Definer

Date	February 2021
Lead Auditor	Shayan Eskandari
Co-auditors	Alex Wade

1 Executive Summary

This report presents the results of our engagement with **DeFiner** to review **DeFiner's SavingAccount protocol**.

The review was conducted over two weeks, from **Feb 8, 2021** to **Feb 19, 2021** by **Shayan Eskandari** and **Alex Wade**. A total of 15 person-days were spent.

2 Scope

Our review focused on the commit hash `880e9aaa883b4d9b68ed9ff3c47e9347345526cc`. The list of files in scope can be found in the Appendix.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under review. It is not a substitute for documentation.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- **Owner**: The Owner has unilateral control over each contract in the Definer protocol. Among other actions, this role may:
  - *Switch out any of the core contracts*: This is equivalent to complete control over the protocol and all its held assets.
  - *Change any system-wide constants*: The Owner may arbitrarily set various rates and ratios. Includes, but is not limited to: the community fund ratio, the liquidation threshold, the liquidation purchase discount ratio, and supply / borrow rates from Compound.
  - *Add supported tokens to the* `TokenRegistry`
  - *Change configuration for tokens in the* `TokenRegistry`: Among other actions, the Owner

  may update the borrow LTV for any token and enable/disable any token.
    - *Pause and unpause the primary contract,* `SavingAccount`
- **Emergency Address**: The emergency address may withdraw any funds held in the `SavingAccount` contract.

3.2 Security Concerns

- It should be noted that DeFiner is fully trusted in this system as they control all the variables and contract addresses used in DeFiner smart contract system.
    - Using time lock in these changes can mitigate some issues and protect users from system take over. However this does not solve the underlying centralization.
- DeFiner is susceptible to market manipulations that results in sudden price changes on ChainLink aggregators and also rate fluctuation on Compound tokens used in this system.

4 Findings

Each issue has an assigned severity:

- `Minor` issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- `Medium` issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- `Major` issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- `Critical` issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Users can withdraw their funds immediately when they are over-leveraged `Critical`

Description

`Accounts.withdraw` makes two checks before processing a withdrawal.

First, the method checks that the amount requested for withdrawal is not larger than the user’ s balance for the asset in question:

code/contracts/Accounts.sol:L197-L201

```
function withdraw(address _accountAddr, address _token, uint256 _amount) external onlyAuthorized retur

    // Check if withdraw amount is less than user's balance
    require(_amount <= getDepositBalanceCurrent(_token, _accountAddr), "Insufficient balance.");
    uint256 borrowLTV = globalConfig.tokenInfoRegistry().getBorrowLTV(_token);
```

Second, the method checks that the withdrawal will not over-leverage the user. The amount to be withdrawn is subtracted from the user’ s current “borrow power” at the current price. If the user’ s total value borrowed exceeds this new borrow power, the method fails, as the user no longer has sufficient collateral to support their borrow positions. However, this `require` is only checked if a user is not already over-leveraged:

code/contracts/Accounts.sol:L203-L211

```
// This if condition is to deal with the withdraw of collateral token in liquidation.
// As the amount if borrowed asset is already large than the borrow power, we don't
// have to check the condition here.
if(getBorrowETH(_accountAddr) <= getBorrowPower(_accountAddr))
    require(
        getBorrowETH(_accountAddr) <= getBorrowPower(_accountAddr).sub(
            _amount.mul(globalConfig.tokenInfoRegistry().priceFromAddress(_token))
            .mul(borrowLTV).div(Utils.getDivisor(address(globalConfig), _token)).div(100)
        ), "Insufficient collateral when withdraw.");
```

If the user has already borrowed more than their “borrow power” allows, they are allowed to withdraw regardless. This case may arise in several circumstances; the most common being price fluctuation.

Recommendation

Disallow withdrawals if the user is already over-leveraged.

