MythX

REPORT 5F857FE08A37B500182C2CF6

Created	Tue Oct 13 2020 10:22:24 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	yannik@chainsulting.de

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
3efdea50-6729-4212-8fc4-29f5ac739c93	browser/lockcontract.sol	34

Analysis 3efdea50-6729-4212-8fc4-29f5ac739c93 MythX

Started	Tue Oct 13 2020 10:22:28 GMT+0000 (Coordinated Universal Time)
Finished	Tue Oct 13 2020 11:07:45 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Browser/Lockcontract.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	14	20

ISSUES

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
101
102 //update balance in address
103 walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[_tokenAddress][msg.sender].add(_amount);
104
105 address _withdrawalAddress = msg.sender;
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
104
105 address _withdrawalAddress = msg.sender;
106 _id = ++depositId;
107 lockedToken[_id].tokenAddress = _tokenAddress;
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
105 address _withdrawalAddress = msg.sender;
106 _id = ++depositId;
107 lockedToken[_id].tokenAddress = _tokenAddress;
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
109 lockedToken[_id].tokenAmount = _amount;
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
106 _id = ++depositId;
107 lockedToken[_id].tokenAddress = _tokenAddress;
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
109 lockedToken[_id].tokenAmount = _amount;
110 lockedToken[_id].unlockTime = _unlockTime;
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
107 lockedToken[_id].tokenAddress = _tokenAddress;
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
109 lockedToken[_id].tokenAmount = _amount;
110 lockedToken[_id].unlockTime = _unlockTime;
111 lockedToken[_id].withdrawn = false;
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`browser/lockcontract.sol`

Locations

```
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
109 lockedToken[_id].tokenAmount = _amount;
110 lockedToken[_id].unlockTime = _unlockTime;
111 lockedToken[_id].withdrawn = false;
112
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`browser/lockcontract.sol`

Locations

```
109 lockedToken[_id].tokenAmount = _amount;
110 lockedToken[_id].unlockTime = _unlockTime;
111 lockedToken[_id].withdrawn = false;
112
113 allDepositIds.push(_id);
```

MEDIUM

Read of persistent state following external call

SWC-107

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`browser/lockcontract.sol`

Locations

```
63  }
64
65  contract lockContract is owned{
66  using SafeMath for uint256;
67
68  /*
69  * deposit vars
70  */
71  struct Items {
72  address tokenAddress;
73  address withdrawalAddress;
74  uint256 tokenAmount;
75  uint256 unlockTime;
76  bool withdrawn;
77  }
78
79  uint256 public depositId;
80  uint256[] public allDepositIds;
81  mapping (address => uint256[]) public depositsByWithdrawalAddress;
82  mapping (uint256 => Items) public lockedToken;
83  mapping (address => mapping(address => uint256)) public walletTokenBalance;
84
85  event LogWithdrawal(address SentToAddress, uint256 AmountTransferred);
86
87  /**
88  * Constrctor function
89  */
90  function lockContract() public {
91
92  }
93
94  /**
95  *lock tokens
96  */
97  function lockTokens(address _tokenAddress, uint256 _amount, uint256 _unlockTime) public returns (uint256 _id) {
98  require(_amount > 0);
99  require(_unlockTime < 10000000000);
100 require(Token(_tokenAddress).transferFrom(msg.sender, this, _amount));
101
102 //update balance in address
103 walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[_tokenAddress][msg.sender].add(_amount);
104
105 address _withdrawalAddress = msg.sender;
106 _id = ++depositId;
107 lockedToken[_id].tokenAddress = _tokenAddress;
108 lockedToken[_id].withdrawalAddress = _withdrawalAddress;
109 lockedToken[_id].tokenAmount = _amount;
110 lockedToken[_id].unlockTime = _unlockTime;
111 lockedToken[_id].withdrawn = false;
112
113 allDepositIds.push(_id);
114 depositsByWithdrawalAddress[_withdrawalAddress].push(_id);
115 }
116
117 /**
```

