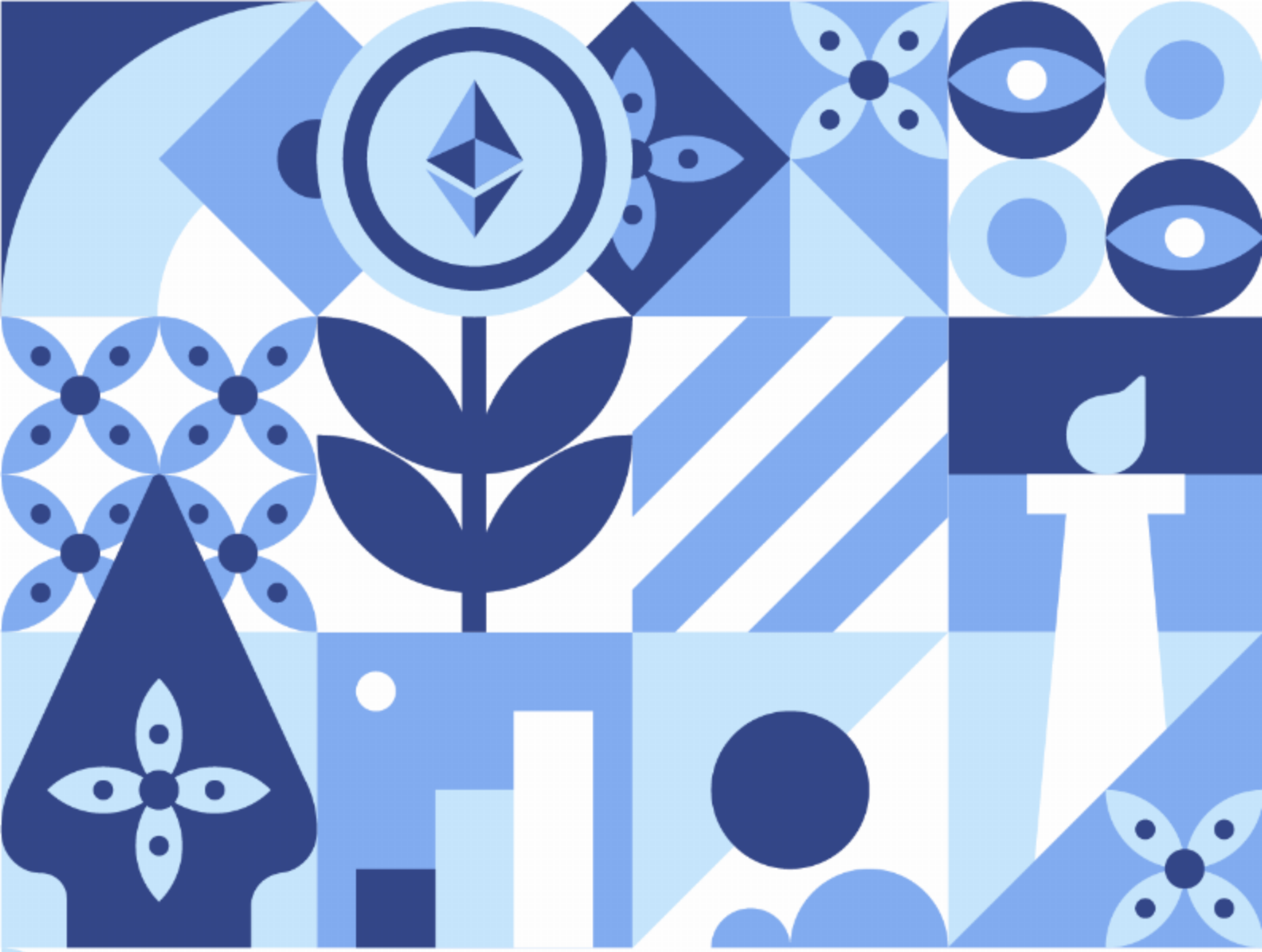

FunFair

Smart Contract Audit Report

SYSFIXED

07 Jul 2017

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
FunFair	FUN	Ethereum

Addresses

Contract address	0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b
Contract deployer address	0x50b26685BC788E164d940F0a73770F4B9196B052

Project Website

https://funtoken.io/

Codebase

https://etherscan.io/address/0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b#code

SUMMARY

The FUNToken is an asset developed specifically for the online gambling and gaming industry. FUNToken combines the qualities of the Ethereum blockchain with a cutting-edge tech stack, making FUN a powerful resource for players, platforms, and developers alike.

