DApp Smart Contract's Audit

October 11, 2018

Initial Data:

Internet address of the provided source code: https://github.com/StayBitDev/RentalContracts

At this address there are 8 smart contracts: BaseEscrowLib.sol, DateTime.sol, FlexibleEscrowLib.sol, ModerateEscrowLib.sol, MyToken.sol, Ownable.sol, StayBitContractFactory.sol, StrictEscrowLib.sol.

Description of work: https://staybit.io/demo/Help.aspx

There is a working DApp in the Ropsten test network and in the Main network: https://staybit.io/demo/ and https://staybit.io/main/. The difference between them is that the test version allows you to simulate the current date and works with a test token. The version in the Main network works with the TrueUSD token and ignores the date simulation.

Brief description of the project:

In the area of short-term daily renting, there is a lot of deception: fake listings on free sites offer to transfer the entire amount in advance for the reservation of housing, and then the tenants come to settle in and find themselves with nothing. Landlords also suffer from frauds who reserve the day before they check in and pay with stolen credit cards or fake checks. This decentralized application allows using the Escrow contract on the Ethereum blockchain to financially secure the housing rental transactions.

The goal of the audit:

Verify that tenants and landlords can use this DApp without fear of losing their financial assets.

Create tests using the Truffle framework to test the functionality of the DApp.

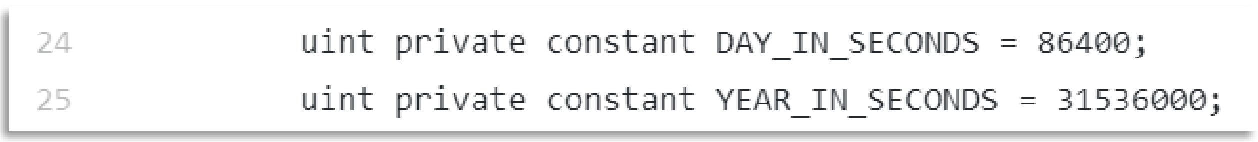
Source Code Analysis

Remarks:

1. The source code of smart contracts is not designed in code style. For this reason, it is hard to understand the logic of the functions, the use of variables and constants. The latest code style version is available on the Internet at: https://solidity.readthedocs.io/en/v0.4.25/style-guide.html

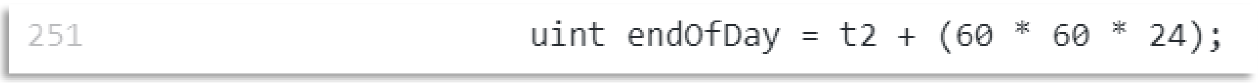
2. None of the provided smart contracts do not use the current version of Solidity – 0.4.25.

1. Contract “DateTime”



```
        uint private constant DAY_IN_SECONDS = 86400;
        uint private constant YEAR_IN_SECONDS = 31536000;
```

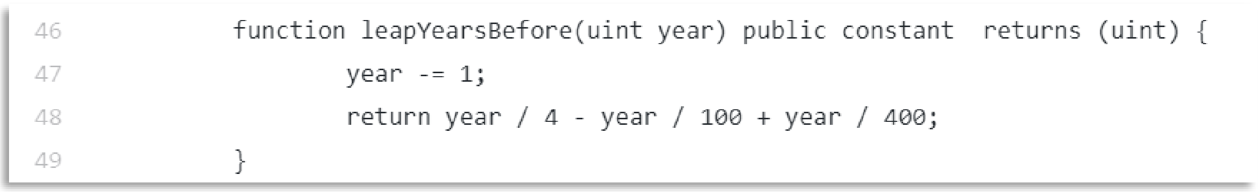
(Take note). In the Solidity language, there are already exist time constants. This code could be written differently, more clearly:
uint private constant DAY _IN_SECONDS = 1 days;
uint private constant YEAR_IN_SECONDS = 1 years;



```
                uint endOfDay = t2 + (60 * 60 * 24);
```

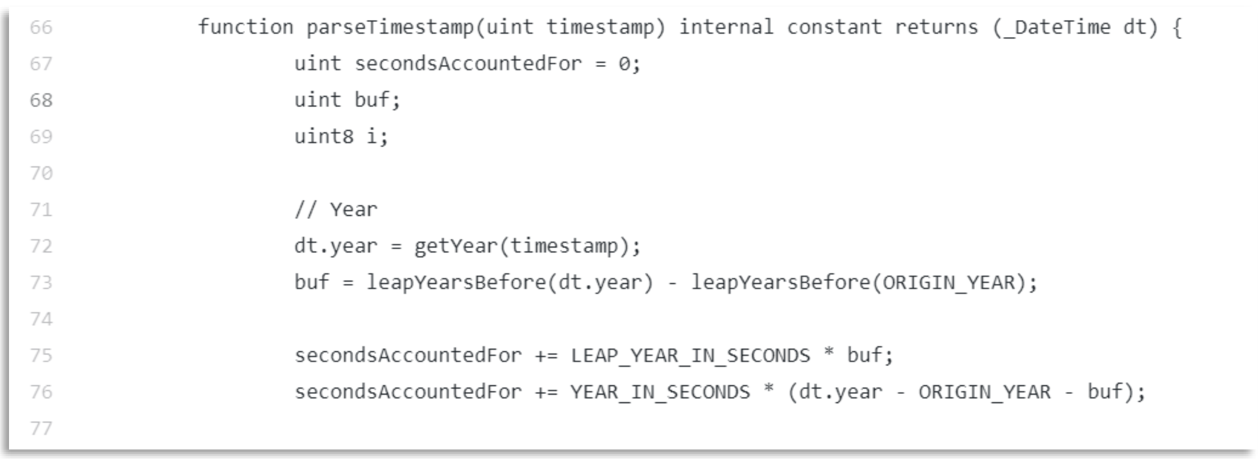
uint endOfDay = t2 + 1 days;

(Critical) There is no safe math in smart contract.
(Critical) The code in the following lines is not safe and its execution can lead to unpredictable consequences.



