SMART CONTRACT AUDIT

zokyo

February 21th 2023 | v. 1.0

Security Audit Score

PASS

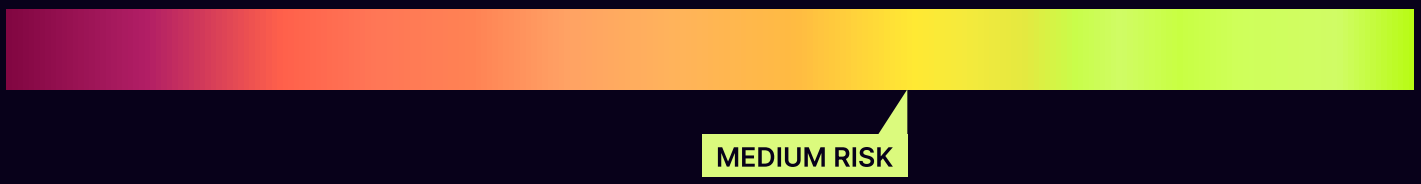
Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

TECHNICAL SUMMARY

This document outlines the overall security of the Unity Chain smart contracts evaluated by the Zokyo Security team.

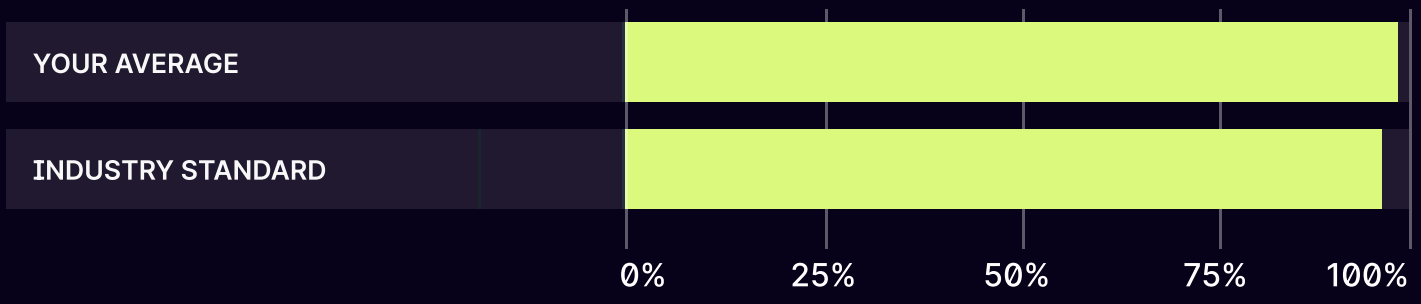
The scope of this audit was to analyze and document the Unity Chain smart contracts codebase for quality, security, and correctness.

Contract Status



There were 2 critical issues found during the audit.

Testable Code



100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Unity Chain team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Unity Chain repository: https://polygonscan.com/token/0xBe788FeAe3C004EE759149C55Db2D173407633f2#code

Last commit: https://polygonscan.com token/0xBe788FeAe3C004EE759149C55Db2D173407633f2#code

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- UnityChain
- BaseToken
- MintableBaseToken
- Address
- SafeERC20
- SafeMath

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Unity Chain smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Truffle/Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contracts by industry leaders.
03	Testing contracts logic against common and uncommon attack vectors.
04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo auditing team has run a deep investigation of Unity Chain's smart contracts. During the auditing process, there were issues with critical, high, medium severity and one informational issue found. All the issues acknowledged, as Unity Chain team know about all found vulnerabilities. They are described in detail in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Unity Chain team and the Unity Chain team is aware of it, but they have chosen to not solved it. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status.sol
1	Minter can burn tokens of other accounts	Critical	Acknowledged
2	Reentrancy in transfer(), mint(), burn(), addNonStakingAccount() and removeNonStakingAccount functions	Critical	Acknowledged
3	Token name and symbol can be changed	High	Acknowledged
4	Possibility of manipulation of rewards	High	Acknowledged
5	Missing zero address check in setYieldTrackers()	Medium	Acknowledged
6	yieldTrackers can be changed anytime	Medium	Acknowledged
7	Missing zero address check in setGov()	Medium	Acknowledged
8	For loop over dynamic array	Medium	Acknowledged
9	ERC20 approve race condition	Medium	Acknowledged
10	Missing events for critical functions	Informational	Acknowledged

CRITICAL | ACKNOWLEDGED

Minter can burn tokens of other accounts

The minter can burn tokens from any address, even from addresses whose private key it does not have.

Recommendation:
It is advised to not allow the minter to burn the tokens directly from other accounts. Review the business and operational logic for the same. Refer this Openzeppelin Burnable ERC20 contract, which allows only the tokens of the msg.sender to be burnt.

CRITICAL | ACKNOWLEDGED

Reentrancy in transfer(), mint(), burn(), addNonStakingAccount() and removeNonStakingAccount functions

The functions _transfer(), _mint(), _burn(), addNonStakingAccount() and removeNonStakingAccount() are susceptible to reentrancy because an external call to YieldTracker contract(out of scope of this audit) is made via the updateRewards() function.

Although addNonStakingAccount() and removeNonStakingAccount() is an onlyAdmin function, and mint() and burn() is an onlyMinter function, it is expected that the function follows a checks-effects-interaction pattern to avoid any kind of reentrancy as a security best practice.