From the comment included in the code sample above, this condition is included to support the `liquidate` method, but its inclusion creates an attack vector that may allow users to withdraw when they should not be able to do so. Consider adding an additional method to support `liquidate`, so that users may not exit without repaying debts.

4.2 Users can borrow funds, deposit them, then borrow more Major Won’t Fix

Resolution

Comment from DeFiner team:

> This is expected behaviour and our contracts are designed like that. Other lending protocols like Compound and AAVE allows this feature as well. So this is not a CRITICAL issue, as the user’s funds are not at risk. The funds of the users are only at risk when their position is over-leveraged, which is expected behaviour.

Description

Users may deposit and borrow funds denominated in any asset supported by the `TokenRegistry`. Each time a user deposits or borrows a token, they earn FIN according to the difference in deposit / borrow rate indices maintained by `Bank`.

Borrowing funds

When users borrow funds, they may only borrow up to a certain amount: the user’ s “borrow power.” As long as the user is not requesting to borrow an amount that would cause their resulting borrowed asset value to exceed their available borrow power, the borrow is successful and the user receives the assets immediately. A user’ s borrow power is calculated in the following function:

code/contracts/Accounts.sol:L333-L353

```
/**
 * Calculate an account's borrow power based on token's LTV
 */
function getBorrowPower(address _borrower) public view returns (uint256 power) {
    for(uint8 i = 0; i < globalConfig.tokenInfoRegistry().getCoinLength(); i++) {
        if (isUserHasDeposits(_borrower, i)) {
            address token = globalConfig.tokenInfoRegistry().addressFromIndex(i);
            uint divisor = INT_UNIT;
            if(token != ETH_ADDR) {
                divisor = 10**uint256(globalConfig.tokenInfoRegistry().getTokenDecimals(token));
            }
            // globalConfig.bank().newRateIndexCheckpoint(token);
            power = power.add(getDepositBalanceCurrent(token, _borrower)
                .mul(globalConfig.tokenInfoRegistry().priceFromIndex(i))
                .mul(globalConfig.tokenInfoRegistry().getBorrowLTV(token)).div(100)
                .div(divisor)
            );
        }
    }
    return power;
}
```

For each asset, borrow power is calculated from the user’ s deposit size, multiplied by the current chainlink price, multiplied and that asset’ s “borrow LTV.”

Depositing borrowed funds

After a user borrows tokens, they can then deposit those tokens, increasing their deposit balance for that asset. As a result, their borrow power increases, which allows the user to borrow again.

By continuing to borrow, deposit, and borrow again, the user can repeatedly borrow assets. Essentially, this creates positions for the user where the collateral for their massive borrow position is entirely made up of borrowed assets.

Conclusion

There are several potential side-effects of this behavior.

First, as described in issue 4.6, the system is comprised of many different tokens, each of which is subject to price fluctuation. By borrowing and depositing repeatedly, a user may establish positions across all supported tokens. At this point, if price fluctuations cause the user’ s account to cross the liquidation threshold, their positions can be liquidated.

Liquidation is a complicated function of the protocol, but in essence, the liquidator purchases a target’ s collateral at a discount, and the resulting sale balances the account somewhat. However, when a user repeatedly deposits borrowed tokens, their collateral is made up of borrowed tokens: the system’ s liquidity! As a result, this may allow an attacker to intentionally create a massively over-leveraged account on purpose, liquidate it, and exit with a chunk of the system liquidity.

Another potential problem with this behavior is FIN token mining. When users borrow and deposit, they earn FIN according to the size of the deposit / borrow, and the difference in deposit / borrow rate indices since the last deposit / borrow. By repeatedly depositing / borrowing, users are able to artificially deposit and borrow far more often than normal, which may allow them to generate FIN tokens at will. This additional strategy may make attacks like the one described above much more economically feasible.

Recommendation

Due to the limited time available during this engagement, these possibilities and potential mitigations were not fully explored. Definer is encouraged to investigate this behavior more carefully.

