QuillAudits

Audit Report
February, 2023

For

ØSPACE

Table of Content

Executive Summary

Project Name xEVMToken by SpaceFi

Overview xEVMToken by SpaceFi is an Evmos Token involved in vesting. SpaceFi is a cross chain web3 platform e.g on Evmos and zkSync. XEvmosToken is an ERC20 type Upgradeable token which has functionality for 'whitelisting' addresses allowed to transfer tokens; 'redeem' functionality to determine compensation amounts to receive when vesting ended and convert XEvmosToken to Evmos; 'convert' to convert Evmos to XEvmosToken.

Scope of Audit The scope of this audit was to analyse XEvmos Token Contract codebase for quality, security, and correctness. This included testing of smart contracts to ensure proper logic was followed, manual analysis ,checking for bugs and vulnerabilities, checks for dead code, checks for code style, security and more. The audited contracts are as follows:

Git Repo link: *https://github.com/SpaceFinance/space-contract/blob/main/XEvmosToken.sol*
Git Branch: main
Commit Hash: 4256795517bd4ea1df1a51cf426b1b8ea8eb06b2

Fixed in: *https://github.com/SpaceFinance/space-contract/blob/main/XEvmosToken.sol*
Git Branch: main
Commit Hash: 094722db24d55dc7fbb917552a8b802329eb4ceb

High Medium Low Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	1
Resolved Issues	1	1	1	6

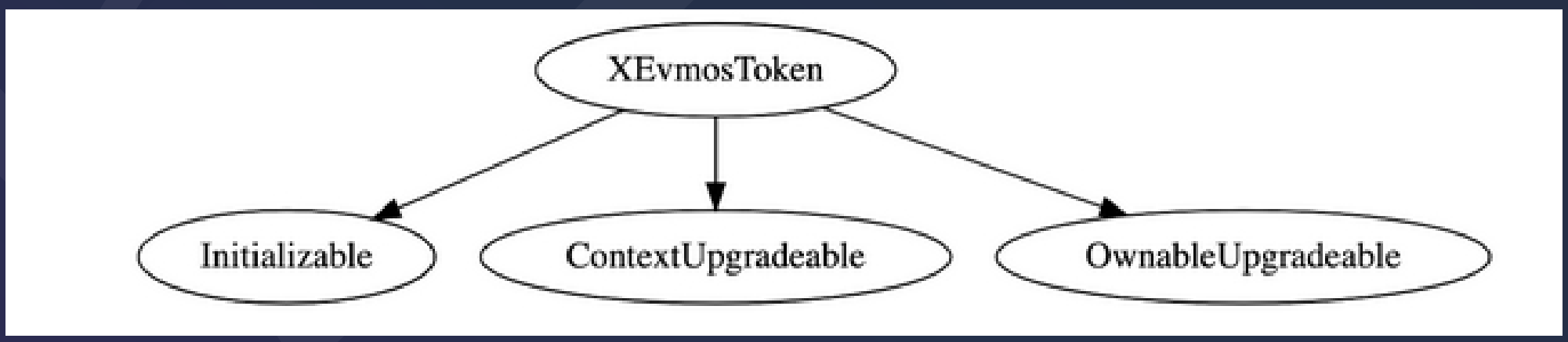
Contracts Information

Contract	Lines	Complexity Score	Capabilities
contracts/XEvmosToken.sol	694	185	paying transactions, unchecked blocks

Dependencies

Dependency / Import Path	Count
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol	1
@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol	1
@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol	1
@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol	1
@openzeppelin/contracts-upgradeable/utils/math/SafeMathUpgradeable.sol	1

