DeFiner Smart Contracts

Security Audit Report

August 25, 2020

Auditor: Alexander Remie

https://takasecurity.com

Table Of Contents

Executive Summary

DeFiner requested a security audit of the DeFiner smart contracts by Taka Security. The security audit focused on verifying that the smart contracts function as expected and also evaluates the overall design of the smart contracts.

Taka Security performed a security review of 7 business days during two weeks in July/August 2020. The security audit uncovered 2 critical, 2 high, 2 medium and 1 low severity security issues. Also, 2 high, 10 medium, and 10 low severity design issues were found. DeFiner has updated the code according to the reported findings and managed to fix all raised security findings. As well as most design findings. The only unresolved design findings are of low or medium severity.

During the audit, as well as during the implementation of fixes, DeFiner has been very receptive to suggestions, questions, and discussions. Also, an initial in-depth call to introduce Taka Security to the DeFiner smart contracts before the audit started, has been very helpful. The good communication has positively impacted the audit by being able to quickly get answers to any questions Taka Security had.

Despite fixing all important issues, and most of the less severe issues, Taka Security would still like to raise several concerns regarding the DeFiner smart contract.

- The amount of uncovered issues is unusually high in regards to the amount of code. Due to this Taka Security thinks given more time more issues would be uncovered. Therefore Taka Security strongly advises DeFiner to get another audit before releasing the project on mainnet.
- Although there are a lot of tests, they are really lacking in effectiveness. Which can be seen when looking at the raised issues which proper unit tests would have uncovered.

About

Taka Security is an Amsterdam-based company whose core business is offering Ethereum smart contract auditing services. Besides performing audits the company also works on the development of Ethereum smart contract analysis tools. The company was founded in 2020 by Alexander Remie.

Alexander is an independent Ethereum smart contract auditor who has previously worked for ChainSecurity and PwC Switzerland. After gaining experience doing dozens of audits for these companies he decided to set up his own Ethereum smart contract auditing company. Alexander has experience auditing various types of Ethereum smart contract projects: ranging from decentralized exchanges, tokens, DeFi projects, and projects based on ENS, amongst others. Before working as an Ethereum smart contract auditor Alexander worked in traditional payments and as an Ethereum smart contract developer.

Audit Overview

Timeline

Taka Security was contracted to perform a security audit of the DeFiner smart contracts for the duration of 7 days. The audit was started on July 28, 2020. The private report was delivered on August 9, 2020. On August 25, 2020 DeFiner provided fixes for the reported findings, and the final public report was delivered to DeFiner.

Scope

Git repo	`https://github.com/DeFinerOrg/Savings`
Git commit	`c1c5e39671244969487d1ea95e39be1be44339f1`
Compiler version	0.5.14
Git commit fixes	`c05bdc71f0e58f195c5a54537f9830ab14bc0fd8`

Review Methodology

During an audit Taka Security will execute Ethereum security analysis tools, as well as perform a thorough manual review. The manual review will focus on finding security related issues, as well as flagging bad practices or inefficient designs. If a specification is provided Taka Security will verify it reflects the implementation. Each issue found will be written into a separate finding in the report.

Findings Resolution

After the initial report has been sent to the client, it is up to the client to update the source code to resolve the reported findings. Once client has provided the updated source code to Taka Security, each resolved finding will be updated accordingly with a short description of the applied code changes.

Limitations

Security auditing cannot uncover all existing vulnerabilities: even a contract in which no vulnerabilities are found during the audit is not a guarantee of a secure smart contract. However, auditing enables the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

Terminology

Each finding is assigned one or more labels each describing a specific aspect of the finding.

Finding types

security	Could cause actual problems with real consequences.
design	Describes shortcomings of the current design.
note	Describes further areas of concern or possible small improvements to the code.

Finding severities

critical	Must be fixed.
high	Highly recommended to fix.
medium	Should be fixed.
low	Could be fixed.

The severity of security findings depends on two ratings.

Impact	How severe are the consequences of the issue being triggered.
Likelihood	how likely is is that the finding is triggered. Either accidentally or on purpose by a malicious actor.

The following table describes the security finding severity according to the Likelihood and Impact rating.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	critical	high	medium
Medium	high	medium	low
Low	medium	low	low

Finding resolvement

When client provides fixes and/or explanations for how they addressed each finding one of the following labels is assigned to each finding.

FIXED	The described finding has been fixed.
PARTIALLY FIXED	The finding has been partially fixed.
ACKNOWLEDGED	Client acknowledged the issue but has decided to not update the code due to certain reasons.

Findings that have not bee fixed and no reason has been provided are not assigned any of the above labels.

Project Overview

NOTE: this overview describes the project as it was at the start of the security review.

The DeFiner smart contract system implements a lending platform where users can deposit and borrow funds. Deposits and borrows both accumulate interest over time, to be earned by lenders and paid by borrowers, respectively. To borrow tokens an account needs to have deposited enough funds as collateral. If loans exceed a certain treshold, any account can trigger a liquidation of the loan. The current implementation supports ETH and the following ERC20 tokens: DAI, USDC, USDT, TUSD, MKR, BAT, ZRX, REP, WBTC. To fetch the prices of tokens DeFiner makes use of Chainlink oracles. These are defined per token inside the token registry.

Files > Libraries

1.1 BitmapLib

Included in `Base`. Implements functions to treat a `uint128` as a bitmap. This is used to store per user which tokens they have deposited/borrowed.

1.2 SafeDecimalMath

Included in `SavingAccount` and `Base`. Implements two functions that both return the same precision to use: `10^18`.

1.3 SymbolsLib

Separately deployed and linked into `SavingAccount` and `Base` by the compiler. Contains mappings from token symbol to address and vice versa, as well as a mapping from token index to symbol. Also defines the address of the `ChainLinkOracle` contract to use. During deployment of `SavingAccount` the `SymbolsLib.initialize` function will be called which will add all of the supported tokens, as well as the `ChainLinkOracle` address. There are no functions to update any of this information after initialization. This library contains a number of `view` functions to retrieve a token's symbol, address or price.

1.4 TokenInfoLib

Separately deployed and linked into `SavingAccount` and `Base` by the compiler. Contains per account borrow (interest) and deposit (interest) information. As well as functions to `withdraw`, `deposit`, `repay`, and `borrow`, which will update the appropriate borrow (interest) and deposit (interest) information.

1.5 strings

Included in `SymbolsLib`. Copied from `Arachnid/solidity-stringutils` and updated to work with solidity `^0.5.0`. Is only used to split a comma separated string of token symbols. Note, this file is out-of-scope for this audit.

1.6 Base

Separately deployed and linked into `SavingAccount` by the compiler. This library contains functions to calculate borrow/deposit rates/balances, Capitalization Ratio, Capital Compound Ratio. There are several mappings which track the amount of tokens in reserve, lent, and deposited into compound. These mappings can be updated using several update functions, which are called internally by the `SavingAccount` contract. If the reserve ratio exceeds the allowed min/max values, the library will withdraw/deposit tokens from/to compound, respectively. To store information a struct called `BaseVariable` is defined, which is stored in `SavingAccount.baseVariable`. The library also defines a function to create a new "rate index checkpoint", which is used to calculate the interest accumulation. The function to make a new checkpoint is called by the following functions inside `SavingAccount`: `borrow`, `withdraw`, `withdrawAll`, `repay`, `deposit`, `liquidate`.

