BLOCK HUNTERS

Audit report for LoanSnap

19/10/2021

bHOME audit report

1. Executive summary

The following audit report presents the effect of the research that Blockhunters team conducted on bHOME smart contracts (ver. 0.2 according to the documentation). The research was conducted in an audit process from 09/09/2021 till 18/10/2021 by the Blockhunters team based on the code delivered by LoanSnap.

Our audit focused on verifying the ver. 0.2 mechanisms – PropertyToken, DevUSDC and upgradeable Pools. We've conducted penetration tests and validated compliance with the documentation.

Blockhunters team has checked the possibility of known Ethereum attacks to be exploited in the code. Fortunately, smart contracts contain basic and well-designed functionalities that are not vulnerable to known Ethereum attacks from SWC Registry. LoanSnap developers have implemented really good practices in the code, including using Open-Zeppelin and SafeMath libraries that significantly lower the risk of possible miscalculations and errors. All of the contracts, methods and state variables were tested and none of them poses any threat to the contract safety.

We have verified the correctness of Interest Rate calculation as one of the most important variables used in the code which serves as a basis for other variables. No issues were found in this case. Blockhunters auditors have found 2 minor vulnerabilities in Pool functions, which serve as a source of information for external calls and therefore do not pose any threat to the smart contracts themselves in the current form.

A comprehensive suite of unit tests was written for this project and is available as an attachment to this report.

To sum up, we are happy to say that the ver. 0.2 of bHOME smart contract suite is safe and can be used in the market and for further development of the mechanism.

For the sake of clarity we introduced the following issues symbols:

✓ works fine!

● works fine although modifications are recommended

✗ major vulnerability (can lead to tokens theft or network failure)

The following, clickable table of contents represents a list of all the issues found.

1.1. Liability clause

Please note that Blockhunters Company doesn’t verify the economic foundation of the project but only its code correctness and security issues. We do not take any responsibility for any misuse or misunderstanding of the information provided and potential economic losses due to faulty investment decisions. This document doesn’t ensure that the code itself is free from potential vulnerabilities that were not found. If any questions arise please contact us by www.blockhunters.io.

1.2. Commit hash and MD5 hashes

Before using the smart contracts, please verify MD5 commit hashes with the following ones, which describe the files that were audited between 09/09/2021 and 18/10/2021.

Commit	a8fda9f596d233d57c575e797f35256ad42b858d

Filename	MD5
Pool0	6e8b2cfc6439cb658ed94e42fccb9063
Pool1	a67450b473f0a9fdd8f81589dd370b0d
DevUSDC	650993890ef8251ac3c93d3399c160fc
PropToken0	74d3b428d491092f114e2f0e51d32a23

2. Table of contents

3. Security audit

3.1. Errors known from Ethereum

✓ Reentrancy attack

Non-susceptible. The contracts adhere to ERC20, ERC777 and ERC721 protocol and use OpenZeppelin standards where possible.

✓ Race conditions

Flow of the system is linear and straightforward. Nothing time-sensitive and requiring synchronicity is performed.

✓ Integer over / underflow

Contracts use the newest solc version where SafeMath library is built-in, which prevents this class of errors.

✓ Timestamps

Custom logic dependent on block.timestamp is a source of many leaks as it can be influenced by the miners. The contract is safe from any such attacks.

✓ Library dependencies

All used dependencies are in the source files.

✓ Front-running

Front running isn’t a risk for integrity of the system with its current capabilities. Foreseeing transactions before visible in the block won’t have any bad results for the users.

✓ DoS

Neither of the contracts can be rendered inoperable by the users

✓ Insufficient gas griefing

Non-susceptible. The contracts don’t use any low-level contract calls, thus this error won’t occur. This attack may be possible on a contract which accepts generic data and uses it to make a call another contract (a 'sub-call') via the low-level address.call() function, as is often the case with multi-signature and transaction relay contracts.

✓ Token deposit and creation

Asset flow follows the specification models and the logic is well tested for integration external smart contracts

3.2. Automated tools

✓ Mythril

- Version number: v0.22.21
- Performed by: AK
- Date, time: 3.10.2021
- Results: No vulnerability detected

- ✓ Slither

- Version number: 0.7.1
- Performed by: PP
- Date, time: 1.10.2021
- Results: No vulnerability detected

4. Business logic audit

4.1. Workflow mechanisms

- ✓ Verification of the governance mechanism based on ERC777 standard, including proposals, voting and distribution among the Lenders.

Governance mechanisms for lenders are not implemented yet. Pool tokens are distributed among Lenders properly.

- ✓ Analysing Pool creation mechanism with support for stablecoins and other coins.

Pool creation mechanism is safe. Contract is upgrading as intended. The use of DevUSDC has been tested and no issues were found.

- ✓ Verification of Lender – HCPool exchange mechanism with capital delivery and locking.

