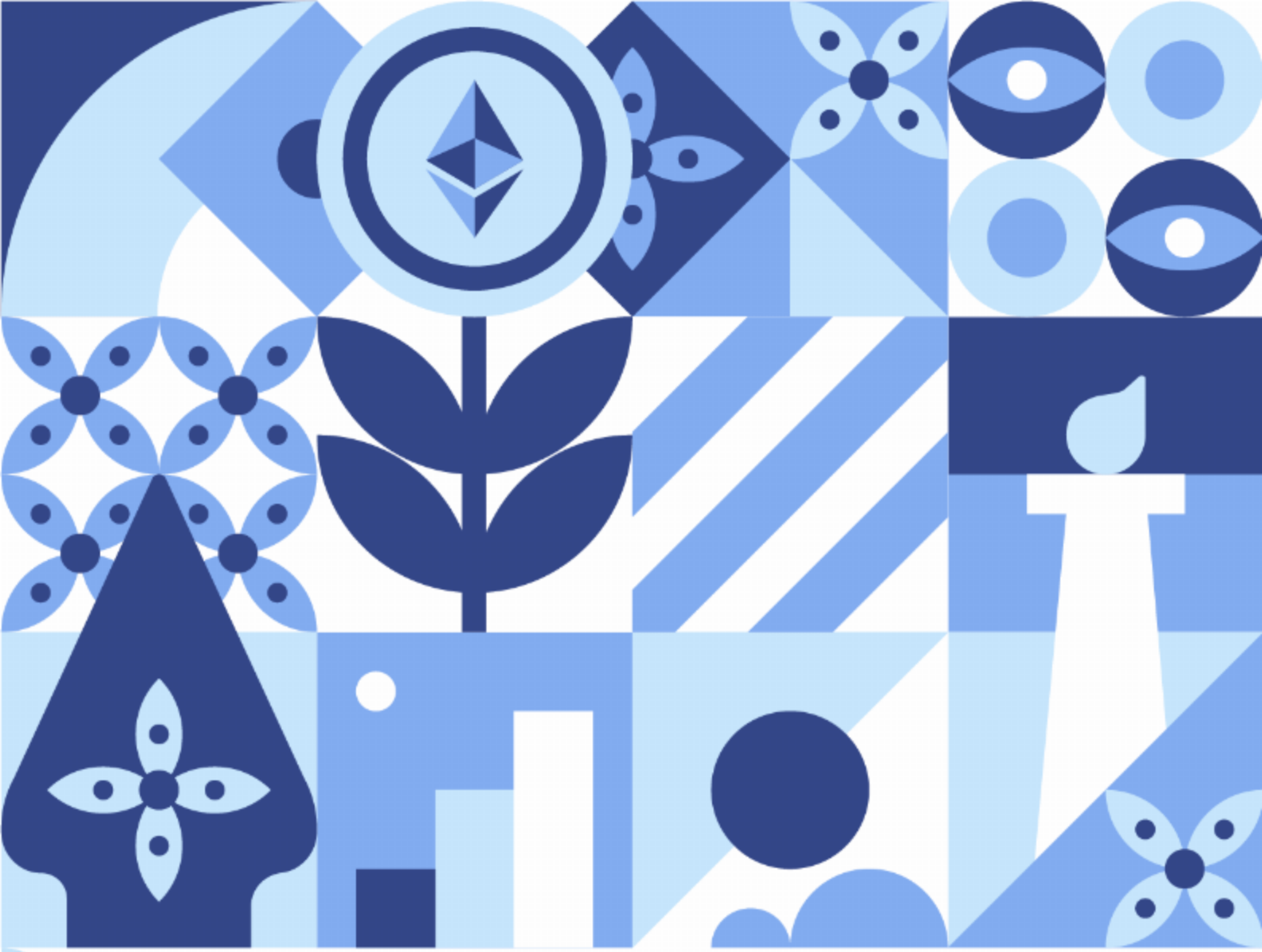

ELTCOIN

Smart Contract Audit Report

SYSFIXED

10 Oct 2017

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
ELTCOIN	ELTCOIN	Ethereum

Addresses

Contract address	0x44197a4c44d6a059297caf6be4f7e172bd56caaf
Contract deployer address	0xdcBd3b5605A525f12819958E0Cb02941C539EAb6

Project Website

https://www.eltcoin.tech/

Codebase

https://etherscan.io/address/0x44197a4c44d6a059297caf6be4f7e172bd56caaf#code

SUMMARY

ELTCOIN is ethereum limited total coin with a community-driven currency that powers an open-sourced library of cool dapps built on Ethereum. the PROJECTS is Get rewarded by contributing to our different projects or by starting your own!

Contract Summary

Documentation Quality

ELTCOIN provides a very good documentation with standard of solidity base code.

- The technical description is provided clearly and structured and also dont have any high risk issue.

Code Quality

The Overall quality of the basecode is standard.

- Standard solidity basecode and rules are already followed by ELTCOIN with the discovery of several low issues.

Test Coverage

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

Audit Findings Summary

- SWC-100 SWC-108 | Explicitly define visibility for all state variables on lines 154, 161, 188, 232, 56 and 97.
- SWC-103 | Pragma statements can be allowed to float when a contract is intended on lines 5.
- 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 32 and 26.
- SWC-111 | It is recommended to use alternatives to the deprecated constructions on lines 12, 18, 25, 30, 44, 78, 89 and 144.

CONCLUSION

We have audited the ELTCOIN project released on October 2017 to discover issues and identify potential security vulnerabilities in ELTCOIN Project. This process is used to find technical issues and security loopholes which might be found in the smart contract.

The security audit report provides a satisfactory result with some low-risk issues.