Files > Contracts

2.1 InitializableReentrancyGuard

Inherited by `SavingAccount`.
OpenZeppelin Reentrancy guard which also works with upgradable contracts.

2.2 GlobalConfig

Separately deployed and address is stored inside `SavingAccount.globalConfig` and `SavingAccount.baseVariable.globalConfigAddress`. Contains the `communityFundRatio`, `minReserveRatio`, `maxReserveRatio`, `liquidationThreshold`, and `liquidationDiscountRatio`. These are all initialized upon declaration but can be updated by the `GlobalConfig` owner.

2.3 ChainLinkOracle

This contract is separately deployed and acts as an abstraction over the possibly different chainlink oracle contracts defined per token in `TokenInfoRegistry`. During deployment of `SavingAccount` the passed in address of the `ChainLinkOracle` will be saved at `SavingAccount.symbols.chainlinkAggregator`. There is no function to allow updating this value after deployment.

Whenever the price is needed of a specific token, the `ChainLinkOracle.getLatestAnswer` will be called. This function will fetch the token's registered chainlink oracle address from the `TokenRegistry`, after which it will call the oracle to get the latest price.

2.4 TokenInfoRegistry

This contract is separately deployed, and its address is stored inside both the `SavingAccount` and `ChainLinkOracle` contracts when they are deployed. The `TokenInfoRegistry` contains a struct for each registered token. The owner of the contract can add new tokens and update existing tokens, removing tokens however is not supported. The `isTokenExists` function is used in the borrow, withdraw, repay, deposit, and liquidate functions inside `SavingAccount` to make sure the token argument of the function call exists inside the token registry.

2.5 SavingAccount

This contract is separately deployed and is the main contract of the DeFiner smart contracts. Users wanting to interact with the DeFiner smart contracts will call functions inside this contract. The following separately deployed libraries are linked in by the compiler: `Base`, `SymbolsLib`, `TokenInfoLib`. This contract is upgradable using the OpenZeppelin Upgradability pattern.

The `SavingAccount` contract contains functions to allow users to deposit/withdraw/borrow tokens, repay borrowed tokens, and liquidate loans. Furthermore, the `definerCommunityFund` account is allowed to withdraw collected commission fees, as well as assign a different `definerCommunityFund` account.

Roles

This section describes, per contract, the privileged roles which are allowed to call certain functions not accessible to regular users.

TokenInfoRegistry

<ins>owner</ins> The owner can call the following functions:

- `updateBorrowLTV`, to update the borrow LTV of a registered token.
- `updateTokenTransferFeeFlag`, to update if a token incurs a transfer fee yes/no.
- `updateTokenSupportedOnCompoundFlag`, to update if a token is supported on compound yes/no.
- `updateCToken`, to update the cToken address of the token.
- `updateChainLinkAggregator`, to update the chainlink oracle address to use for the token.
- `enableToken`/`disableToken`, to set the `enable` field of the token to `true` or `false` respectively.

GlobalConfig

<ins>owner</ins> The owner can call the following functions:

- `updateCommunityFundRatio`, to update the `communityFundRatio`.
- `updateMinReserveRatio`, to update the `minReserveRatio`.
- `updateMaxReserveRatio`, to update the `maxReserveRatio`.
- `updateLiquidationThreshold`, to update the `liquidationThreshold`.
- `updateLiquidationDiscountRatio`, to update the `liquidationDiscountRatio`.

SavingAccount

<ins>deFinerCommunityFund</ins> This role is stored inside `SavingAccount.baseVariable.definerCommunityFund` and can call the following functions:

- `recycleCommunityFund`, to withdraw collected commission fees.
- `setDeFinerCommunityFund`, to assign the role to another account.

Trust Model

Compound The compound smart contracts are *trusted* to function correct.

Chainlink The chainlink oracle(s) are *trusted* to function correct.

Deployer The deployer is *trusted* to use the correct code during deployment and set the right parameters. Furthermore, as the `SavingAccount` contract is upgradable, the deployer is *trusted* to upgrade the contract without users losing (access to) their funds.

Owner The owner of each contract is *trusted* to call the right functions with sensible parameters.

Borrower A borrower is *untrusted* and assumed to be potentially malicious.

Depositor A depositor is *untrusted* and assumed to be potentially malicious.

Upgradability

The DeFiner project makes use of OpenZeppelin Upgradability to allow the `SavingAccount` contract to be upgradable. This contract has two separately deployed libraries linked into it, which makes those two libraries also upgradable (by simply deploying a new library and linking in the new library's address in the upgraded version of the `SavingAccount` contract). However, the current set up causes some problems, see **DEF-009**.

Test suite

The test suite of DeFiner contains a significant amount of tests, a little over 200, which all pass. However, due to the high number of issues found, as well as the types of issues, it becomes apparent that the current unit tests are significantly lacking. Also, Taka Security was unable to get the coverage report working, which would have shown that certain parts are not tested sufficiently.

Findings

This section lists the issues found during the audit of the DeFiner smart contracts.

DEF-001 Incorrect LTV check in `borrow` security critical FIXED

Likelihood: high
Impact: high

The `borrow` and `withdraw` functions inside `SavingAccount` contain a check to make sure the LTV does not exceed the max allowed (60%). This check is implemented correct inside `withdraw`. However, the same check in `borrow` incorrectly handles the precision. This leads to the already borrowed amount not fully being taken into account when calculating the LTV, and thereby allowing borrows far surpassing the LTV, to succeed.
The current borrow LTV check:

```
require(baseVariable.getBorrowETH(msg.sender,
    symbols).add(_amount.mul(symbols.priceFromAddress(_token))).mul(100)
  <= baseVariable.getDepositETH(msg.sender, symbols).mul(divisor).mul(borrowLTV),
      "Insufficient collateral.");
```

SavingAccount.sol

This is best explained through an example. Eve has made a total deposit worth 10 ETH, and has already borrowed worth 6 ETH. Her LTV is therefore currently the maximum allowed (60%), and trying to borrow more should fail. However, this is not what will happen. Instead, if Eve tries to now borrow 1000 DAI the following will happen, and the LTV check and borrow will succeed:

```
 - chainlink DAI-ETH exchange rate = 0.00253794
   =                          2537940000000000
 $1 baseVariable.getBorrowETH(msg.sender, symbols) = 6 ETH
   =                       6000000000000000000
 $2 _amount.mul(symbols.priceFromAddress(_token)) = 1000 DAI * 2537940000000000
   =   2563321000000000000000000000000000000
 $3 (existing borrow + new borrow) * 100 = ($1 + $2) * 100
   = 256332100000000000060000000000000000000
 $4 baseVariable.getDepositETH(msg.sender, symbols) = 10 ETH
   =                      10000000000000000000
 $5 deposited eth * divisor * borrowLTV = 10*10**18 * 10**18 * 60
   = 600000000000000000000000000000000000000
 $6 require($3 <= $5), which will succeed
```

A few things to take away from this are:

- The bug is caused by not correctly handling the precision. To be precise, the already borrowed ETH in $1 is not multiplied by `10^18`. Although this issue can be resolved in multiple ways, see Recommendations below.
- The already borrowed amount ends up somehwere in the middle of $3. This means that this bug could be exploited consecutively, since each time the already borrowed amount will be in the middle and not noticeably influence the outcome. This could easily be used to drain the contract's funds.
- This bug requires that the attacker has deposited at least some tokens. Also, the more he has deposited, the more he can borrow with each call.