Contract Summary

Documentation Quality

FunFair provides a very poor documentation with standard of solidity base code.

- The technical description is provided unclear and disorganized.

Code Quality

The Overall quality of the basecode is poor.

- Solidity basecode and rules are unclear and disorganized by FunFair.

Test Coverage

Test coverage of the project is 100% (Through Codebase)

Audit Findings Summary

- SWC-101 | It is recommended to use vetted safe math libraries for arithmetic operations consistently on lines 229, 93 and 231.
- SWC-107 | It is recommended to use a reentrancy lock, reentrancy weaknesses detected on lines 79, 168, 147, 158, 147, 168 and 158.
- SWC-110 SWC-123 | It is recommended to use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM on lines 147, 80, 168, 158, 93, 95, 141, 195, 115, 132, 96, 178, 59, 219, 229, 231, 163, 77, 111, 49, 203, 116, 32, 188, 107, 97, 183, 45, 123, 209, 179, 57, 153, 119, 145, 112, 79, 135, 166, 120, 126 and 156.
- SWC-113 SWC-128 | It is recommended to implement the contract logic to handle failed calls and block gas limit on lines 80, 168, 158 and 147.

CONCLUSION

We have audited the FunFair project released in June 2017 to find issues and identify potential security vulnerabilities in the FunFair project. This process is used to find technical issues and security loopholes that may be found in smart contracts.

The security audit report gave unsatisfactory results with the discovery of high-risk issues and several other low-risk issues.

Writing a contract that does not follow the Solidity style guide can pose a significant risk. The high-risk, medium, and low problems we found in the smart contract are the arithmetic operation can underflow, an assertion violation was triggered, multiple calls are executed in the same transaction, a call to a user-supplied address is executed, an assertion violation was triggered, multiple calls are executed in the same transaction. We not recommended to take invest to this kind of risky smart contract.

AUDIT RESULT

Article	Category	Description	Result
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	PASS
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	ISSUE FOUND
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	PASS
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	PASS
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	PASS
Unprotected Ether Withdrawal	SWC-105	Due to missing or insufficient access controls, malicious parties can withdraw from the contract.	PASS
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	PASS
Reentrancy	SWC-107	Check effect interaction pattern should be followed if the code performs recursive call.	ISSUE FOUND
Uninitialized Storage Pointer	SWC-109	Uninitialized local storage variables can point to unexpected storage locations in the contract.	PASS
Assert Violation	SWC-110 SWC-123	Properly functioning code should never reach a failing assert statement.	ISSUE FOUND
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	PASS
Delegate call to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	PASS

DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	ISSUE FOUND
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	PASS
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	PASS
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	PASS
Signature Unique ID	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	PASS
Incorrect Constructor Name	SWC-118	Constructors are special functions that are called only once during the contract creation.	PASS
Shadowing State Variable	SWC-119	State variables should not be shadowed.	PASS
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	PASS
Write to Arbitrary Storage Location	SWC-124	The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations.	PASS
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.	PASS
Insufficient Gas Griefing	SWC-126	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract.	PASS
Arbitrary Jump Function	SWC-127	As Solidity doesnt support pointer arithmetics, it is impossible to change such variable to an arbitrary value.	PASS

Typographical Error	SWC-129	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable.	PASS
Override control character	SWC-130	Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.	PASS
Unused variables	SWC-131 SWC-135	Unused variables are allowed in Solidity and they do not pose a direct security issue.	PASS
Unexpected Ether balance	SWC-132	Contracts can behave erroneously when they strictly assume a specific Ether balance.	PASS
Hash Collisions Variable	SWC-133	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision.	PASS
Hardcoded gas amount	SWC-134	The transfer() and send() functions forward a fixed amount of 2300 gas.	PASS
Unencrypted Private Data	SWC-136	It is a common misconception that private type variables cannot be read.	PASS

SMART CONTRACT ANALYSIS

Started	Thursday Jul 06 2017 05:32:46 GMT+0000 (Coordinated Universal Time)
Finished	Friday Jul 07 2017 20:42:39 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	Token.sol

Detected Issues

ID	Title	Severity	Status
SWC-101	THE ARITHMETIC OPERATION CAN UNDERFLOW.	high	acknowledged
SWC-101	THE ARITHMETIC OPERATION CAN UNDERFLOW.	high	acknowledged
SWC-101	THE ARITHMETIC OPERATOR CAN OVERFLOW.	high	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	medium	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	medium	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	medium	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	medium	acknowledged
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	medium	acknowledged
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	medium	acknowledged
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	medium	acknowledged
SWC-107	A CALL TO A USER-SUPPLIED ADDRESS IS EXECUTED.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged

SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged

SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	low	acknowledged

SWC-101 | THE ARITHMETIC OPERATION CAN UNDERFLOW.

