www.securing.biz

Smart Contracts Security Review of UnboundFinance for Opin Foundation

Retest

document version: 1.1

author: Damian Rusinek

test time period: 2021-02-03 – 2021-02-18

report date: 2021-05-14

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Testing overview

The security review of the UnboundFinance 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:

- Manual source code review,
- Automatic scans using security verification tools,
- Verification of compliance with Smart Contracts Security Verification Standard (SCSVS) in a form of code review,
- Q&A sessions with the client's representatives which allowed to gain knowledge about architecture and the technical details behind the platform.

1.2. Summary of test results

- During the security review no critical vulnerabilities were found (vulnerabilities which are easy to exploit and have high risk impact).
- There were 2 vulnerabilities with high, 1 with medium and 5 with low impact on risk were identified. They most important consequences are:
    - Unlocking more uToken than allowed according to the collateralization ratio (4.1).
    - Minting uToken without any collateral (4.2).
    - With access to the Owner address the attacker could (4.3):
        - Mint under-collateralized uTokens by setting the loan ratio greater than 100%.
        - Perform the Denial of Service and lock user's LPTs permanently.
        - Revert transactions submitted by users but not yet approved (waiting in mempool) leading to the user's fee loss.
- The scope included preparation of a security checklist for Liquidity Pool Tokens to be included in the UnboundFinance project. It is available in a separate chapter (6 Security Checklist for Liquidity Pool Tokens).
- The results of SCSVS compliance verification are attached to the report in the *SCSVS-UnboundFinance.xlsx* file.
- Additionally, eleven recommendations have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.

1.3. Summary of retest results

- All vulnerabilities and recommendations has been addressed by the Unbound Finance team. Only two were not fixed.
    - For one vulnerability - 4.3 Excessively powerful Owner address – the risk has been accepted, because the team plans to introduce the decentralized governance in the future.
    - The other vulnerability - 4.8 Outdated documentation – has been partially fixed. New articles describing the Block Limit Lock Mechanism and new Price Oracle Solution have been added but no modification in the white paper have been introduced yet.
- Eight of the 11 recommendations have been implemented. The following 3 have been partially implemented:
    - 5.1 Function names should correspond to their implementation
    - 5.8 Remove unused contracts and libraries
    - 5.11 Optimize the gas consumption
- The status of retested vulnerabilities and recommendations has been added to the table – see 2.2 Identified vulnerabilities.

1.4. 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. Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.

2. SUMMARY OF IDENTIFIED VULNERABILITIES

2.1. 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). 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. 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. 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. 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. The decision whether to take mitigating actions should be made by the client.	

2.2. Identified vulnerabilities

Vulnerability	Risk impact	Retest 2021-05-14
4.1 Collateralization ratio mechanism bypass	High	Fixed
4.2 Minting uToken without collateral via reentrancy	High	Fixed
4.3 Excessively powerful Owner address	Medium	Accepted risk
4.4 Lack of possibility to pause sensitive operations	Low	Fixed
4.5 Missing zero-address verification	Low	Fixed
4.6 Potential underflow leading to reverting all transactions	Low	Fixed
4.7 Front-runnable approve function	Low	Fixed
4.8 Outdated documentation	Low	Partially fixed
Recommendations		
5.1 Function names should correspond to their implementation		Partially implemented
5.2 Use widely accepted implementation of pausability		Implemented
5.3 Use consistent function naming		Implemented
5.4 Use clear require error messages		Implemented
5.5 Correct comments		Implemented
5.6 Improve the tests coverage		Implemented
5.7 Do not use multiple variables to store the same value		Implemented
5.8 Remove unused contracts and libraries		Partially implemented
5.9 Remove unused code		Implemented
5.10 Consider verifying if the on-chain oracle data is up-to-date		Implemented
5.11 Optimize the gas consumption		Partially implemented

3. PROJECT DESCRIPTION

3.1. Basic information

Testing team	Damian Rusinek
Testing time period	2021-02-03 – 2021-02-18
Retests time period	2021-05-12 - 2021-05-14
Report date	2021-05-14
Version	1.1

3.2. Target in scope

The object being analysed was UnboundFinance platform accessible in the following GitHub repository:

- https://github.com/unbound-finance/UnboundContracts

The contracts reviewed were the following:

- contracts/LiquidityLockContracts/
  - LiquidityLockContract.sol
  - LLC_EthDai.sol
  - OracleLibrary.sol
- contracts/Valuators/
  - valuing_01.sol
- contracts/UnboundTokens/
  - unboundDollar.sol

Commit ID: **4580675df56e9eda8569858b39f37bad88d6fa30**.

3.3. Threat analysis

The key threats were identified as follows:

1. Incorrect uToken minting calculation.
2. Flash Loan assisted arbitrage exploitation.
3. Vulnerable or malicious LPT pair.
4. Insufficient LP liquidity.
5. Vulnerable or malicious AMM platform.
6. Valuator uToken confusion.

7. Compromised Owner keys.

3.4. Scope

Following the specification, the tests covered:

- Manual source code review, including:
  - Oracle implementation for security vulnerabilities,
  - Administrative actions for business logics and other vulnerabilities,
  - Correctness of Minting and burning functionality (economic viability for LTVs during minting),
  - Blocklimit functionality (i.e. ability to do multiple sensitive actions in one transactions),
  - Unlocking functionality (Collateralization Ratio for partial unlock),
- Security checklist for LPT to be listed, including:
  - Reproduce economical, market based attack scenarios and also review correctness of LPT pairs with respect to security (Uniswap pairs with ETH cryptocurrency),
  - The checklist was based on the following pairs:
    - WBTC-ETH
      Mainnet: 0xbb2b8038a1640196fbe3e38816f3e67cba72d940
    - USDC-ETH
      Mainnet: 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc
    - DAI-ETH
      Mainnet: 0xa478c2975ab1ea89e8196811f51a7b7ade33eb11
    - ETH-USDT
      Mainnet: 0x0d4a11d5eeaac28ec3f61d100daf4d40471f1852
    - DAI-USDT
      Mainnet: 0xb20bd5d04be54f870d5c0d3ca85d82b34b836405
    - USDC-USDT
      Mainnet: 0x3041cbd36888becc7bbcbc0045e3b1f144466f5f
    - DAI-USDC
      Mainnet: 0xae461ca67b15dc8dc81ce7615e0320da1a9ab8d5
    - WBTC-USDC
      Mainnet: 0x004375dff511095cc5a197a54140a24efef3a416
    - LINK-ETH
      Mainnet: 0xa2107fa5b38d9bbd2c461d6edf11b11a50f6b974
    - AAVE-ETH
      Mainnet: 0xdfc14d2af169b0d36c4eff567ada9b2e0cae044f
- Verification of the test coverage and proposal of gas optimizations,
- Use of automatic security verification tools,

- Compliance with Smart Contracts Security Verification Standard (SCSVS), version 1.1.