A few things could have caused this bug to be detected:

- Proper unit tests would have caught this bug.
- Spreading out the code on multiple lines would significantly increase the readability of the check and likely made the incorrect precision handling to be noticed by a developer.

Recommendations
Correctly handle the precision, i.e. replace the existing borrow LTV check with:

```
require(
  baseVariable.getBorrowETH(msg.sender, symbols).add(
    _amount.mul(symbols.priceFromAddress(_token)).div(divisor)
  )).mul(100)
  <=
  baseVariable.getDepositETH(msg.sender, symbols).mul(borrowLTV),
  "Insufficient collateral.");
```

Also, add proper unit tests to make sure the borrow and withdraw LTV checks work as expected, and spread the code out on multiple lines so that is readable.

Mitigation

The incorrect check has been replaced with a correct check.

DEF-002 Flawed `recycleCommunityFund` security critical FIXED

Likelihood: high
Impact: high

The `recycleCommunityFund` function contains multiple bugs. Calling this function will therefore not perform the epxected actions and there is a very high chance to mess up the internal ETH balance of the contract. This could lead to other functions failing as there will be less ETH in the contract than what the accounting variables signify.

The `recycleCommunityFund` function can only be called by the DeFiner-controlled `deFinerCommunityFund` account. This function allows DeFiner to withdraw income earned from the commission fees, which is one of the main sources of income for DeFiner.

Whenever a user calls `withdraw` or `withdrawAll` to withdraw a particular token (ETH or ERC20) from their deposits, a 10% commission fee will be deducted from the amount of deposit interest withdrawn (See **DEF-004**, as this is not actually the case). This commission fee will not first be converted to ETH, but will be collected in that particular token, and is added to the `definerFund[token]` mapping.

The `recycleCommunityFund` function is defined as:

```
576    function recycleCommunityFund(address _token) public {
577            require(msg.sender == baseVariable.deFinerCommunityFund, "Unauthorized call");
578            baseVariable.deFinerCommunityFund.transfer(uint256(baseVariable.definerFund[_token]));
579            baseVariable.definerFund[_token] == 0;
580    }
```

SavingAccount.sol

There are three bugs inside this code:

- Line 578 will try to transfer ETH to the `definerCommuntiyFund`, not ERC20 tokens. However, the `definerFund[_token]` mapping contains the amount of that token, not ETH, that has been collected in the community fund. For example, if trying to "recycle" 20 DAI which has been collected inside `definerFund[DAI]`, calling this function will instead try to send 20 ETH to the `definerCommunityFund` account.
- Line 579 compares zero against the amount of the token in the `definerFund`. It will not reset the `definerFund[_token]` to zero. This allows this function to be called multiple times to withdraw the entire commission fee. Instead of allowing a commission fee to be withdrawn once and be reset back to zero.
- By (trying to) reset the fund balance after the transfer call, this contract is susceptible to reentrancy. Although it is unlikely that the DeFiner-controlled `definerCommunityFund` account will set up a contract to exploit this, it is still recommended to adhere to the Checks-Effects-Interactions pattern to prevent the possibility of a reentrancy. Adding the `nonReentrant` modifier would also prevent reentrancy.

The first two bugs effectively result in not being able to call the `recycleCommunityFund` function without introducing problems in other parts of the contract.

Recommendations

- The code should be updated to transfer the correct token or ETH depending on `_token`.
- Replace == with = on line 579.

- The fund reset should be performed before the transfer call.

Mitigation
The code has been updated to send the correct token or ETH, correctly reset the withdrawn amount, and the reentrancy issue has been mitigated.

DEF-003 No way to pause contract security high FIXED

Likelihood: medium
Impact: high

The `SavingAccount` contract is upgradable, and with it the `SymbolsLib` and `Base` libraries. Since there is no way to pause the contract there is no way to prevent the bug from being exploited until a fix is deployed. Since a fix might be not trivial and/or take time to implement, having a way to pause the contract would be desired. This would give the DeFiner team enough time to work on a proper fix without risking more exploitation of the bug.

A pause could be implemented as a contract wide pause which prevents any function from being called. On the other hand, a pause could be implemented in such a way to block all functions except the ones that allow users to still withdraw their funds. However, such a pause is difficult to implement since it could be that the bug resides in the withdraw functions.

Even though there is an `enabled` field inside the `TokenInfoRegistry.TokenInfo` struct, it is currently not used (**DEF-014**). If it were used this would atleast enable a pay to temporarily "disable" all tokens. Still, since ETH is not part of the token registry, ETH could still be used.

Recommendations
DeFiner should carefully consider if a pause functionality would be beneficial to the DeFiner smart contract system, and what type of pause would be best suited, and implement it if found to be desired.

Mitigation
The `SavingAccount` contract is now pausable by making use of OpenZeppelin `Pausable`.

DEF-004 Commission fee can be skipped security high FIXED

Likelihood: high
Impact: medium

The calculation of the commission fee of withdrawn amounts is incorrect and depending on the withdrawn amount and existing deposit interest could result in not taking a 10% commission fee, or just flat out skipping the entire commission fee.

The `SavingAccount.withdraw` function contains the following code to calculate the 10% commission fee over the withdrawn deposit interest amount.

```
// Update tokenInfo for the user
TokenInfoLib.TokenInfo storage tokenInfo = baseVariable.accounts[_from].tokenInfos[_token];
uint accruedRate = baseVariable.getDepositAccruedRate(_token,
    tokenInfo.getLastDepositBlock());
tokenInfo.withdraw(_amount, accruedRate, this.getBlockNumber());

// Unset deposit bitmap if the deposit is fully withdrawn
if(tokenInfo.getDepositPrincipal() == 0)
    baseVariable.unsetFromDepositBitmap(msg.sender, tokenRegistry.getTokenIndex(_token));

// DeFiner takes 10% commission on the interest a user earn
// sichaoy: 10 percent is a constant?
uint256 commission = tokenInfo.depositInterest <= _amount ?
    tokenInfo.depositInterest.div(10) : _amount.div(10);
baseVariable.deFinerFund[_token] = baseVariable.deFinerFund[_token].add(commission);
uint256 amount = _amount.sub(commission);
```

SavingAccount.sol

On line 366 the `tokenInfo.withdraw` function is called which will accordingly decrease `tokenInfo.depositInterest`. However, only on line 374 the commission fee is calculated, when the `depositInterest` has already been decreased. This leads to unexpected and incorrect results.

