PUBLIC

Security Audit

of BLOCKIMMO's Smart Contracts

April 1, 2019

Produced for

BLOCKIMMO

by

CHAINSECURITY

Table Of Content

Foreword

We first and foremost thank BLOCKIMMO for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The BLOCKIMMO smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts.

The communication with BLOCKIMMO was always professional and BLOCKIMMO gave a good introduction into their code. CHAINSECURITY found several medium and low severity issues which have all been fixed or addressed by BLOCKIMMO. Additional critical severity issues have been uncovered in a custom exchange smart contract. BLOCKIMMO reacted to these issues by removing the exchange smart contract and switching to Uniswap, a popular decentralized exchange. The Uniswap exchange was outside the scope of this audit, but ChainSecurity has no reason to suspect security vulnerabilities in it.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on February 15, 2019. The corresponding git commits are:

`blockimmo-contracts` branch `v2` at commit `f0c92cc1ed43c9ec2390def3e2e3a50b0f4f3847` and
`stx-contracts` branch `v2` at commit `a25efa3d6e53d9ac9fa8bd9d8be401b17a5e6fc8`
The latest update has been received on March 18, 2019. The corresponding Git commits are:
`blockimmo-contracts` branch `v2` at commit `4e4ee2ef6248a235e5a81b241d2b3fb43419f256` and
`stx-contracts` branch `v2` at commit `7376ff83b132f26331053e7c0ac8744713d1b582`

File	SHA-256 checksum
DividendDistributingToken.sol	`8e1593d2419663d1a2d82607b8bfb40c26e03be56058886763f4a35de42cad0e`
TokenSale.sol	`cc43a56171d9890a020ef12eb2975a7c9f84b9e212d76153b123b024281afa1e`
WhitelistProxy.sol	`951f7aa5dc3b9eac73f748d7c6b54fe909eb5045933bcc3825c6aee16e1c2fbd`
ShareholderDAO.sol	`8a77e534842c2cb6d2452cb9ebfeb208d6e6f0be17c3419452da88a81fb3a231`
LandRegistry.sol	`842c76c2fccd74ee19a11ba8538fe4c67642c28f5838d99a2247daf5949cec4c`
LandRegistryProxy.sol	`ff9a4953e64d13e716796f84c67f770ad69507e8bf49eb2b0d28244944d1ca03`
TokenizedProperty.sol	`78cdd66f8ca462d1b77e7095156dd50c660db56551020c0b7b69e72e0c1f0d1e`
LoanEscrow.sol	`fe645e431400cb3ac6164e243827fb35d8a125cd500d65c0ed7a795cc6dd82a5`
PaymentsLayer.sol	`4fc187a0b4a0be8dbe8cd52ab2c89f7ca2db7bab02c5275a3df99dd9d2172d0a`

Depth of Audit

The scope of the security audit conducted by ChainSecurity was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

[1] `https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology`

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of the potential security impact, we assign the following labels:

- ✓ No Issue : no security impact
- ✓ Fixed : during the course of the audit process, the issue has been addressed technically
- ✓ Addressed : issue addressed otherwise by improving documentation or further specification
- ✓ Acknowledged : issue is meant to be fixed in the future without immediate changes to the code

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, ChainSecurity has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

BLOCKIMMO offers the tokenization of real estate via smart contracts on the Ethereum blockchain. Furthermore, BLOCKIMMO enables the investors to manage their real estate investment via a DAO (decentralized autonomous organization), receive dividend payments on-chain. Investments are converted via Kyber into DAI. To get interest on currently not invested funds (held in DAI), these funds are continuously invested into Compound.finance. As most of the features under BLOCKIMMO's control, investors need to heavily trust BLOCKIMMO regarding the correct system setup and system management.

In detail BLOCKIMMO's system consists of a land registry contract, which bridges the gap between the common off-chain swiss land registry by mapping the unique eGrid number to a corresponding smart contract, which tokenizes the property. Each property is split into one million tokens. But not all of these tokens need to be publicly available for sale. The token sale is managed by a token sale contract. Investors need to be whitelisted by BLOCKIMMO on a whitelist contract to be able to invest and transfer the security token. Another smart contract distributes the dividends placed into this contract by BLOCKIMMO or a managing company between the investors. BLOCKIMMO's infrastructure allows investors to vote on important decisions (a proposal needs to be filed on-chain). This feature is managed by a DAO contract. Proxies allow to update the land registry and whitelist contracts.

Feature overview:

Land registry A contract connecting the off-chain legal swiss land registry entries to the corresponding security token smart contracts.

Security token A token which represents a share of the real estate property.

Token sale A contract to sell initially created security tokens to investors.

Dividend distributing contract A contract to distribute the dividend payments between the investors of the corresponding property.

DAO contract A contract to vote on proposals to manage the property (only important decisions).

Whitelist A contract managing who is allowed to invest and transfer the issued tokens.

Loan escrow A contract to keep the DAI funds and invest them if necessary to get interest on funds.

Best Practices in BLOCKIMMO's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when BLOCKIMMO's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the BLOCKIMMO's project can be audited by CHAINSECURITY.

- [✓] The code is provided as a Git repository to allow the review of future code changes.
- [✓] Code duplication is minimal, or justified and documented.
- [✓] The code compiles with the latest Solidity compiler version. If BLOCKIMMO uses an older version, the reasons are documented.
- [✓] There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to BLOCKIMMO.

- [ ] There are migration scripts.
- [✓] There are tests.
- [ ] The tests are related to the migration scripts and a clear separation is made between the two.
- [ ] The tests are easy to run for CHAINSECURITY, using the documentation provided by BLOCKIMMO.
- [ ] The test coverage is available or can be obtained easily.
- [✓] The output of the build process (including possible flattened files) is not committed to the Git repository.
- [ ] The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- [✓] There is no dead code.
- [✓] The code is well documented.
- [✓] The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- [✓] Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- [✓] There are no getter functions for public variables, or the reason why these getters are in the code is given.
- [✓] Function are grouped together according either to the Solidity guidelines[2], or to their functionality.

[2] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Security Issues

This section relates our investigation into security issues. It is meant to highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Minority Token Holder can single-handedly win a Vote M ✓ Addressed

An malicious investor can manipulate the voting if the investor at least owns one third of the shares. To manipulate the voting, the malicious investor sets up a new proposal (e.g. to transfer the ownership) by calling `extendProposal()` (at best with a very short voting period). After voting for their own proposal the malicious investor burns their token and calls `finalize()`. The vote is evaluated by

```
if (_tallyFor > property.totalSupply() / 2) {
```

ShareholderDAO.sol

The `totalsSupply` now dropped due to the burn to only two thirds from the initial supply but still there are valid votes with a voting power of one third of the initial supply, which now is a majority. Thus, the vote would pass.

Likelihood: Medium
Impact: Medium

Addressed: BLOCKIMMO explained that the implications are minor. A DAO vote are suggestions that BLOCKIMMO and/or the property's management company **will try to take**. Furthermore, `ShareholderDAO` has no real power (i.e. it can't move funds/value, modify state, etc. . .).

Locked tokens M ✓ Acknowledged

If tokens especially DAI are accidentally sent to the following contracts and not via the correct function, the tokens will be locked.

- `TokenSale`
- `TokenizedProperty`
- `DividendDistributingToken`
- `Exchange`
- `LoanEscrow`
- `PaymentsLayer`

Note:
ETH and token can be forced into any contract and be locked there if no "recover" function exists. If contracts are not supposed to handle/process Ether or token transfers, CHAINSECURITY does not further report this kind of locked tokens. But CHAINSECURITY reports locked token or locked ETH for contracts which are supposed to handle/process tokens or ETH in some way.

Likelihood: Medium
Impact: Medium

Acknowledged: BLOCKIMMO acknowledged the issue. BLOCKIMMO wants to prevent this on UI level because this is the intended way to use their system.

Unintentional call to `renounceOwnership()` could block LandRegistry L ✓ Acknowledged

The BLOCKIMMO contract `LandRegistry` inherits from the `Ownable` contract and therefore contains the function `renounceOwnership()`. This function can be called by the existing owner to renounce his ownership and leave the contract without any owner. Any accidental call to this function from the current owner would leave the `LandRegistry` contract without any owner. Then, no more properties could be tokenized or untokenized.

Likelihood: Low
Impact: Medium

Acknowledged: BLOCKIMMO acknowledged the issue and plans to update the `LandRegistry` via its proxy if this issue arises.

Trust Issues

This section mentions functionality which is not fixed inside the smart contract and hence requires additional trust into BLOCKIMMO, including in BLOCKIMMO's ability to deal with such powers appropriately.

`TokenSale` ETH sent to wallet M ✓ Fixed

`Crowdsale.buyTokens()` function is used to send the DAI to the `TokenSale` contract. This function is a `payable` function which can also receive ETH. However, BLOCKIMMO has modified the code of the `Crowdsale` contract and apparently only wants to receive DAI (or at least only accounts for DAI).

But if accidentally ETH are sent along with the transaction call, the function would accept it and "silently" forward the ETH to the `_wallet` address (provided by the BLOCKIMMO at the time of deployment of `TokenSale` contract). Without counting the contribution in any way. The same way the fallback function present in the `Crowdsale` contract is also used to send ETH.

Fixed: BLOCKIMMO fixed the issue by adding `msg.value == 0` to the require, to prevent sending ETH when calling the `buyTokens` function.

Unlimited token minting by BLOCKIMMO M ✓ Acknowledged

Even though it is an intended feature, it is also a trust issue that BLOCKIMMO can mint unlimited number of new tokens using `TokenizeProperty.mint()` function. When minting is done by BLOCKIMMO, it would affect the dividend and share ownership of the existing holder.

Acknowledged: BLOCKIMMO is aware of the issue and acknowledged the issue. BLOCKIMMO told CHAIN-SECURITY that they tried to minimalize the centralization. But the business model does need some trust inevitably.

No check to ensure that sufficient tokens are transferred to TokenSale L ✓ Acknowledged

To payout tokens to the investors, BLOCKIMMO needs to deposit the minted tokens in the `TokenSale` contract. When finalizing the token sale `totalTokens` is set to `token().balanceOf(address(this))`. Additionally, BLOCKIMMO sets a `cap` in the `TokenSale`'s constructor. Thus, the token contract can sell tokens up to the `cap`. BLOCKIMMO never checks the integrity of these values by e.g. comparing them. Hence, it could happen that the `TokenSale` contract sells the amount of token specified in `cap` but there was not the correct amount of tokens transferred to the `TokenSale` contract. Maybe less than the `cap`. In the end, this would be noticed too late (when finalizing the sale).

BLOCKIMMO could consider checking if these values match earlier to guarantee no issues arise in the end.

Acknowledged: BLOCKIMMO acknowledged the issue with reference to the issue and explanation in the issue above. Some trust needs to be put in BLOCKIMMO due to their regulated business model.

Potentially no tokens inside the sale L ✓ Acknowledged

The `TokenSale` contract contains no checks whether it actually has control over any tokens. Only during finalization, the contract checks the amount of available tokens. Hence, only when distributing tokens through the `withdrawTokens` function, a shortage of tokens would become apparent. Therefore, users need to check that the sold tokens are actually available. Instead, the contract could ensure this.

Acknowledged: BLOCKIMMO acknowledged the issue with reference to the two issues and explanations above.

High dependency on third party providers L ✓ Acknowledged

BLOCKIMMO excessively uses the services of three third-party providers, namely DAI, Compound.finance and Kyber. Although BLOCKIMMO can change from DAI to other stable coins, there is a remaining risk that the DAI service will crash for some reason. The remaining third-party services also do have risks. For example; flawed code in Compound.finance could result in the loss of a significant amount of funds.

However, CHAINSECURITY will further assume these services will work as expected but still needs to point out this risk factor.

Acknowledged: BLOCKIMMO is aware of the issue and also told CHAINSECURITY that they are prepared to quickly adopt.

Design Issues

This section lists general recommendations about the design and style of BLOCKIMMO's project. They highlight possible ways for BLOCKIMMO to further improve the code.

Code optimization not used L ✓ Fixed

There is no optimization flag in the Truffle configuration. Truffle compiles Solidity without optimization by default, which can lead to higher gas consumption. Why has this choice been made?

Fixed: BLOCKIMMO fixed the problem by adding optimization flag in truffle configuration file.

Token burn and mint operations do not update the Dividend M ✓ Fixed

BLOCKIMMO calls `creditAccount()` to account for token changes when transferring tokens to update the dividend. However, when burning and minting token, it does not call `creditAccount()`. Hence, dividend are not updated.

Hence, in case of a token burn, dividends are "lost" and will remain as locked tokens inside the contract. On the other hand, in case of a token mint, there are not sufficient funds to pay out all dividends. Therefore, legitimate token holders would not be able to collect their dividends.

Fixed: BLOCKIMMO solved the issue by adding `creditAccount(_to)` to the `mint()` and `burn()` functions.