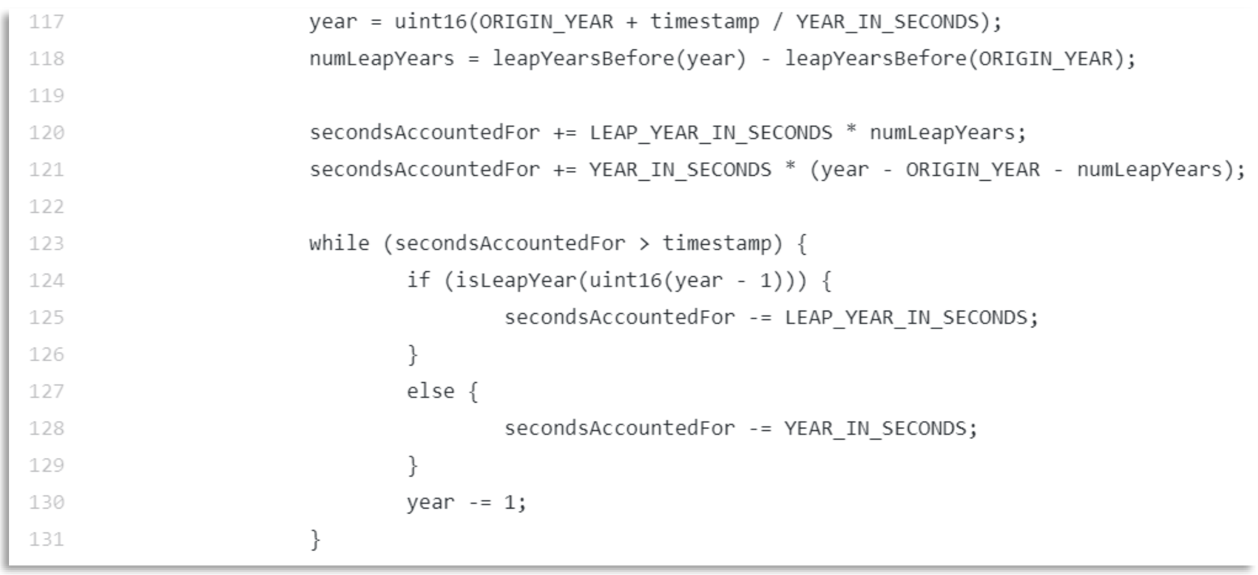
```
46          function leapYearsBefore(uint year) public constant  returns (uint) {
47                  year -= 1;
48                  return year / 4 - year / 100 + year / 400;
49          }
```

Lines 47, 48



```
66          function parseTimestamp(uint timestamp) internal constant returns (_DateTime dt) {
67                  uint secondsAccountedFor = 0;
68                  uint buf;
69                  uint8 i;
70
71                  // Year
72                  dt.year = getYear(timestamp);
73                  buf = leapYearsBefore(dt.year) - leapYearsBefore(ORIGIN_YEAR);
74
75                  secondsAccountedFor += LEAP_YEAR_IN_SECONDS * buf;
76                  secondsAccountedFor += YEAR_IN_SECONDS * (dt.year - ORIGIN_YEAR - buf);
77
```

Lines 73, 76

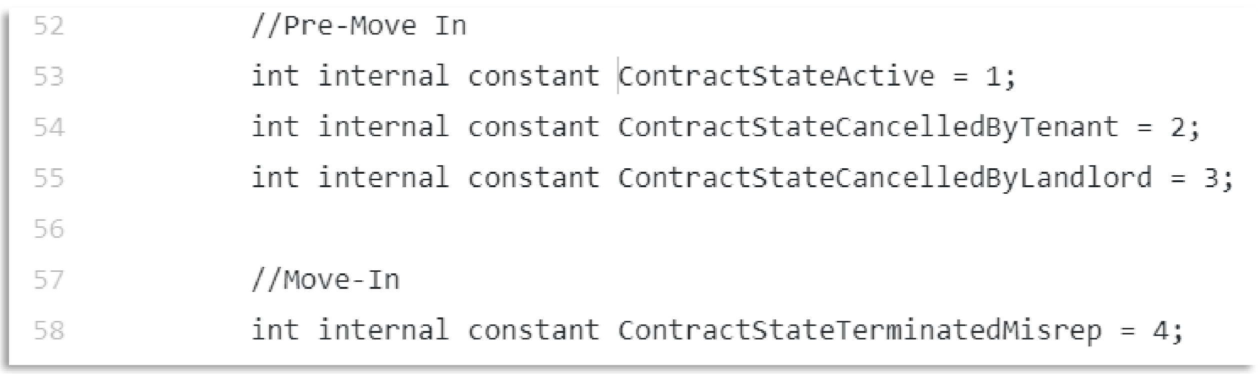


```
117                         year = uint16(ORIGIN_YEAR + timestamp / YEAR_IN_SECONDS);
118                         numLeapYears = leapYearsBefore(year) - leapYearsBefore(ORIGIN_YEAR);
119
120                         secondsAccountedFor += LEAP_YEAR_IN_SECONDS * numLeapYears;
121                         secondsAccountedFor += YEAR_IN_SECONDS * (year - ORIGIN_YEAR - numLeapYears);
122
123                         while (secondsAccountedFor > timestamp) {
124                                 if (isLeapYear(uint16(year - 1))) {
125                                         secondsAccountedFor -= LEAP_YEAR_IN_SECONDS;
126                                 }
127                                 else {
128                                         secondsAccountedFor -= YEAR_IN_SECONDS;
129                                 }
130                                 year -= 1;
131                         }
```

Lines 118, 121, 124, 125, 128, 130
To make it safer to use this code allows you to check for validity of the values used in these variable expressions.

2. <u>Contract “BaseEscrowLib”</u>

(Take note) Using “enum” data types will make the code more compact. Instead of this code:

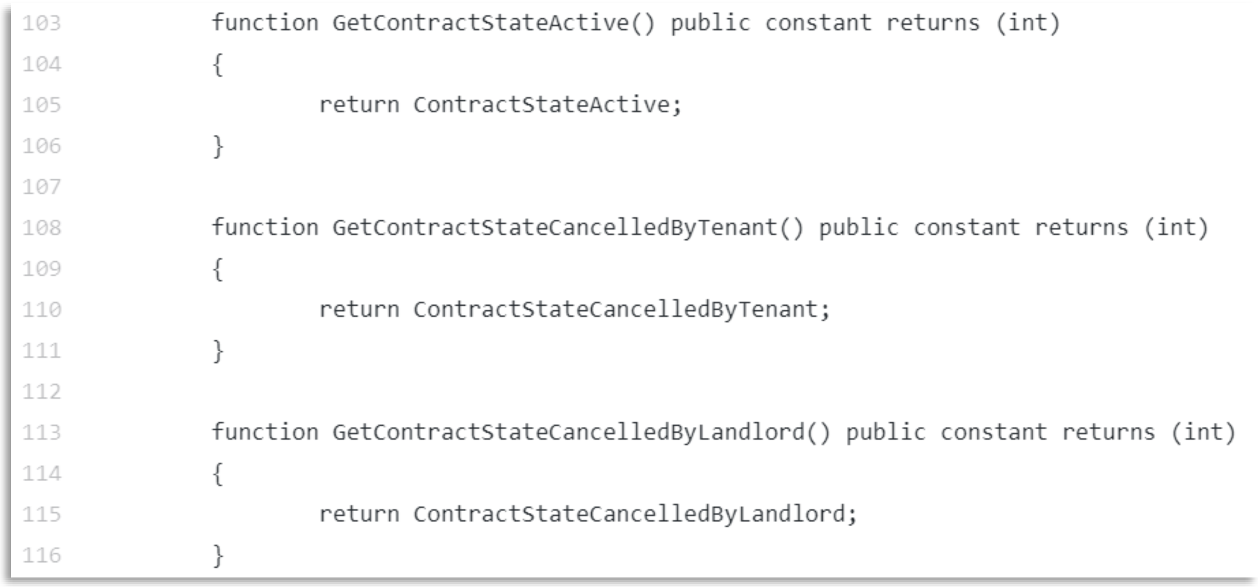


```
52          //Pre-Move In
53          int internal constant ContractStateActive = 1;
54          int internal constant ContractStateCancelledByTenant = 2;
55          int internal constant ContractStateCancelledByLandlord = 3;
56
57          //Move-In
58          int internal constant ContractStateTerminatedMisrep = 4;
```

It is possible to use this code:

enum ContractState {ACTIVE, CANCELLED_BY_TENANT, CANCELLED_BY_LANDLORD, TERMINATED_MISREP }

The descriptions of the functions were as follows:



```
        function GetContractStateActive() public constant returns (int)
        {
                return ContractStateActive;
        }

        function GetContractStateCancelledByTenant() public constant returns (int)
        {
                return ContractStateCancelledByTenant;
        }

        function GetContractStateCancelledByLandlord() public constant returns (int)
        {
                return ContractStateCancelledByLandlord;
        }
```

It is possible to shorten the source code. Instead of describing the call to GetContractState..() functions in 9 places, it is sufficient to describe the getter method only once.