A depositor could call the withdraw function to withdraw 10 tokens while having a `depositInterest` of 10 tokens. Line 366 would decrease the `depositInterest` from 10 to 0. Then on line 374 the comparison will be `0 <= 10`, which will result in calculating 10% of the updated `depositInterest` of 0, resulting in 0 commission fee.

If a depositor currently has a `depositInterest` of 10 tokens and tries to withdraw 30 tokens. Line 366 would decrease the `depositInterest` from 10 to 0. Then on line 374 the comparison will again be `0 <= 10`, and the resulting commission fee will be 0.

If a depositor currently has a `depositInterest` of 10 tokens and tries to withdraw 7 tokens. Line 366 would decrease the `depositInterest` from 10 to 3. Then on line 374 the comparison will be `3 <= 7`, and the resulting commission fee will be 10% of 3, instead of 10% of 7.

This effectively means that whenever the original `depositInterest` will be entirely used when withdrawing an amount of tokens, the caller can skip the paying of a commission fee.

Recommendations
The commission fee should be calculated over the amount with which the `depositInterest` is lowered during a `withdraw` call.

Mitigation
The commission fee during withdrawals has been removed, for now.

DEF-005 Cannot liquidate ETH loans security medium FIXED

Likelihood: high
Impact: low

The `borrow`, `withdraw`, `deposit`, `repay`, and `withdrawAll` functions inside `SavingAccount` all have the `onlySupported` modifier applied. This allows any registered ERC20 token as well as ETH to be accepted by these functions.

```
    modifier onlySupported(address _token) {
        if(!_isETH(_token)) {
            require(tokenRegistry.isTokenExist(_token), "Unsupported token");
        }
        _;
    }
```

SavingAccount.sol

On the other hand, the `liquidate` function does not have this modifier applied but instead performs the following check on the first line:

```
    require(tokenRegistry.isTokenExist(_targetToken), "Unsupported token");
```

SavingAccount.sol

This check will prevent ETH loans from being liquidated. However, because the contract is upgradable a fix could be deployed by DeFiner. Therbey limiting the impact of this issue.

Recommendations
Remove the check inside `liquidate` and instead apply the `onlySupported` `modifier`.

Mitigation
The modifier is now also applied to the `liquidate` function.

DEF-006 Missing checks in `GlobalConfig` security medium FIXED

Likelihood: low
Impact: high

The GlobalConfig contract update functions are missing checks to prevent the new values from exceeding the maximum allowed value (100). These functions can only be called by the owner. Still, if such incorrect values where accidentally supplied to these functions this could have immediate and severe consequences for multiple other important calculations throughout the contracts.

For example, the maxReserveRatio could be set to 1000.

```
    function updateMaxReserveRatio(uint256 _maxReserveRatio) external onlyOwner {
        require(_maxReserveRatio != 0, "Max Reserve Ratio is zero");
        require(_maxReserveRatio > minReserveRatio, "Max reserve less than or equal to
            Min reserve");
        maxReserveRatio = _maxReserveRatio;
    }
```

GlobalConfig.sol

Futhermore, since the minReserveRatio can only be above zero, the check that the maxReserveRatio != 0 is redundant. Also, there are no checks that prevent a value to be updated to the same value.

Recommendations

- Add checks to make sure the new value is below the max allowed value.
- Add checks to make sure the new value differs from the current value.
- Remove the maxReserveRatio != 0 check.

Mitigation

The recommended checks have been added.

DEF-007 Deposited funds locked for newly added tokens security low FIXED

Likelihood: medium
Impact: low

The SymbolsLib library stores a static list of supported tokens inside SavingAccount during deployment. On the other hand, the TokenInfoRegistry contract contains a dynamic list of supported tokens. Due to this, adding a token to the dynamic list after deployment will not update the static list. This will lead to user's deposited newly added tokens to be locked, as they cannot be withdrawn. However, since the SavingAccount contract is upgradable a fix could still be deployed.

The TokenInfoRegistry contract contains a function that checks whether a token exists inside the registry.

```
    function isTokenExist(address _token) public view returns (bool isExist) {
        isExist = tokenInfo[_token].chainLinkAggregator != address(0);
    }
```

TokenInfoRegistry.sol

This function is used in a modifier applied to most functions inside SavingAccount. In particular, the deposit, withdraw and withdrawAll functions.

If a token is added to the registry the isTokenExist check will succeed. Calling the deposit function will succeed. However, the withdraw functions also call Base.getBorrowETH and Base.getDepositETH, and since these functions make use of the static list they will revert as the new token was not added to the static list. This effectively allows newly added tokens to the registry to be deposited, but prevents them from being withdraw.

The repay and borrow functions also call functions which use the static list and will therefore also revert. So the only thing that can be done with a newly added token is depositing it. All the other functions will fail.

As it seems reasonably likely that tokens will be added, the likelihood of this issue is medium. The impact on the other hand is set to low as there will be no loss of user funds and since the contract is upgradable a fix can always be deployed by DeFiner.

Recommendations

To mitigate this issue it should either be allowed to update both lists, or not update any of the two lists. Also, if allowed to update, the changing of this list in the two separate places should be dealt with atomically, i.e. one transaction will update both lists. However, as explained in issue **DEF-013** it makes much more sense to merge the contracts/libraries that keep track of supported tokens.

Mitigation
The token related contracts/libraries have been merged into a new contract called `TokenRegistry`.

DEF-008 `SavingAccount` contract too big design high FIXED

The `SavingAccount` contract bytecode size is currently 28802 bytes, whereas the maximum allowed size is 24576 bytes. Therefore, the current `SavingsAccount` contract will not deploy on an Ethereum chain. DeFiner should reduce the contract size so that it is falls below the maximum allowed size. There are various ways to reduce the contract size.

Recommendations
Short term, apply all tips to make contracts smaller [1] and remove unused imports (**DEF-028**).

Long term, extract the `SymbolsLib` and `Base` libraries into their own separate contracts. See **DEF-013** and **DEF-010**, respectively.

Update
Through refactoring, the `SavingAccount` contract now takes up less than the max allowed bytecode size.

DEF-009 Problems with upgradability of library structs design high FIXED

The DeFiner smart contracts make use of OpenZeppelin Upgradability to allow them be upgraded in case of new features and/or bug fixes. Specifically, the `SavingAccount` contract is upgradable. The `SymbolsLib` and `Base` libraries are separately deployed and linked into the `SavingAccount` contract by the compiler. Both these libraries define a `struct` which contains variables used by that particular library and which is stored inside a variable in `SavingAccount`. Due to the use of such structs in these libraries it is not possible to add fields to these structs during an upgrade.