Code duplication of interface definition M ✓ Acknowledged

There are a few places where a contract just for function interface purposes defined in multiple contracts. For example `LandRegistryProxyInterface` contract/interface is defined in `TokenizedProperty.sol`, `TokenSale.sol`, and `ShareholderDAO.sol`.

Same is the case with the other interface like `WhitelistInterface` and `WhitelistProxyInterface`.

CHAINSECURITY recommend keeping the interface defined in separate contract/interface files to minimize the code duplication and code maintainability.

Acknowledged: BLOCKIMMO acknowledged and justified the issue. Therefore, BLOCKIMMO is fine with these small pieces of duplicated code.

Front running possible in `extendProposal` function L ✓ Acknowledged

The function `extendProposal()` present in the `ShareholderDAO` is called from the `isAuthorized` accounts to propose a new proposal for the DAO. Here `isAuthorized` is the function modifier which checks that an account is a shareholder of the property or BLOCKIMMO account.

However, anyone who `isAuthorized` can front run a `extendProposal()` transaction with duplicated transaction with higher gas price. Following are the steps an `isAuthorized` account can do to front run the transaction:

- An attacker listens for the transaction on `ShareholderDao` contract.
- BLOCKIMMO from its `owner` account calls `ShareholderDao.extendProposal()` function with some parameters.
- The attacker finds that a new transaction is initiated, he tries to front-run this transaction with a higher gas price.
- If the attacker's transaction is executed first then his address is registered as `proposer` in the `Proposal` struct.
  However, for this, an attacker must be a shareholder of the corresponding property.

The implications are low but the malicious account would be the `proposer` instead of the other account. The following transaction fails.

Acknowledged: BLOCKIMMO informed CHAINSECURITY that the ShareholderDAO is not used in the current live system. Issues related to ShareholderDAO will be resolved or addressed by BLOCKIMMO if needed, in the future.

Inconsistent use of SafeMath library L ✓ Fixed

SafeMath library is used in the project to do the mathematical operations. However, the library is not used consistently. Following are the places where SafeMath can be used.

First case in ShareholderDAO.tallyVotes().

```
  function tallyVotes(uint256 _tallyFor) internal view returns (Outcomes
      outcome) {
    if (_tallyFor > property.totalSupply() / 2) {
      outcome = Outcomes.Accept;
    } else {
      outcome = Outcomes.Reject;
    }
  }
```

ShareholderDAO.sol

Second case in the constructor of TokenizedProperty.

```
  constructor(string memory _eGrid, string memory _grundstuck) public
      ERC20Detailed(_eGrid, _grundstuck, 18) {
    uint256 totalSupply = NUM_TOKENS * (uint256(10) ** decimals());
    _mint(msg.sender, totalSupply);
  }
```

TokenizedProperty.sol

Fixed: BLOCKIMMO fixed the issue by using safemath in the sections mentioned above.

Calls to untrusted contracts in PaymentsLayer L ✓ Addressed

BLOCKIMMO implements a critical feature in the functions PaymentsLayer.forwardEth(). The PaymentsLayer gives any user the chance to call three untrusted contracts (namely _kyberNetworkProxy, _srcToken and _destinationAddress) by providing the addresses as function arguments in forwardEth(). This feature is basically a proxy contract to call anything. It also does not prevent reentrancy. Such a critical feature should be handled with extreme care.

CHAINSECURITY suggest reviewing if this functionality and the implications are really needed and intended. If this is the case, BLOCKIMMO might consider to put a restriction on these features or at least prevent reentrancy. Currently, the implications are limited as the PaymentsLayer contract is not supposed to hold any funds, but if this changes (by accident or by design) such funds are at risk.

Addressed: BLOCKIMMO addressed the issue and explained that the payment layer is not supposed to hold any funds or have any special permissions. In case the contract will have special permissions or hold funds, BLOCKIMMO is aware of the issue and told CHAINSECURITY to implement the necessary security features. Nevertheless, BLOCKIMMO made sure that the function can not be re-entered.

Vote outcome not taking care for unissued tokens L ✓ Fixed

BLOCKIMMO calculates the outcome of a vote by checking if a majority exists. This is done by

```
if (_tallyFor > property.totalSupply() / 2)
```

ShareholderDAO.sol

BLOCKIMMO told CHAINSECURITY that some sales do not sell all one million tokens. Nevertheless, always one million tokens are minted. The tokens which are not issued are held back in a contract. The code just checks the totalSupply(), thus not accounting for that.