Inheritance Graph



Call Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
XEvmosToken
initialize
name
symbol
decimals
totalSupply
balanceOf
unbondingBalanceOf
farmStakingBalanceOf
approve
allowance
decreaseAllowance
increaseAllowance
transferFrom
convert
updateTransferWhitelist
updateRedeemSettings
transfer
cancelRedeem
finalizeRedeem
redeem
getUserRedeemsLength
isTransferWhitelisted
validateRedeem
<Receive Ether>
<Fallback>
__ERC20_init
__Ownable_init_unchained
__Context_init_unchained
__ERC20_init_unchained
_approve
_msgSender
_convert
_transfer
_deleteRedeemEntry
_finalizeRedeem
_currentBlockTimestamp
getEvmosByVestingDuration
_mint
_burn
_afterTokenTransfer
_beforeTokenTransfer
amount
mul

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

High Severity Issues

1. Reentrancy

Line 415
function finalizeRedeem(uint256 redeemIndex) external validateRedeem(msg.sender, redeemIndex) {

Above function does not follow checks effects interaction patterns where an external call is made in internal function Line 444 _finalizeRedeem() and an attacker contract can reenter the finalizeRedeem function again to get same EvmosAmount on the same RedeemInfo in userRedeems that has not yet been deleted which only happens in Line 436 _deleteRedeemEntry(redeemIndex);

Recommendation

It is recommended to make use of checks and effects interaction patterns to protect the above function or make use of upgradeable safe ReentrancyGuard nonReentrant protection.

Auditor's Response: nonReentrant modifier applied

Status

Resolved

Medium Severity Issues

2. Centralization Risks / Overpowered Ownership

Without clear documentation if the Ownable contract will make use of decentralised control such as MultiSig. The owner has control to update whitelists, update redeem settings, upgrade contracts. Any compromise to the Owner account may allow the hacker to take advantage of this.

Recommendation

We advise the client to carefully manage the Owner account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., Multisignature wallets

Auditor's Response: Administrator authority will be handed over to MultiSig

Status

Resolved

Low Severity Issues

3. Uninitialized and unused logic

State variable Line 56 mapping(address => uint256) private _farmStakingBalances; is never initialised and or updated in the contracts or used in any other functionality. Function Line 155 function farmStakingBalancOf(address) will always return 0 for all addresses.

Recommendation

It is recommended to review this functionality as it may be indicative of missing logic or to remove if it is not used.

Auditor's Response: Unused code removed

Status

Resolved

Informational Issues

4. Floating pragma

Contracts make use of pragma ^0.8.0 which allows for variable solidity compiler versions. This can result in different versions being used for testing and production.

Recommendation

It is recommended to deploy contracts using the same compiler version/flags with which they have been tested. The solidity version must be fixed by locking the pragma by avoiding using ^. Consider using later stable stable versions like pragma 0.8.14, 0.8.15, 0.8.16

Auditor's Response: Later version of Solidity ^0.8.12 used. Floating pragma still applied

Status

Resolved

5. Natspec

The code is lacking complete commenting. Comments inline, particularly in Natspec format help to clarify what the code does especially if it uses @params, @return and more.

Recommendation

It is recommended to use Natspec, a form of comments in Solidity that provides rich documentation for functions, return variables and more.
E.g */// @notice Returns the amount of leaves the tree has.*
/// @dev Returns only a fixed number.
/// @param ....

Auditor's Response: Functions @params and @return applied

Status

Resolved

6. Order Layout

The ordering and layout of functions, constructors, variables and general layout of the Solidity files may need to be changed

Recommendation

It is recommended to follow the widely accepted style in how to layout a file from **Pragma -> Imports -> Interfaces -. Libraries -> Type Declarations -> State Variables -> Constructor -> Fallback -> Receive => External Functions -> Public Functions -> Internal Functions -> Private Functions**. See Solidity styleguide *https://docs.soliditylang.org/en/v0.5.3/style-guide.html.* Not only does the above improve code readability and maintainability but by moving external functions to the top if they are most frequently called functions, it can save on gas. It is recommended to consider ordering functions based on how frequently they are expected to be called to save on gas. However there is a tradeoff with readability so consider fully.

Auditor's Response: Order is for readability

Status

Resolved

7. Unused Imports or unnecessary imports

The contract make use of SafeMath library
i*mport "@openzeppelin/contracts/utils/math/SafeMath.sol"*;
This is never used in the contracts. Additionally SafeMathUpgradeable the appropriate one for upgradeable contracts is also imported.