LINE 229

high SEVERITY

It is possible to cause an arithmetic underflow. Prevent the underflow by constraining inputs using the require() statement or use the OpenZeppelin SafeMath library for integer arithmetic operations. Refer to the transaction trace generated for this issue to reproduce the underflow.

Source File

- Token.sol

Locations

```
228
229   string public motd;
230   event Motd(string message);
231   function setMotd(string _m) onlyOwner {
232   motd = _m;
233
```

SWC-101 | THE ARITHMETIC OPERATION CAN UNDERFLOW.

LINE 93

high SEVERITY

It is possible to cause an arithmetic underflow. Prevent the underflow by constraining inputs using the require() statement or use the OpenZeppelin SafeMath library for integer arithmetic operations. Refer to the transaction trace generated for this issue to reproduce the underflow.

Source File

- Token.sol

Locations

```
92
93   contract Token is Finalizable, TokenReceivable, SafeMath, EventDefinitions {
94
95   string public name = "FunFair";
96   uint8 public decimals = 8;
97
```

SWC-101 | THE ARITHMETIC OPERATOR CAN OVERFLOW.

LINE 231

high SEVERITY

It is possible to cause an integer overflow or underflow in the arithmetic operation.

Source File

- Token.sol

Locations

```
230   event Motd(string message);
231   function setMotd(string _m) onlyOwner {
232   motd = _m;
233   Motd(_m);
234   }
235
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 147

medium SEVERITY

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
146
147   success = controller.approve(msg.sender, _spender, _value);
148   if (success) {
149   Approval(msg.sender, _spender, _value);
150   }
151
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 80

medium SEVERITY

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
79   uint balance = token.balanceOf(this);
80   if (token.transfer(_to, balance)) {
81   logTokenTransfer(_token, _to, balance);
82   return true;
83   }
84
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 168

medium SEVERITY

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
167   if (success) {
168   uint newval = controller.allowance(msg.sender, _spender);
169   Approval(msg.sender, _spender, newval);
170   }
171   }
172
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 158

medium SEVERITY

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
157   if (success) {
158   uint newval = controller.allowance(msg.sender, _spender);
159   Approval(msg.sender, _spender, newval);
160   }
161   }
162
```

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 80

medium SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source File

- Token.sol

Locations

```
79   uint balance = token.balanceOf(this);
80   if (token.transfer(_to, balance)) {
81   logTokenTransfer(_token, _to, balance);
82   return true;
83   }
84
```

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 168

medium SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source File

- Token.sol

Locations

```
167   if (success) {
168   uint newval = controller.allowance(msg.sender, _spender);
169   Approval(msg.sender, _spender, newval);
170   }
171   }
172
```

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 158

medium SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source File

- Token.sol

Locations

```
157   if (success) {
158   uint newval = controller.allowance(msg.sender, _spender);
159   Approval(msg.sender, _spender, newval);
160   }
161   }
162
```

SWC-107 | A CALL TO A USER-SUPPLIED ADDRESS IS EXECUTED.

LINE 79

low SEVERITY

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

- Token.sol

Locations

```
78   IToken token = IToken(_token);
79   uint balance = token.balanceOf(this);
80   if (token.transfer(_to, balance)) {
81   logTokenTransfer(_token, _to, balance);
82   return true;
83
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 168

low SEVERITY

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

- Token.sol

Locations

```
167   if (success) {
168   uint newval = controller.allowance(msg.sender, _spender);
169   Approval(msg.sender, _spender, newval);
170   }
171   }
172
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL

LINE 147

low SEVERITY

The contract account state is accessed after an external call to a fixed 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

- Token.sol

Locations

```
146
147   success = controller.approve(msg.sender, _spender, _value);
148   if (success) {
149   Approval(msg.sender, _spender, _value);
150   }
151
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 158

low SEVERITY

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

- Token.sol

Locations