4.3 Stale Oracle prices might affect the rates Major

Description

It’ s possible that due to network congestion or other reasons, the price that the ChainLink oracle returns is old and not up to date. This is more extreme in lesser known tokens that have fewer ChainLink Price feeds to update the price frequently. The codebase as is, relies on `chainLink().getLatestAnswer()` and does not check the timestamp of the price.

Examples

/contracts/registry/TokenRegistry.sol#L291-L296

```
    function priceFromAddress(address tokenAddress) public view returns(uint256) {
        if(Utils._isETH(address(globalConfig), tokenAddress)) {
            return 1e18;
        }
        return uint256(globalConfig.chainLink().getLatestAnswer(tokenAddress));
    }
```

Recommendation

Do a sanity check on the price returned from the oracle. If the price is older than a threshold, revert or handle in other means.

4.4 Overcomplicated unit conversions Medium

Description

There are many instances of unit conversion in the system that are implemented in a confusing way. This could result in mistakes in the conversion and possibly failure in correct accounting. It’ s been seen in the ecosystem that these type of complicated unit conversions could result in calculation mistake and loss of funds.

Examples

Here are a few examples:

- /contracts/Bank.sol#L216-L224

```
function getBorrowRatePerBlock(address _token) public view returns(uint) {
    if(!globalConfig.tokenInfoRegistry().isSupportedOnCompound(_token))
    // If the token is NOT supported by the third party, borrowing rate = 3% + U * 15%.
        return getCapitalUtilizationRatio(_token).mul(globalConfig.rateCurveSlope()).div(INT_UNIT

    // if the token is suppored in third party, borrowing rate = Compound Supply Rate * 0.4 + Comp
    return (compoundPool[_token].depositRatePerBlock).mul(globalConfig.compoundSupplyRateWeights(
        add((compoundPool[_token].borrowRatePerBlock).mul(globalConfig.compoundBorrowRateWeights(
}
```

- /contracts/Bank.sol#L350-L351

```
            compoundPool[_token].depositRatePerBlock = cTokenExchangeRate.mul(UNIT).div(lastCToke
                .sub(UNIT).div(blockNumber.sub(lastCheckpoint[_token]));
```

- /contracts/Bank.sol#L384-L385

```
    return lastDepositeRateIndex.mul(getBlockNumber().sub(lcp).mul(depositRatePerBlock).add(INT_U
```

Recommendation

Simplify the unit conversions in the system. This can be done either by using a function wrapper for units to convert all values to the same unit before including them in any calculation or by better documenting every line of unit conversion

4.5 Commented out code in the codebase Medium

Description

There are many instances of code lines (and functions) that are commented out in the code base. Having commented out code increases the cognitive load on an already complex system. Also, it hides the important parts of the system that should get the proper attention, but that attention gets to be diluted.

The main problem is that commented code adds confusion with no real benefit. Code should be code, and comments should be comments.

Examples

Here’s a few examples of such lines of code, note that there are more.

- /contracts/SavingAccount.sol#L211-L218

```
struct LiquidationVars {
    // address token;
    // uint256 tokenPrice;
    // uint256 coinValue;
    uint256 borrowerCollateralValue;
    // uint256 tokenAmount;
    // uint256 tokenDivisor;
    uint256 msgTotalBorrow;
```

- contracts/Accounts.sol#L341-L345

```
            if(token != ETH_ADDR) {
                divisor = 10**uint256(globalConfig.tokenInfoRegistry().getTokenDecimals(token));
            }
            // globalConfig.bank().newRateIndexCheckpoint(token);
            power = power.add(getDepositBalanceCurrent(token, _borrower)
```

- Many usage of `console.log()` and also the commented import on most of the contracts

```
// import "@nomiclabs/buidler/console.sol";
...
//console.log("tokenNum", tokenNum);
```