```
*transfer locked tokens
*/
function transferLocks(uint256 _id, address _receiverAddress) public {
require(block.timestamp < lockedToken[_id].unlockTime);
require(msg.sender == lockedToken[_id].withdrawalAddress);
lockedToken[_id].withdrawalAddress = _receiverAddress;

//decrease sender's token balance
walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);

//increase receiver's token balance
walletTokenBalance[lockedToken[_id].tokenAddress][_receiverAddress] = walletTokenBalance[lockedToken[_id].tokenAddress][_receiverAddress].add(lockedToken[_id].tokenAmount);

}

/**
*withdraw tokens
*/
function withdrawTokens(uint256 _id) public {
require(block.timestamp >= lockedToken[_id].unlockTime);
require(msg.sender == lockedToken[_id].withdrawalAddress);
require(!lockedToken[_id].withdrawn);
require(Token(lockedToken[_id].tokenAddress).transfer(msg.sender, lockedToken[_id].tokenAmount));

lockedToken[_id].withdrawn = true;

//update balance in address
walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);

LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
}

/*get total token balance in contract*/
function getTotalTokenBalance(address _tokenAddress) view public returns (uint256)
{
return Token(_tokenAddress).balanceOf(this);
}

/*get total token balance by address*/
function getTokenBalanceByAddress(address _tokenAddress, address _walletAddress) view public returns (uint256)
{
return walletTokenBalance[_tokenAddress][_walletAddress];
}

/*get allDepositIds*/
function getAllDepositIds() view public returns (uint256[])
{
return allDepositIds;
}

/*get getDepositDetails*/
function getDepositDetails(uint256 _id) view public returns (address tokenAddress, address withdrawalAddress, uint256 tokenAmount, uint256 unlockTime, bool withdrawn)
{
return(lockedToken[_id].tokenAddress,lockedToken[_id].withdrawalAddress,lockedToken[_id].tokenAmount,
lockedToken[_id].unlockTime,lockedToken[_id].withdrawn);
}

/*get number of active deposits of an address*/
function numOfActiveDeposits(address _withdrawalAddress) public view returns (uint256) {
uint256 staked = 0;
for (uint i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
if (!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
staked++;
```

```
181 }
182 }
183 return staked;
184 }
185
186 /*get getWithdrawableDepositsByAddress*/
187 function getWithdrawableDepositsByAddress(address _withdrawalAddress) view public returns (uint256[])
188 {
189 uint256[] memory deposits = new uint256[](numOfActiveDeposits(_withdrawalAddress));
190 uint256 tempIdx = 0;
191 for(uint256 i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
192 if(!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
193 deposits[tempIdx] = depositsByWithdrawalAddress[_withdrawalAddress][i];
194 tempIdx ++;
195 }
196 }
197 return deposits;
198 }
199
200 /*get getAllDepositsByAddress*/
201 function getAllDepositsByAddress(address _withdrawalAddress) view public returns (uint256[])
202 {
203 return depositsByWithdrawalAddress[_withdrawalAddress];
204 }
205
206 }
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`browser/lockcontract.sol`

Locations

```
111 lockedToken[_id].withdrawn = false;
112
113 allDepositIds.push(_id);
114 depositsByWithdrawalAddress[_withdrawalAddress].push(_id);
115 }
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`browser/lockcontract.sol`

Locations

```
112
113 allDepositIds.push(_id);
114 depositsByWithdrawalAddress[_withdrawalAddress].push(_id);
115 }
116
```

MEDIUM

Write to persistent state following external call

SWC-107

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
140 require(Token(lockedToken[_id].tokenAddress).transfer(msg.sender, lockedToken[_id].tokenAmount));
141
142 lockedToken[_id].withdrawn = true;
143
144 //update balance in address
```