It will be possible to replace this code in the BaseEscrowLib library with the following code:

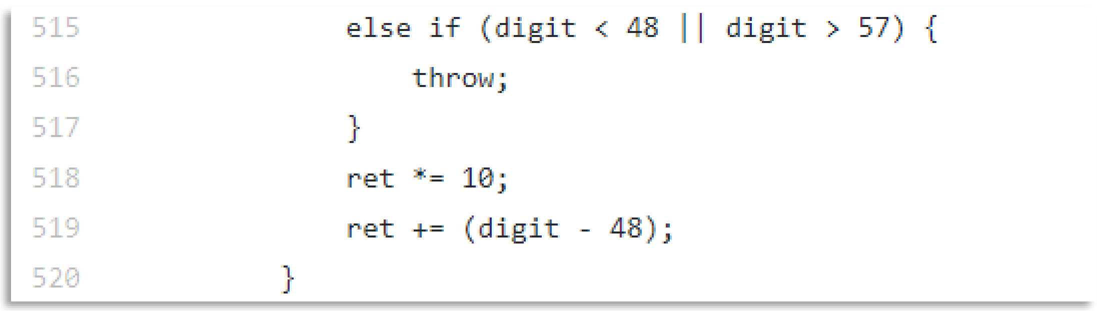
```
function getContractState(ContractState _value) public pure returns (uint result) {
	result = uint256(_value);
}
```

Then the appeal will be carried out, for example, as follows:

getContractState(ContractState.CANCELLED_BY_TENANT);

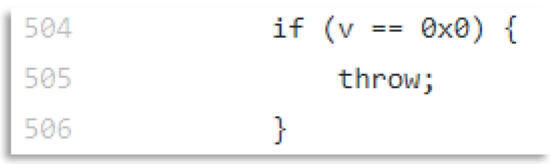
Similarly, it is possible to do the same for 3 functions of the type GetContractStage...().

Consider next the source code of this contract.

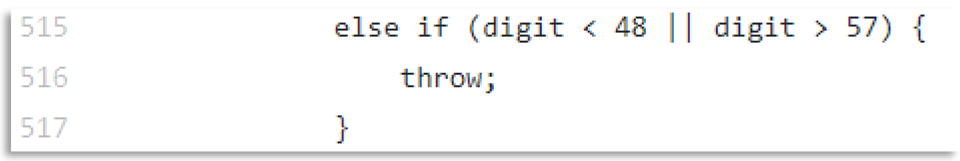


```
515                 else if (digit < 48 || digit > 57) {
516                     throw;
517                 }
518                 ret *= 10;
519                 ret += (digit - 48);
520             }
```

The code in line 519 is safe, because line 515 checks for the validity of the variable value.



```
504             if (v == 0x0) {
505                 throw;
506             }
```



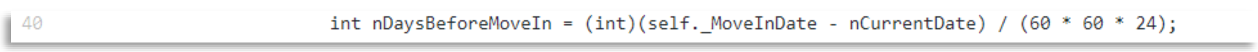
```
515                 else if (digit < 48 || digit > 57) {
516                     throw;
517                 }
```

(Take note) In lines 505 and 516, the use of “throw” is considered obsolete.
In the version Solidity 0.4.10, the functions assert (), require () and revert () were introduced. The call to revert () occurs in more "light" cases (for example, if an if / else call), and the call to assert () in more severe cases (for example: exceeding the limit for a variable, a condition that should not occur, etc.).

In this case, it would be preferable to use revert ().

Also, using “throw” will consume all the gas sent, and using revert () will return unused gas to the caller.

3. Contract “FlexibleEscrowLib”

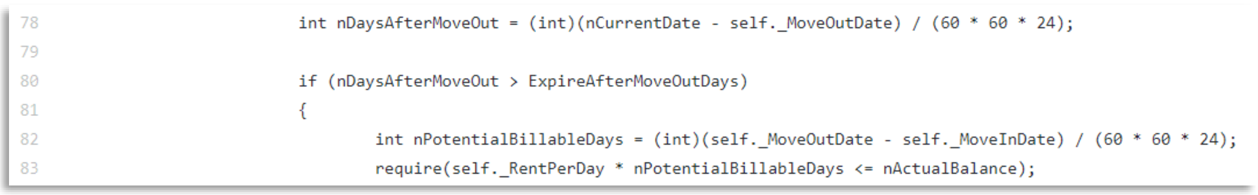


```
40                      int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (60 * 60 * 24);
```

(Take note) The code in line 40 can be replaced by

int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (1 days);

(Critical) Running code in line 40 without checking the values of variables can lead to unpredictable consequences.



```
78            int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (60 * 60 * 24);
79
80            if (nDaysAfterMoveOut > ExpireAfterMoveOutDays)
81            {
82                int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
83                require(self._RentPerDay * nPotentialBillableDays <= nActualBalance);
```

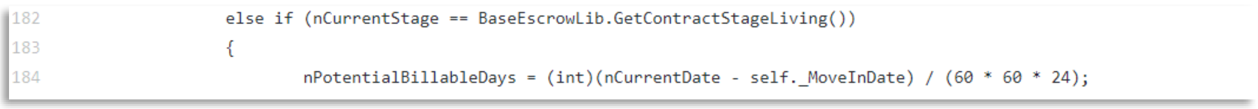
(Take note) The code in line 78 can be replaced by

int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (1 days);

(Take note) The code in line 82 can be replaced by

int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

(Critical) Executing code in lines 78 and 82 without checking the values of variables can lead to unpredictable consequences.

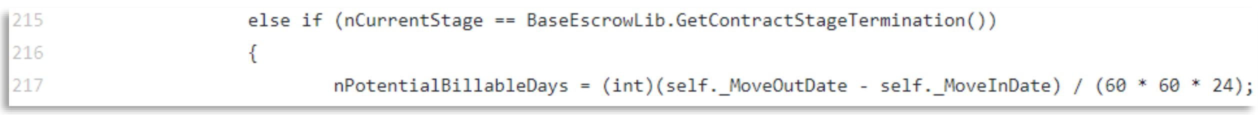


```
182        else if (nCurrentStage == BaseEscrowLib.GetContractStageLiving())
183        {
184            nPotentialBillableDays = (int)(nCurrentDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 184 can be replaced by

nPotentialBillableDays = (int)(nCurrentDate - self._MoveInDate) / (1 days);

(Critical) Executing code in line 184 without checking the values of variables can lead to unpredictable consequences.



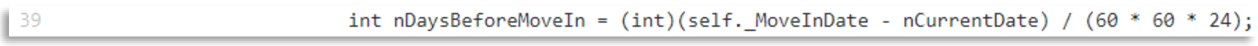
```
215        else if (nCurrentStage == BaseEscrowLib.GetContractStageTermination())
216        {
217            nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 217 can be replaced by

nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

(Critical) Running the code in line 217 without checking the values of variables can lead to unpredictable consequences.

4. Contract "ModerateEscrowLib"

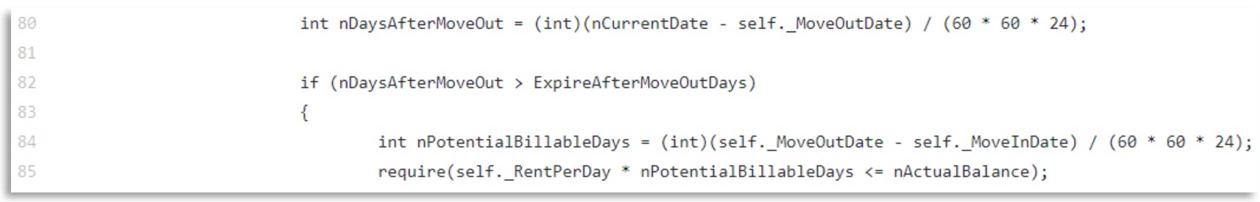


```
39            int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (60 * 60 * 24);
```

(Take note) The code in line 39 can be replaced by

int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (1 days);

(Critical) Executing code in line 39 without checking the values of variables can lead to unpredictable consequences.



```
80                      int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (60 * 60 * 24);
81
82                      if (nDaysAfterMoveOut > ExpireAfterMoveOutDays)
83                      {
84                              int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
85                              require(self._RentPerDay * nPotentialBillableDays <= nActualBalance);
```

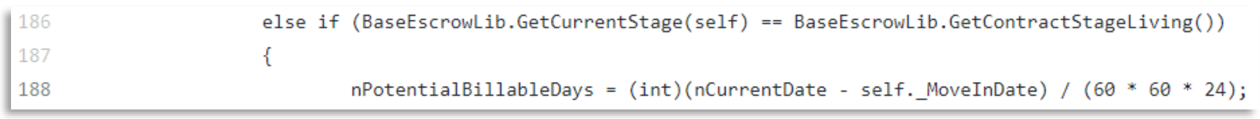
(Take note)   The code in line 80 can be replaced by

int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (1 days);

(Take note)   The code in line 84 can be replaced by

int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

(Critical) Executing code in lines 80 and 84 without checking the values of variables can lead to unpredictable consequences.

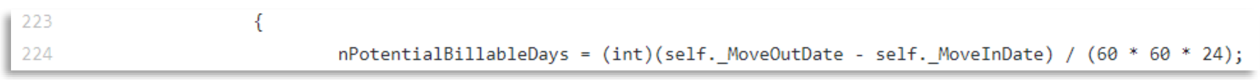


```
186                 else if (BaseEscrowLib.GetCurrentStage(self) == BaseEscrowLib.GetContractStageLiving())
187                 {
188                         nPotentialBillableDays = (int)(nCurrentDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 188 can be replaced by

nPotentialBillableDays = (int)(nCurrentDate - self._MoveInDate) / (1 days);

(Critical) Executing code in line 188 without checking the values of variables can lead to unpredictable consequences.



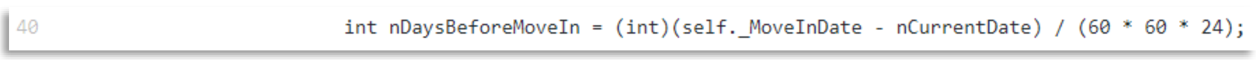
```
223                 {
224                         nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 224 can be replaced by

nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

(Critical) Executing code in line 224 without checking the values of variables can lead to unpredictable consequences.

5. <u>Contract "StrictEscrowLib"</u>

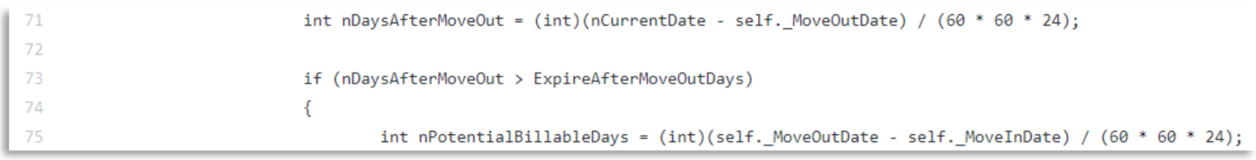


```
40                      int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (60 * 60 * 24);
```

(Take note) The code in line 40 can be replaced by

int nDaysBeforeMoveIn = (int)(self._MoveInDate - nCurrentDate) / (1 days);

(Critical) Executing code in line 40 without checking the values of variables can lead to unpredictable consequences.



```
71                      int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (60 * 60 * 24);
72
73                      if (nDaysAfterMoveOut > ExpireAfterMoveOutDays)
74                      {
75                          int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
```

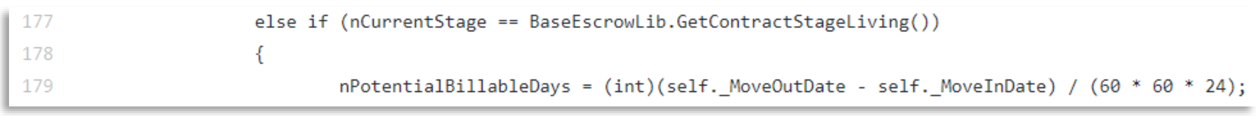
(Take note) The code in line 71 can be replaced by

int nDaysAfterMoveOut = (int)(nCurrentDate - self._MoveOutDate) / (1 days);

(Take note) The code in line 75 can be replaced by

int nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

(Critical) Executing code in lines 71 and 75 without checking the values of variables can lead to unpredictable consequences.

