Chi Gastoken by 1inch

Smart Contract Audit Report

SYSFIXED

13 Feb 2021

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
Chi Gastoken by 1inch	CHI	Binance Smart Chain

Addresses

Contract address	0x0000000000004946c0e9f43f4dee607b0ef1fa1c
Contract deployer address	0x7E1E3334130355799F833ffec2D731BCa3E68aF6

Project Website

https://app.1inch.io/

Codebase

https://bscscan.com/address/0x0000000000004946c0e9f43f4dee607b0ef1fa1c#code

SUMMARY

An entry point to the 1inch Network's tech. The 1inch dApp is the #1 DeFi aggregator, offering access to the most liquidity and the best token swap rates on various DEXes, with unique features, including partial fill, the Chi gas token, and the ability to find the best swap paths across multiple liquidity sources.

Contract Summary

Documentation Quality

Chi Gastoken by 1inch 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 Chi Gastoken by 1inch.

Test Coverage

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

Audit Findings Summary

- SWC-103 | Pragma statements can be allowed to float when a contract is intended on lines 203, 227, 375 and 425.
- SWC-104 | It is recommended to use handle at low-level call methods on lines 468.
- SWC-107 | It is recommended to use a reentrancy lock, reentrancy weaknesses detected on lines 468, 468 and 468.
- SWC-113 SWC-128 | It is recommended to implement the contract logic to handle failed calls and block gas limit on lines 468, 468 and 468.

CONCLUSION

We have audited the Chi Gastoken by 1inch project released on June 2021 to find issues and identify potential security vulnerabilities in the Chi Gastoken by 1inch project. This process is used to find technical issues and security loopholes that may be found in smart contracts.

The security audit report yielded unsatisfactory results, discovering medium-risk and low-risk issues.

Writing a contract that does not follow the Solidity style guide can pose a significant risk. The serious and low problems we found in the smart contract are Unchecked return values from the low-level external calls and multiple calls being executed in the same transaction. For the low-risk issues, a floating pragma is set and Read or Write to persistent state following the external call. Low-level external calls return a boolean value. If the callee halts with an exception, 'false' is returned, and execution continues in the caller. The caller should check whether an exception happened and react accordingly to avoid unexpected behavior. For example, wrapping low-level external calls in require() is often desirable, so the transaction is reverted if the call fails. This call is executed following another call within the same transaction. The call may never get executed if an initial 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 ensure that all callees can be trusted (i.e. they're part of your codebase).

We were recommended to keep being aware of investing in this 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.	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.	ISSUE FOUND
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.	PASS
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	Friday Feb 12 2021 11:43:02 GMT+0000 (Coordinated Universal Time)
Finished	Saturday Feb 13 2021 03:07:08 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	ChiToken.sol

Detected Issues

ID	Title	Severity	Status
SWC-104	UNCHECKED RETURN VALUE FROM LOW-LEVEL EXTERNAL CALL.	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-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	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-104 | UNCHECKED RETURN VALUE FROM LOW-LEVEL EXTERNAL CALL.

LINE 468

medium SEVERITY

Low-level external calls return a boolean value. If the callee halts with an exception, 'false' is returned and execution continues in the caller. The caller should check whether an exception happened and react accordingly to avoid unexpected behavior. For example it is often desirable to wrap low-level external calls in require() so the transaction is reverted if the call fails.

Source File

- ChiToken.sol

Locations

```
467   }
468   }
469
```

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

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

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

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

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

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 203

low SEVERITY

The current pragma Solidity directive is ""^0.6.0"". 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

- ChiToken.sol

Locations

```
202   function div(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
203   // Solidity only automatically asserts when dividing by 0
204   require(b > 0, errorMessage);
205   uint256 c = a / b;
206   // assert(a == b * c + a % b); // There is no case in which this doesn't hold
207
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 227

low SEVERITY

The current pragma Solidity directive is ""^0.6.0"". 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

- ChiToken.sol

Locations

```
226   /**
227   * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
modulo),
228   * Reverts with custom message when dividing by zero.
229   *
230   * Counterpart to Solidity's `%` operator. This function uses a `revert`
231
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 375

low SEVERITY

The current pragma Solidity directive is ""^0.6.0"". 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

- ChiToken.sol

Locations

```
374
375   function _burn(address account, uint256 amount) internal {
376   _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds
balance");
377   emit Transfer(account, address(0), amount);
378   }
379
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 425

low SEVERITY

The current pragma Solidity directive is ""^0.6.0"". 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

- ChiToken.sol

Locations

```
424   for {let i := and(value, 0x1F)} i {i := sub(i, 1)} {
425   pop(create2(0, 0, 30, offset))
426   offset := add(offset, 1)
427   }
428   }
429
```

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

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 468

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

- ChiToken.sol

Locations

```
467   }
468   }
469
```

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.