```
157   if (success) {
158   uint newval = controller.allowance(msg.sender, _spender);
159   Approval(msg.sender, _spender, newval);
160   }
161   }
162
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL

LINE 147

low SEVERITY

The contract account state is accessed after an external call to a fixed 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

- Token.sol

Locations

```
146
147   success = controller.approve(msg.sender, _spender, _value);
148   if (success) {
149   Approval(msg.sender, _spender, _value);
150   }
151
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL

LINE 168

low SEVERITY

The contract account state is accessed after an external call to a fixed 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

- Token.sol

Locations

```
167   if (success) {
168   uint newval = controller.allowance(msg.sender, _spender);
169   Approval(msg.sender, _spender, newval);
170   }
171   }
172
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL

LINE 158

low SEVERITY

The contract account state is accessed after an external call to a fixed 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

- Token.sol

Locations

```
157   if (success) {
158   uint newval = controller.allowance(msg.sender, _spender);
159   Approval(msg.sender, _spender, newval);
160   }
161   }
162
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 93

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
92
93   contract Token is Finalizable, TokenReceivable, SafeMath, EventDefinitions {
94
95   string public name = "FunFair";
96   uint8 public decimals = 8;
97
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 95

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
94
95   string public name = "FunFair";
96   uint8 public decimals = 8;
97   string public symbol = "FUN";
98
99
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 141

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
140
141   function approve(address _spender, uint _value)
142   onlyPayloadSize(2)
143   returns (bool success) {
144   //promote safe user behavior
145
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 195

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
194
195   bool public multilocked;
196
197   modifier notMultilocked {
198   assert(!multilocked);
199
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 115

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
114
115   function totalSupply() constant returns (uint) {
116   return controller.totalSupply();
117   }
118
119
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 132

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
131
132   function transferFrom(address _from, address _to, uint _value)
133   onlyPayloadSize(3)
134   returns (bool success) {
135   success = controller.transferFrom(msg.sender, _from, _to, _value);
136
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 96

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
95   string public name = "FunFair";
96   uint8 public decimals = 8;
97   string public symbol = "FUN";
98
99   Controller controller;
100
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 178

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
177
178   function burn(uint _amount) {
179   controller.burn(msg.sender, _amount);
180   Transfer(msg.sender, 0x0, _amount);
181   }
182
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 59

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
58
59   function finalize() onlyOwner {
60   finalized = true;
61   }
62
63
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 219

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
218
219   function multiApprove(uint[] bits) onlyOwner notMultilocked {
220   if (bits.length % 3 != 0) throw;
221   for (uint i=0; i<bits.length; i += 3) {
222   address owner = address(bits[i]);
223
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 229

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
228
229   string public motd;
230   event Motd(string message);
231   function setMotd(string _m) onlyOwner {
232   motd = _m;
233
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 231

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
230   event Motd(string message);
231   function setMotd(string _m) onlyOwner {
232   motd = _m;
233   Motd(_m);
234   }
235
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 163

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
162
163   function decreaseApproval (address _spender, uint _subtractedValue)
164   onlyPayloadSize(2)
165   returns (bool success) {
166   success = controller.decreaseApproval(msg.sender, _spender, _subtractedValue);
167
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 77

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
76
77   function claimTokens(address _token, address _to) onlyOwner returns (bool) {
78   IToken token = IToken(_token);
79   uint balance = token.balanceOf(this);
80   if (token.transfer(_to, balance)) {
81
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 111

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
110
111   function balanceOf(address a) constant returns (uint) {
112   return controller.balanceOf(a);
113   }
114
115
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 49

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
48
49   function acceptOwnership() {
50   if (msg.sender == newOwner) {
51   owner = newOwner;
52   }
53
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 203

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
202   //do we want lock permanent? I think so.
203   function lockMultis() onlyOwner {
204   multilocked = true;
205   }
206
207
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 116

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
115   function totalSupply() constant returns (uint) {
116   return controller.totalSupply();
117   }
118
119   function allowance(address _owner, address _spender) constant returns (uint) {
120
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 32

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
31   contract Owned {
32   address public owner;
33
34   function Owned() {
35   owner = msg.sender;
36
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 188

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
187
188   function controllerApprove(address _owner, address _spender, uint _value)
189   onlyController {
190   Approval(_owner, _spender, _value);
191   }
192
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 107

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
106
107   function setController(address _c) onlyOwner notFinalized {