The structs are:

```
    struct Symbols {
        uint count;
        mapping(uint => string) indexToSymbol;
        mapping(string => uint256) symbolToPrices;
        mapping(address => string) addressToSymbol;
        mapping(string => address) symbolToAddress;
        ChainLinkOracle chainlinkAggregator;
    }
```

SymbolsLib.sol

```
    struct BaseVariable {
        // The amount for the whole saving pool
        mapping(address => uint256) totalLoans;     // amount of lended tokens
        mapping(address => uint256) totalReserve;   // amount of tokens in reservation
        mapping(address => uint256) totalCompound;  // amount of tokens in compound
        mapping(address => address) cTokenAddress;  // cToken addresses
        // Token => block-num => rate
        mapping(address => mapping(uint => uint)) depositeRateIndex; // the index curve of
            deposit rate
        // Token => block-num => rate
        mapping(address => mapping(uint => uint)) borrowRateIndex;   // the index curve of
            borrow rate
        // token address => block number
        mapping(address => uint) lastCheckpoint;            // last checkpoint on the index
            curve
        // cToken address => rate
        mapping(address => uint) lastCTokenExchangeRate;    // last compound cToken exchange
            rate
        // Store per account info
        mapping(address => Account) accounts;
        address payable deFinerCommunityFund;   // address allowed to withdraw the community
            fund
        address globalConfigAddress;            // global configuration contract address
        address savingAccountAddress;           // the SavingAccount contract address
```

[1] https://blog.aragon.one/rage-against-the-ev-machine-part-1/

```
        mapping(address => uint) deFinerFund;    // Definer community fund for the tokens
        // Third Party Pools
        mapping(address => ThirdPartyPool) compoundPool;     // the compound pool
    }
```

Base.sol

These two libraries are initialized during deployment of `SavingAccount` and stored inside a contract storage variable inside `SavingAccount`. The `SavingAccount` storage variables are:

```
    SymbolsLib.Symbols symbols;
    Base.BaseVariable baseVariable;

    TokenInfoRegistry public tokenRegistry;
    GlobalConfig public globalConfig;

    // Following are the constants, initialized via upgradable proxy contract
    // This is emergency address to allow withdrawal of funds from the contract
    address payable public EMERGENCY_ADDR;
    address public ETH_ADDR;
    uint256 public ACCURACY;
    uint256 public BLOCKS_PER_YEAR;
    uint256 public UINT_UNIT;
```

SavingAccount.sol

When using upgradable contracts the storage layout of contract variables is very important. You can only append new variables after the last one during an upgrade. The two library's variables inside `SavingAccount` are defined as the first two variables. Both of these represent the struct defined in the library. Structs are laid out in storage by laying out all their members (packing small fields when possible).

The `Symbols` struct contains 6 fields and therefore takes up 6 storage slots. The `BaseVariable` struct contains 14 fields and therefore takes up 14 storage slots. Therefore the `SavingAccount` state variables are laid out as:

```
    // slot 0:  Symbols.count
    // slot 1:  Symbols.indexToSymbol
    // slot 2:  Symbols.symbolToPrices
    // slot 3:  Symbols.addressToSymbol
    // slot 4:  Symbols.symbolToAddress
    // slot 5:  Symbols.chainlinkAggregator
    // slot 6:  BaseVariable.totalLoans
    // slot 7:  BaseVariable.totalReserve
    // slot 8:  BaseVariable.totalCompound
    // slot 9:  BaseVariable.cTokenAddress
    // slot 10: BaseVariable.depositeRateIndex
    // slot 11: BaseVariable.borrowRateIndex
    // slot 12: BaseVariable.lastCheckpoint
    // slot 13: BaseVariable.lastCTokenExchangeRate
    // slot 14: BaseVariable.accounts
    // slot 15: BaseVariable.deFinerCommunityFund
    // slot 16: BaseVariable.globalConfigAddress
    // slot 17: BaseVariable.savingAccountAddress
    // slot 18: BaseVariable.deFinerFund
    // slot 19: BaseVariable.compoundPool
    // slot 19: tokenRegistry
    // slot 20: globalConfig
    // slot 21: EMERGENCY_ADDR
    // slot 22: ETH_ADDR
    // slot 23: ACCURACY
    // slot 24: BLOCKS_PER_YEAR
    // slot 25: UINT_UNIT
```

To update the `SymbolsLib` or `Base` libraries a new version needs to be deployed and the address should be linked into the new version of `SavingAccount`. However, because of the above described storage layout it's not possible to add any new fields to the `Symbols` or `BaseVariable` structs as there are already other variables right after the structs.

If for example the `BaseVariable` was instead declared as the last storage variable in `SavingAccount`, it would be possible to add new fields to the `BaseVariable`. However, once a new field would be added to the `SavingAccount`, it would not be possible anymore to add new fields to the `BaseVariable` struct.

Fixing this problem is not easy. You could define a mapping in which you store the structs [2]. Which due to the storage location of mapping entries would make it possible to add variables to both of the structs as well as the SavingAccount contract. However, this would be an ugly hack and is discouraged. Another possible solution would be to add a uint256[100] variable right after both struct variables.

```
    SymbolsLib.Symbols symbols;
    uint256[100] symbolsExtensionSpace;
    Base.BaseVariable baseVariable;
    uint256[100] baseExtensionSpace;

    TokenInfoRegistry public tokenRegistry;
    GlobalConfig public globalConfig;

    // Following are the constants, initialized via upgradable proxy contract
    // This is emergency address to allow withdrawal of funds from the contract
    address payable public EMERGENCY_ADDR;
    address public ETH_ADDR;
    uint256 public ACCURACY;
    uint256 public BLOCKS_PER_YEAR;
    uint256 public UINT_UNIT;
```

SavingAccount.sol

These two uint arrays would initially take up 100 slots each. Whenever a new field needs to be added to one of the structs the following extension space array size needs to be decreased by the amount of new storage slots occupied by the added struct fields. However, this is error prone as it should also be taken into account that depending on the added struct field types multiple fields could be packed into the same storage slot.

Recommendations
First off, add tests to test upgrading all the different parts of SavingAccount and its linked libraries!

Short term, add the extension space uint arrays after both of the struct variables.

Long term, get rid of both libraries by turning them into a separately deployed and upgradable contract. There are other reasons to get rid of these libraries. See **DEF-013** for SymbolsLib, and **DEF-010** for Base.

Update
Both of the mentioned libraries have been converted (and split) into separate contracts. This means the SavingAccount contract does no longer contain variables referring to structs defined in a separate library. As such, the above described upgradability problems have been resolved.

DEF-010 Split Base into Bank and Accounts **design** **medium** **PARTIALLY FIXED**

The Base library is separaterly deployed and linked into the SavingAccount contract. It contains data structures and functions that could be divided functionally in two distinct parts.

- Keeping track and updating of tokens in the reserve, deposited into compound, and lent out to borrowers.
- Per account infomation regarding borrowed and deposited tokens (and interest).

As explained in **DEF-009** the current set up of having a struct in Base leads to problems when trying to add variables to BaseVariables during an upgrade of SavingAccount. Besides that, there being two distinct parts of functionality inside Base suggests that splitting Base into two separate contracts would benefit both the code readability, complexity, as well as upgradability.

Recommendations
Split the Base library into two upgradable contracts: Accounts and Bank. The Accounts contract would contain all account related functions and data. The Bank contract would contain all information regarding tokens in reserve, compound, lent to borrowers, and the commision fund.

Update
The Base library has been split into two contracts: Accounts and Bank. Although they both contain an initialize function, they do not inherit from Initializable. Thereby making it possible that initialize is called more than once.