- /contracts/Accounts.sol#L426-L429

```
    // require(
    //     totalBorrow.mul(100) <= totalCollateral.mul(liquidationDiscountRatio),
    //     "Collateral is not sufficient to be liquidated."
    // );
```

- /contracts/registry/TokenRegistry.sol#L298-L306

```
// function _isETH(address _token) public view returns (bool) {
//     return globalConfig.constants().ETH_ADDR() == _token;
// }

// function getDivisor(address _token) public view returns (uint256) {
//     if(_isETH(_token)) return INT_UNIT;
//     return 10 ** uint256(getTokenDecimals(_token));
// }
```

- /contracts/registry/TokenRegistry.sol#L118-L121

```
    // require(_borrowLTV  = 0, "Borrow LTV is zero");
    require(_borrowLTV < SCALE, "Borrow LTV must be less than Scale");
    // require(liquidationThreshold > _borrowLTV, "Liquidation threshold must be greater than Bor
```

Recommendation

In many of the above examples, it’s not clear if the commented code is for testing or obsolete code (e.g. in the last example, can `_borrowLTV ==0`?) . All these instances should be reviewed and the system should be fully tested for all edge cases after the code changes.

4.6 Price volatility may compromise system integrity Medium Won't Fix

Resolution

Comment from DeFiner team:

> The issue says that due to price volatility there could be an attack on DeFiner. However, price volatility is inherent in the Cryptocurrency ecosystem. All the other lending platforms like MakerDAO, Compound and AAVE also designed like that, in case of price volatility(downside) more liquidation happens on these platforms as well. Liquidations are in a sense good to keep the market stable. If there is no liquidation during those market crash, the system will be at risk. Due to this, it is always recommended to maintain the collateral and borrow ratio by the user. A user should keep checking his risk in the time when the market crashes.

Description

`SavingAccount.borrow` allows users to borrow funds from the bank. The funds borrowed may be denominated in any asset supported by the system-wide `TokenRegistry`. Borrowed funds come from the system's existing liquidity: other users' deposits.

Borrowing funds is an instant process. Assuming the user has sufficient collateral to service the borrow request (as well as any existing loans), funds are sent to the user immediately:

code/contracts/SavingAccount.sol:L130-L140

```
function borrow(address _token, uint256 _amount) external onlySupportedToken(_token) onlyEnabledToken(

    require(_amount != 0, "Borrow zero amount of token is not allowed.");

    globalConfig.bank().borrow(msg.sender, _token, _amount);

    // Transfer the token on Ethereum
    SavingLib.send(globalConfig, _amount, _token);

    emit Borrow(_token, msg.sender, _amount);
}
```

Users may borrow up to their "borrow power", which is the sum of their deposit balance for each token, multiplied by each token's `borrowLTV`, multiplied by the token price (queried from a chainlink oracle):

code/contracts/Accounts.sol:L344-L349

```
// globalConfig.bank().newRateIndexCheckpoint(token);
power = power.add(getDepositBalanceCurrent(token, _borrower)
    .mul(globalConfig.tokenInfoRegistry().priceFromIndex(i))
    .mul(globalConfig.tokenInfoRegistry().getBorrowLTV(token)).div(100)
    .div(divisor)
);
```

If users borrow funds, their position may be liquidated via `SavingAccount.liquidate`. An account is considered liquidatable if the total value of borrowed funds exceeds the total value of collateral (multiplied by some liquidation threshold ratio). These values are calculated similarly to "borrow power:" the sum of the deposit balance for each token, multiplied by each token's `borrowLTV`, multiplied by the token price as determined by chainlink.

Conclusion

The instant-borrow approach, paired with the chainlink oracle represents a single point of failure for the Definer system. When the price of any single supported asset is sufficiently volatile, the entire liquidity held by the system is at risk as borrow power and collateral value become similarly volatile.

Some users may find their borrow power skyrocket and use this inflated value to drain large amounts of system liquidity they have no intention of repaying. Others may find their held collateral tank in value and be subject to sudden liquidations.