Locking of the capital is not implemented yet. HCPool exchange mechanism is tested and working properly.

✓ Examining the borrowing mechanism – PropertyToken creation and dynamic interest rate for pools.

Due to the linear flow of the program, the mechanism of taking a loan is safe. There is no possibility of a reentrancy attack. Interest is calculated correctly, which has been tested over a long period of time. The *getInterestRate* function returns the correct result for the given size of the potential loan.

✓ Running through PropertyToken mechanisms – Registry management and storage, ownership and Pool upgrading.

Property Tokens used in Pool1 are working correctly. Their use does not create an opportunity to attack.

✓ Analysing loan repayment mechanism with emphasis on per-block interest rate and possible vulnerabilities / errors there.

Loan repayment mechanism is safe. The different repayment cases are well separated. Servicer will receive a fee with each loan repayment. The repay function does not create any opportunity for a reentrancy attack.

✓ Verification of repayment mechanisms, including voting and reclaiming value for the Lenders.

Redeem function successfully burns the sender's hcPool tokens and transfers the DevUSDC back to them. No vulnerabilities were found.

4.2. HomeDAO methods check

4.2.1. Pool0

Method	Status	Information
Pool0.initialize	OK	
Pool0.setApprovedAddress	OK	
Pool0.isApprovedAddress	OK	
Pool0.isApprovedServicer	OK	
Pool0.getContractData	OK	
Pool0.getPoolValueWithInterest	OK	
Pool0.getPoolBorrowed	OK	
Pool0.getSupplyableTokenAddress	OK	
Pool0.getServicerAddress	OK	
Pool0.getUserLoans	OK	
Pool0.getAllLoans	OK	
Pool0.getActiveLoans	OK	
Pool0.getLoanAccruedInterest	OK	
Pool0.getLoanDetails	OK	
Pool0.getAverageInterest	Minor	division by zero

Pool0.mintProportionalPoolTokens	OK	
Pool0.lend	OK	
Pool0.redeem	OK	
Pool0.getInterestRate	OK	
Pool0.borrow	OK	
Pool0.repay	OK	
Pool0.hasUpgradedFunction	Minor	should return false

4.2.2. Pool1

Method	Status	Information
Pool1.initialize	OK	
Pool1.setApprovedAddress	OK	
Pool1.isApprovedAddress	OK	
Pool1.isApprovedServicer	OK	
Pool1.getContractData	OK	
Pool1.getPoolValueWithInterest	OK	
Pool1.getPoolBorrowed	OK	
Pool1.getSupplyableTokenAddress	OK	

Pool1.getServicerAddress	OK	
Pool1.getUserLoans	OK	
Pool1.getAllLoans	OK	
Pool1.getActiveLoans	OK	
Pool1.getLoanAccruedInterest	OK	
Pool1.getLoanDetails	OK	
Pool1.getAverageInterest	Minor	division by zero
Pool1.mintProportionalPoolTokens	OK	
Pool1.lend	OK	
Pool1.redeem	OK	
Pool1.getInterestRate	OK	
Pool1.borrow	OK	
Pool1.repay	OK	
Pool1.hasUpgradedFunction	OK	
Pool1.onERC721Received	OK	

4.2.3. DevUSDC

Method	Status	Information
BUSDC.constructor	OK	

4.2.4. PropToken0

Method	Status	Information
PropToken0.initialize	OK	
PropToken0.isApprovedServicer	OK	
PropToken0.getLienValue	OK	
PropToken0.getPropTokenCount	OK	
PropToken0.getPoolAddress	OK	
PropToken0.getPropTokenData	OK	
PropToken0.mintPropToken	OK	

4.3. interestAccrued test

This variable is one of the most important source of information for the whole smart contract suite to operate smoothly. Therefore we have tested it’s value for further steps in time.

Formula for calculating the interest:

```
Loan = 1000000000000000
loan.interest = 2000000
numberOfSecondsInADay = 86400
interestPerSecond = (principal * loan.interest) / (1000000 * 31622400)
interestPerDay = interestPerSecond * numberOfSecondsInADay

interest accrued up to the Nth day = interestPerDay * N
```

	interest accrued	
day	computed by the smart contract	computed from the formula
0	0	0
1	5464480838400	5464480838400
2	10928961676800	10928961676800
3	16393442515200	16393442515200
4	21857923353600	21857923353600
…		
50	273224041920000	273224041920000
51	278688522758400	278688522758400
52	284153003596800	284153003596800
53	289617484435200	289617484435200
54	295081965273600	295081965273600
55	300546446112000	300546446112000
…		
95	519125679648000	519125679648000
96	524590160486400	524590160486400
97	530054641324800	530054641324800
98	535519122163200	535519122163200
99	540983603001600	540983603001600
100	546448083840000	546448083840000