[2] https://forum.openzeppelin.com/t/how-to-use-a-struct-in-an-upgradable-contract/832

DEF-011 Missing difference checks in `TokenInfoRegistry` design medium PARTIALLY FIXED

The `TokenInfoRegistry` contract update functions do not check that the new value differs from the current value. If the value does not differ, the function will still succeed and a `TokenUpdated` event will be emitted. This would be confusing as there was actually no change.

Recommendations
Add a check to make sure the new value differs from the current value.

Update
The update functions now check that the new value differs from the existing value. However, instead of throwing they simply `return` if the check fails. Although in such cases no event is emitted, letting the function succeed without any changes is a bit misleading.

DEF-012 Token `decimals`, fetch once and store design medium FIXED

Whenever the decimals of an ERC20 token is necessary to calculate the divisor used in a calculation, the below call is made. This happens in multiple places in both `SavingAccount` and `Base`.

```
IERC20Extended(< token address >).decimals()
```

The decimals of an ERC20 token should not ever change. It is therefore unnecessary to constantly refetch the decimals of a token. Instead, fetching the decimals inside `TokenInfoRegistry.addToken` and storing it in a field in the `TokenInfoRegistry.TokenInfo` struct should be sufficient.

As it happens, the `addToken` function already contains a `decimals` argument. However, it needs to be manually supplied instead of using the token address to fetch the decimals value directly from the token contract.

Recommendations
Remove the `decimals` function argument from `addToken` and instead use the supplied token address to fetch the `decimals` from the token contract by calling `decimals()`, Furthermore, replace all other calls of `decimals()` throughout the contracts with a call to fetch the `decimals` variable from the `TokenInfoRegistry.TokenInfo` struct.

Update
The code no longer refetches the token decimals.

DEF-013 Duplicate token data can get out-of-sync design medium FIXED

There are several contracts that contain information regarding tokens. Some pieces of information are stored in multiple contracts, and some of these can get out-of-sync due to only (being able to) updating it in one place.

Currently the token information is spread out across:

- `SavingAccountParamaters` contains list of symbols of the enabled tokens.
- `TokenInfoRegistry` contains (updateable) list of token structs with information.
- `SymbolsLib` contains (non-updateable) list of token symbols, addresses and indexes.
- `Base` contains mapping from token address to cToken address (also exists inside token registry).

As all this information is about tokens it would be recommended to put it all in one contract. Besides that, some information is stored in multiple places and could get out of sync. For example, both the `Base` and `TokenInfoRegistry` define the cToken address per token. The `Base` list cannot be updated after deployment, whereas the `TokenInfoRegistry` can be updated. See **DEF-007** for another example of out-of-sync token data.

During deployment of `SavingAccount` the token data is loaded into `Base` and `SymbolsLib`. However, the token data inside `TokenInfoRegistry` is not added automatically and should be added manually by calling `addToken`. This means extreme care should be taken to add the same tokens, and in the same order, when calling `addToken`.

By merging all token related functionality into one contract the readability of the code would greatly increase. Also, all information would only be present in a single place, instead of being duplicated - and possibly out of sync.

The addToken function accepts a decimals and symbol parameter. However, instead of manually supplying this the addToken function could use the token address to fetch both the decimals and symbol from the token contract. Also see **DEF-012** regarding the decimals.

Recommendations

- Merge the contents of the SymbolsLib library into the TokenInfoRegistry contract.
- Move the token to cToken mapping in Base to the TokenInfoRegistry contract.
- Remove the SavingAccountParameters contract.
- Inside addToken fetch the token symbols and decimals values from the token contract.
  - This would also remove the need for splitting the comma separated token symbol list, and thereby strings.sol would no longer be needed.

Update
All token related data is now store in a new contract called TokenRegistry.

DEF-014 token.enabled not used design medium FIXED

The TokenInfo struct inside the TokenInfoRegistry contains a bool enabled field. This field can be updated by the owner. The enabled field is only used inside the external view function TokenInfoRegistry.isTokenEnabled, which is not called in any of the other contracts. Instead of checking this field, the smart contracts only check if a token exists before proceeding, regardless of being enabled. This means that even though a token might not be enabled, it can still be used by any of the functions inside SavingAccount.

Recommendations
Either remove the enabled field, or actually use it. Also see **DEF-032** regarding the update functions for the enabled field.

Update
The modifier to check if a token is valid inside SavingAccount now also checks if a token's enabled field is true.

DEF-015 Redundant notZero modifier design medium FIXED

The TokenInfoRegistry contract defines a modifier notZero. This modifier is used in the updateCToken and updateChainLinkAggregator functions. However, since the modifier isTokenExists is used before notZero, the _token being zero will already have been caught inside isTokenExists.

It could be that DeFiner actually wanted to make sure the other address argument to these functions is not address zero. As currently both the cToken as chainlinkAggregator token fields can be updated to address zero.

Recommendations
If the other address argument should not be allowed to be set to zero, apply the notZero modifier to this argument. If that is not what DeFiner intended, remove the notZero modifier.

Update
The modifier has been removed.

DEF-016 Missing token exists check transfer design medium FIXED

The SavingAccount.transfer function does not have the onlySupported(_token) modifier. It will call the internal withdraw and deposit functions, which also do not have the modifier (the public functions of both these functions do have it). If transfer is called with a _token that does not exist, the following check inside withdraw will cause the transaction to revert with the message Insufficient balance. This is a bit misleading as the reason it failed is that it's an Unsupported token, which the modifier would have thrown.

```
    require(_amount <= baseVariable.getDepositBalance(_token, _from), "Insufficient
        balance.");
```

SavingAccount.sol

Recommendations
Apply the `onlySupported(_token)` modifier to the `transfer` function.

Update
The modifier is now also applied to the `transfer` function.

DEF-017 `getTokenState` missing `view` visibility design medium FIXED

The `getTokenState` function inside `SavingAccount` returns the state of a given token. Internally the `Base.getTokenState` function is called to retrieve the token state. Both of the `getTokenState` functions are not marked as `view`.

If `SavingAccount.getTokenState` is currently called from an EOA, a transaction is sent on-chain, costing ETH. Furthermore, the return data of a transaction is always the transaction hash, not the return value(s) of the called function. So currently an EOA cannot call the function and retrieve the token state.

If instead both of the `getTokenState` functions were marked as `view`, an EOA calling `SavingAccount.getTokenState` would be able to retrieve the current token state, and it wouldn't submit an on-chain transaction.

Recommendations
Mark both `getTokenState` functions as `view`.

Update
The function is now marked as `view`.

DEF-018 Ambiguous names of chainlink contract/variables design medium FIXED

The `SymbolsLib.Symbols` struct contains a `ChainLinkOracle chainlinkAggregator` field. This field will contain the address of the `ChainLinkOracle` contract, which acts as an abstraction over the chainlink oracle defined for each token inside `TokenInfoRegistry`.

Inside the `TokenInfoRegistry.TokenInfo` struct an `address chainlinkAggregator` field is defined for each token. This will contain the address of the actual chainlink oracle contract to use for fetching the price of this token.

Having the name `chainlinkAggregator` refer to two different things in different places is confusing. Furthermore, the `ChainLinkOracle` name for the contract is not very appriopriate, as this is actually the "aggregator".