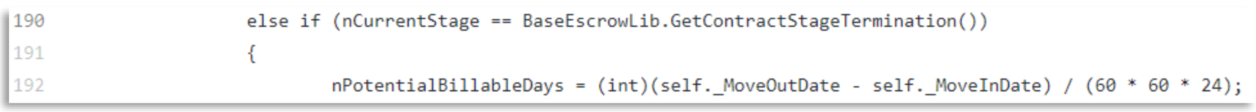


```
177                 else if (nCurrentStage == BaseEscrowLib.GetContractStageLiving())
178                 {
179                     nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 179 can be replaced by

nPotentialBillableDays = (int)(nCurrentDate - self._MoveInDate) / (1 days);

(Critical) Executing code in line 179 without checking the values of variables can lead to unpredictable consequences.



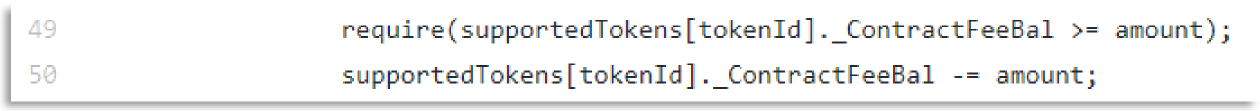
```
190                 else if (nCurrentStage == BaseEscrowLib.GetContractStageTermination())
191                 {
192                     nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (60 * 60 * 24);
```

(Take note) The code in line 192 can be replaced by

nPotentialBillableDays = (int)(self._MoveOutDate - self._MoveInDate) / (1 days);

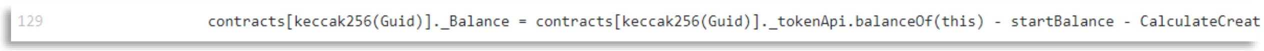
(Critical) Executing code in line 192 without checking the values of variables can lead to unpredictable consequences.

6. Contract “StayBitContractFactory”



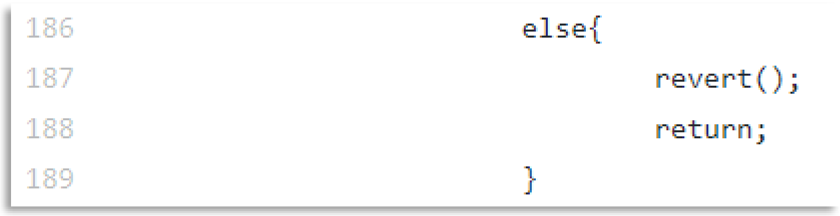
```
49                      require(supportedTokens[tokenId]._ContractFeeBal >= amount);
50                      supportedTokens[tokenId]._ContractFeeBal -= amount;
```

The code in line 50 is safe, because line 49 checks for the validity of variable values.

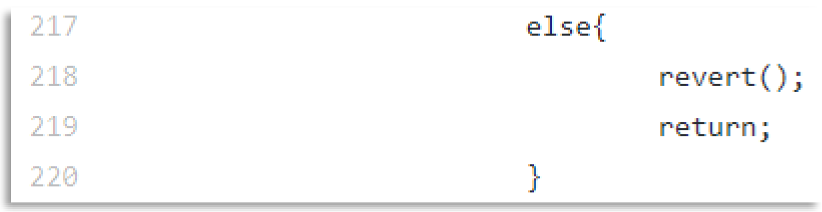


```
129             contracts[keccak256(Guid)]._Balance = contracts[keccak256(Guid)]._tokenApi.balanceOf(this) - startBalance - CalculateCreat
```

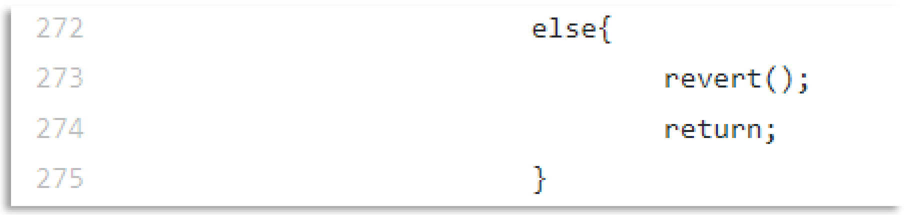
(Critical) Executing code in line 129 without checking the values of variables can lead to unpredictable consequences.



```
186                     else{
187                             revert();
188                             return;
189                     }
```

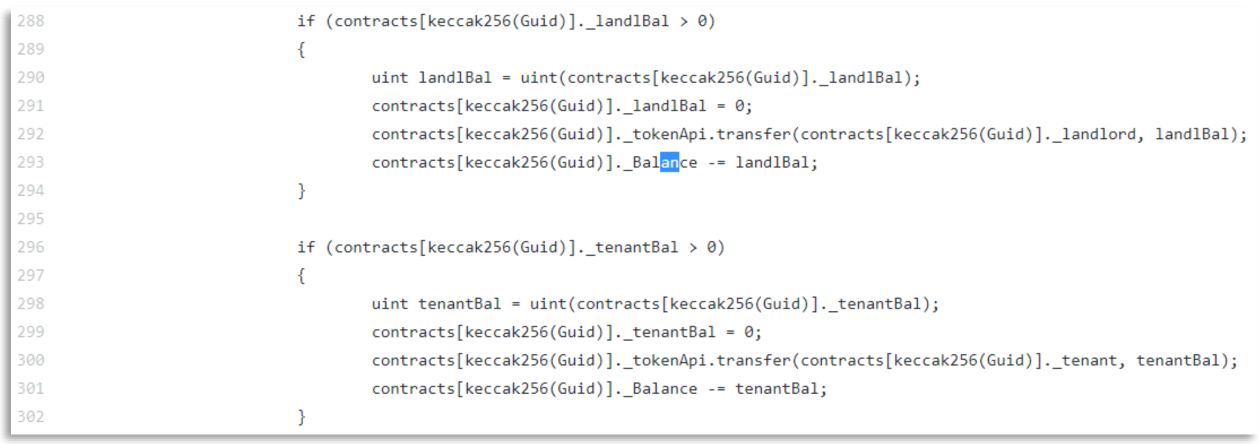


```
217                     else{
218                             revert();
219                             return;
220                     }
```



```
272                     else{
273                             revert();
274                             return;
275                     }
```

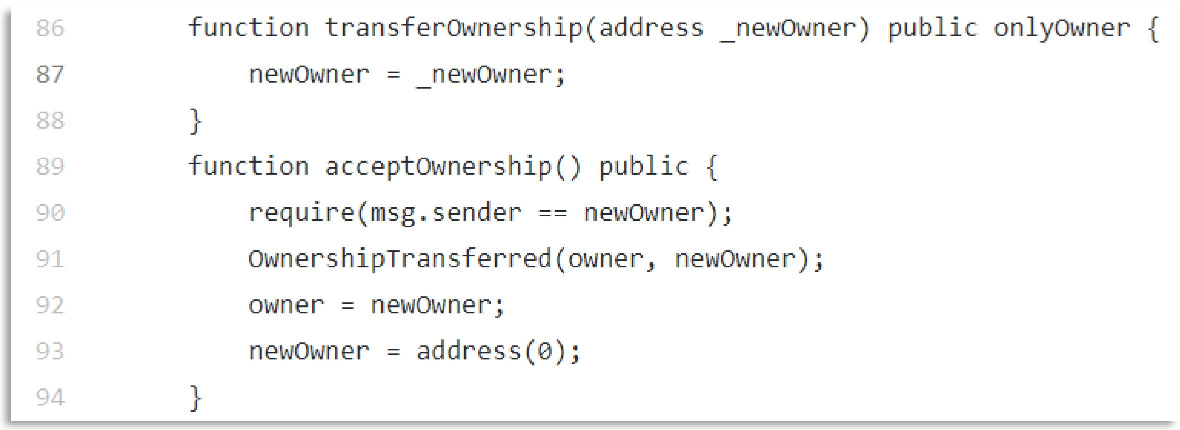
(Take note) Calling the return operator in lines 188, 219, 274 will never be executed, since always executing the code in lines 187, 218, 273 will lead to the exit of the function and the completion of the transaction.