…		
996	5442622915046400	5442622915046400
997	5448087395884800	5448087395884800
998	5453551876723200	5453551876723200
999	5459016357561600	5459016357561600
1000	5464480838400000	5464480838400000

5. Suggestions

contract: Pool1

"// contracts/Pool0.sol" should be replaced with "// contracts/Pool1.sol "

```
// contracts/Pool0.sol
// SPDX-License-Identifier: MIT
```

contracts: Pool0, Pool1

functions: *getPoolValueWithInterest*, *getPoolBorrowed*

These functions have wrong descriptions:

```
/**
   *   @dev Function getPoolValueWithInterest() returns the contract address of
ERC20 this pool accepts (ususally USDC)
   */
   function getPoolValueWithInterest() public view returns (uint256) {
       uint256 totalWithInterest = poolLent;

       for (uint i=0; i<loans.length; i++) {
           totalWithInterest = totalWithInterest.add(getLoanAccruedInterest(i));
       }

       return totalWithInterest;
   }

   /**
   *   @dev Function getPoolBorrowed() returns the contract address of ERC20 this
pool accepts (ususally USDC)
   */
   function getPoolBorrowed() public view returns (uint256) {
       return poolBorrowed;
   }

   /**
   *   @dev Function getSupplyableTokenAddress() returns the contract address of
ERC20 this pool accepts (ususally USDC)
   */
   function getSupplyableTokenAddress() public view returns (address) {
       return ERCAddress;
   }
```

contract: Pool1
function: *onERC721Received*
Documentation for this function is missing.

```
   function onERC721Received(
       address,
       address,
       uint256,
       bytes memory
   ) public pure override returns (bytes4) {
       return this.onERC721Received.selector;
   }
```

contract: Pool0
function: *hasUpgradedFunction*
Pool0 has not been upgraded yet. *hasUpgradedFunction* should return *false*.

```
/**
   *   @dev Function hasUpgradedFunction returns bool depending on if contract has
been upgraded or not
   */
   function hasUpgradedFunction() public pure returns (bool) {
       return true;
   }
```

contracts: Pool0, Pool1
function: *borrow*
Documentation states that *borrow* returns the loan ID and fixed Interest Rate, but the function returns nothing.

```
/**
   *   @dev Function borrow creates a new Loan, moves the USDC to Borrower, and
returns the loan ID and fixed Interest Rate
   *   - Also creates an origination fee for the Servicer in HC_Pool
   *   @param amount The size of the potential loan in (probably usdc).
   *   @param maxRate The size of the potential loan in (probably usdc).
   *   EDITED in pool1 to also require a PropToken
   *   EDITED in pool1 - borrower param was removed and msg.sender is new recepient
of USDC
   */
   function borrow(uint256 amount, uint256 maxRate, uint256 propTokenId) public {
       //for v2 require this address is approved to transfer propToken
       require(PropToken0(propTokenContractAddress).getApproved(propTokenId)       ==
address(this), "pool is not approved to move propToken");
       //also require msg.sender is owner of token
       require(PropToken0(propTokenContractAddress).ownerOf(propTokenId)           ==
msg.sender, "msg.sender is not propToken owner");

       //check the requested interest rate is still available
       uint256 fixedInterestRate = uint256(getInterestRate(amount));
```

```
        require(fixedInterestRate <= maxRate, "interest rate no longer available");

        //require the propToken approved has a lien value less than or equal to the
requested loan size
        require(PropToken0(propTokenContractAddress).getLienValue(propTokenId)      >=
amount, "Loan amount too large for propToken value");

        //first take the propToken
        PropToken0(propTokenContractAddress).safeTransferFrom(msg.sender,
address(this), propTokenId);

        //create new Loan
        Loan memory newLoan = Loan(loanCount, msg.sender, fixedInterestRate, amount,
0, block.timestamp);
        loans.push(newLoan);
        userLoans[msg.sender].push(loanCount);

        //map new loanID to Token ID
        loanToPropToken[loanCount] = propTokenId;

        //update system variables
        loanCount = loanCount.add(1);
        poolBorrowed = poolBorrowed.add(amount);

        //finally move the USDC
        IERC20Upgradeable(ERCAddress).transfer(msg.sender, amount);

        //then mint HC_Pool for the servicer (fixed 1% origination is better than
standard 2.5%)
        mintProportionalPoolTokens(servicer, amount.div(100));
    }
```

contracts: Pool0, Pool1

function: *getAverageInterest*

Division by zero if *borrowedCounter* is equal to 0.

```
function getAverageInterest() public view returns (uint256) {
        uint256 sumOfRates = 0;
        uint256 borrowedCounter = 0;

       for (uint i = 0; i < loans.length; i++) {
           if(loans[i].principal != 0){
               sumOfRates =
sumOfRates.add(loans[i].interestRate.mul(loans[i].principal));
               borrowedCounter = borrowedCounter.add(loans[i].principal);
           }
       }
       return sumOfRates.div(borrowedCounter);
    }
```