108   controller = Controller(_c);
109   }
110
111
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 97

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
96   uint8 public decimals = 8;
97   string public symbol = "FUN";
98
99   Controller controller;
100   address owner;
101
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 183

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
182
183   function controllerTransfer(address _from, address _to, uint _value)
184   onlyController {
185   Transfer(_from, _to, _value);
186   }
187
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 45

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
44
45   function changeOwner(address _newOwner) onlyOwner {
46   newOwner = _newOwner;
47   }
48
49
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 123

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
122
123   function transfer(address _to, uint _value)
124   onlyPayloadSize(2)
125   returns (bool success) {
126   success = controller.transfer(msg.sender, _to, _value);
127
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 209

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
208
209   function multiTransfer(uint[] bits) onlyOwner notMultilocked {
210   if (bits.length % 3 != 0) throw;
211   for (uint i=0; i<bits.length; i += 3) {
212   address from = address(bits[i]);
213
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 179

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
178   function burn(uint _amount) {
179   controller.burn(msg.sender, _amount);
180   Transfer(msg.sender, 0x0, _amount);
181   }
182
183
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 57

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
56   contract Finalizable is Owned {
57   bool public finalized;
58
59   function finalize() onlyOwner {
60   finalized = true;
61
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 153

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
152
153   function increaseApproval (address _spender, uint _addedValue)
154   onlyPayloadSize(2)
155   returns (bool success) {
156   success = controller.increaseApproval(msg.sender, _spender, _addedValue);
157
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 119

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
118
119   function allowance(address _owner, address _spender) constant returns (uint) {
120   return controller.allowance(_owner, _spender);
121   }
122
123
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 145

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
144   //promote safe user behavior
145   if (controller.allowance(msg.sender, _spender) > 0) throw;
146
147   success = controller.approve(msg.sender, _spender, _value);
148   if (success) {
149
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 112

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
111   function balanceOf(address a) constant returns (uint) {
112   return controller.balanceOf(a);
113   }
114
115   function totalSupply() constant returns (uint) {
116
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 79

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
78   IToken token = IToken(_token);
79   uint balance = token.balanceOf(this);
80   if (token.transfer(_to, balance)) {
81   logTokenTransfer(_token, _to, balance);
82   return true;
83
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 135

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
134   returns (bool success) {
135   success = controller.transferFrom(msg.sender, _from, _to, _value);
136   if (success) {
137   Transfer(_from, _to, _value);
138   }
139
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 166

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
165   returns (bool success) {
166   success = controller.decreaseApproval(msg.sender, _spender, _subtractedValue);
167   if (success) {
168   uint newval = controller.allowance(msg.sender, _spender);
169   Approval(msg.sender, _spender, newval);
170
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 120

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
119   function allowance(address _owner, address _spender) constant returns (uint) {
120   return controller.allowance(_owner, _spender);
121   }
122
123   function transfer(address _to, uint _value)
124
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 126

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
125   returns (bool success) {
126   success = controller.transfer(msg.sender, _to, _value);
127   if (success) {
128   Transfer(msg.sender, _to, _value);
129   }
130
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 156

low SEVERITY

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source File

- Token.sol

Locations

```
155   returns (bool success) {
156   success = controller.increaseApproval(msg.sender, _spender, _addedValue);
157   if (success) {
158   uint newval = controller.allowance(msg.sender, _spender);
159   Approval(msg.sender, _spender, newval);
160
```

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 147

low SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they’re part of your own codebase).

Source File

- Token.sol

Locations

```
146
147   success = controller.approve(msg.sender, _spender, _value);
148   if (success) {
149   Approval(msg.sender, _spender, _value);
150   }
151
```

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to, or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Sysfixed's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Sysfixed to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model, or legal compliance.

This is a limited report on our findings based on our analysis, in accordance with good industry practice as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Sysfixed and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Sysfixed) owe no duty of care.

ABOUT US

Sysfixed is a blockchain security certification organization established in 2021 with the objective to provide smart contract security services and verify their correctness in blockchain-based protocols. Sysfixed automatically scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. Sysfixed a comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution—can accurately detect security vulnerabilities to provide an in-depth analysis report. With a vibrant ecosystem of world-class integration partners that amplify developer productivity, Sysfixed can be utilized in all phases of your project's lifecycle. Our team of security experts is dedicated to the research and improvement of our tools and techniques used to fortify your code.