```
288                 if (contracts[keccak256(Guid)]._landlBal > 0)
289                 {
290                         uint landlBal = uint(contracts[keccak256(Guid)]._landlBal);
291                         contracts[keccak256(Guid)]._landlBal = 0;
292                         contracts[keccak256(Guid)]._tokenApi.transfer(contracts[keccak256(Guid)]._landlord, landlBal);
293                         contracts[keccak256(Guid)]._Balance -= landlBal;
294                 }
295
296                 if (contracts[keccak256(Guid)]._tenantBal > 0)
297                 {
298                         uint tenantBal = uint(contracts[keccak256(Guid)]._tenantBal);
299                         contracts[keccak256(Guid)]._tenantBal = 0;
300                         contracts[keccak256(Guid)]._tokenApi.transfer(contracts[keccak256(Guid)]._tenant, tenantBal);
301                         contracts[keccak256(Guid)]._Balance -= tenantBal;
302                 }
```

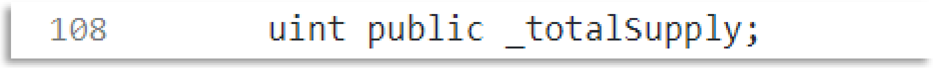
(Critical) Executing code in lines 293 and 301 without checking the values of variables can lead to unpredictable consequences.

7. Contract (MyToken)



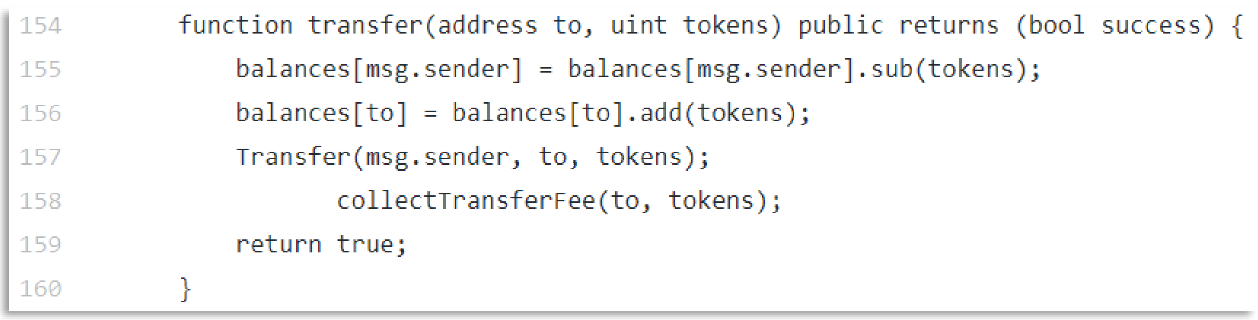
```
86          function transferOwnership(address _newOwner) public onlyOwner {
87              newOwner = _newOwner;
88          }
89          function acceptOwnership() public {
90              require(msg.sender == newOwner);
91              OwnershipTransferred(owner, newOwner);
92              owner = newOwner;
93              newOwner = address(0);
94          }
```

(Critical) Before performing the assignment of a new address value, there is no checking of the value of this address is equal 0. If an incorrect value is entered, it is possible that a situation of loss of control over contract management may arise.
(Critical) The function in line 89 can be called by any user and changed to another owner of the contract. In the case of the incorrect value of the newOwner variable, for example, immediately after the contract is installed on the network, a function call by any user will lead to the loss of the ability to manage the contract.



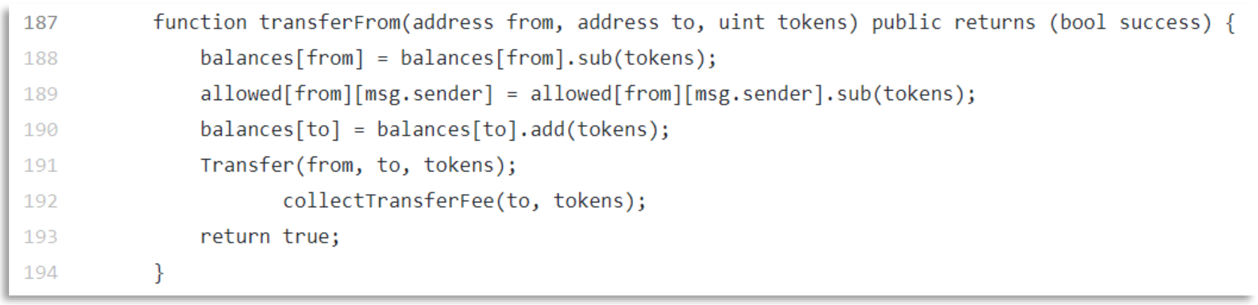
```
108         uint public _totalSupply;
```

(Take note) In line 108, one should either call the variable totalSupply and then remove the call to the totalSupply () function from line 136, or make the _totalSupply variable internal:
uint private _totalSupply;



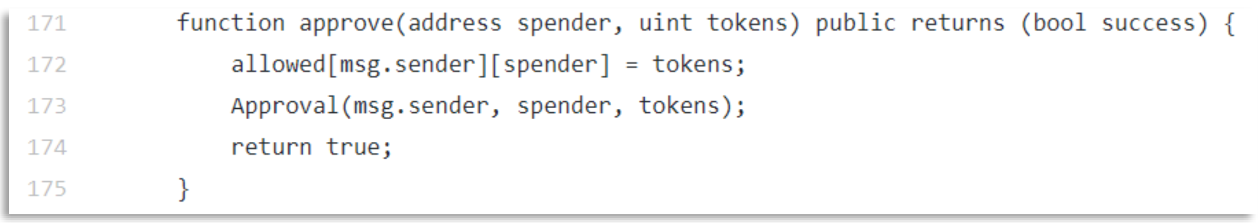
```
154         function transfer(address to, uint tokens) public returns (bool success) {
155             balances[msg.sender] = balances[msg.sender].sub(tokens);
156             balances[to] = balances[to].add(tokens);
157             Transfer(msg.sender, to, tokens);
158                     collectTransferFee(to, tokens);
159             return true;
160         }
```

(Critical) In lines 155 and 156, the change in the number of tokens occurs without checking the value of these tokens for validity. Calling this function with incorrect values will lead to unpredictable consequences.
It is necessary to check for valid values.
require(tokens <= _balances[msg.sender]);
require(to != address(0));