3.5. Exclusion and remarks

The initial scope included up to 20 LP pairs to be reviewed, but eventually 10 Uniswap pairs were delivered and reviewed. These pairs used the same Uniswap pair v2 code.

4. VULNERABILITIES

4.1. Collateralization ratio mechanism bypass

Risk impact	High
Exploitation conditions	None.
Exploitation results	Unlocking more uToken than allowed according to the collateralization ratio.
Remediation	The “>” sign in the *if* statement should be changed to “<=” sign.

Retest 2021-05-14:
Fixed
The sign has been changed.

Vulnerability description:
The *getLPTokensToReturn* function in *LiquidityLockContract* contract incorrectly calculates the amount of liquidity pool tokens to return in the partial unlock functionality.

Test case:
The *if* statement in *LiquidityLockContract #453* should check whether the current collateralization ratio is greater that specified in the contract.
The current *if* statement contains “<” sign and is the following:

```
if (CREnd.mul(10**18).div(CRNorm) > CRNow) {
```

This equations checks the opposite condition than expected. It returns *true* when the current collateralization ratio is lower than required.

4.2. Minting uToken without collateral via reentrancy

Risk impact	High
Exploitation conditions	The *LiquidityLockContract* must use a liquidity pool token that calls a callback function defined by the sender when transferring the token (e.g. ERC777).
Exploitation results	Minting uToken without any collateral.
Remediation	The blocklimit variable (*nextBlock*) should be updated right after it is verified (*LiquidityLockContract #205*and *#352*). The *_tokensLocked* variable should be increased after the transfer of liquidity pool token. The code from *LiquidityLockContract #213* should be moved after the calls of function *transferLPTPermit* in *LiquidityLockContract #232* and *#249*.

Retest 2021-05-14:
Fixed
The internal function *lockLPTBody* has been removed and the value is calculated in the *lockLPT* and *lockLPTWithPermin* functions directly.
The update of blocklimit variable has been moved and now it is updated before the transfer call. Additionally, the update of locked amount is moved after the transfer call.

Vulnerability description:
The *LiquidityLockContract* implements a blocklimit functionality which blocks calling the locking and unlocking functions for a specified period of time (10 blocks by default). The incorrect implementation of blocklimit functionality allows to call these functions multiple times in the same transaction call.
Additionally, the locked liquidity pool tokens are added to user's balance before they are transferred which allows to take them back using *unlockLPT* call via reentrancy.
The Proof of Concept has been commited and pushed on the *attack/reentrancy* branch.

Test case:
The blocklimit functionality firstly verifies whether user has not called locking or unlocking function (e.g. *LiquidityLockContract #205*) within the specified last blocks and then updates its value (e.g. *LiquidityLockContract #255* in function *lockLPT*).

```
function lockLPTBody(uint256 LPTamt) internal returns (uint256 LPTValueInDai) {
    require(!killSwitch, "LLC: This LLC is Deprecated");
    require(LPTContract.balanceOf(msg.sender) >= LPTamt, "LLC: Insufficient
LPTs");
    require(nextBlock[msg.sender] <= block.number, "LLC: user must wait");

    uint256 totalLPTokens = LPTContract.totalSupply();

    // Acquire total baseAsset value of pair
    uint256 totalUSD = getValue();

    // map locked tokens to user address
    _tokensLocked[msg.sender] = _tokensLocked[msg.sender].add(LPTamt);

    // This should compute % value of Liq pool in Dai. Cannot have decimals in
Solidity
    LPTValueInDai = totalUSD.mul(LPTamt).div(totalLPTokens);
}

function lockLPT(uint256 LPTamt, uint256 minTokenAmount) public {
    uint256 LPTValueInDai = lockLPTBody(LPTamt);

    // transfer LPT to the address
    transferLPT(LPTamt);

    // Call Valuing Contract
    valuingContract.unboundCreate(LPTValueInDai, msg.sender, minTokenAmount);
    // sets nextBlock
    nextBlock[msg.sender] = block.number.add(blockLimit);

    // emit lockLPT event
    emit LockLPT(LPTamt, msg.sender);
}
```

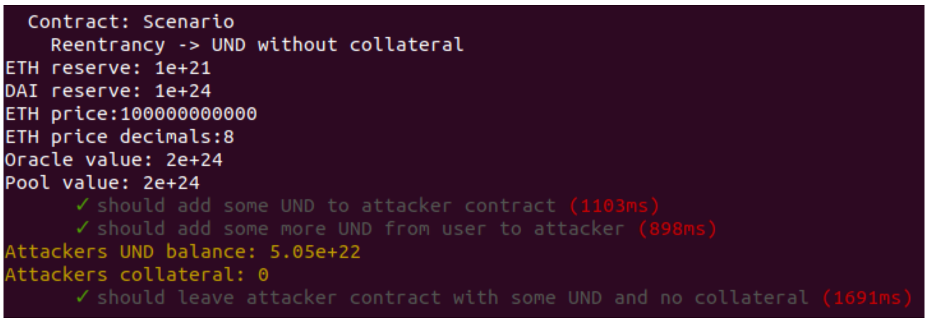
Meanwhile, the *transferLPT* function is called, which transfers the liquidity pool tokens and calls back the contract defined by the sender (attacker who locks the LPT).
The attacker's callback function is the following (compliant with ERC777 as an example):

```
function tokensToSend(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes calldata userData,
    bytes calldata operatorData
) external override     {
    llc.unlockLPT(und.checkLoan(address(this), address(llc)));
}
```

Now, during the locking liquidity pool tokens the *unlockLPT* function is called which fulfils the *nextBlock* variable requirement, because it is not yet updated.
Also, as highlighted (in red) in the first snippet in the description, the *_tokensLocked* is already increased, therefore during the nested *unlockLPT* call:

- The attacker's locked LPT balance in *_tokensLocked* is already updated.
- The uToken for the LPTs being locked is not yet minted.