contract: Pool0, Pool1

variable: *ERCAddress*

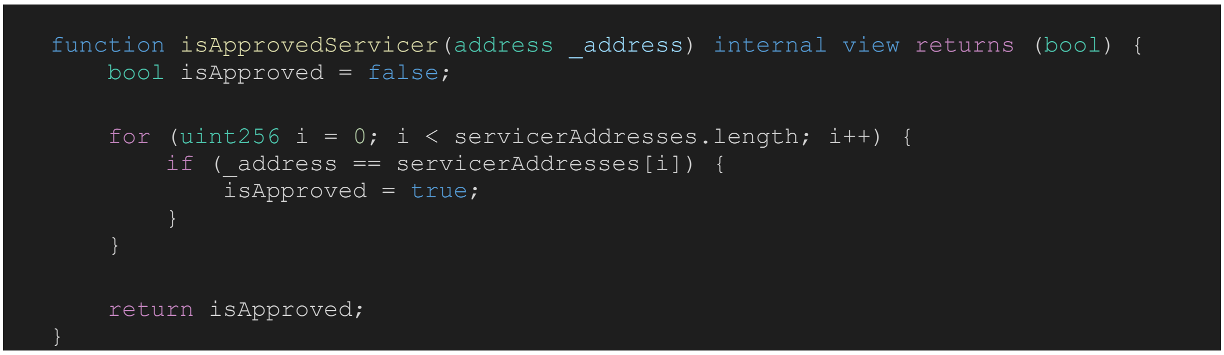
ERCAddress should be constant.

```
address ERCAddress;
```

contract: Pool1

function: *isApprovedServicer*

isApprovedServicer function is internal, never used and should be removed.

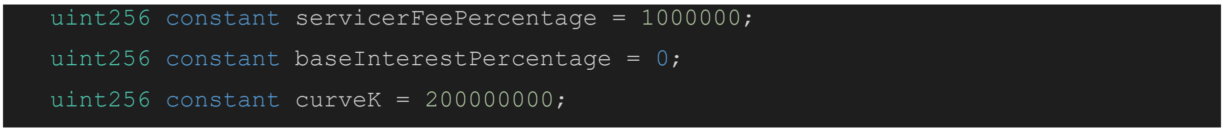


```
function isApprovedServicer(address _address) internal view returns (bool) {
    bool isApproved = false;

    for (uint256 i = 0; i < servicerAddresses.length; i++) {
        if (_address == servicerAddresses[i]) {
            isApproved = true;
        }
    }

    return isApproved;
}
```

contract: Pool0, Pool1

variables: *servicerFeePercentage*, *baseInterestPercentage*, *curveK*

These constant variables are not UPPER_CASE_WITH_UNDERSCORES.

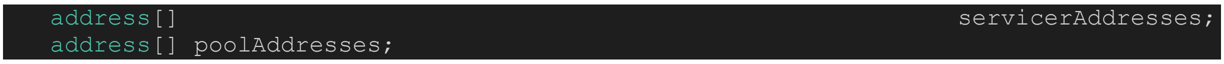


```
uint256 constant servicerFeePercentage = 1000000;
uint256 constant baseInterestPercentage = 0;
uint256 constant curveK = 200000000;
```

contract: PropToken0

variables: *servicerAddresses*, *poolAddresses*

servicerAddresses and *poolAddresses* are arrays, but they only store one value.

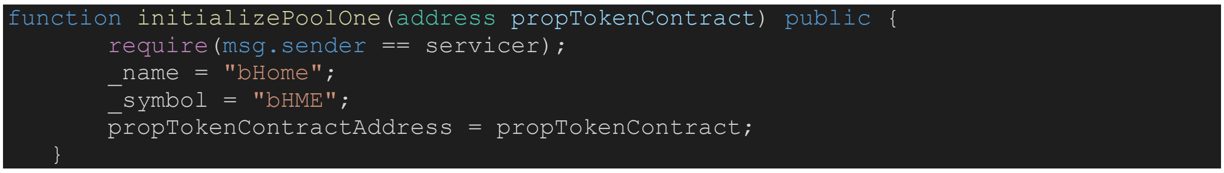


```
address[]                                                          servicerAddresses;
address[] poolAddresses;
```

contract: Pool1

function: *initializePoolOne*

This function should be able to be called only once.



```
function initializePoolOne(address propTokenContract) public {
        require(msg.sender == servicer);
        _name = "bHome";
        _symbol = "bHME";
        propTokenContractAddress = propTokenContract;
    }
```

Thank you!

Contact us at:

heyhunters@blockhunters.io

www.blockhunters.io