```
187         function transferFrom(address from, address to, uint tokens) public returns (bool success) {
188             balances[from] = balances[from].sub(tokens);
189             allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
190             balances[to] = balances[to].add(tokens);
191             Transfer(from, to, tokens);
192                     collectTransferFee(to, tokens);
193             return true;
194         }
```

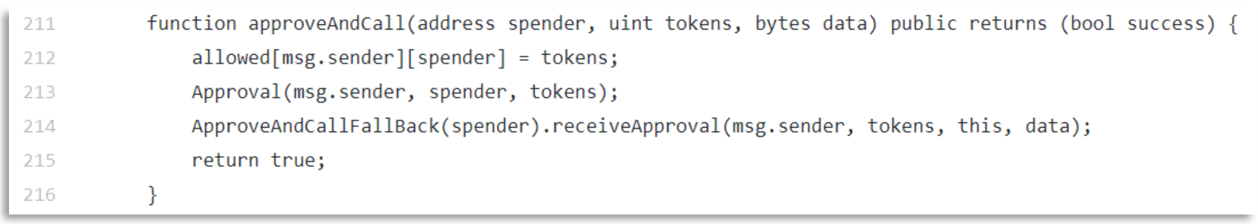
(Critical) In lines 188, 189 and 190, the change in the number of tokens occurs without checking the value of these tokens for validity. Calling this function with incorrect values will lead to unpredictable consequences.
It is necessary to check for valid values.
require(tokens <= _balances[from]);
require(to != address(0));
require(tokens <= _allowed[from][msg.sender]);



```
171         function approve(address spender, uint tokens) public returns (bool success) {
172             allowed[msg.sender][spender] = tokens;
173             Approval(msg.sender, spender, tokens);
174             return true;
175         }
```

(Critical) Before executing the code in line 172 did not check for correctness of the entered value of the address.
It is necessary to check for valid values:

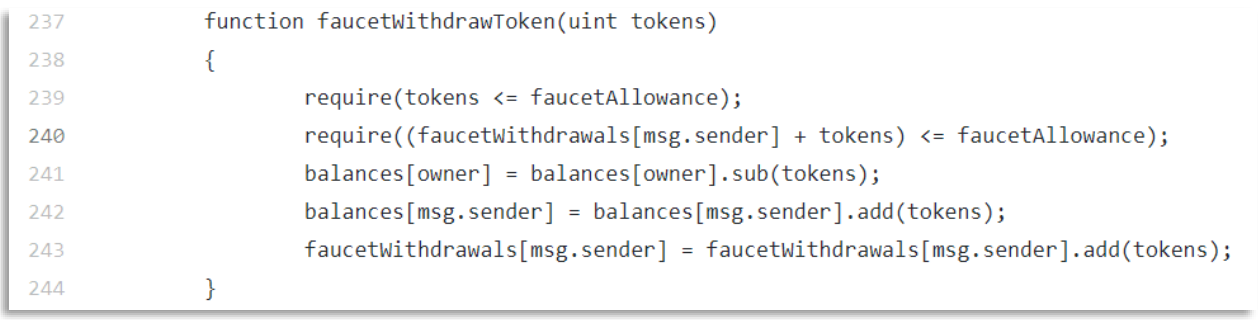
require(spender != address(0));



```
211         function approveAndCall(address spender, uint tokens, bytes data) public returns (bool success) {
212             allowed[msg.sender][spender] = tokens;
213             Approval(msg.sender, spender, tokens);
214             ApproveAndCallFallBack(spender).receiveApproval(msg.sender, tokens, this, data);
215             return true;
216         }
```

(Critical) Before executing the code in line 212 did not check for correctness of the entered value of the address.
It is necessary to check for valid values:

require(spender != address(0));

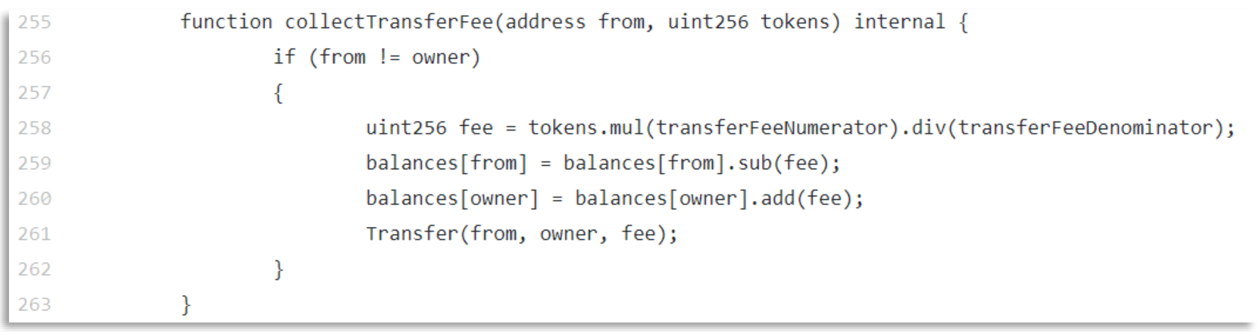


```
237             function faucetWithdrawToken(uint tokens)
238             {
239                     require(tokens <= faucetAllowance);
240                     require((faucetWithdrawals[msg.sender] + tokens) <= faucetAllowance);
241                     balances[owner] = balances[owner].sub(tokens);
242                     balances[msg.sender] = balances[msg.sender].add(tokens);
243                     faucetWithdrawals[msg.sender] = faucetWithdrawals[msg.sender].add(tokens);
244             }
```

(Critical) In lines 241, 242, 243, the change in the number of tokens occurs without checking the value of these tokens for validity. Calling this function with incorrect values will lead to unpredictable consequences.

It is necessary to check for valid values.

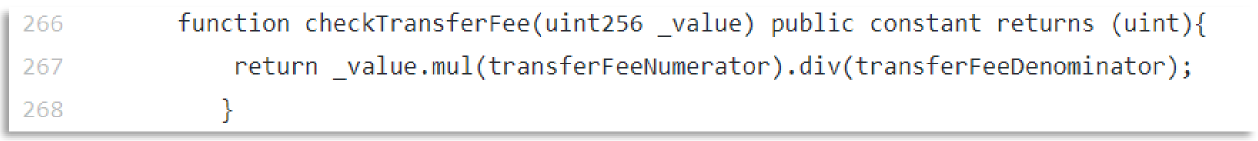
require(tokens <= _balances[owner]);



```
255             function collectTransferFee(address from, uint256 tokens) internal {
256                     if (from != owner)
257                     {
258                             uint256 fee = tokens.mul(transferFeeNumerator).div(transferFeeDenominator);
259                             balances[from] = balances[from].sub(fee);
260                             balances[owner] = balances[owner].add(fee);
261                             Transfer(from, owner, fee);
262                     }
263             }
```

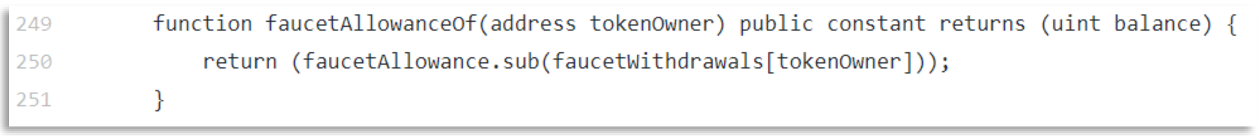
(Critical) In lines 258, 259, 260, the change in the number of tokens occurs without checking the value of these tokens for validity. Calling this function with incorrect values will lead to unpredictable consequences.

It is necessary to check for valid values.require(transferFeeDonominator > 0);
require(fee <= _balances[from]);



```
266        function checkTransferFee(uint256 _value) public constant returns (uint){
267            return _value.mul(transferFeeNumerator).div(transferFeeDenominator);
268          }
```

(Critical) In line 267 it is necessary to check the value of the variable for validity. Calling this function with incorrect values will lead to unpredictable consequences.
require(transferFeeDonominator > 0);