The unlock call would burn the current attacker's loan (without not yet minted uTokens) and return all LPTs (including the currently being locked). After this nested unlock call is finished, the rest of the *lockLPT* function is executed that mints new uToken for the attacker.
The consequence is that the attacker has minded uTokens and no LPTs as collateral.
Below is the output of the PoC from *attack/reeentrancy* branch:


```
  Contract: Scenario
    Reentrancy -> UND without collateral
ETH reserve: 1e+21
DAI reserve: 1e+24
ETH price:100000000000
ETH price decimals:8
Oracle value: 2e+24
Pool value: 2e+24
      ✓ should add some UND to attacker contract (1103ms)
      ✓ should add some more UND from user to attacker (898ms)
Attackers UND balance: 5.05e+22
Attackers collateral: 0
      ✓ should leave attacker contract with some UND and no collateral (1691ms)
```

4.3. Excessively powerful Owner address

Risk impact	Medium
Exploitation conditions	Access to the Owner's wallet.
Exploitation results	Possibility to perform the following actions: • Mint under-collateralized uTokens by setting the loan ratio greater than 100%. • Perform the Denial of Service and lock user's LPTs permanently. • Revert transactions submitted by users but not yet approved (waiting in mempool) leading to the user's fee loss.
Remediation	The remediation is divided into 3 categories: • Remove or change update functions that can lead to DoS, permanent lock of users' LPTs and minting under-collateralized uTokens. • In short term we recommend to implement timelocks to delay the updates that should be kept. • In long term we recommend to implement the governance with upgradability process. The remediation is explained in details below.

Retest 2021-05-14:

Accepted risk

The Unbound Finance team is planning to replace the owner-based management with governance contract in the future.

Vulnerability description:

Audited contracts contain a lot of functions that can be called by the Owner only, which is an Externally Owned Account (not a contract). These functions include updating addresses of other contracts (within the scope of the audit) and business logic parameters. Some of the function calls can have significant consequences as listed in the exploitation results section.

The proposed remediations should not be introduced for the pausing functionality before the governance is implemented.

Test case:

Denial of Service and locking users' LPTs temporary and permanently

The Owner can change:

- the valuator address using *changeValuator* in *UnboundDollar* contract,
- the *valuingContract* address using *setValuingAddress* in *LiquidityLockContract* contract.

Both these changes could lead to the situation when unlocking LPTs is not possible as the *UnboundDollar* contract would not accept call to *unboundRemove* or the new valuing contract would revert all calls. As the consequence all transactions would be reverted.

The Owner can also set a very big block limit value using *setBlockLimit* function in *LiquidityLockContract* contract, which would not allow to unlock LPT for a very long time.

Another potentially dangerous function is *setCREnd* in *LiquidityLockContract* contract which allows the Owner to set a very high collateralization ratio and make it very unprofitable to unlock LPTs.

Mint under-collateralized uTokens

The Owner can set the value of *loanRate* parameter greater than *rateBalance* parameter using *addLLC* and *changeLoanRate* functions in Valuing_01 contract. That would lead to minting more uTokens than value of the collateral.

Detailed remediation:

Removing functions

The following functions should be removed:

- *UnboundDollar*
  - *changeValuator* – the valuator address should be constant,
- *LiquidityLockContract*
  - *setValuingAddress* – the valuator address should be constant.

Modify functions

The following functions should be modified:

- *Valuing_01*
  - *addLLC* – the loan rate should be limited to not be greater than *rateBalance*, because it would allow to mint under-collateralized uTokens,
  - *changeLoanRate* – the loan rate should be limited to not be greater than *rateBalance*, because it would allow to mint under-collateralized uTokens,
- *LiquidityLockContract*
  - *setBlockLimit* – the function could be removed to make the limit parameter constant or the limit parameter should have a maximum reasonable value,

  - *setCREnd* – the function could be removed to make the collateralization ratio constant or the collateralization ratio parameter should have a maximum reasonable value,
  - *setMaxPercentDifference* – the function could be removed to make the max percentage difference parameter constant or the parameter should have a minimum value.

Timelocks

The timelock contract (Compound's example) allows to delay the updates in time (in terms of blocks) and thus it does not allow the Owner to get their transactions accepted before the users' transactions due to the transaction order dependency.
The following functions should be time-locked:

- *UnboundDollar*
  - *changeSafuShare*
  - *changeStakeShare*
- *Valuing_01*
  - *addLLC*
  - *disableLLC*
  - *changeLoanRate*
  - *changeFeeRate*
- *LiquidityLockContract*
  - *setBlockLimit*
  - *setCREnd*
  - *setMaxPercentDifference*

Governance

In the long term, a governance contract should be implemented (on top of the timelock contract). All updates should be voted and if they are accepted they should be queued in the timelock contract to be later activated.
The only exception is the pausing functionality which could be activated without voting, after a small number of signers accept it (e.g. 2 signers).

4.4. Lack of possibility to pause sensitive operations

Risk impact	Low
Exploitation conditions	Large fluctuations in token prices reflected in on-chain oracles.
Exploitation results	Partial unlocking resulting in gaining more uTokens than allowed.
Remediation	Adding a requirement in the partial unlock branch of *unlockLPT* function that checks whether the contract is not paused.

Retest 2021-05-14:

Fixed

The *LiquidityLockContract* now uses *Pausable* contract. Function for locking and unlocking can now be paused and there is an *emergencyunlockLPT* function that allows to unlock all locked LPTs which is compliant with the requirement that users' funds should not be locked forever.

Vulnerability description:

The *LiquidityLockContract* implements a pause functionality (called kill switch) which allows to block the *lockLPTBody* function (*LiquidityLockContract #203*). The goal of pause functionality is to temporarily stop the sensitive functions that rely on the external sources of information in case of unexpected situations (e.g. large fluctuations, vulnerabilities discovered).

However, the partial unlocking functionality also depends on the external sources of information and cannot be paused. If there is a temporary significant increase of the tokens' price the users would be able to unlock their LPTs for high prices and when the token prices are restored to the real values, users' loans become under-collateralized.

Test case:

The pause requirement is added to the *lockLPTBody* function (*LiquidityLockContract #203*) but it is not present in the partial unlock branch of *unlockLPT* function (*LiquidityLockContract #376)*.

4.5. Missing zero-address verification

Risk impact	Low
Exploitation conditions	Sending a transaction with zero-address as parameter (e.g. mistaken function call without the argument).
Exploitation results	Temporal denial of service and loss of fees.
Remediation	Adding the *require* statement which makes sure that the address set is not equal to zero: `Require(variable != address(0), “Change to the zero address”)`

Retest 2021-05-14:
Fixed
The zero-address verification has been added in listed functions.

