to flag the listing as spam. If a listing is flagged, an arbitration period starts during which REX users can vote for or against the listing with REXtokens.

At the end of the arbitration period, the pool with the most votes wins and the tokens from the losing pool are distributed to the winning pool. Fees and rewards are managed bny the `ListingRewards` contract. The listor receives a reward in REXToken if the listing is successful.

1.2.2 - Smart Contracts in Scope

The audit covered the following smart contracts:

Contract Name	Filename	Description
Subscription	subscription.sol	Subscriber registry. Stores a mapping between Pseudo contract addresses and subscriber status.
Enterprize	enterprize.sol	Smart contract used by the "Enterprize" subscription type. This contract stores a special type of listing called "Enterprize" listings that can be added in bulk.
Rating	rating.sol	Stores rating information for users (i.e. `Pseudo` contract addresses). Inherited by `ListingRewards` .
ListorData	listordata.sol	The listings registry. Contains functions to add, delete and update listings. The listing managent functions can only be called by subscribers, i.e. the sender address must be mapped to a non-zero subscriber type in the subscriber registry.

Contract Name	Filename	Description
ListingRewards	listingrewards.sol	Manages reward requests, flagging of listings, arbitration, and token payouts to users.
Pseudo	pseudo.sol	Proxy contract that is deployed for each user. Rex will send transactions through the Proxy contract (essentially impersonating the users). This is done to cover the gas costs for subscribers.
REXToken	Etherscan	Token contract that manages balances of REXToken holders.

All source code except REXToken was obtained from the RexSoftware Bitbucket repository.

- Branch: develop
- Commit: 84203c9

Notes:

- REXCoordinator and REXToken had already been deployed on the mainnet before the start of the audit (and the REX token sale had aleady been completed). The deployed bytecode does *not* correspond to the code in the Solidity files provided for the audit. REXCoordinator was therefore excluded from the audit.
- For REXToken, the deployed code was obtained from Etherscan instead.
- Besides Ethereum smart contracts, the REX technology stack also includes a NodeJS web app and IPFS storage. Note that only the smart contracts were in scope of this audit.

1.2.3 - Overview of Findings

Overall, we found that the Rex smart contract behaves as intended, and did not discover any game-breaking vulnerabilities. However, the architecture appears to be overly complicated in some places (see issue 3.1 and issue 3.4). We recommend simplifying the architecture for better readability and to prevent mistakes in future updates, and applying readability and efficiency of the code in some locations.

List of Findings

Finding	Severity
Pseudo: Ineffective Re-Entrancy Mitigation	Medium
Inconsistencies Between White Paper and Smart Contracts	Minor
Visibility Not Explicitly Specified	Minor
Pragma Not Locked To Specific Compiler Version	Minor
Pseudo: Excessive Logic in Forward Function	Minor
ListingRewards: Store Reward Parameters in State Variables	Minor
REXToken: Complex Fallback Function	Minor
ListingRewards: Free Unused Storage	Optimization
ListingRewards: Simplify Management of Reward Request State	Optimization
Use Require In Place of Assert	Optimization

1.2.4 - General Recommendations

- Currently, most `ListingRewards` parameters, including the lengths of flagging and arbitration periods, are hardcoded into the contract. This means that the whole contract must be upgraded if any of the parameters is to be changed. It would be preferable to store those values in state variables that are configurable by the contract owner (the same priciples should be applied throughout the whole codebase). See also 3.3 - ListingRewards: Store Reward Parameters in State Variables.

- Even though the NodeJS web app was not in scope, it should be pointed out that access to any server-side functionality that generates transactions to the subscribers' `Pseudo` contracts should be rate-limited. Otherwise, malicious users could very cheaply execute a denial-of-service attack by triggering a large amount of transactions and using up the Ether (gas) available to the web server.

2 - General Findings

2.1 - Inconsistencies Between White Paper and Smart Contracts

Severity: Minor

When comparing the provided code with the specifications outlined in the white paper, we noticed several inconsistencies including:

Whitepaper	Implementation
Flagging period is *5 days* (4.9)	Flagging period is *14 days*
Arbitration period is *3 days* (4.9, 4.10)	Arbitration period is *7 days*
No mention of tie breaker	Arbitration period is extended by *3 days* in case of tie
User can challenge existing listing (4.12)	Not implemented

Recommendation

When developing smart contracts, it is recommended to create detailed final specifications before any code is written. This makes it possible to verify correctness of the code in respect to the specifications. When specifications are still unclear (and the code might still change), no guarantee of correctness can be given.