Recommendations

- Rename the `ChainLinkOracle` contract to `ChainlinkAggregator`.
- Rename the `TokenInfoRegistry.TokenInfo.chainlinkAggregator` field to `chainlinkOracle`.

Update
The above recommendations have been implemented.

DEF-019 Ambiguous naming `TokenInfoLib/TokenRegistryLib` design medium PARTIALLY FIXED

The `TokenInfoLib` library contains borrow and deposit (interest) information per account and token. The `TokenInfoRegistry` contains information about registered tokens. Since these names are quite similar, but their information differs significantly, it is confusing and lowers the overall readability of the code. Furthermore, `TokenInfoLib` is not a suitable name as it's about accounts and their tokens.

Besides the ambiguous naming of the library/contract, the decision to name the struct in each of these `TokenInfo` is even more ambiguous. Every time `tokenInfo` is used inside the other contracts it's not immediately clear if this is the `TokenInfoLib.TokenInfo` or `TokenRegistryLib.TokenInfo` struct.

Lastly, since the `TokenRegistryLib` is not a library, appending `Lib` is a bit confusing. Also, the `TokenRegistryLib.TokenInfo` struct could be renamed to `Token`, as it's obvious that a struct contains "info".

Recommendations
Consider applying the following changes:

- Rename `TokenInfoLib` to `AccountLib`.
- Rename the `TokenInfoLib.TokenInfo` struct to `Account`.
- Rename `TokenRegistryLib` to `TokenRegistry`.
- Rename the `TokenRegistryLib.TokenInfo` struct to `Token`.

Applying all these changes would significantly improve the readability and remove the ambiguity.

Update
The `TokenInfoLib` has been renamed to `AccountTokenLib`, and the `TokenInfoRegistry` has been renamed to `TokenRegistry`. However, the structs in both files is still named `TokenInfo`.

DEF-020 `payable` constructor `SavingAccountParamters` design low FIXED

The `SavingAccountParamters` contract is deployed in the initialize function of `SavingAccount`. The `initialize` function serves the purpose of constructor due to the use of OpenZeppelin Upgradability.

`SavingAccount` does not transfer ETH with the `SavingAccountParameters` deployment call. Therefore, the `SavingAccountParameters` constructor does not need to be `payable`.

Recommendations
Remove `payable` from the `SavingAccountParameters` constructor. However, as explained in **DEF-013**, better to get rid of the `SavingAccountParameters` contract entirely.

Update
The `SavingAccountParamters` contract has been removed.

DEF-021 Redundant check in `liquidate` design low FIXED

The `liquidate` function starts with the following line:

```
require(tokenRegistry.isTokenExist(_targetToken), "Unsupported token");
```

SavingAccount.sol

Later on in the function the following check is performed:

```
require(_targetToken != address(0), "Token address is zero");
```

SavingAccount.sol

The first check will implicilty check that the `_targetToken` address is not zero, making the second check redundant.

Recommendations
Remove the second check.
Update
The duplicate check has been removed.

DEF-022 Call prerequisite function from within `updateTotalReserve` design low ACKNOWLEDGED

The `Base.updateTotalReserve` function natspec comment reads:

```
\\ * Update the total reservation. Before run this function, make sure that totalCompound has
    been updated
\\ * by calling updateTotalCompound. Otherwise, self.totalCompound may not equal to the exact
    amount of the
\\ * token in Compound.
```

SavingAccount.sol

Looking at the `SavingAccount`, this prerequisite is fullfilled as the function is everytime called like:

```
baseVariable.updateTotalCompound(_token);
baseVariable.updateTotalLoan(_token);
baseVariable.updateTotalReserve(_token, _amount, <Some Action>);
```

Instead of each time calling these three functions as shown above, updateTotalCompound and updateTotalLoan could be called at the start of updateTotalReserve. This would alleviate the prerequisite to call the updateTotalCompound function before calling updateTotalReserve, and would therefore automatically prevent a possible future mistake of not fullfilling this prerequisite when upgrading the contract.

Recommendations
Call updateTotalCompound and updateTotalLoan at the start of updateTotalReserve.
Update
DeFiner acknowledged the finding but explained this will be updated in a future version of the code.

DEF-023 Define constants as constant design low FIXED

The SavingAccount contract declares the following variables:

```
address payable public EMERGENCY_ADDR;
address public ETH_ADDR;
uint256 public ACCURACY;
uint256 public BLOCKS_PER_YEAR;
uint256 public UINT_UNIT;
```

SavingAccount.sol

These variables are assigned a value inside initialize, which is only executed during deployment of the initial version of the contract.

```
EMERGENCY_ADDR = 0xc04158f7dB6F9c9fFbD5593236a1a3D69F92167c;
ETH_ADDR = 0x000000000000000000000000000000000000000E;
ACCURACY = 10**18;
BLOCKS_PER_YEAR = 2102400;
UINT_UNIT = 10 ** 18;
```

SavingAccount.sol

Since these variables cannot be updated through function calls after deployment, they could be defined as constant. These variables are also very unlikely to ever change.

Recommendations
Declare and set the variables as constant.

Update
A new contract Constant has been added which contains all the constants originally in SavingAccount.

DEF-024 Missing events design low FIXED

It is considered good practice to emit events when important contract variables change or actions are performed. This makes it possible to easily track down when a certain important value changed or action was performed. Throughout the contracts DeFiner emits events in certain functions. However, there are a number of functions for which the emitting of an event should also be considered.

- All of the functions inside GlobalConfig.
- liquidate, recycleCommunityFund, and setDeFinerCommunityFund inside SavingAccount.

Furthermore, each of the token update functions inside TokenInfoRegistry emits the following event:

```
emit TokenUpdated(_token);
```

Since there are no arguments to indicate what changed, the usefulness of this event is suboptimal.

Recommendations
Add events to all the mentioned functions.

Update
Appropriate events are now emitted from the mentioned functions.

DEF-025 Superfluous `SafeDecimalMath` library design low FIXED

The `SafeDecimalsMath` library, used in `Base` and `SavingAccount`, is defined as:

```
library SafeDecimalMath {

    /* Number of decimal places in the representations. */
    uint8 public constant decimals = 18;
    uint8 public constant highPrecisionDecimals = 27;

    /* The number representing 1.0. */
    uint256 public constant UNIT = 10 ** uint256(decimals);
    uint256 public constant UINT_UNIT = 10 ** uint256(18);

    function getUNIT() internal pure returns (uint256) {
        return UNIT;
    }

    function getUINT_UNIT() internal pure returns (uint256) {
        return UINT_UNIT;
    }
}
```

SafeDecimalMath.sol

Note that both functions return the same value, `10**18`, and therefore having just one of the two would be sufficient. Still, having all this code just to return `10**18` is complete overkill. Also, `highPrecisionDecimals` is never used.

The `SavingAccount` contract contains a variable called `UINT_UNIT`, whose value is `10**18`, and which it sometimes uses. However, it also sometimes calls `SafeDecimalMath.getUINT_UNIT()` to retrieve `10**18`.