MEDIUM

Read of persistent state following external call

SWC-107

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
143
144 //update balance in address
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
```

MEDIUM

Read of persistent state following external call

SWC-107

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
143
144 //update balance in address
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
```

MEDIUM

SWC-107

Write to persistent state following external call

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
143
144 //update balance in address
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.4.16"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

browser/lockcontract.sol

Locations

```
1 pragma solidity ^0.4.16;
2
3 /**
```

LOW

SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Source file

browser/lockcontract.sol

Locations

```
151 function getTotalTokenBalance(address _tokenAddress) view public returns (uint256)
152 {
153 return Token(_tokenAddress).balanceOf(this);
154 }
155
```

LOW

SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Source file

browser/lockcontract.sol

Locations

```
98 require(_amount > 0);
99 require(_unlockTime < 10000000000);
100 require(Token(_tokenAddress).transferFrom(msg.sender, this, _amount));
101
102 //update balance in address
```

LOW

SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Source file

browser/lockcontract.sol

Locations

```
138 require(msg.sender == lockedToken[_id].withdrawalAddress);
139 require(!lockedToken[_id].withdrawn);
140 require(Token(lockedToken[_id].tokenAddress).transfer(msg.sender, lockedToken[_id].tokenAmount));
141
142 lockedToken[_id].withdrawn = true;
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
101
102 //update balance in address
103 walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[_tokenAddress][msg.sender].add(_amount);
104
105 address _withdrawalAddress = msg.sender;
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
143
144 //update balance in address
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
143
144 //update balance in address
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

browser/lockcontract.sol

Locations

```
145 walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);
146
147 LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
148 }
```

LOW

SWC-111

Use of the "constant" state mutability modifier is deprecated.

Using "constant" as a state mutability modifier in function "mul" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source file

browser/lockcontract.sol

Locations

```
14
15 library SafeMath {
16 function mul(uint256 a, uint256 b) internal constant returns (uint256) {
17 if (a == 0) {
18 return 0;
19 }
20 uint256 c = a * b;
21 require(c / a == b);
22 return c;
23 }
24
25 function div(uint256 a, uint256 b) internal constant returns (uint256) {
```

LOW

Use of the "constant" state mutability modifier is deprecated.

Using "constant" as a state mutability modifier in function "div" is disallowed as of Solidity version 0.5.0. Use "view" instead.

SWC-111

Source file

browser/lockcontract.sol

Locations

```
23  }
24
25  function div(uint256 a, uint256 b) internal constant returns (uint256) {
26  uint256 c = a / b;
27  return c;
28  }
29
30  function sub(uint256 a, uint256 b) internal constant returns (uint256) {
```

LOW

Use of the "constant" state mutability modifier is deprecated.

Using "constant" as a state mutability modifier in function "sub" is disallowed as of Solidity version 0.5.0. Use "view" instead.

SWC-111

Source file

browser/lockcontract.sol

Locations

```
28  }
29
30  function sub(uint256 a, uint256 b) internal constant returns (uint256) {
31  require(b <= a);
32  return a - b;
33  }
34
35  function add(uint256 a, uint256 b) internal constant returns (uint256) {
```

LOW

Use of the "constant" state mutability modifier is deprecated.

Using "constant" as a state mutability modifier in function "add" is disallowed as of Solidity version 0.5.0. Use "view" instead.

SWC-111

Source file

browser/lockcontract.sol

Locations

```
33  }
34
35  function add(uint256 a, uint256 b) internal constant returns (uint256) {
36  uint256 c = a + b;
37  require(c >= a);
38  return c;
39  }
40
41  function ceil(uint256 a, uint256 m) internal constant returns (uint256) {
```

LOW

SWC-111

Use of the "constant" state mutability modifier is deprecated.

Using "constant" as a state mutability modifier in function "ceil" is disallowed as of Solidity version 0.5.0. Use "view" instead.

Source file

browser/lockcontract.sol

Locations