Client Response

At this point, the values for various parameters are not finalized. This will be done in the near future.

2.2 - Pragma Not Locked To Specific Compiler Version

Severity: Minor

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

This issue affects all smart contracts in the test scope.

Recommendation

Use a pragma statement to enforce the compiler version. For example, in `listingrewards.sol`, change the line:

```
pragma solidity ^0.4.15;
```

To:

```
pragma solidity 0.4.21;
```

2.3 - Visibility Not Explicitly Specified

Severity: Minor

As a best practice, the visibility of functions and state variables should always be labeled explictly. Labeling the visibility explicitly will make it easier to catch incorrect assumptions about who can call the function or access the variable.Functions can be specified as being external, public, internal or private. For state variables, external is not possible. Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can call the function or access the variable.

Recommendation

Explicitly specify visibility throughout all contracts. For example, in the contract Pseudo, the state variables between line 21 and 24 could be declared as public in order to allow for easier debugging of the deployed contract:

```
address public user; // Listor account address
address public coordinator; // RexCoordinator address
address public owner; // Address of account that deploys this contract
bool public  transferred; // Turns true after tokens are transferred(To avoid reentrancy)
```

Moreover, the function `forward` could be declared as `external`, which would lead to considerable gas savings over time.

References

- Solc: Warning: No visibility specified. Defaulting to "public".
- Solium: No visibility specified explicitly for multiple functions.
- Solhint: 'Explicitly mark visibility of state [state-visibility]'
- Smart contract best practices

3 - Specific Findings

3.1 - Pseudo: Excessive Logic in Forward Function

Severity: Minor

The `forward` function in the contract `Pseudo` contains quite complicated logic for deducting REXToken when calls are proxied to the `ListorData` contract:

- The amount of REX Token to be deducted from the user's account is passed in the function argument `value`.
- If a listing request with non-zero value is submitted, the forwarded calldata (function argument `data`) is expected to *also* contain the same token value.
- The call is forwarded to the target function `ListorData`. This function may contain further logic, such as verifying that the token amount is sufficient.
- If the proxy call returns successfully, it is verified that the amount in the forwarded calldata bytes matches the amount in the function argument `value`. This is done with a piece of inline assembly (pseudo.sol, line 76):

```
bytes32 v;

assembly {
  v := mload(164)
}

require(v == bytes32(amount));
```

- If the values match, the specified token amount is transferred to the `ListorData` contract.

This approach has several drawbacks:

- *High complexity.* Implementing a "special case" in the Proxy contract makes the architecture complex and difficult to understand.
- Any function call that transfers value have the same predefined argument list with uint256 value at a specific position. It is easy to break things if `ListorData` is updated.
- Increased gas cost due to additional instructions and function calls being executed (call to `listorData.isFreeListing` in line 73).

Recommendation

We recommend keeping the `Pseudo` contract as simple as possible to avoid issues. Simplify the `forward` function to only forward calldata, and move all business logic to the `ListorData` contract. E.g.:

```
function forward(bytes _data, address destination) isAuthorised isValidAddress(destination) {
    require(destination.call(_data));
}
```

3.2 - Pseudo: Ineffective Re-Entrancy Mitigation

Severity: Medium

The function `forward` in `Pseudo` implements a mutex (boolean `transferred`) that is meant to prevent re-entrancy attacks. However, the implementation is ineffective due to several reasons.

1. `transferred` is declared as a local variable. Because `transferred` is set to `false` at the beginning of the function (line 66), it would always be reset to `false` after re-entrancy, and therefore doesn't offer any protection.

2. Further, `transferred` is set to `true` only *after* the messagecall to REXToken.transfer (line 84). To be effective, it would have to be set *before* the call.

3. In a re-entrancy attack, the proxy call on line 68 directly to call `REXToken.transfer` . Therefore, the mutex would have to be set at the beginning of the function `forward` .