Vulnerability description:
The audited contracts do not verify in their functions whether the addresses of external contracts are zero.
It would result in a temporal denial of service in the following cases:

- *UnboundDollar*
    - *changeValuator*
- *LiquidityLockContract*
    - *setValuingAddress*

It would result in the permanent loss of fees in the following cases:

- *UnboundDollar*
    - *constructor*
    - *changeStaking*
    - *changeSafuFund*
    - *changeDevFund*

Test case:
The following functions do not verify whether the parameter value is not zero-address:

- *UnboundDollar*
    - *constructor*
    - *changeStaking*
    - *changeSafuFund*
    - *changeDevFund*
    - *changeValuator*
- *LiquidityLockContract*
    - *setValuingAddress*

4.6. Potential underflow leading to reverting all transactions

Risk impact	Low
Exploitation conditions	Deploying *LiquidityLockContract* for a base asset with number of decimals lower than 2.
Exploitation results	Reverting all locking and unlocking transactions leading to a useless *LiquidityLockContract*.
Remediation	Add requirement in the *LiquidityLockContract* constructor (#155) that makes sure the number of decimals is greater than 2. `require(baseAssetDecimal >= 2, “Base asset must have at least 2 decimals”);`

Retest 2021-05-14:
Fixed
The vulnerable code is not present anymore because there is a new price oracle implementation.

Vulnerability description:
The *LiquidityLockContract* contract uses *OracleLibrary* to make sure that the base asset price is close to the price in the pool.
The unchecked number of decimals can lead to underflow and revert all transactions.

Test case:
The *OracleLibrary* calculates value of the base asset in line 28:

```
_baseAssetValue = _baseAssetValue / (10**(_decimals - 2));
```

If the *_decimals* variable is lower than 2, the subtraction would underflow and the *_baseAssetValue* would be zero and not pass requirement in the next line:

```
require(
   _baseAssetValue <= (100 + percentDiff) && _baseAssetValue >= (100 -
percentDiff), "stableCoin not stable"
);
```

4.7. Front-runnable approve function

Risk impact	Low
Exploitation conditions	Front-running victim's transaction that changes existing victim's allowance for attacker as spender (by using *approve* function).
Exploitation results	Transferring victim's uTokens on behalf of the previous allowance as well as the new allowance.
Remediation	As there is not general remediation for this vulnerability, encourage users to not use this functionality. Also, do not use the *approve* function in a DApp. Use the *increaseAllowance* and *decreaseAllowance* instead.

Retest 2021-05-14:
Fixed
Fixed on the frontend application side.

Vulnerability description:
The *UnboundDollar* contract implements ERC20 standard which contains *approve* function. The function allows to set a given allowance of sender's tokens to a defined spender.
Front-running the transaction that changes existing allowance makes it possible for the malicious spender to transfer victim's tokens before the *approve* function call is added to block and again, using another transaction, to transfer the newly set amount of tokens. As the consequence, the attacker has transferred the sum of both allowances and not the second allowance only, as it was desired by the victim.
This vulnerability is common vulnerability for ERC20 implementations and there is no general mitigation.

Test case:
The vulnerable implementation of *approve* function in *UnboundDollar* contract can be found below:

```
function approve(address spender, uint256 amount) public override returns (bool)
{
    _approve(msg.sender, spender, amount);
    return true;
}

function _approve(
    address owner,
    address spender,
    uint256 amount
) internal {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");
    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}
```

Consider the following calls:

```
Victim submits: approve(Spender, 1000) – ADDED TO MEMPOOL
Transaction validated: Victim: approve(Spender, 1000)

# Victim wants to set allowance to 100 because 1000 was a mistake
Victim submits: approve(Spender, 100) – ADDED TO MEMPOOL

# Spender becomes MaliciousSpender
MaliciousSpender submits: transferFrom(Victim, 1000) – ADDED TO MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 1000)
Transaction validated: Victim: approve(Spender, 100)
MaliciousSpender submits: transferFrom(Victim, 100) – ADDED TO MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 100)
```

This scenario presents a chain of transactions where the malicious spender was able to transfer 1100 tokens, while victim wanted to set the allowance to 100.

4.8. Outdated documentation

Risk impact	Low
Exploitation conditions	None.
Exploitation results	UnboundFinance image loss due to unclear and outdated documentation (white paper). Potentially incorrect integrations with other DeFi projects.
Remediation	Add description of new functionalities to the documentation.

Retest 2021-05-14:
Partially fixed
No changes have been found in the whitepaper, but there are new articles that describe the Block Limit Lock Mechanism and new Price Oracle Solution.

Vulnerability description:
The documentation (white paper) does not include the full description of all functionalities and specifies some parameters that can be easily changed by the Owner.

Test case:
The white paper lacks the documentation for the partial unlocking and collateralization ratio functionalities.
The documentation specifies the following fee distribution:

a) SAFU Fund (40%),
b) UND-DAI Liquidity Pool (40%),
c) Team Fund (20%),

However, the Owner can easily change it using the following functions from *Valuing_01* contract:

- changeSafuShare,
- changeStakeShare.

Moreover, there is an asterisk sign next to the following documentation chapters, but it is not explained:

- Tokenomics,
- Fee Distribution.

5. RECOMMENDATIONS

5.1. Function names should correspond to their implementation

Retest 2021-05-14:
Partially implemented
The *disableLock* function has been removed, but *addLLC* function is still present and not updated.

Description:
Some function names do not correspond to the business logic they implement.

- In *Valuing_01* contract function *addLLC* not only allows to add new LLC, but also allows the Owner to modify any existing LLC parameters (e.g. loan rate).
- In *LiquidityLockContract* function *disableLock* not only can disable the locking but also enable it.

How to implement:
Change the function names (and documentation) or their implementation to reflect the name.

5.2. Use widely accepted implementation of pausability

Retest 2021-05-14:
Implemented
The *Pausable* contract has been used to protect *LiquidityLockContract*.

Description:
The *LiquidityLockContract* uses *killSwitch* boolean value to pause the locking LPT functionality. It is recommended to use the well-known implementation of pausability.

How to implement:
Use *Pausable* contract. Note that the contract mentioned inherits *Context* contract. Make sure this inheritance is required and if not, remove it and change *_msgSender* function calls to *msg.sender*.

5.3. Use consistent function naming

Retest 2021-05-14:
Implemented
The names of functions have been changed.

Description:
Functions and contract fields starting with underscore sign "_" are usually meant to be internal or private. Some non-internal function names start with underscore:

- *UnboundDollar*
  - *_mint* – is external,
  - *_burn* – is external.

How to implement:
Rename functions and remove the underscore sign.

5.4. Use clear require error messages

Retest 2021-05-14:
Implemented
The error messages have been changes according to the description. A few were removed because the functions that included them were removed as well.

Description:
Some error messages in *require* statements are not clear. Examples:

- *UnboundDollar*:
  - Line 280 - "UND: Tx took too long", proposal: “UND: minting less tokens than minimum amount”,
  - Line 358 – “bad input”, proposal: “Too big value for Safu share”,
  - Line 365 – “bad input”, proposal: “Too big value for Stake share”,
- *Valuing_01*:
  - Lines 89 – “value too small”, proposal: “Cannot mint 0 loan value”
  - Lines 95 – “amount too small”, proposal: “Too small loan value to pay the fee”,
- *LiquidityLockContract*:
  - Line 157 – “invalid”, proposal: “Mismatch of base asset and pool assets”,
  - Lines 173 and 174 – “invalid address args”, proposal: “Invalid number of price feeds”,
  - Line 476 – “invalid number”, proposal: “Block limit cannot be 0”,
  - Line 490 – “cannot be beyond 100”, proposal: “Max percentage difference cannot be greater than 100”

How to implement:
Correct the error messages.

5.5. Correct comments

Retest 2021-05-14:
Implemented
The comments were corrected.

Description:
The comments documenting the source code do not follow the implementations:

- *Valuing_01*
  - Line 131

```
// Enter 2500 for 0.25%, 250 for 2.5%, and 25 for 25%.
```

should be:

```
// Enter 2500 for 0.25%, 25000 for 2.5%, and 250000 for 25%.
```

How to implement:
Correct the comments.

5.6. Improve the tests coverage

Retest 2021-05-14:
Implemented
The tests for scenarios pointed out in the description have been added.

Description:
The unit tests do not cover all important protocol flows. The tests do not cover:

- reentrancy scenario to check blocklimit,
- partial unlocking and correctness of Collateralization Ration calculation.

How to implement:
Implement unit tests for before-mentioned functionalities and protocol flows.

5.7. Do not use multiple variables to store the same value

Retest 2021-05-14:
Implemented
The *pair* field has been removed.

Description:
In the *LiquidityLockContract* contract the liquidity pool address is kept in two variables:

- *pair*,
- *LPTContract* (casted to *IUniswapV2Pair_0* interface).

It is not recommended to keep the same value in different variables.

How to implement:
Remove *pair* field and change the line 498 to:

```
require(_tokenAddr != address(LPTContract), "Cannot move LP tokens");
```

5.8. Remove unused contracts and libraries

Retest 2021-05-14:
Partially implemented

The *LiquidityLockContract* does not use *Address* library but it is still imported and present in comments. Also, the *Valuing_01* contract still uses *Address* library.
The inheritance of *Context* contract has been removed.

Description:
All audited contracts use OpenZeppelin's Address library for address type. However, it is never used.
Additionally, the *UnboundDollar* contract inherits from *Context* contract and never uses *_msgSender* function.

How to implement:
Consider removing unused contracts and libraries (Context, Address).

5.9. Remove unused code

Retest 2021-05-14:
Implemented
The code has been removed.

Description:
In the *UnboundDollar* contract the modifier *onlyValuator* is implemented and never used.

How to implement:
Remove unused code.

5.10. Consider verifying if the on-chain oracle data is up-to-date

Retest 2021-05-14:
Implemented
The freshness of chainlink's data is checked in *getChainlinkPrice* function.

Description:
OracleLibrary gets the current price of verified assets from on-chain oracle – Chainlink. However, the library assumes that the retrieved data is up-to-date and never checks its timestamp.

How to implement:
Consider checking the *updatedAt* value returned by the *latestRoundData* function (4th returned value) to make sure the on-chain oracle is not outdated.

5.11. Optimize the gas consumption

Retest 2021-05-14:
Partially implemented

Most of the proposed changes have been implemented. Only the following were not included:

- Removing unused code (see 5.8).
- Using public constants.

Description:
A few changes are proposed to optimize gas consumption.

How to implement:
Consider the following changes:

- Change public functions to external if they are not called by the contracts:
  - *UnboundDollar*
    - *approve*
    - *increaseAllowance*
    - *decreaseAllowance*
    - *changeSafuShare*
    - *changeStakeShare*
    - *changeStaking*
    - *changeSafuFund*
    - *changeDevFund*
    - *changeValuator*
    - *setOwner*
    - *claimOwner*
    - *claimTokens*
  - *Valuing_01*
    - *addLLC*
    - *changeLoanRate*
    - *changeFeeRate*
    - *disableLLC*
    - *setOwner*
    - *claimOwner*
    - *claimTokens*
- Remove some fragments in the code:
  - *UnboundDollar*
    - Function *_burn*

      ```
      require(_balances[account] >= toBurn, "Insufficient UND to pay
      back loan");
      ```

      This can be removed, because this requirement is later checked in:

      ```
      _balances[account] = _balances[account].sub(toBurn, "ERC20:
      burn amount exceeds balance");
      ```
- Store frequently changed values in uint256 instead of smaller slots. Storing values in *uint8* takes more gas than storing in *uint256*, because when updating such variables EVM must firstly load whole slot and then update it.

- Remove unused contracts, libraries and code to decrease the size of contracts.
    - See 5.8 Remove unused contracts and libraries and 5.9 Remove unused code.
- Use public constant variables instead of private ones and getters.
    - Example of fields definition:

    ```
    string constant public name = "Unbound Finance";
    string constant public symbol = "UND";
    uint8 constant public decimals = 18;
    ```

    - Some functions are required by the interfaces and must not be removed, therefore the *_totalSupply* private field cannot be removed as well.

    ```
    function totalSupply() public override view returns (uint256) {
        return _totalSupply;
    }
    ```

- Remove *minTokenAmount* parameter from *UnboundDollar _mint* function and check its requirement in *Valuing_01* contract's *unboundCreate* function.
  Add the following require statement to *Valuing_01* contract in line 98:

  ```
  require(minTokenAmount <= loanAmt.sub(feeAmt), "UND: minting less tokens
  than minimum amount");
  ```

  If the requirement is not met, there is not call to *unboundContract*.
- Do not repeat code. The function *unlockLPT* in *LiquidityLockContract* contains the same code in both branches of *if* statement (line 61). It can be changed to:

  ```
  // check if repayment is partial or full
  uint256 LPTokenToReturn;
  if (currentLoan == uTokenAmt) {
      LPTokenToReturn = _tokensLocked[msg.sender];
  } else {
      LPTokenToReturn = getLPTokensToReturn(currentLoan, uTokenAmt);
  }

  // Burning of uToken will happen first
  valuingContract.unboundRemove(uTokenAmt, msg.sender);

  // update mapping
  _tokensLocked[msg.sender] =
  _tokensLocked[msg.sender].sub(LPTokenToReturn);

  // send LP tokens back to user
  require(LPTContract.transfer(msg.sender, LPTokenToReturn), "LLC: Transfer
  Failed");

  // emit unlockLPT event
  emit UnlockLPT(_tokensLocked[msg.sender], msg.sender);
  ```

6. SECURITY CHECKLIST FOR LIQUIDITY POOL TOKENS

The following list includes security requirements that must and should be fulfilled by the Liquidity Pool Tokens considered to be included in the UnboundFinance project.

The Liquidity Pool Tokens **must not**:

- Implement functions that change reserves, users' balances and total supply other than those which add and remove liquidity.

The Liquidity Pool Tokens **must**:

- Have an on-chain price oracle in Chainlink (used by the UnboundFinance) – or its equivalent.
- Use base assets with at least 2 decimals.
- Implement the following functions according to the standard:
    - transferFrom
    - transfer
    - getReserves
    - totalSupply
    - token0
    - token1
    - balanceOf

The Liquidity Pool Tokens **should not**:

- Allow external calls in the implementation of the following functions (e.g. in ERC777):
    - transferFrom
    - transfer

The Liquidity Pool Tokens **should**:

- Implement *permit* function (according to the standard).
- Verify the signature malleability in *permit* function.
- Check whether the owner address is not zero, because *ecrecover* fails silently to 0.

7. SCSVS

The application has been verified for compliance with the Smart Contracts Security Verification Standard (SCSVS), v. 1.1.

Legend:

- Passed – Smart contracts meet the requirement.
- Failed – Smart contracts do not meet the requirements.
- N/A – The requirement does not apply to system (e.g. the system does not include the feature to which the requirement applies).

The results of SCSVS compliance verification are attached to the report in the *SCSVS-UnboundFinance.xlsx* file.

8. AUTOMATIC TOOLS

The tests included use of automatic tools such as: Mythril, Slither and Echidna.

8.1. Slither

Slither is a static analysis tool for smart contracts. The presented list includes all findings with the comment whether they are false positives (have no impact on risk) or are addressed in vulnerabilities or recommendations.

8.1.1. Array length with a user-controlled value

```
LiquidityLockContract
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#39-535) contract sets
array length with a user-controlled value:
     - tokenFeedDecimals.push(OracleLibrary.getDecimals(priceFeedAddress[i]))
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#194)
```

Comment: False positive.

8.1.2. Multiplication on the result of a division

```
LiquidityLockContract.getLPTokensToReturn(uint256,uint256)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#395-468) performs a
multiplication on the result of a division:
     -valueOfSingleLPT = poolValue.mul(10 ** 18).div(totalLP)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#448)
     -CRNow = (valueOfSingleLPT.mul(_tokensLocked[msg.sender])).div(_currentLoan)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#451)
LiquidityLockContract.getLPTokensToReturn(uint256,uint256)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#395-468) performs a
multiplication on the result of a division:
     -valueOfSingleLPT = poolValue.mul(10 ** 18).div(totalLP)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#448)
     -valueStart = valueOfSingleLPT.mul(_tokensLocked[msg.sender])
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#458)
LiquidityLockContract.getLPTokensToReturn(uint256,uint256)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#395-468) performs a
multiplication on the result of a division:
     -valueAfter = CREnd.mul(loanAfter).div(CRNorm).mul(10 ** 18)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#463)
Valuing_01.unboundCreate(uint256,address,uint256)
(contracts/Valuators/valuing_01.sol#75-101) performs a multiplication on the
result of a division:
     -loanAmt = amount.mul(listOfLLC[msg.sender].loanRate).div(rateBalance)
(contracts/Valuators/valuing_01.sol#88)
     -require(bool,string)(loanAmt.mul(listOfLLC[msg.sender].feeRate) >=
rateBalance,amount is too small) (contracts/Valuators/valuing_01.sol#95)
Valuing_01.unboundCreate(uint256,address,uint256)
(contracts/Valuators/valuing_01.sol#75-101) performs a multiplication on the
result of a division:
     -loanAmt = amount.mul(listOfLLC[msg.sender].loanRate).div(rateBalance)
(contracts/Valuators/valuing_01.sol#88)
     -feeAmt = loanAmt.mul(listOfLLC[msg.sender].feeRate).div(rateBalance)
(contracts/Valuators/valuing_01.sol#96)
```

Comment: False positive.

8.1.3. Reentrancy

```
LiquidityLockContract.lockLPT(uint256,uint256)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#245-259)

LiquidityLockContract.lockLPTWithPermit(uint256,uint256,uint8,bytes32,bytes32,uin
t256) (contracts/LiquidityLockContracts/LiquidityLockContract.sol#221-242)

LiquidityLockContract.unlockLPT(uint256)
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#350-393)
```

Comment: Some of the findings are false positives (re-entrancy by trusted contracts) and other were addressed in 4.2.

8.1.4. Assembly code

```
ddress.isContract(address)
(node_modules/@openzeppelin/contracts/utils/Address.sol#26-35) uses assembly
     - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#33)
Address._functionCallWithValue(address,bytes,uint256,string)
(node_modules/@openzeppelin/contracts/utils/Address.sol#119-140) uses assembly
     - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#132-
135)
```

Comment: Code is safe, but is unused and thus should be removed.

8.1.5. Different versions of Solidity

Different versions of Solidity is used in :

```
     - Version used: ['0.7.5', '>=0.4.23<0.8.0', '>=0.6.0', '^0.7.0']
     - ^0.7.0 (node_modules/@openzeppelin/contracts/math/SafeMath.sol#3)
     - ^0.7.0 (node_modules/@openzeppelin/contracts/utils/Address.sol#3)
     - >=0.4.23<0.8.0 (contracts/Interfaces/IERC20.sol#2)
     - >=0.4.23<0.8.0 (contracts/Interfaces/IUnboundToken.sol#2)
     - >=0.4.23<0.8.0 (contracts/Interfaces/IUniswapV2Pair.sol#2)
     - >=0.4.23<0.8.0 (contracts/Interfaces/IValuing_01.sol#2)
     - >=0.6.0 (contracts/Interfaces/chainlinkOracleInterface.sol#2)
     - 0.7.5 (contracts/LiquidityLockContracts/LiquidityLockContract.sol#2)
     - 0.7.5 (contracts/LiquidityLockContracts/OracleLibrary.sol#2)
     - 0.7.5 (contracts/UnboundTokens/unboundDollar.sol#2)
     - 0.7.5 (contracts/Valuators/valuing_01.sol#2)
```

Comment: Consider using one Solidity version.

8.1.6. Version too recent to be trusted

```
Pragma version^0.7.0 (node_modules/@openzeppelin/contracts/math/SafeMath.sol#3)
necessitates a version too recent to be trusted. Consider deploying with 0.6.11
```

```
Pragma version^0.7.0 (node_modules/@openzeppelin/contracts/utils/Address.sol#3)
necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version>=0.4.23<0.8.0 (contracts/Interfaces/IERC20.sol#2) is too complex
Pragma version>=0.4.23<0.8.0 (contracts/Interfaces/IUnboundToken.sol#2) is too
complex
Pragma version>=0.4.23<0.8.0 (contracts/Interfaces/IUniswapV2Pair.sol#2) is too
complex
Pragma version>=0.4.23<0.8.0 (contracts/Interfaces/IValuing_01.sol#2) is too
complex
Pragma version>=0.6.0 (contracts/Interfaces/chainlinkOracleInterface.sol#2)
allows old versions
Pragma version0.7.5
(contracts/LiquidityLockContracts/LiquidityLockContract.sol#2) necessitates a
version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.5 (contracts/LiquidityLockContracts/OracleLibrary.sol#2)
necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.5 is not recommended for deployment
Parameter OracleLibrary.getPriceFeeds(bool,address[])._triangulatePriceFeed
(contracts/LiquidityLockContracts/OracleLibrary.sol#84) is not in mixedCase
Parameter OracleLibrary.getPriceFeeds(bool,address[])._addresses
(contracts/LiquidityLockContracts/OracleLibrary.sol#84) is not in mixedCase
Parameter
OracleLibrary.checkBaseAssetPrices(bool,uint8,address[])._triangulateBaseAsset
(contracts/LiquidityLockContracts/OracleLibrary.sol#94) is not in mixedCase
Parameter
OracleLibrary.checkBaseAssetPrices(bool,uint8,address[])._maxPercentDiffBaseAsset
(contracts/LiquidityLockContracts/OracleLibrary.sol#95) is not in mixedCase
Parameter OracleLibrary.checkBaseAssetPrices(bool,uint8,address[])._addresses
(contracts/LiquidityLockContracts/OracleLibrary.sol#96) is not in mixedCase
```

Slither's recommendation:

Deploy with any of the following Solidity versions:

- 0.5.11 - 0.5.13,
- 0.5.15 - 0.5.17,
- 0.6.8,
- 0.6.10 - 0.6.11. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

8.1.7. Low level call

```
Low level call in Address.sendValue(address,uint256)
(node_modules/@openzeppelin/contracts/utils/Address.sol#53-59):
     - (success) = recipient.call{value: amount}()
(node_modules/@openzeppelin/contracts/utils/Address.sol#57)
Low level call in Address._functionCallWithValue(address,bytes,uint256,string)
(node_modules/@openzeppelin/contracts/utils/Address.sol#119-140):
     - (success,returndata) = target.call{value: weiValue}(data)
(node_modules/@openzeppelin/contracts/utils/Address.sol#123)
```

Comment: Code is safe, but is unused and thus should be removed.

8.1.8. Conformance to Solidity naming conventions

There are multiple places where non-mixedCase naming is used. Here are some of the examples:

```
Contract IERC20_2 (contracts/Interfaces/IERC20.sol#4-19) is not in CapWords
Function IUnboundToken._mint(address,uint256,uint256,address,uint256)
(contracts/Interfaces/IUnboundToken.sol#5) is not in mixedCase
Parameter IUnboundToken._mint(address,uint256,uint256,address,uint256).LLCAddr
(contracts/Interfaces/IUnboundToken.sol#5) is not in mixedCase
Function IUnboundToken._burn(address,uint256,address)
(contracts/Interfaces/IUnboundToken.sol#6) is not in mixedCase
Parameter IUnboundToken._burn(address,uint256,address).LLCAddr
(contracts/Interfaces/IUnboundToken.sol#6) is not in mixedCase
Contract IUniswapV2Pair_0 (contracts/Interfaces/IUniswapV2Pair.sol#4-53) is not
in CapWords
Function IUniswapV2Pair_0.DOMAIN_SEPARATOR()
(contracts/Interfaces/IUniswapV2Pair.sol#19) is not in mixedCase
Function IUniswapV2Pair_0.PERMIT_TYPEHASH()
(contracts/Interfaces/IUniswapV2Pair.sol#20) is not in mixedCase
Function IUniswapV2Pair_0.MINIMUM_LIQUIDITY()
(contracts/Interfaces/IUniswapV2Pair.sol#37) is not in mixedCase
Contract IValuing_01 (contracts/Interfaces/IValuing_01.sol#4-15) is not in
CapWords
Function UnboundDollar._mint(address,uint256,uint256,address,uint256)
(contracts/UnboundTokens/unboundDollar.sol#271-296) is not in mixedCase
Parameter UnboundDollar._mint(address,uint256,uint256,address,uint256).LLCAddr
(contracts/UnboundTokens/unboundDollar.sol#275) is not in mixedCase
Function UnboundDollar._burn(address,uint256,address)
(contracts/UnboundTokens/unboundDollar.sol#299-321) is not in mixedCase
Parameter UnboundDollar._burn(address,uint256,address).LLCAddr
(contracts/UnboundTokens/unboundDollar.sol#302) is not in mixedCase
Parameter UnboundDollar.setOwner(address)._newOwner
(contracts/UnboundTokens/unboundDollar.sol#400) is not in mixedCase
Parameter UnboundDollar.claimTokens(address,address)._tokenAddr
(contracts/UnboundTokens/unboundDollar.sol#417) is not in mixedCase
Variable UnboundDollar.DOMAIN_SEPARATOR
(contracts/UnboundTokens/unboundDollar.sol#58) is not in mixedCase
Variable UnboundDollar._owner (contracts/UnboundTokens/unboundDollar.sol#81) is
not in mixedCase
Variable UnboundDollar._valuator (contracts/UnboundTokens/unboundDollar.sol#88)
is not in mixedCase
```

Comment: Some reported places are false positives (e.g. DOMAIN_SEPARATOR). Other were addressed in 5.3.

8.1.9. Public function that could be declared external

There are multiple *public* functions that are never called by the contract should be declared *external* to save gas.
Comment: Addressed in 5.11.

8.1.10. Missing zero address validation

There are multiple functions that do not require non-zero address.
Comment: Addressed in 4.5.

8.1.11. Dangerous usage of block.timestamp

```
Dangerous comparisons:
     - require(bool,string)(deadline >= block.timestamp,UnboundDollar: EXPIRED)
(contracts/UnboundTokens/unboundDollar.sol#185)
```

Comment: False positive.

8.2. Mythril

Mythril is a security analysis tool that uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. The presented list includes all findings with the comment whether they are false positives (have no impact on risk) or are addressed in vulnerabilities or recommendations.

8.2.1. Setup

The contracts were simplified to speed up the analysis. The following actions were included during simplifications:

- Removed function covering permit functionality.
- Removed owner's functions that set parameters.
- Removed well-known IERC20 functions (allowance, approve).

8.2.2. External Call To User-Supplied Address

```
Contract: Valuing_01
Function name: claimTokens(address,address)
In file: ../contracts/simplified/valuing_01.sol:183

IERC20(_tokenAddr).transfer(to, tokenBal)
```

```
Contract: Valuing_01
Function name: claimTokens(address,address)
In file: ../contracts/simplified/valuing_01.sol:111

unboundContract._burn(user, toUnlock, msg.sender)
```

```
Contract: Valuing_01
Function name: claimTokens(address,address)
In file: ../contracts/simplified/valuing_01.sol:100

unboundContract._mint(user, loanAmt, feeAmt, msg.sender, minTokenAmount)
```

Comment: False positive.