The issues found in the ELTCOIN smart contract code do not pose a considerable risk. The writing of the contract is close to the standard of writing contracts in general. The low-risk issues found are a floating pragma is set, a state variable visibility is not set, an assertion violation is triggered and the use of "constant" state mutability that has been deprecated. It is recommended to use alternatives to the deprecated constructions.

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.	ISSUE FOUND
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	PASS
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.	ISSUE FOUND
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.	PASS
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.	ISSUE FOUND
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.	PASS
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	Monday Oct 09 2017 07:06:22 GMT+0000 (Coordinated Universal Time)
Finished	Tuesday Oct 10 2017 01:20:07 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	ELTCoin.sol

Detected Issues

ID	Title	Severity	Status
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-100	FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-108	STATE VARIABLE VISIBILITY IS NOT SET.	low	acknowledged
SWC-108	STATE VARIABLE VISIBILITY IS NOT SET.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged
SWC-110	AN ASSERTION VIOLATION WAS TRIGGERED.	low	acknowledged

SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged
SWC-111	USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.	low	acknowledged

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 154

low SEVERITY

The function definition of "increaseApproval" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- ELTCoin.sol

Locations

```
153   */
154   function increaseApproval (address _spender, uint _addedValue) returns (bool
success) {
155   require(isPreSaleReady);
156   allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
157   Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
158
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 161

low SEVERITY

The function definition of "decreaseApproval" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- ELTCoin.sol

Locations

```
160
161   function decreaseApproval (address _spender, uint _subtractedValue) returns (bool
success) {
162   require(isPreSaleReady);
163   uint oldValue = allowed[msg.sender][_spender];
164   if (_subtractedValue > oldValue) {
165
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 188

low SEVERITY

The function definition of "Ownable" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- ELTCoin.sol

Locations

```
187   */
188   function Ownable() {
189   owner = msg.sender;
190   }
191
192
```

SWC-100 | FUNCTION VISIBILITY IS NOT SET (PRIOR TO SOLIDITY 0.5.0)

LINE 232

low SEVERITY

The function definition of "ELTCoin" lacks a visibility specifier. Note that the compiler assumes "public" visibility by default. Function visibility should always be specified explicitly to assure correctness of the code and improve readability.

Source File

- ELTCoin.sol

Locations

```
231   */
232   function ELTCoin() {
233   totalSupply = INITIAL_SUPPLY;
234   balances[msg.sender] = INITIAL_SUPPLY;
235   }
236
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 5

low SEVERITY

The current pragma Solidity directive is ""^0.4.15"". 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

- ELTCoin.sol

Locations

```
4
5   pragma solidity ^0.4.15;
6
7   /**
8   * @title SafeMath
9
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 56

low SEVERITY

It is best practice to set the visibility of state variables explicitly. The default visibility for "balances" is internal. Other possible visibility settings are public and private.

Source File

- ELTCoin.sol

Locations

```
55
56   mapping(address => uint256) balances;
57
58   /**
59   * @dev transfer token for a specified address
60
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 97

low SEVERITY

It is best practice to set the visibility of state variables explicitly. The default visibility for "allowed" is internal. Other possible visibility settings are public and private.

Source File

- ELTCoin.sol

Locations

```
96
97   mapping (address => mapping (address => uint256)) allowed;
98   bool public isPreSaleReady = false;
99
100   /**
101
```

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

- ELTCoin.sol

Locations

```
31   uint256 c = a + b;
32   assert(c >= a);
33   return c;
34   }
35   }
36
```

SWC-110 | AN ASSERTION VIOLATION WAS TRIGGERED.

LINE 26

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

- ELTCoin.sol

Locations

```
25   function sub(uint256 a, uint256 b) internal constant returns (uint256) {
26   assert(b <= a);
27   return a - b;
28   }
29
30
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 12

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
11   library SafeMath {
12   function mul(uint256 a, uint256 b) internal constant returns (uint256) {
13   uint256 c = a * b;
14   assert(a == 0 || c / a == b);
15   return c;
16
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 18

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
17
18   function div(uint256 a, uint256 b) internal constant returns (uint256) {
19   // assert(b > 0); // Solidity automatically throws when dividing by 0
20   uint256 c = a / b;
21   // assert(a == b * c + a % b); // There is no case in which this doesn't hold
22
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 25

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
24
25   function sub(uint256 a, uint256 b) internal constant returns (uint256) {
26   assert(b <= a);
27   return a - b;
28   }
29
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 30

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
29
30   function add(uint256 a, uint256 b) internal constant returns (uint256) {
31   uint256 c = a + b;
32   assert(c >= a);
33   return c;
34
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 44

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
43   uint256 public totalSupply;
44   function balanceOf(address who) public constant returns (uint256);
45   function transfer(address to, uint256 value) public returns (bool);
46   event Transfer(address indexed from, address indexed to, uint256 value);
47   }
48
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 78

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
77   */
78   function balanceOf(address _owner) public constant returns (uint256 balance) {
79   return balances[_owner];
80   }
81
82
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 89

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
88   contract ERC20 is ERC20Basic {
89   function allowance(address owner, address spender) public constant returns
(uint256);
90   function transferFrom(address from, address to, uint256 value) public returns
(bool);
91   function approve(address spender, uint256 value) public returns (bool);
92   event Approval(address indexed owner, address indexed spender, uint256 value);
93
```

SWC-111 | USE OF THE "CONSTANT" STATE MUTABILITY MODIFIER IS DEPRECATED.

LINE 144

low SEVERITY

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

Source File

- ELTCoin.sol

Locations

```
143   */
144   function allowance(address _owner, address _spender) public constant returns
(uint256 remaining) {
145   return allowed[_owner][_spender];
146   }
147
148
```

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.