Recommendation

Ideally, the contract architecture should be changed such that a mutex at the proxy level is not needed.

Re-entrancy protection could however be added to the `forward` function as an in-depth security measure. To be effective, the re-entrancy lock should be implemented as a private state variable. A working implementation is available in the Zeppelin-Solidity repository (copied below).

```
pragma solidity ^0.4.18;

/**
 * @title Helps contracts guard agains reentrancy attacks.
 * @author Remco Bloemen <remco@2π.com>
 * @notice If you mark a function `nonReentrant`, you should also
 * mark it `external`.
 */
contract ReentrancyGuard {

  /**
   * @dev We use a single lock for the whole contract.
   */
  bool private reentrancy_lock = false;

  /**
   * @dev Prevents a contract from calling itself, directly or indirectly.
   * @notice If you mark a function `nonReentrant`, you should also
   * mark it `external`. Calling one nonReentrant function from
   * another is not supported. Instead, you can implement a
```

```
   * `private` function doing the actual work, and a `external`
   * wrapper marked as `nonReentrant`.
   */
  modifier nonReentrant() {
    require(!reentrancy_lock);
    reentrancy_lock = true;
    _;
    reentrancy_lock = false;
  }

}
```

References

- Smart Contract Security Best Practices

3.3 - ListingRewards: Store Reward Parameters in State Variables

Severity: Minor

Currently, important parameters of the reward system are hardcoded in several locations. For example, references to the arbitration period (7 days) are found in the functions `checkForVetoWinner` (line 317), `isValidVoteRequest` (lines 212 and 215). `getNumberOfVotesInFavor` (line 285) and others. This makes it necessary to ensure consistency between the hardcoded numbers manually and increases the likelihood of errors being introduced during updates.

Recommendation

Use constant state variables to store key parameters, e.g.:

```
uint256 constant private ARBITRATION_PERIOD = 7 days;
```

References

- Solidity documentation

3.4 - REXToken: Complex Fallback Function

Fallback functions are called when a contract is sent a message with no arguments (or when no function matches). If another contract sends Ether to the this contract via `send()` or `transfer()` , only a gas stipend of 2,300 gas is available. Complex functionality in the fallback function will cause such transfers to fail.

Recommendation

The most that should be done in a fallback function is log an event. Use a proper function if a computation or more gas is required.

```
// bad
function() payable { balances[msg.sender] += msg.value; }

// good
function deposit() payable external { balances[msg.sender] += msg.value; }

function() payable { LogDepositReceived(msg.sender); }
```

References

- Smart Contract Security Best Practices

4 - Optimizations

4.1 - ListingRewards: Free Unused Storage

The function `cancelRewardRequest` in `ListingRewards` is used to cancel existing reward requests. To delete the existing request, the `listing` element of the respective `ListingRewardRequestsStruct` is set to zero (listingrewards.sol,cancelRewardRequest, line 170):

```
requests[_listing].listing = bytes32(0);
```

In function listorPayout (listingrewards.sol, line 268):

```
// Avoid reentrancy/Clear the data
 delete requests[_listing].listing;
```

The same code is used to delete a listing in the function `vetosTiePayout` (listingrewards.sol, line 417).

However, this the remaining contents of the struct are not cleared, forfeiting the gas refund for returning the storage.

Recommendation

Delete the complete request struct as follows:

```
    delete requests[_listing];
```

References

- Solidity developer documentation

4.2 - ListingRewards: Simplify Management of Reward Request State

The `ListorData` and `Enterprize` contracts store the reward request state for every listing (e.g. `bool listingInfoMapping[_hash].rewardRequested in listorData`). However, this state is only used in `ListingRewards` . The result of this architecture is that functions in `ListingRewards` must make message calls into `RexCoordinator` , `ListorData` and `Enterprize` contracts to get or set the request status.