```
39 }
40
41 function ceil(uint256 a, uint256 m) internal constant returns (uint256) {
42 uint256 c = add(a,m);
43 uint256 d = sub(c,1);
44 return mul(div(d,m),m);
45 }
46 }
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

browser/lockcontract.sol

Locations

```
135 */
136 function withdrawTokens(uint256 _id) public {
137 require(block.timestamp >= lockedToken[_id].unlockTime);
138 require(msg.sender == lockedToken[_id].withdrawalAddress);
139 require(!lockedToken[_id].withdrawn);
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

browser/lockcontract.sol

Locations

```
119 */
120 function transferLocks(uint256 _id, address _receiverAddress) public {
121 require(block.timestamp < lockedToken[_id].unlockTime);
122 require(msg.sender == lockedToken[_id].withdrawalAddress);
123 lockedToken[_id].withdrawalAddress = _receiverAddress;
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

browser/lockcontract.sol

Locations

```
98  require(_amount > 0);
99  require(_unlockTime < 10000000000);
100 require(Token(_tokenAddress).transferFrom(msg.sender, this, _amount));
101
102 //update balance in address
```

Source file

browser/lockcontract.sol

Locations

```
63  }
64
65  contract lockContract is owned{
66  using SafeMath for uint256;
67
68  /*
69  * deposit vars
70  */
71  struct Items {
72  address tokenAddress;
73  address withdrawalAddress;
74  uint256 tokenAmount;
75  uint256 unlockTime;
76  bool withdrawn;
77  }
78
79  uint256 public depositId;
80  uint256[] public allDepositIds;
81  mapping (address => uint256[]) public depositsByWithdrawalAddress;
82  mapping (uint256 => Items) public lockedToken;
83  mapping (address => mapping(address => uint256)) public walletTokenBalance;
84
85  event LogWithdrawal(address SentToAddress, uint256 AmountTransferred);
86
87  /**
88  * Constrctor function
89  */
90  function lockContract() public {
91
92  }
93
94  /**
95  *lock tokens
96  */
97  function lockTokens(address _tokenAddress, uint256 _amount, uint256 _unlockTime) public returns (uint256 _id) {
98  require(_amount > 0);
99  require(_unlockTime < 10000000000);
100 require(Token(_tokenAddress).transferFrom(msg.sender, this, _amount));
101
102 //update balance in address
103 walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[_tokenAddress][msg.sender].add(_amount);
104
105 address _withdrawalAddress = msg.sender;
106 _id = ++depositId;
107 lockedToken[_id].tokenAddress = _tokenAddress;
```

```
lockedToken[_id].withdrawalAddress = _withdrawalAddress;
lockedToken[_id].tokenAmount = _amount;
lockedToken[_id].unlockTime = _unlockTime;
lockedToken[_id].withdrawn = false;

allDepositIds.push(_id);
depositsByWithdrawalAddress[_withdrawalAddress].push(_id);
}

/**
*transfer locked tokens
*/
function transferLocks(uint256 _id, address _receiverAddress) public {
require(block.timestamp < lockedToken[_id].unlockTime);
require(msg.sender == lockedToken[_id].withdrawalAddress);
lockedToken[_id].withdrawalAddress = _receiverAddress;

//decrease sender's token balance
walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);

//increase receiver's token balance
walletTokenBalance[lockedToken[_id].tokenAddress][_receiverAddress] = walletTokenBalance[lockedToken[_id].tokenAddress][_receiverAddress].add(lockedToken[_id].tokenAmount);

}

/**
*withdraw tokens
*/
function withdrawTokens(uint256 _id) public {
require(block.timestamp >= lockedToken[_id].unlockTime);
require(msg.sender == lockedToken[_id].withdrawalAddress);
require(!lockedToken[_id].withdrawn);
require(Token(lockedToken[_id].tokenAddress).transfer(msg.sender, lockedToken[_id].tokenAmount));

lockedToken[_id].withdrawn = true;

//update balance in address
walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount);

LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
}

/*get total token balance in contract*/
function getTotalTokenBalance(address _tokenAddress) view public returns (uint256)
{
return Token(_tokenAddress).balanceOf(this);
}

/*get total token balance by address*/
function getTokenBalanceByAddress(address _tokenAddress, address _walletAddress) view public returns (uint256)
{
return walletTokenBalance[_tokenAddress][_walletAddress];
}

/*get allDepositIds*/
function getAllDepositIds() view public returns (uint256[])
{
return allDepositIds;
}

/*get getDepositDetails*/
function getDepositDetails(uint256 _id) view public returns (address tokenAddress, address withdrawalAddress, uint256 tokenAmount, uint256 unlockTime, bool withdrawn)
{
```

```
return(lockedToken[_id].tokenAddress,lockedToken[_id].withdrawalAddress,lockedToken[_id].tokenAmount,
lockedToken[_id].unlockTime,lockedToken[_id].withdrawn);
}

/*get number of active deposits of an address*/
function numOfActiveDeposits(address _withdrawalAddress) public view returns (uint256) {
uint256 staked = 0;
for (uint i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
if (!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
staked++;
}
}
return staked;
}

/*get getWithdrawableDepositsByAddress*/
function getWithdrawableDepositsByAddress(address _withdrawalAddress) view public returns (uint256[])
{
uint256[] memory deposits = new uint256[](numOfActiveDeposits(_withdrawalAddress));
uint256 tempIdx = 0;
for(uint256 i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
if(!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
deposits[tempIdx] = depositsByWithdrawalAddress[_withdrawalAddress][i];
tempIdx ++;
}
}
return deposits;
}

/*get getAllDepositsByAddress*/
function getAllDepositsByAddress(address _withdrawalAddress) view public returns (uint256[])
{
return depositsByWithdrawalAddress[_withdrawalAddress];
}

}
```