4.7 Emergency withdrawal code present Medium

Description

Code and functionality for emergency stop and withdrawal is present in this code base.

Examples

/contracts/lib/SavingLib.sol#L43-L48

```
    // ============================================
    // EMERGENCY WITHDRAWAL FUNCTIONS
    // Needs to be removed when final version deployed
    // ============================================
    function emergencyWithdraw(GlobalConfig globalConfig, address _token) public {
        address cToken = globalConfig.tokenInfoRegistry().getCToken(_token);
...
```

/contracts/SavingAccount.sol#L307-L309

```
    function emergencyWithdraw(address _token) external onlyEmergencyAddress {
        SavingLib.emergencyWithdraw(globalConfig, _token);
    }
```

/contracts/config/Constant.sol#L7-L8

```
...
    address payable public constant EMERGENCY_ADDR = 0xc04158f7dB6F9c9fFbD5593236a1a3D69F92167c;
...
```

Recommendation

To remove the emergency code and fully test all the affected contracts.

4.8 `Accounts` contains expensive looping Medium

This issue describes `Accounts.getBorrowETH` in-depth as an example of potentially-problematic looping in `Accounts`. Similar issues exist in `Accounts.getBorrowPower`, `Accounts.getDepositETH`, `Accounts.isAccountLiquidatable`, and `Accounts.claim`. Definer is encouraged to investigate all of these cases in detail.

Description

`Accounts.getBorrowETH` performs multiple external calls to `GlobalConfig` and `TokenRegistry` within a for loop:

code/contracts/Accounts.sol:L381-L397

```
function getBorrowETH(
    address _accountAddr
) public view returns (uint256 borrowETH) {
    uint tokenNum = globalConfig.tokenInfoRegistry().getCoinLength();
    //console.log("tokenNum", tokenNum);
    for(uint i = 0; i < tokenNum; i++) {
        if(isUserHasBorrows(_accountAddr, uint8(i))) {
            address tokenAddress = globalConfig.tokenInfoRegistry().addressFromIndex(i);
            uint divisor = INT_UNIT;
            if(tokenAddress != ETH_ADDR) {
                divisor = 10 ** uint256(globalConfig.tokenInfoRegistry().getTokenDecimals(tokenAddress
            }
            borrowETH = borrowETH.add(getBorrowBalanceCurrent(tokenAddress, _accountAddr).mul(globalCo
        }
    }
    return borrowETH;
}
```

The loop also makes additional external calls and delegatecalls from:

- `TokenRegistry.priceFromIndex`:

code/contracts/registry/TokenRegistry.sol:L281-L289

```
function priceFromIndex(uint index) public view returns(uint256) {
    require(index < tokens.length, "coinIndex must be smaller than the coins length.");
    address tokenAddress = tokens[index];
    // Temp fix
    if(Utils._isETH(address(globalConfig), tokenAddress)) {
        return 1e18;
    }
    return uint256(globalConfig.chainLink().getLatestAnswer(tokenAddress));
}
```

- `Accounts.getBorrowBalanceCurrent`:

code/contracts/Accounts.sol:L313-L331

```
function getBorrowBalanceCurrent(
    address _token,
    address _accountAddr
) public view returns (uint256 borrowBalance) {
    AccountTokenLib.TokenInfo storage tokenInfo = accounts[_accountAddr].tokenInfos[_token];
    uint accruedRate;
    if(tokenInfo.getBorrowPrincipal() == 0) {
        return 0;
    } else {
        if(globalConfig.bank().borrowRateIndex(_token, tokenInfo.getLastBorrowBlock()) == 0) {
            accruedRate = INT_UNIT;
        } else {
            accruedRate = globalConfig.bank().borrowRateIndexNow(_token)
            .mul(INT_UNIT)
            .div(globalConfig.bank().borrowRateIndex(_token, tokenInfo.getLastBorrowBlock()));
        }
        return tokenInfo.getBorrowBalance(accruedRate);
    }
}
```

In a worst case scenario, each iteration may perform a maximum of 25+ calls/delegatecalls. Assuming a maximum `tokenNum` of 128 (`TokenRegistry.MAX_TOKENS`), the gas cost for this method may reach upwards of 2 million for external calls alone.