Note for example the calls at the beginning and end of `newRewardRequest` in `ListingRewards` :

```
function addListing(bytes32 _hash, uint256 amount, bytes4 _feedCode) {
        // Check whether user making the call is registered

require(SubscriptionInterface(RexCoordinatorInterface(coordinator).getAddress("subscription")).subscriptior
 != 0);
        require(listingAmount <= amount);
        // // Prevent 2 listings from having the same hash and hence being overwritten
        require(listingInfoMapping[_hash].listing == bytes32(0));

        pseudoListorMapping[msg.sender].listingCount += 1;
        pseudoListingMapping[msg.sender].push(_hash);

        listingInfoMapping[_hash].pseudoAddress = msg.sender;
        listingInfoMapping[_hash].rewardRequested = false;
        listingInfoMapping[_hash].listing = _hash;
        listingInfoMapping[_hash].feedCode = _feedCode;

        uint balance =
TokenInterface(RexCoordinatorInterface(coordinator).getAddress("token")).balanceOf(msg.sender);

        ListingAdded(_hash, msg.sender, _feedCode, balance);
    }
```

Recommendation

Keep all state related to reward requests in the ListingRewards contract only to reduce complexity and save gas.

The reward request state could be saved in a mapping in `ListingRewards`, such as:

```
mapping (address => boolean) rewardRequested
```

Client Response

The architecture was designed with upgradeability in mind: If `ListingRewards` was updated, the reward request details of all listings would have to be retrieved and updated in the new contract. This was seen as not feasible.

4.3 Use Require in Place of Assert

Currently, whenever a contract throws the remaining gas is used up. However, after the Byzantium hard fork, calling the `REVERT` instruction will result in the excess gas being refunded to the caller. Note that `require()` uses the 0xfd (REVERT) opcode to cause an error condition, while `assert()` uses the 0xfe opcode. This means that gas will be refunded only if `require()` is used.

Rex uses `assert()` for checks that are likely to fail often. For example, it is used to check the return value of the proxy call in `Pseudo` (function `forward`, line 68):

```
        assert(destination.call(_data));
```

Recommendation

Consider using `require` instead of `assert` except when enforcing invariants (i.e. conditions that should *never* occur. For example, in `Pseudo`, use `require` to verify the return value of the forwarded calls (function `forward`, line 68):

```
        require(destination.call(_data));
```

This will save significant gas in the long run, as excess gas will be refunded if the forwarded call fails.

5 - Tool-Based Verification

An automated anaysis was performed with the Mythril analysis tool. The analysis results were then investigated by the auditor. The following section contains the tool output along with a manual assessment of the results. None of the reported issues were found to be exploitable in practice.

5.1 - CALL with gas to dynamic address

- Type: Warning
- Contract: Enterprize
- Function name: `addListings(bytes32[],bytes4[])`
- PC address: 3638

Description

The function `addListings(bytes32[],bytes4[])` contains a function call to an address provided as a function argument. The available gas is forwarded to the called contract. Make sure that the logic of the calling contract is not adversely affected if the called contract misbehaves (e.g. reentrancy).

In *contracts/enterprize.sol:44*

```
SubscriptionInterface(RexCoordinatorInterface(coordinator).getAddress("subscription")).subscriptionStatus(m
```

Auditor Comment

The message call was reviewed. There is no risk of re-entrancy or other types of attack.

5.2 - Integer Underflow

- Type: Warning
- Contract: ListorData
- Function name: `deleteListing(bytes32)`
- PC address: 5455

Description

A possible integer underflow exists in the function `deleteListing(bytes32)` . The SUB instruction at address 5455 may result in a value < 0.

In *contracts/listordata.sol:103*

```
pseudoListorMapping[msg.sender].listingCount -= 1
```

Auditor Comment

While there is no explicit underflow check, the underflow is prevented by the smart contract logic. There must be at least one existing listing to successfully execute the function. This is due to the `require` statement in line 100:

```
require(listingInfoMapping[rootHash].pseudoAddress == msg.sender);
```

5.3 - Integer Underflow

- Type: Warning
- Contract: ListingRewards
- Function name: `listorPayout(bytes32)`
- PC address: 15311

Description

A possible integer underflow exists in the function `listorPayout(bytes32)` . The SUB instruction at address 15311 may result in a value < 0.

In *contracts/rating.sol:25*

```
ratingInfo[user].numOfActions - 1
```

Auditor Comment

While there is no explicit underflow check, the underflow is prevented by the smart contract logic.