```
249        function faucetAllowanceOf(address tokenOwner) public constant returns (uint balance) {
250            return (faucetAllowance.sub(faucetWithdrawals[tokenOwner]));
251        }
```

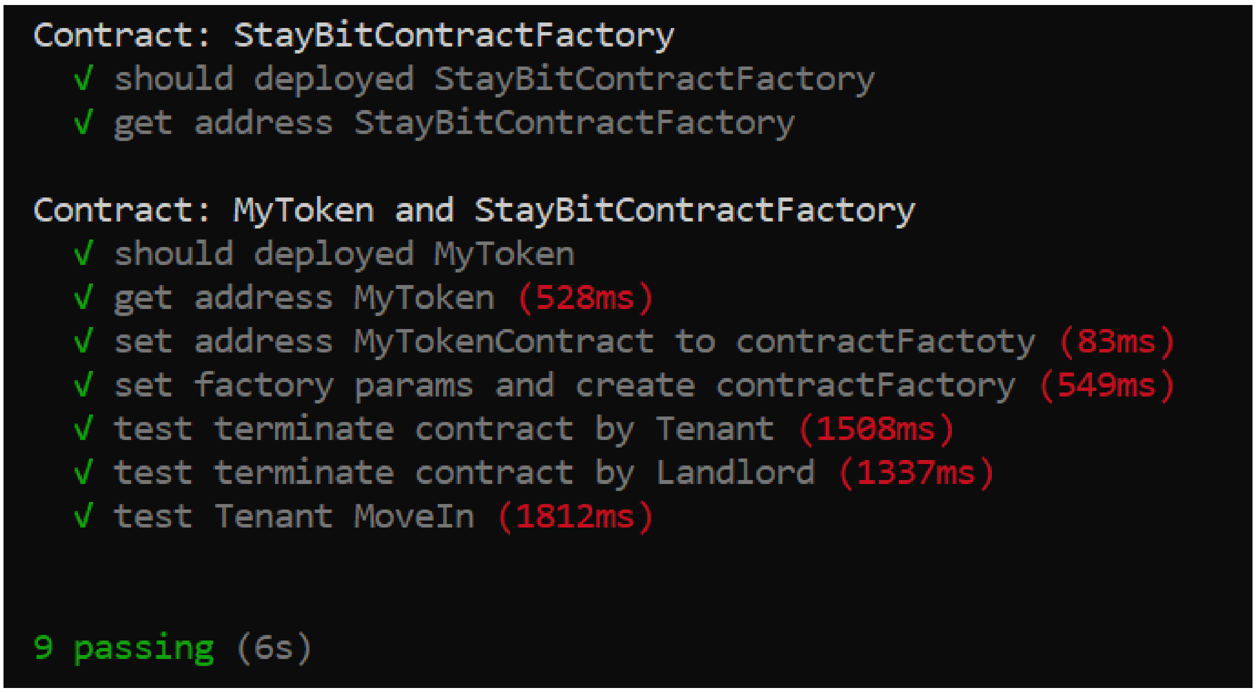
(Critical) Before executing the code in line 250, it is necessary to check the value of the variable for validity. Calling this function with incorrect values will lead to unpredictable consequences.
require(faucetAllowance >= faucetWithdrawals[tokenOwner]);

8. Контракт (Ownable)

There are no notes to this contract. The contract code is written safely.

Testing of the smart contracts

There were created tests using the Truffle framework to test the performance of DApp, the logic of work and collaboration of all smart contracts. The results of the test run for the main life stages of DApp are shown in the screenshot.



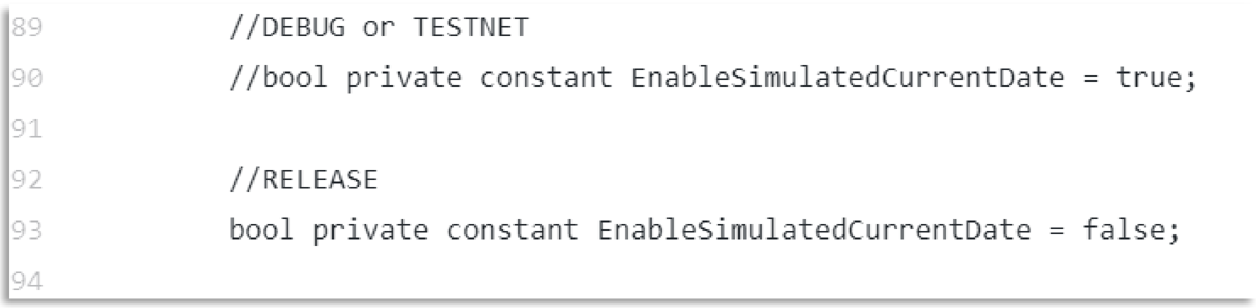
```
Contract: StayBitContractFactory
  √ should deployed StayBitContractFactory
  √ get address StayBitContractFactory

Contract: MyToken and StayBitContractFactory
  √ should deployed MyToken
  √ get address MyToken (528ms)
  √ set address MyTokenContract to contractFactoty (83ms)
  √ set factory params and create contractFactory (549ms)
  √ test terminate contract by Tenant (1508ms)
  √ test terminate contract by Landlord (1337ms)
  √ test Tenant MoveIn (1812ms)


9 passing (6s)
```

Internet address of the test's source code:

https://github.com/vpomo/AuditRentalContracts

No changes were made to the source code of the smart contracts, except for the inclusion of the possibility of changing the current date during testing.

For this, changes were made to the source code of the BaseEscrowLib contract. Line 93 was commented out and line 90 was uncommented.



```
89          //DEBUG or TESTNET
90          //bool private constant EnableSimulatedCurrentDate = true;
91
92          //RELEASE
93          bool private constant EnableSimulatedCurrentDate = false;
94
```

Also during the tests, it was assumed that the number of decimal places in the token is 0.

Successful testing suggests that, in general, the source code of the contract is operational. The results of the functions correspond to the declared logic of work.

Conclusion

1. The basic logic of the operation of all smart contracts is workable. This confirms the conduct of successful tests to verify the perfomance of the main life stages of DApp. No changes were made to the source code of the smart contracts, except for the inclusion of the possibility of changing the current date during testing. The source code of the tests is available at: https://github.com/vpomo/AuditRentalContracts
2. All important functions relating to the life cycle of the rental of real estate are protected from the performance of users who are not parties to the transaction.
3. During the audit many critical comments were found. They are mainly related to the lack of use of safe math SafeMath and the incorrect logic of some functions.
4. None of the contracts submitted to the study does not use the current version of Solidity - 0.4.25.
5. Names of variables and constants not by codestyle: https://solidity.readthedocs.io/en/v0.4.25/style-guide.html
6. It is recommended in accordance with the principles of object-oriented programming - SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) to separate them into independent entities and install each separately. The installation of contracts (not all) as separate entities will allow in the course of further work with DApp to change individual DApp modules without disrupting the operation of the entire application.
7. Before the commercial operation beginning of the DApp, it is necessary to substantially rework the entire source code of the smart contracts.