Recommendation:
It is advised to move the _updateRewards() function call on line: 192 and 193 in _transfer to the end of the function instead. Also, it is advised to move the _updateRewards() function call on line: 157 to the end of the _mint() function instead in order to mitigate this issue. Similarly, it is advised to move the _updateRewards() function call on line: 172 to the end of the _burn() function instead, in order to mitigate this issue. The same is advised for the addNonStakingAccount() and removeNonStakingAccount() functions. This will ensure checks-effects interactions pattern to be followed.

HIGH | ACKNOWLEDGED

Token name and symbol can be changed

The setInfo() function can be used to change the name and symbol of the token even after the token has been deployed. This is not advisable because the gov can change the name and symbol of the token anytime.

Recommendation:
Add a requirement check to prevent calling the setInfo() function more than once.

HIGH | ACKNOWLEDGED

Possibility of manipulation of rewards

There is a possibility that the transferring of tokens to yourself can lead to increases in rewards due to the call to _updateRewards() function. The existence and impact of this issue is unknown, as the YieldTracker contract is out of scope of this audit.

Recommendation:
It is advised to review the business and operational logic for the same.

MEDIUM | ACKNOWLEDGED

Missing zero address check in setYieldTrackers()

The yieldTrackers array can have zero address being set in its array of addresses via the setYieldTrackers function. This could lead to revert on function calls to recoverClaim() and claim() functions.

Recommendation:
It is advised to add the missing zero address checks for yieldTrackers. It is also advised to review business and operational logic.

MEDIUM | ACKNOWLEDGED

yieldTrackers can be changed anytime

It should also be noted that the yieldTrackers addresses can be changed by the gov which could lead to undiscovered or unknown bugs such as Denial of Service(which depends on the YieldTracker contracts that are out of scope of this audit).

Recommendation:
It is advised to provide necessary checks to prevent this from happening and make the process of changing of yieldTrackers addresses more decentralized.

MEDIUM | ACKNOWLEDGED

Missing zero address check in setGov()

There is missing zero address check for _gov in setGov() function. This could lead to gov being accidentally set to zero address and its access to gov thus being lost forever. This is because only gov can change the address of gov, but once the gov is set as a zero address, the gov address can never be changed. This would result in all the onlyGov becoming uncallable.

Recommendation:
It is advised to add missing zero address check for the gov address.

MEDIUM | ACKNOWLEDGED

For loop over dynamic array

The functions _updateRewards(), recoverClaim() and claim() function have a for loop over a dynamic array. This can lead to out of gas issues if the array becomes too large.

Recommendation:
It is advised to add a check on the max length that the array can take.

MEDIUM | ACKNOWLEDGED

ERC20 approve race condition

The ERC20 contract has a race condition in its approve function, which can be exploited by attackers via a frontrun. For example, let's say Alice approves Bob for 10 tokens in transaction 1, but then changes the approve to 5 tokens in transaction 2. Bob can exploit this by spending the 10 tokens in a transaction 3 and executing it before transaction 2 by paying a higher gas fee.

Recommendation:
This can be mitigated by using the increaseAllowance() and decreaseAllowance() functions along with the approve function, such as in the Openzeppelin version of ERC20 tokens. Refer to this and this for more information on its implementation.

INFORMATIONAL | ACKNOWLEDGED

Missing events for critical functions

Missing events for critical events: setInPrivateTransferMode(), setHandler(), addAdmin(), removeAdmin(), addNonStakingAccount() and removeNonStakingAccount()

Recommendation:
Add and emit event for the above mentioned functions carrying out critical operations. This would be a best practice for offchain monitoring.

	UnityChain BaseToken MintableBaseToken Address SafeERC20 SafeMath
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Unity Chain in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the Unity Chain contract requirements for details about issuance amounts and how the system handles these.

Contract: BaseToken

setGov()
✓ sets gov to a new address (94ms)
setInfo()
✓ setS token name and symbol (84ms)
addAdmin()
✓ adds address to list of admins (73ms)
withdrawToken()
✓ transfers token to receipient (163ms)
setInPrivateTransferMode()
✓ toggles private mode
setHandler()
✓ sets an address as handler
addNonStakingAccount()
✓ adds a non staking account (117ms)
removeNonStakingAccount
✓ removes non staking account (119ms)
recoverClaim()
✓ recovers claims (63ms)
Claim
✓ Claim() (44ms)
totalStaked()
✓ Should return correct amount of total staked
stakedBalance()
✓ SHould return staked balance of account (62ms)
✓ Should return staked balance of account
transfer()
✓ Should transfer tokens (140ms)
allowance()
✓ Should return allowance for an approved spender

```
    transferFrom()
    ✓ should transferFrom approver to spender (45ms)
    ✓ transferFrom (57ms)
    balanceOf()
    ✓ Returns balance of an address
Contract: MintableBaseToken
    setMinter
    ✓ setMinter
    mint
    ✓ mint (72ms)
    burn
    ✓ burn (100ms)
Contract: UnityChain
    id()
    ✓ Returns id

22 passing (4s)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
BaseToken.sol	100	96.55	100	100	
UnityChain.sol	100	100	100	100	
MintableBaseToken.sol	100	100	100	100	
FILE	**100**	**96.97**	**100**	**100**	

We are grateful for the opportunity to work with the Unity Chain team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Unity Chain team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo