# Smart Contracts Security Review for PandiFi

# Report

**document version:** 1.1

**author:** Damian Rusinek (SecuRing)

**test time period:** 2021-10-29 – 2021-11-05

**report date:** 2021-12-08

## TABLE OF CONTENTS

# 1. EXECUTIVE SUMMARY

## 1.1. Summary of retest results

- Two of four vulnerabilities have been fixed. The other two have been acknowledged by the development team and the risk has been decreased.
- All recommendation have been implemented.

## 1.2. Summary of test results

- The project operates on the border of decentralized network and off chain resources, therefore entities enabling this connection have a great influence on it.
- There is one vulnerability with **high** impact on risk found. Its potential exploitation result is:
  - Theft of stable coins equal to the loan *cashHeld* parameter during homogenization (4.1).
- There are two vulnerabilities with **medium** impact on risk found. Their potential exploitation results are:
  - Depending on the test case (4.2):
    - Draining the pool,
    - Introducing inconsistencies in protocol workflow,
    - Minting the same loan multiple times or a fake one.
  - Setting the non-stable coin address (4.3).
- There is one vulnerability with **low** impact on risk found. Its potential exploitation result is:
  - Invalid value of *lockedTokenCount* parameter which in future might influence the security of the system (4.4).
- Additionally, **nine** recommendations have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.
- Protocol security depends on other contracts that protocol is integrated with and which are not present in repository (and are out of scope). These are eligibility contracts. Security review of those contracts is highly recommended before the main deploy.

# 2. PROJECT OVERVIEW

## 2.1. Project description

The analyzed system makes it possible to exchange digitized loans and mortgages on the Ethereum blockchain. It distinguishes roles such as *admin*, *minter*, *service*, *trader*, *keeper*, *sweeper*, *borrower and investor*:

- ***Admin*** - role with the highest privileges responsible for protocol governance.
- ***Minter*** – role with privileges to mint new loan tokens.
- ***Servicer*** – role with privileges to distribute cash flow and modify loan state.
- ***Trader*** **–** role with privileges to set reservations.
- ***Keeper*** – role with privileges to run various functions (e.g. emit events to reindex post-issuance) to ensure that API data is current.
- ***Sweeper*** – role with privileges to receive cash flow distributions made to homogenized loans.
- ***Borrower*** - user that is permitted to digitize his loans to ERC-721 standard token.
- ***Investor*** – user that is permitted to buy digitized loan, homogenize it, dehomogenize other loan and redeem the underlying loan.

An example usage scenario:

1. *PandiFi* digitizes Borrower's loans (as an ERC-721 standard token) by minting new NFTs that represent off-chain loans.
2. Investor buys loan outside *PandiFi* and *PandiFi* transfers it (an ERC-721 token) to Investor.
3. Investor homogenizes desired digitalized loan to ERC-20 standard token and uses this token to buy other digitized loans.
4. Investor dehomogenizes owned homogenized loans and redeem the underlying loan.

## 2.2. Target in scope

The object being analysed were selected smart contracts accessible in the following:

- GitHub repository: [https://github.com/pandifi/pandifi-v1-core](https://github.com/pandifi/pandifi-v1-core)
- Commit ID: *e9d435514b9ad963f6ef9bbdcfd234ed945345b7*

The files reviewed were the following:

- Digitizer.sol
- HomogenizerFactoryClone.sol
- HomogenizerUpgradable.sol
- base/

- EligibilityBase.sol
- ERC20LookthroughUpgradable.sol

## 2.3. Threat analysis

This section summarized the potential threats that were identified during initial threat modeling. The audit was mainly focused on, but not limited to, finding security issues that might be exploited to achieve these threats.

The key threats were identified from particular perspectives as follows:

1. General (apply to all of the roles mentioned)
    - Bypassing the business logic of the system.
    - Theft of users’ funds.
    - Errors in arithmetic operation.
    - Possibility to lock the contract.
    - Possibility to lock users' funds in the contract.
    - Incorrect access control for roles used by the contracts.
    - Existing of known vulnerabilities (e.g. front-running, reentrancy, overflows).
    - Reentrancy during homogenization before homogenization ends (and opposite).
    - Create malicious homogenizer that is used automatically in the business process.
    - Incorrect calculation of current cash required parameters.
    - Sweeping deposit cash flow to malicious address.
    - Reinitialization of the contract.
    - Unprotected initialization.
    - Inconsistency with the whitepaper.
2. Admin
    - Too much power in relation to the declared one.
    - Unintentional loss of the ability to governance the system.
    - Admin mints unlimited tokens.
    - Admin locks whole funds.
    - Admin receives whole cash flow.
    - Admin sets incorrect stable coin address.
    - Admin private key leak.
3. Minter
    - Minter mints unlimited digitized loans (e.g. reentrancy).
4. Service
    - Service receives whole cash flow.
    - Service lock whole fund in the contract.
5. Trader
    - Denial of Service via finished reservation.

- Blocking users using front-funning.
- Denial of Service via invalid reservation address.

6. Investor
    - Investor bypasses USDC deposit.
    - Investor homogenizes locked loan.
    - Investor dehomogenizes locked loan.
    - Investor homogenize the same digitized loan twice in one batch.
    - Investor homogenize the sae digitized loan twice via reentrancy.
    - Investor homogenizes not his digitized loan.
    - Investor gets more ERC20 tokens than loan value.
    - Incorrect calculation of current cash required parameter.

## 2.4. Testing overview

The security review of the *PandiFi* smart contracts were meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Source code review,
- Automated tests through various tools (static analysis),
- Q&A sessions with the client's representatives which allowed to gain knowledge about the project and the technical details behind the platform.

## 2.5. Basic information

| | |
|---|---|
| **Testing team** | Damian Rusinek<br>Paweł Kuryłowicz<br>Jakub Zmysłowski |
| **Testing time period** | 2021-10-29 – 2021-11-05 |
| **Retesting time period** | 2021-12-01 – 2021-12-01 |
| **Report date** | 2021-12-08 |
| **Version** | 1.1 |

## 2.6. Disclaimer

**The security review described in this report is not a security warranty.**
It does not guarantee the security or correctness of reviewed smart contracts. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best security practices, security reviews and formal verification, ending with constant monitoring and incident response.

<u>Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.</u>

# 3. SUMMARY OF IDENTIFIED VULNERABILITIES

## 3.1. Risk classification

| Vulnerabilities | |
|---|---|
| **Risk impact** | **Description** |
| **Critical** | Vulnerabilities that affect the security of the entire system, all users or can lead to a significant financial loss (e.g. tokens).<br>It is recommended to take immediate mitigating actions or limit the possibility of vulnerability exploitation. |
| **High** | Vulnerabilities that can lead to escalation of privileges, a denial of service or significant business logic abuse.<br>It is recommended to take mitigating actions as soon as possible. |
| **Medium** | Vulnerabilities that affect the security of more than one user or the system, but might require specific privileges, or custom prerequisites.<br>The mitigating actions should be taken after eliminating the vulnerabilities with critical and high risk impact. |
| **Low** | Vulnerabilities that affect the security of individual users or require strong custom prerequisites.<br>The mitigating actions should be taken after eliminating the vulnerabilities with critical, high, and medium risk impact. |
| **Recommendations** | |
| Methods of increasing the security of the system by implementing good security practices or eliminating weaknesses. No direct risk impact has been identified.<br>The decision whether to take mitigating actions should be made by the client. | |

## 3.2. Identified vulnerabilities

| Vulnerability | Risk impact | Retest 2021-12-08 |
|---|---|---|
| 4.1 Multiple dehomogenization of loans | **High** | **Fixed** |
| 4.2 Centralized entities with excess permissions | **Medium** | Accepted the risk |
| 4.3 Lack of stable coin address validation | **Medium** | Accepted the risk |
| 4.4 Invalid calculation of *lockedTokenCount* parameter | **Low** | **Fixed** |
| **Recommendations** | | |
| 5.1 Use specific Solidity compiler version | | **Implemented** |
| 5.2 Functions do not emit proper events | | **Implemented** |
| 5.3 Remove unnecessary code | | **Implemented** |
| 5.4 Correct require messages to match the action | | **Implemented** |
| 5.5 Rename variable to better suit its purpose | | **Implemented** |
| 5.6 Gas optimization | | **Implemented** |
| 5.7 Remove direct calls to revert function | | **Implemented** |
| 5.8 Do not repeat the same code | | **Implemented** |
| 5.9 Use consistent function names | | **Implemented** |

# 4. VULNERABILITIES

## 4.1. Multiple dehomogenization of loans

| Risk impact | High | SCSVS V8 |
|---|---|---|
| Vulnerable contract | *Digitizer, HomogenizerUpgradable* | |
| Exploitation conditions | User's loan has current *UPB* equal to 0. | |
| Exploitation results | Theft of stable coins equal to the loan *cashHeld* parameter during homogenization. | |
| Remediation | Clearing the *cashHeld* parameter of dehomogenized loan. | |

**Status 2021-12-08:**
**Fixed**
Checked on commit ID: *92fdbeedf13fbd44763816dc72c26a4051e784d8t*

**Vulnerability description:**
Lack of clearing the *cashHeld* parameter allows malicious actor to steal the stable coins equal to the amount required to pay during homogenization. As the consequence, abuser is able to drain whole homogenizer.

**Test case:**

1. Abuser homogenizes (HomogenizerUpgradable.sol#L107) and dehomogenizes (HomogenizerUpgradable#L121) loan.
2. Abuser waits for the current *UPB* of the loan to be 0 – the value is assigned in *_receiveCashFlow* function (Digitizer.sol#L132).
3. Abuser sends the loan (ERC1155 token) to homogenizer.
4. Abuser dehomogenizes the same loan again.
5. Homogenizer sends stable coins to the abuser. The amount of that transfer is equal to the *cashHeld* parameter set during the homogenization process in step 1.

**References:**
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.1/0x17-V8-Business-Logic.html

## 4.2. Centralized entities with excess permissions

| Risk impact | Medium | SCSVS V1 |
|---|---|---|
| Vulnerable contract | *Digitizer, HomogenizerUpgradable* | |
| Exploitation conditions | Depending on the test case, compromise of the key belonging to the address with one of the following roles: *DEFAULT_ADMIN_ROLE, MINTER_ROLE or SERVICER_ROLE*. | |
| Exploitation results | Depending on the test case:<br>• Draining the pool,<br>• Introducing inconsistencies in protocol workflow,<br>• Minting the same loan multiple times or a fake one. | |
| Remediation | Short-term:<br>• Implementation of distributed multi-signature system. All entities control one of the signature key and submit their signed data to one entity called propagator. This entity combines all signatures and submits a multisigned transaction.<br>• Use of timelock mechanism may be considered so that users have time to become familiar with the decision and the change that will be made. However, it should be taken into account that in the event of a mistake or vulnerability, it may also be delayed to repair it.<br>Long-term:<br>• Using DAO as a source of decisions, introducing predetermined maximum and minimum values where possible and a clear message in the documentation regarding the privileges of each role. **Note:** The *PandiFi* team plans to implement governance contract. | |

**Status 2021-12-01:**
Accepted the risk
The dev team acknowledged the issue and accepted the risk. The team is going to use multisig to decrease the risk.

**Vulnerability description:**
Currently, the project is very much dependent on roles with excessive permissions. Contracts contain functions that strongly influence the operation of the project. It allows privileged accounts to manage the implementation and control system behaviour.

**Test case:**

*Excessive permission for DEFAULT_ADMIN_ROLE*

- The *mint* function (Digitizer.sol#L181) allows *DEFAULT_ADMIN_ROLE* to mint any loan with arbitrary *UPB* and homogenize it to the empty pool.
- The user with *DEFAULT_ADMIN_ROLE* has rights to change the service fee to very high and current *UPB* to 0 for a minted and homogenized loan and then dehomogenize it using *_dehomogenize* function (HomogenizerUpgradable.sol#L126) to steal stable coin from homogenizer contract.
- The *mint* function (Digitizer.sol#L181) allows users with *DEFAULT_ADMIN_ROLE* to mint a batch of new loan tokens. It may result in minting the same loan multiple times.
- Using *setStablecoinAddress* function (Digitizer.sol#L93), users with *DEFAULT_ADMIN_ROLE* can set a new stable coin address which can be an address of any (even malicious) contract.
- The *lockLoanTokens* function (Digitizer.sol#L193) allows user with *DEFAULT_ADMIN_ROLE* to lock arbitrary loans.

*Excessive permission for MINTER_ROLE*

- The *mint* function (Digitizer.sol#L181) allows *MINTER_ROLE* to mint any loan with arbitrary *UPB* and homogenize it to the empty pool.
- The *mint* function (Digitizer.sol#L181) allows users with *MINTER_ROLE* to mint a batch of new loan tokens. It may result in minting the same loan multiple times.

*Excessive permission for SERVICER_ROLE*

- The *lockLoanTokens* function (Digitizer.sol#L193) allows user with *SERVICER_ROLE* to lock arbitrary loans.

**References:**
SCSVS V1: Architecture, design and threat modeling
https://securing.github.io/SCSVS/1.2/0x10-V1-Architecture-Design-Threat-modelling.html
SCSVS V14: Decentralized Finance
https://securing.github.io/SCSVS/1.2/0x23-V14-Decentralized-Finance.html

## 4.3. Lack of stable coin address validation

| Risk impact | Medium | SCSVS V6 |
|---|---|---|
| Vulnerable contract | *Digitizer* | |
| Exploitation conditions | The *DEFAULT_ADMIN_ROLE* private key leakage. | |
| Exploitation results | Setting the non-stable coin address. | |
| Remediation | Hardcode the address of the stable coin that the project should use at the beginning or consider creating an allowlist to which other stable coins can be added in the future. | |

**Status 2021-12-01:**

Accepted the risk

The dev team acknowledged the issue and accepted the risk. The team is going to use multisig to decrease the risk.

**Vulnerability description:**

The current implementation allows to choose any address (including zero address), not only stable coins.

**Test case:**

The following functions allow to set a stable coin address without any validation:

- *Digitizer*
  - *constructor(string memory name, string memory symbol, address* stablecoinAddress_*)* – Digitizer.sol#55
  - *setStablecoinAddress(address* newStablecoinAddress*)* - Digitizer.sol#L93

**References:**

SCSVS V6; Malicious input handling

https://securing.github.io/SCSVS/1.2/0x15-V6-Malicious-Input-Handling.html

## 4.4. Invalid calculation of *lockedTokenCount* parameter

| Risk impact | Low | SCSVS V5 |
|---|---|---|
| Vulnerable contract | *Digitizer* | |
| Exploitation conditions | The *SERVICER_ROLE* or *DEFAULT_ADMIN_ROLE* private key leakage. | |
| Exploitation results | Invalid value of *lockedTokenCount* parameter which in future might influence the security of the system. | |
| Remediation | Check if a given *tokenId* is already locked and increase the value of *lockedTokenCount* only in the case of locking the so far unlocked *tokenID*. | |

**Status 2021-12-01:**
**Fixed**
The proposed remediation has been implemented.
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Vulnerability description:**
The *lockedTokenCount* parameter increments regardless of whether the *tokeId* was already locked or not.

**Test case:**
Calling *lockLoanTokens* function (Digitizer.sol#L195) with an array containing a list of 10 elements with the same *tokenId*, *lockedTokenCount* would increase by 10, when in fact it should increase by 1 or don't increase at all (if it was already locked).

```
function lockLoanTokens(uint256[] memory tokenIds) external override nonReentrant
{
        require(hasRole(SERVICER_ROLE, msg.sender) || hasRole(DEFAULT_ADMIN_ROLE,
msg.sender), 'Digitizer: Only the admin can lock loans');
        lockedTokenCount += tokenIds.length;
        for (uint256 i = 0; i < tokenIds.length; i++) {
            require(_exists(tokenIds[i]), 'Digitizer: Loan token does not
exist');
            isLocked[tokenIds[i]] = true;
        }
    }
```

**References:**
SCSVS V5: Arithmetic
https://securing.github.io/SCSVS/1.2/0x14-V5-Arithmetic.html

# 5. RECOMMENDATIONS

## 5.1. Use specific Solidity compiler version

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
Audited contracts use the following pragma:

```
pragma solidity ^0.8.6;
```

It allows to compile contracts with various versions of compiler (including those too recent) and introduces the risk of using a different version when deploying than during testing. .

**How to implement:**
Use a specific version of Solidity compiler (latest stable):

```
pragma solidity 0.8.6;
```

**References:**
Floating Pragma
https://swcregistry.io/docs/SWC-103
SCSVS V1: Architecture, design and threat modelling
https://securing.github.io/SCSVS/1.1/0x10-V1-Architecture-Design-Threat-modelling.html

## 5.2. Functions do not emit proper events

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
The following state-changing functions do not emit the appropriate events:

- *Digitizer*
  - *setTokenURI(uint256 tokenId, string calldata URI)* – Digitizer.sol#74
  - *_setReservation(uint256 tokenId, address owner)* – Digitizer.sol#80
  - setStablecoinAddress(address newStablecoinAddress) – Digitizer.sol#93
  - *lockLoanTokens(uint256[] memory tokenIds)* – Digitizer.sol#193

**How to implement:**
Consider adding events to functions that change state.

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

## 5.3. Remove unnecessary code

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
There is no need to use *if* statement when you return *true* or *false*.

- *HomogenizerUpgradable*
  - *isEligible(uint256 tokenId)* – HomogenizerUpgradable.sol#71
- *EligibilityBase*
  - *_isEligibleAndValid(uint256 tokenId, address digitizerAddress, address eligibilityContractAddress)* – EligibilityBase.sol#L12

**How to implement:**
Remove *if* statement:

- *HomogenizerUpgradable*
  - *isEligible*

```
return noteRateCriteria &&
            _isEligibleAndValid(tokenId, _underlyingContractAddress,
homogenizerParameters.eligibilityContractAddress);
```

- *EligibilityBase*
  - *_isEligibleAndValid*

```
return eligibilityCriteria && success;
```

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html
Solidity ^0.8.0 breaking changes
https://docs.soliditylang.org/en/v0.8.7/080-breaking-changes.html

## 5.4. Correct require messages to match the action

**Retest 2021-12-08:**
**Implemented**
Checked on commit ID: *4436299f97dfe6ce9740d8d78be9d9a2e1da91d5*

**Description:**
Some require messages are inconsistent with the function's operation and its description in the documentation.

- *Digitizer*
    - *setTokenURI(uint256 tokenId, string calldata URI)* – Digitizer.sol#L76
    - *_setReservation(uint256 tokenId, address owner)* – Digitizer.sol#L81
    - *_mintLoan(address to, LoanSchema.Original memory originalTokenData, LoanSchema.Current memory currentTokenData, string memory offChainDataURI)* – Digitizer.sol#L100
    - *lockLoanTokens(uint256[] memory tokenIds)* – Digitizer.sol#L194
- HomogenizerFactoryClone
    - *setEligibilityContractRestrictions(EligibilityContractRestrictions calldata eligibilityContractRestrictions_)* – HomogenizerFactoryClone#L78
    - *setHomogenizerRestrictions(HomogenizerRestrictions calldata homogenizerRestrictions_)* - HomogenizerFactoryClone#L87

**How to implement:**
Correct require messages to match the actual action as follow:

```
 function setTokenURI(uint256 tokenId, string calldata URI) external override {
     require(_exists(tokenId), 'Digitizer: URI query for nonexistent token');
     require(hasRole(SERVICER_ROLE, msg.sender) ||hasRole(DEFAULT_ADMIN_ROLE,
msg.sender), 'Digitizer: Only the admin and service can modify servicing data');
_tokenURI[tokenId] = URI;
}
```

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

## 5.5. Rename variable to better suit its purpose

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
In accordance with Solidity's best practices, names should best fit their purpose.

**How to implement:**
Rename *homogenizationFeeCurrent* to *cashRequiredOnDehomogenization.*
(HomogenizerUpgradable.sol#140).

**References:**
SCSVS V11: Code clarity

https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html
Solidity Style Guide page 220
https://docs.soliditylang.org/_/downloads/en/v0.8.9/pdf/

## 5.6. Gas optimization

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
Storing a state variable in memory instead of retrieving it multiple times from storage.

- *Digitizer*
  - *_receiveCashFlow(uint256 tokenId, LoanSchema.Current memory newTokenData, LoanSchema.CashFlow memory cashFlowData, address collectionWallet)* - Digitizer.sol#L145

```
upbOfOwner[ownerOf(tokenId)] -= (beginningUpb - newTokenData.UPB);
(…)
safeStablecoin.safeTransfer(ownerOf(tokenId), cashFlow);
```

**How to implement:**
Save a state variable in temporary memory variable.

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

## 5.7. Remove direct calls to revert function

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
It is a good practice to use *require* function instead of the *if* statement with *revert* function.

- *Digitizer*
  - *_receiveCashFlow(uint256 tokenId, LoanSchema.Current memory newTokenData, LoanSchema.CashFlow memory cashFlowData, address collectionWallet)* - Digitizer.sol#L169

```
if (isLocked[tokenId]) {
(…)
} else {
    revert('Digitizer: Loan must be locked prior to distributing cash flows');
}
```

**How to implement:**
Change *if* statement to *require* function call.

```
require(isLocked[tokenId], 'Digitizer: Loan must be locked prior to distributing
cash flows');
```

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

## 5.8. Do not repeat the same code

**Retest 2021-12-01:**
Implemented
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
It is a good practice to not repeat the same code but rather create a function that will handle the repeated logic or save the result in a temporary variable.
Repeated code could become inconsistent when its not updated in all places.

- *HomogenizerFactoryClone*
  - *createHomogenizer(string memory eligibilityContractSymbol, uint256 maxRate, uint256 minRate, uint256 buydownRatio, bool useArmMargin, bool useCurrentRate)* - HomogenizerFactoryClone.sol#L127

```
require(uniqueHomogenizerAddress[abi.encodePacked(eligibilityContractSymbol,
maxRate, minRate, buydownRatio, useArmMargin, useCurrentRate)] == address(0),
            'HomogenizerFactoryClone: An identical homogenizer already exists');
(…)
uniqueHomogenizerAddress[abi.encodePacked(eligibilityContractSymbol, maxRate,
minRate, buydownRatio, useArmMargin, useCurrentRate)] = joinedAddresses[0];
```

- *ERC20LookthroughUpgradable*
  - *_mint(address account, uint256 amount)* - ERC20LookthroughUpgradable.sol#L68

```
_balances[account] += (amount * _totalSupply) / uTotalSupply;
_totalSupply += (amount * _totalSupply) / uTotalSupply;
```

**How to implement:**
Save the unique key of homogenizer in a variable and use it in the mapping.

- *HomogenizerFactoryClone*
  - *createHomogenizer(string memory eligibilityContractSymbol, uint256 maxRate, uint256 minRate, uint256 buydownRatio, bool useArmMargin, bool useCurrentRate)* - HomogenizerFactoryClone.sol#L127

```
bytes memory homogenizerKey = abi.encodePacked(eligibilityContractSymbol,
maxRate, minRate, buydownRatio, useArmMargin, useCurrentRate);
(…)
require(uniqueHomogenizerAddress[homogenizerKey] == address(0),
            'HomogenizerFactoryClone: An identical homogenizer already exists');
(…)
uniqueHomogenizerAddress[homogenizerKey] = joinedAddresses[0];
```

- *ERC20LookthroughUpgradable*
    - *_mint(address account, uint256 amount)* - ERC20LookthroughUpgradable.sol#L68

```
uint256 balanceChange = (amount * _totalSupply) / uTotalSupply;
_balances[account] += balanceChange;
_totalSupply += balanceChange;
```

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

## 5.9. Use consistent function names

**Retest 2021-12-01:**
**Implemented**
Checked on commit ID: *76139cb6637ca5eb5ac44ce0f06a69f654afe0e2*

**Description:**
It is a good practice to consistently name functions. For example, if names of all private functions start with underscore, there should not be functions that start with underscore and are not private.
We recommend to change the name of *_dehomogenize* function (HomogenizerUpgradable.sol#L126) which is public.

**How to implement:**
Change name of the function to *dehomogenizeForUser*.

**References:**
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

## 6. CODE COVERAGE FOR CONTRACTS

The following table represent tests code coverage identified using *solidity-coverage* package. Overall contract's code coverage is considered as **excellent**.

| File | Statements | | Branches | | Functions | | Lines | |
|---|---|---|---|---|---|---|---|---|
| contracts/base/ | | | | | | | | |
| ERC20LookthroughUpgradable.sol | 100% | 44/44 | 100% | 22/22 | 100% | 8/8 | 100% | 44/44 |
| EligibilityBase.sol | 100% | 6/6 | 100% | 4/4 | 100% | 1/1 | 100% | 4/4 |
| contracts/ | | | | | | | | |
| Digitizer.sol | 100% | 84/84 | 100% | 44/44 | 100% | 15/15 | 100% | 83/83 |
| HomogenizerFactoryClone.sol | 100% | 106/106 | 100% | 60/60 | 100% | 14/14 | 100% | 106/106 |
| HomogenizerUpgradable.sol | 100% | 57/57 | 100% | 28/28 | 100% | 11/11 | 100% | 54/54 |

# 7. TERMINOLOGY

This section explains the terms that are related to the methodology used in this report.

Risk = Threat + Vulnerability

**Threat**

Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.[1]

**Vulnerability**

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.[1]

**Risk**

The level of impact on organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals resulting from the operation of an information system given the potential impact of a threat and the likelihood of that threat occurring.[1]

The risk impact can be estimated based on the complexity of exploitation conditions (representing the likelihood) and the severity of exploitation results.

| | | Complexity of exploitation conditions | | |
|---|---|---|---|---|
| | | Simple | Moderate | Complex |
| Severity of exploitation results | Major | Critical | High | Medium |
| | Moderate | High | Medium | Low |
| | Minor | Medium | Low | Low |

[1] NIST FIPS PUB 200: Minimum Security Requirements for Federal Information and Information Systems. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

## 8. CONTACT

Person responsible for providing explanations:

**Damian Rusinek**
e-mail: Damian.Rusinek@securing.pl
tel.: +48 12 425 25 75

http://www.securing.pl
e-mail: info@securing.pl
Kalwaryjska 65/6
30-504 Kraków
tel./fax.: +48 (12) 425 25