5.4 - Integer Underflow

- Type: Warning
- Contract: ListingRewards
- Function name: `voteInFavorOfListing(bytes32,uint256)`
- PC address: 16624

Description

A possible integer underflow exists in the function `voteInFavorOfListing(bytes32,uint256)` . The SUB instruction at address 16624 may result in a value < 0.

In *contracts/listingrewards.sol:210*

```
now - requests[_listing].vetoDateCreated
```

Auditor Comment

Assuming that the blockchain behaves normally, the `now` timestamp would always be higher than the `requests[_listing].vetoDateCreated` .

5.5 - Integer Underflow

- Type: Warning
- Contract: ListingRewards
- Function name: `flagListing(bytes32,uint256)`
- PC address: 8752

Description

A possible integer underflow exists in the function `flagListing(bytes32,uint256)` . The SUB instruction at address 8752 may result in a value < 0.

In *contracts/listingrewards.sol:182*

```
now - requests[_listing].dateCreated
```

Auditor Comment

Assuming that the blockchain behaves normally, the `now` timestamp would always be higher than the `requests[_listing].dateCreated` .

5.6 - CALL with gas to dynamic address

- Type: Warning
- Contract: Pseudo
- Function name: `forward(bytes,address,uint256,uint256)`
- PC address: 1681

Description

The function `forward(bytes,address,uint256,uint256)` contains a function call to an address provided as a function argument. The available gas is forwarded to the called contract. Make sure that the logic of the calling contract is not adversely affected if the called contract misbehaves (e.g. reentrancy).

Auditor Comment

`Pseudo` is a proxy contract, so this is expected behavior.

6 - Test Coverage Analysis

Testing is implemented using the Truffle Framework.

Automated measurement was done using Solidity-Coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	70.57	64.48	70.3	70.41	
enterprize.sol	66.67	31.82	60	60.61	... 4,95,96,100
listingrewards.sol	98.64	89.06	100	98.71	142,157
listordata.sol	86.84	70	90.91	80	... 107,108,110
migrations.sol	75	50	75	60	20,21
pseudo.sol	85.71	81.82	66.67	85.19	42,43,58,92
rating.sol	100	100	100	100	
rexcoordinator.sol	53.33	50	41.67	53.85	... 163,167,168
rextoken.sol	22.22	18.97	26.32	22.68	... 284,285,289
safemath.sol	100	50	100	100	
standardtoken.sol	100	100	100	100	
subscription.sol	100	100	100	100	
All files	**70.57**	**64.48**	**70.3**	**70.41**	

- Coverage Rating: **** (4 out of 5)

Appendix 1 - Audit Details

A.1.1 - Audit Participants

Security audit was performed by Bernhard Mueller.

A.1.2 - File Hashes

The SHA256 hashes of the source code files provided were as follows.

```
d46967cebf159a95ecb650e620b2b4185a04a9200fd85fb2b3361f722bba075c  enterprize.sol
120b21d80ea8323d1c1523909762b1a40e5fcae3a83cc0de281e023f5c4ea883  listingrewards.sol
b4edc5781ba6ecaf22799a5bc01146fa957085a7cad2f9f233725f083923068c  listordata.sol
d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376  migrations.sol
abaacadcebee9fe806e60b40908190c3ef8b4cfb772c038eae4d3eb3c60b0600  pseudo.sol
0bbe2ed1c68b67dfe60a308d7b1ce695130a02175e1093407bbfc823dd9fe84b  rating.sol
c0a411c065ca0c2a4b55d70c5a72149eacbf61f675e8640b2fc3aca3d6c82dd5  rexcoordinator.sol
dc5a3660c7610b974fca95348410b268b66cc23118c7b49228c1d75f440a2cca  rextoken.sol
ae555adbe9dd2aceb340770e482ee937fa9e4d4d9758ee7f24cdfbf71f9bd2c9  safemath.sol
3dc6137c5d271bc265d37ce963a1211f0b07a8263fe0af0c22c9b898e4c8d517  standardtoken.sol
db5204603645205b3fa360d2aceb31feb285660da9bb108faceb55e629754047  subscription.sol
```

Appendix 2 - Terminology

A.2.1 - Severity

Measurement of magnitude of an issue.

A.2.1.1 - Minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgement as to whether addressing these issues improves the codebase.

A.2.1.2 - Medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.2.1.3 - Major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.2.1.4 - Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.