8.2.3. Multiple Calls in a Single Transaction

```
Function name: claimTokens(address,address)
In file: ../contracts/simplified/valuing_01.sol:183

IERC20(_tokenAddr).transfer(to, tokenBal)
```

Comment: False positive.

8.3. Echidna

Echidna is a Haskell program designed for fuzzing/property-based testing of Ethereum smarts contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions.

8.3.1. Setup

The Echidna tool requires to define and implement the test scenarios to be fuzzed. The following scenarios have been defined:

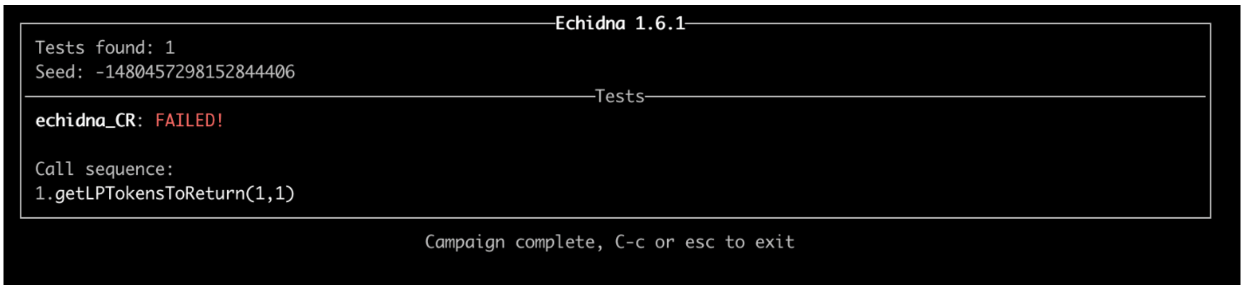
- Unlocking LPT with lower Collateralization Ratio (*LiquidityLockContract*)
- Incorrect fees distribution and clearance (*UnboundDollar*)
- Zero fee for locking LPT (*Valuing*_01)

8.3.2. Unlocking LPT with lower Collateralization Ratio

The scenario defined CR, tokens locked (LPTs), loan taken (uTokens), first token reserve and price as constant and fuzzed *getLPTokensToReturn* function.
The amount of uToken to unlock has been defined as variable as user can manipulate it, but it cannot be greater than the loan taken (user parameter _*uTokenAmt*).
The second token reserve has been defined as variable to reflect the price change (market parameter _*token1*).
The function declaration has been changed to:

```
function getLPTokensToReturn(uint256 _uTokenAmt, uint112 _token1) public returns
(uint256 _LPTokenToReturn)
```

Result:



```
Echidna 1.6.1
Tests found: 1
Seed: -1480457298152844406
Tests
echidna_CR: FAILED!

Call sequence:
1.getLPTokensToReturn(1,1)

Campaign complete, C-c or esc to exit
```

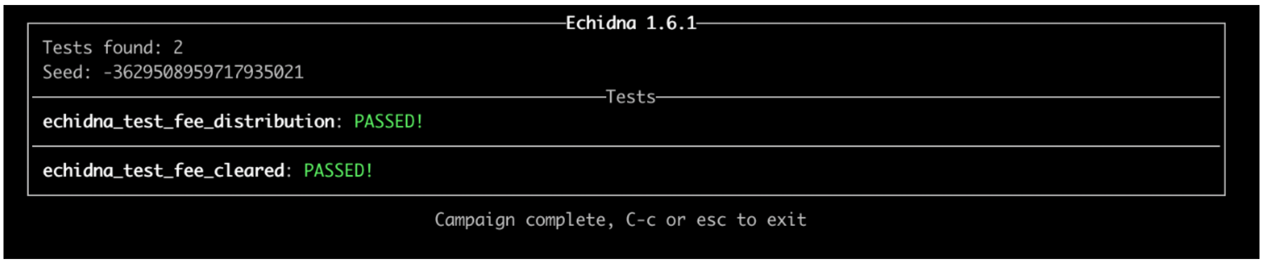
Comment: Addressed in 4.1.

8.3.3. Incorrect fees distribution and clearance

The scenario verified whether the function *distributeFee* does not mint additional of burn existing tokens and whether the fee to be distributed is correctly zeroed.
The amount of collected fee to be distributed has been defined as variable to reflect the current fee (market parameter *_storedFee*).
The function declaration has been changed to:

```
function distributeFee(uint256 _storedFee) public returns (bool)
```

Result:



```
Echidna 1.6.1
Tests found: 2
Seed: -3629508959717935021
Tests
echidna_test_fee_distribution: PASSED!
echidna_test_fee_cleared: PASSED!
Campaign complete, C-c or esc to exit
```

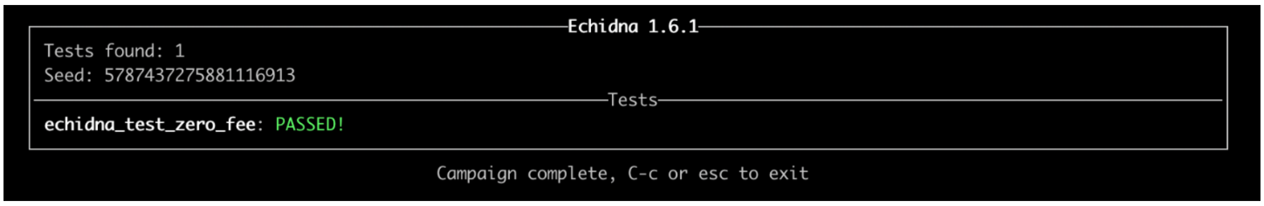
Comment: Passed.

8.3.4. Zero fee for locking LPT

The scenario verified whether the function *unboundCreate* allows to create a loan with zero fee.
The amount to be locked has been defined as variable as user can manipulate it (user parameter *_amount*).
The function declaration has been changed to:

```
function unboundCreate(uint256 amount, address user, uint256 minTokenAmount)
public
```

Result:



```
Echidna 1.6.1
Tests found: 1
Seed: 5787437275881116913
Tests
echidna_test_zero_fee: PASSED!
Campaign complete, C-c or esc to exit
```

Comment: Passed.

9. 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.

10. CONTACT

Person responsible for providing explanations:

Damian Rusinek
e-mail: Damian.Rusinek@securing.pl
tel.: +48 12 425 25 75
mob.: +48 502 118 491

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