Given that this figure would only be a portion of the total transaction gas cost, `getBorrowETH` may represent a DoS risk within the `Accounts` contract.

Recommendation

- Avoid for loops unless absolutely necessary
- Where possible, consolidate multiple subsequent calls to the same contract to a single call, and store the results of calls in local variables for re-use. For example,

Instead of this:

```
uint tokenNum = globalConfig.tokenInfoRegistry().getCoinLength();
for(uint i = 0; i < tokenNum; i++) {
  if(isUserHasBorrows(_accountAddr, uint8(i))) {
    address tokenAddress = globalConfig.tokenInfoRegistry().addressFromIndex(i);
    uint divisor = INT_UNIT;
    if(tokenAddress != ETH_ADDR) {
      divisor = 10 ** uint256(globalConfig.tokenInfoRegistry().getTokenDecimals(tokenAddress));
    }
    borrowETH = borrowETH.add(getBorrowBalanceCurrent(tokenAddress, _accountAddr).mul(globalConfig.tok
  }
}
```

Modify `TokenRegistry` to support a single call, and cache intermediate results like this:

```
TokenRegistry registry = globalConfig.tokenInfoRegistry();
uint tokenNum = registry.getCoinLength();
for(uint i = 0; i < tokenNum; i++) {
  if(isUserHasBorrows(_accountAddr, uint8(i))) {
    // here, getPriceFromIndex(i) performs all of the steps as the code above, but with only 1 ext cal
    borrowETH = borrowETH.add(getBorrowBalanceCurrent(tokenAddress, _accountAddr).mul(registry.getPric
  }
}
```

4.9 Naming inconsistency `Minor`

Description

There are some inconsistencies in the naming of some functions with what they do.

Examples

- /contracts/registry/TokenRegistry.sol#L272-L274

```
function getCoinLength() public view returns (uint256 length) { //@audit-info coin vs token
    return tokens.length;
}
```

Recommendation

Review the code for the naming inconsistencies.

Appendix 1 - Files in Scope

This audit covered the following files:

File Name	SHA-1 Hash
contracts/Accounts.sol	f562f6404758d252ed3662155349a1c4a4638c57
contracts/lib/AccountTokenLib.sol	a29fa0969c67697568cdb64d298263ef43a93b85
contracts/lib/BitmapLib.sol	2c1ba5411a775e8a427e59c31a95a2aa4c0c6199
contracts/lib/Utils.sol	7404d204b028244fab937088286452c720f4347e
contracts/config/GlobalConfig.sol	fe217933c8f24bab1e4ad611b10a645c5ddec75d
contracts/registry/TokenRegistry.sol	58746c39254f0b06985cbbec3ab3dd6c175439f8
contracts/SavingAccount.sol	3f7c60c2083006d710115543b99b873dabfca6d1
contracts/config/Constant.sol	15b61e1384788e555ba0bd07770a90d0ccd9b963
contracts/lib/SavingLib.sol	69eaa76ce1bd3b61df6226ab0646b0f242149381
contracts/InitializableReentrancyGuard.sol	9ba6c6b7dd1397c3700901e2ec818a17e100f2b7
contracts/InitializablePausable.sol	7e1ef3878154b66b351f749601b5328fe0c5fde1
contracts/Bank.sol	d8b954351cc2c5cdf38e60086190ad67d9d40e39
contracts/oracle/ChainLinkAggregator.sol	c51b0a052ffc8bf4b65e24bd0f944b9a914df0d9

Appendix 2 - Disclosure

areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.