The `Base` contract will call both the `SafeDecimalMath.getUNIT` and `SafeDecimalMath.getUINT_UNIT` functions whenever `10**18` is needed.

Recommendations
Get rid of the `SafeDecimalMath` library and add a constant to `Base` whose value is `10**18`. Furthermore, the `SavingAccount` contract should be updated to use the `UINT_UNIT` constant. However, since other contracts call this variable `BASE`, consider also renaming `UINT_UNIT` to `BASE`.

Update
The `SafeDecimalsMath` library has been removed.

DEF-026 Store `ETH_ADDR` as `constant` design low FIXED

The `SymbolsLib` library declares a local variable `ETH_ADDR` inside it's `initialize` function. However, inside the `_isETH` function the value is directly used.

Recommendations
Consider adding an `ETH_ADDR` constant to the contract and using that in both places.

Update
The `SymbolsLib` library has been removed.

DEF-027 Superfluous `getBlockNumber` function design low ACKNOWLEDGED

The `SavingAccount` contract contains the following function:

```
    function getBlockNumber() public view returns (uint) {
        return block.number;
    }
```

SavingAccount.sol

At various places throughout `Base` this function is called using:

```
    IBlockNumber(self.savingAccountAddress).getBlockNumber()
```

Base.sol

The only thing this function does is return `block.number`, which has nothing to do with the `SavingAccount` contract. Therefore, Taka Security sees no reason for the `getBlockNumber` function, let alone call this function in such a contrived manner.

Recommendations
Get rid of `getBlockNumber` and replace all `IBlockNumber(self.savingAccountAddress).getBlockNumber()` with `block.number`.

Update
DeFiner acknowledged but explained that this function is necessary to be able to fast-forward the blocknumber when testing the contract.

DEF-028 Unused imports design low PARTIALLY FIXED

The DeFiner smart contracts contain multiple imports that are never used and could be removed.

- `SignedSafeMath` library is included in several contracts, but never used.
- `BitmapLib` is imported in `SavingAccount`, but not used.
- `TokenInfoRegistry` is imported in `GlobalConfig`, but not used.

Recommendations
Remove all unused imports.

Update
Most unused imports have been removed. However, some still remain, for example `ICETH` and `ICToken` inside `SavingAccount`.

DEF-029 Unused struct fields design low FIXED

The `isTransferFeeEnabled`, `isSupportedOnCompound`, and `decimals` fields inside the `TokenInfoRegistry.TokenInfo` struct are never used. Also, the `symbolToPrices` field inside the `SymbolsLib.Symbols` struct is never used.

Recommendations
Consider removing the unused fields. However, as explained in **DEF-012**, the `TokenInfoRegistry.TokenInfo.decimals` field should be kept and used.

Update
The `isSupportedOnCompound` and `decimals` fields are now actually used. Also, DeFiner explained that the `isTransfer FeeEnabled` field will be used in the future.

DEF-030 Simplify return check note

The `SavingAccount.isAccountLiquidatable` function contains the following code:

```
    if (
        totalBalance.mul(100) > totalETHValue.mul(liquidationThreshold) &&
        totalBalance.mul(liquidationDiscountRatio) <= totalETHValue.mul(100)
    ) {
        return true;
    }
    return false;
```

SavingAccount.sol

This could be simplified to:

```
    return (
        totalBalance.mul(100) > totalETHValue.mul(liquidationThreshold) &&
        totalBalance.mul(liquidationDiscountRatio) <= totalETHValue.mul(100)
    );
```

DEF-031 Rename `initialize` to `init` note

When using the OpenZeppelin Upgradability, instead of a `constructor` a function named `initialize` serves the purpose of constructor. The OpenZeppelin `Initializable` contract contains checks to make sure the `initialize` function can only be called once. This can be seen inside `SavingAccount` which inherits `Initializable` and contains an `initialize` function.

The `SymbolsLib` and `Base` libraries also contain a function called `initialize`. Both of these functions will be called from `SavingAccount.initialize`. However, since these two libraries have nothing to do with OpenZeppelin Upgradability, DeFiner could consider renaming the `initialize` functions to something else to make this immediately clear. They could for example be renamed to `init`.

DEF-032 Merge enable/disable token functions note

The `TokenInfoRegistry` contract contains two functions to set the `bool enabled` field of a token.

```
    function enableToken(address _token) external onlyOwner whenTokenExists(_token) {
        require(tokenInfo[_token].enabled == false, "Token already enabled");

        tokenInfo[_token].enabled = true;

        emit TokenUpdated(_token);
    }

    function disableToken(address _token) external onlyOwner whenTokenExists(_token) {
        require(tokenInfo[_token].enabled == true, "Token already disabled");

        tokenInfo[_token].enabled = false;

        emit TokenUpdated(_token);
    }
```

TokenInfoRegistry.sol

Whereas the `bool isTransferFeeEnabled` field only has one function for updating.

```
    function updateEnabled(address _token, bool _enabled) external onlyOwner
        whenTokenExists(_token) {
        require(tokenInfo[_token].enabled != _enabled, "Enabled field already set to value");
        tokenInfo[_token].enabled = _enabled;
        emit TokenUpdated(_token);
    }
```

DeFiner should consider also merging the two `enabled` field functions, such that the code style is consistent. However, as explained in **DEF-014**, the `enabled` field is currently not used and could instead be removed.

DEF-033 `Base.approveAll` could have more descriptive name note

The `Base.approveAll` function can be called with a chosen token address. The corresponding Compound cToken will be fetched from `Base.cTokenAddress`. Then `safeApprove` will be used to set the allowance of the cToken to spend the chosen token belonging to the `SavingAccount` contract.

Taka Security thinks the `approveAll` could have a more descriptive name, as the current name does not in any way signify that it is the cToken of the chosen token which will be approved to spend tokens. The same applies to the `approveAll` function inside `SavingAccount`.

DEF-034 `DepositorOperations` indexed event parameters note

The `DepositorOperations` event is defined as:

```
    event DepositorOperations(uint256 indexed code, address token, address from, address to,
        uint256 amount);
```

SavingAccount.sol

By only indexing the `code` parameter it is not possible to easily filter on `token`, `from`, or `to`. DeFiner could consider also marking two out of those three parameters as `indexed` (Ethereum allows max 3 event parameters to be `indexed`).

DEF-035 Tokens switching from on compound to not on comopound **note**

Throughout the `Base` contract there are many points where a different path is taken depending on if a token is on compound or not, i.e. has a `cToken` or not. In case a token first is on compound for some time and used by depositors and borrowers, and then suddenly is taken off compound (or vice versa), this could have considerable effects on the execution of the code. As now the other branches would be taken when performing the numerous calculations.

Due to the high number of other issues found, and time constraints, Taka Security was unable to fully verify if this causes any problems. DeFiner should carefully inspect the above to see if it causes any problems.

DEF-036 Add more comments in `liquidate` **note**

The `liquidate` function is likely the most complex function throughout the smart contracts. Although there are several comments throughout this function, adding more comments to explain what each step/check does would significantly lower the diffculty of understanding this function.

Disclaimer

Contact:

https://takasecurity.com
contact@takasecurity.com