LOW

SWC-128

Implicit loop over unbounded data structure.

Gas consumption in function "getAllDepositIds" in contract "lockContract" depends on the size of data structures that may grow unboundedly. The highlighted statement involves copying the array "allDepositIds" from "storage" to "memory". When copying arrays from "storage" to "memory" the Solidity compiler emits an implicit loop.If the array grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Source file

`browser/lockcontract.sol`

Locations

```
function getAllDepositIds() view public returns (uint256[])
{
return allDepositIds;
}
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "numOfActiveDeposits" in contract "lockContract" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Source file

browser/lockcontract.sol

Locations

```
176 function numOfActiveDeposits(address _withdrawalAddress) public view returns (uint256) {
177 uint256 staked = 0;
178 for (uint i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
179 if (!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
180 staked++;
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getWithdrawableDepositsByAddress" in contract "lockContract" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Source file

browser/lockcontract.sol

Locations

```
189 uint256[] memory deposits = new uint256[](numOfActiveDeposits(_withdrawalAddress));
190 uint256 tempIdx = 0;
191 for(uint256 i = 0; i < depositsByWithdrawalAddress[_withdrawalAddress].length; i++) {
192 if(!lockedToken[depositsByWithdrawalAddress[_withdrawalAddress][i]].withdrawn) {
193 deposits[tempIdx] = depositsByWithdrawalAddress[_withdrawalAddress][i];
```

LOW

SWC-128

Implicit loop over unbounded data structure.

Gas consumption in function "getAllDepositsByAddress" in contract "lockContract" depends on the size of data structures that may grow unboundedly. The highlighted statement involves copying the array "depositsByWithdrawalAddress[_withdrawalAddress]" from "storage" to "memory". When copying arrays from "storage" to "memory" the Solidity compiler emits an implicit loop.If the array grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Source file

browser/lockcontract.sol

Locations

```
201 function getAllDepositsByAddress(address _withdrawalAddress) view public returns (uint256[])
202 {
203 return depositsByWithdrawalAddress[_withdrawalAddress];
204 }
205
```