Fixed: BLOCKIMMO fixed the problem by deducting the unissued tokens from total supply of tokens before calculating the tally for votes.

Tokens locked until crowdsale finishes L ✓ Fixed

If all tokens are sold in a token sale but there is still time left for crowdsale to finish, the investors cannot withdraw their tokens. The tokens are in locked state until the token sale is finalized. As owner allowed to call `finalize()` only after `_closingTime` has passed.

Fixed: BLOCKIMMO fixed the problem by checking that either the cap is reached or the crowdsale expired after timeout.

`amountInterest` can be influenced L ✓ Fixed

BLOCKIMMO uses the following code to calculate the interest amount which shall be withdrawn from Compound.finance.

```
uint256 amountInterest = moneyMarket.getSupplyBalance(address(this),
    DAI_ADDRESS).add(dai.balanceOf(address(this))).add(pulled).sub(deposited);
```

LoanEscrow.sol

By checking the balance of the contract with `dai.balanceOf(address(this)` the `amountInterest` can be influenced if a malicious accounts sends DAI to the contract. In consequence the `amountInterest` will increase and too many DAI is withdrawn from Compound.finance.

Fixed: BLOCKIMMO solved the issue by checking if the amount of interest can be paid with the DAI balance in the contract and if not, only withdrawing the needed difference.

Inefficient storage layout L ✓ Fixed

BLOCKIMMO defines a the struct `Order` in `Exchange` and `Proposal` in `ShareholderDAO`. As the Solidity compiler does not perform optimizations as one might expect[3] it is necessary to ensure that structs and variables are aligned with 32 byte word boundaries. CHAINSECURITY identified an opportunity to do so and recommends BLOCKIMMO to adopt such reorderings to save significant gas costs.

Optimized:

```
struct Proposal {

    uint256 closingTime;
    uint256 tallyFor;
    uint256 tallyAgainst;
    uint256 blockNumber;

    address owner;
    address proposer;

    ProposalStatus status;
    Actions action;

    string message;
    mapping (address => Vote) voters;
}
```

ShareholderDAO.sol

This reordering saves around 19750 gas per call to `extendProposal` function

[3]https://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-variables-in-storage

Fixed: BLOCKIMMO informed CHAINSECURITY that the ShareholderDAO is not used in the currentlive system. Issues related to ShareholderDAO will be resolved or addressed by BLOCKIMMO if needed, in the future.

Inefficient transfer operation L ✓ Fixed

In ERC20 tokens, the transfer operation is typically the most frequently used function. In the BLOCKIMMO contracts, this function is fairly inefficient, as it wastes gas in common cases. When transfer is executed, the freshness of the dividend information is checked for sender and receiver. Given that BLOCKIMMO plans to emit dividends rather infrequently, the dividend information is quite possibly up-to-date for both accounts. However, even then four SSTORE operations, costing $20{,}000$ gas are triggered. These SSTORE operations could be omitted if unnecessary, in order to save gas.

Fixed: BLOCKIMMO fixed the issue by checking if the values differ or not and only updating them if needed.

Interest could be withdrawn differently L ✓ Fixed

When withdrawInterest is called, the funds are withdrawn from the money market and then transferred to the caller. During the withdrawal any funds residing inside the contract are ignored. Instead, as in pull only the necessary amount could be withdrawn from the money market.

This would also prevent the theoretical issue, where someone transfers DAI into the contract to prevent a successful call to withdrawInterest, as the full amount cannot be withdrawn from the money market.

Fixed: BLOCKIMMO fixed the issue by withdwing only the amount needed.

Missing check in transferFrom() L ✓ Acknowledged

The contract TokenizedProperty has a transfer() and transferFrom() functions defined. Each token holder can set his minimum transfer limit by calling setMinTransfer() function.

This minimum transfer is checked in the transfer() function like below:

```
function transfer(address _to, uint256 _value) public isValid returns (bool) {
    require(_value >= minTransferAccepted[_to], "_value must exceed _to's 
        minTransferAccepted");
    transferBookKeeping(msg.sender, _to);
    return super.transfer(_to, _value);
  }
```

TokenizedProperty.sol

However, the same check is missing from the transferFrom() function.

```
  function transferFrom(address _from, address _to, uint256 _value) public
      isValid returns (bool) {
    transferBookKeeping(_from, _to);
    return super.transferFrom(_from, _to, _value);
  }
```

TokenizedProperty.sol

As discussed with BLOCKIMMO before, there needs to be an explicit approval before. BLOCKIMMO nevertheless could consider adding this additional check.

Acknowledged: BLOCKIMMO acknowledged the issue because the investor has to approve an allowance anyway and can also set it back to zero.

`string` as key is used in mapping L ✓ Acknowledged

In the contract `LandRegistry` there is a mapping `landRegistry` which maps the `property` address to the `eGrid` number.

```
mapping(string => address) private landRegistry;
```

LandRegistry.sol

CHAINSECURITY wants to make client aware of the fact that there is the possibility to generate similar looking strings but different ultimately different keys (accidentally or intentionally). Here is an example how this could be used maliciously[4].

Acknowledged: BLOCKIMMO acknowledged the issue because as an offical identifier the string needs to be unique and exact.

[4]`https://github.com/Arachnid/uscc/tree/master/submissions-2017/marcogiglio`

Recommendations / Suggestions

✓ BLOCKIMMO has maintained a copy of OpenZeppelin library code to their repository at this location[5], which is being used in the project. BLOCKIMMO has also modified the implementation of `Crowdsale` contract slightly[6] to make it support for DAI token.

Battle-tested libraries should be used as much as possible, since they are less likely to contain bugs than custom code. There are different ways to use library code however, and copy-pasting code without further justification could end up problematic.

CHAINSECURITY does understand BLOCKIMMO's justification. BLOCKIMMO pointed out, they have an open merge request at the OZ repository and will wait for it to be merged.

✓ BLOCKIMMO often stores addresses as constants and later cast them into interfaces like in the `TokenSale` contract:

```
address public constant LAND_REGISTRY_PROXY_ADDRESS = 0
    x0f5Ea0A652E851678Ebf77B69484bFcD31F9459B;
address public constant WHITELIST_PROXY_ADDRESS = 0
    xEC8bE1A5630364292E56D01129E8ee8A9578d7D8;

LandRegistryProxyInterface public registryProxy =
    LandRegistryProxyInterface(LAND_REGISTRY_PROXY_ADDRESS);
WhitelistProxyInterface public whitelistProxy = WhitelistProxyInterface(
    WHITELIST_PROXY_ADDRESS);
```

TokenSale.sol

A single variable is sufficient, which reduces the storage consumption and the code size, as currently getter functions are generated for both variables. These getters return identical results.

☐ Both the `TokenSale` and `TokenizedProperty` contract inherit from the `Pausable` contract. Usually, one might expect if a sale or token contract can be paused that the transfers or the sale is paused. But in BLOCKIMMO's implementation, it only pauses the funds being invested into the Compound.finance. BLOCKIMMO could consider making this more unambiguous.

✓ BLOCKIMMO has a function called `forwardEth()` in the `PaymentsLayer` contract. This name is misleading as this function can handle more or less any compatible ERC20 token. BLOCKIMMO might consider renaming the function.

[5] https://github.com/blockimmo-ch/openzeppelin-solidity/

[6] https://github.com/blockimmo-ch/openzeppelin-solidity/commit/8a7d4a702f59be243f37d2577d6481591aa34ff1

Addendum and general considerations for Ethereum

Blockchains an especially the Ethereum Blockchain might often behave different from common software. There are many pitfalls which apply to all smart contracts on the Ethereum blockchain. ChainSecurity mentions general issues in this section. These are relevant for the project but not problematic. Additionally, ChainSecurity mentions information in this section, to clarify or support the information in the security report. This section, therefore, is a reminder to make Blockimmo and potential users aware of these general issues.

Dependence on block information ✓ Acknowledged

Blockimmo uses `block.timestamp` in the `ShareholderDAO` contract. Although block manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by around 15 sec. compared to the actual time.

ChainSecurity notes that Blockimmo and its users should be aware of this and adhere to the 15 seconds rule[7]

`SafeERC20.safeApprove` minor fix in the OpenZeppelin 2.1.3 ✓ Acknowledged

In the new release of OpenZeppelin 2.1.3 an issue with `SafeERC20.safeApprove()` function is fixed. ChainSecurity recommend checking the fix present in OpenZeppelin library latest 2.1.3 version. However, the release with the function itself is not being used by Blockimmo. Therfore, ChainSecurity wants to highlight that extra care needs to be taken if Blockimmo plans to move to 2.1.3. [8]

[7] `https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule`
[8] `https://github.com/OpenZeppelin/openzeppelin-solidity/releases/tag/v2.1.3`

Disclaimer