Contract makes use of the SafeMathUpgradeable library.
import "@openzeppelin/contracts-upgradeable/utils/math/SafeMathUpgradeable.sol";
However the contracts use Solidity version 0.8.0 and above which have overflow and underflow checks by default.

Contract makes use of SafeERC20Upgradeable library
import "@openzeppelin/contracts-upgradeable/utils/math/SafeMathUpgradeable.sol";
This is attached on IERC20Upgradeable Line 46 which is never imported or used in contracts. SafeERC20 is a wrapper that solves the problem of needing to handle return values on tokens. In contracts it appears only this token is the one being used and or created contracts.

Recommendation

It is recommended to consider removing SafeMath imports. It is recommended to consider removing SafeMathUpgradeable imports. It is recommended to consider removing SafeERC20Upgradeable imports.

Auditor's Response: Unused imports removed. SafeMathUpgradeable retained

Status

Resolved

8. Unindexed event parameters

Certain events like Transfer, Approval, SetTransferWhitelist are lacking indexed parameters for addresses. This is especially important for Transfer and Approve which in ERC20 conformance, standards and best practices have indexed parameters.

Recommendation

It is recommended to add indexed event parameters to Transfer and Approval
event Transfer(address indexed sender, address recipient, uint256 amount);
event Approval(address indexed sender, address recipient, uint256 amount);
This is why earlier it was suggested to inherit from ERC20Upgradeable to ensure best practices, standards, conformance and limit errors or critical omissions for XEvmosERC20 Token.

Auditor's Response: indexing applied on Transfer and Approval events address _sender; must also be applied to address _recipient

Status

Partially Fixed

9. Function parameters use of underscore _

It appears some code parts for function parameters make use of _param, other parts param_ wheres others just param

Recommendation

It is recommended for code to be consistent and make use of _param for function parameters across all functions e.g.
function updateTransferWhitelist(address _account, bool _add)
Ensure that necessary adjustments are made to state variables that may have used _param to param

Auditor's Response: Use of _ (underscore) applied consistently

Status

Resolved

10. Boolean comparisons

Function *_beforeTokenTransfer(address _from, address _to, uint256 _amount) internal view.*
does boolean comparisons
_transferWhitelist[_from] == true and _transferWhitelist[_to] == true

Recommendation

It is recommended in the if statement to use *_transferWhitelist[_from]* , *_transferWhitelist[_to]* only as these values are already booleans

Status

Resolved

Functional Testing

Some of the tests performed are mentioned below:

✓ initialize	function	PASS
✓ __ERC20_init	modifier	PASS
✓ __ERC20_init_unchained	function	PASS
✓ name	function	PASS
✓ symbol	function	PASS
✓ decimals	function	PASS
✓ totalSupply	function	PASS
✓ balanceOf	function	PASS
✓ unbondingBalancOf	function	PASS
✓ farmStakingBalancOf	function	PASS
✓ transfer	function	PASS
✓ allowance	function	PASS
✓ approve	function	PASS
✓ transferFrom	function	PASS
✓ increaseAllowance	function	PASS
✓ decreaseAllowance	function	PASS
✓ _transfer	function	PASS
✓ _mint	function	PASS
✓ _burn	function	PASS
✓ _approve	function	PASS
✓ updateTransferWhitelist	function	PASS
✓ updateRedeemSetting	function	PASS
✓ convert	function	PASS
✓ _convert	function	PASS
✓ redeem	function	PASS
✓ finalizeRedeem	function	PASS
✓ cancelRedeem	function	PASS
✓ _finalizeRedeem	function	PASS
✓ _deleteRedeemEntry	function	PASS
✓ getUserRedeemsLength	function	PASS
✓ getEvmosByVestingDuration	function	PASS
✓ isTransferWhitelisted	function	PASS
✓ validateRedeem	modifier	PASS
✓ _beforeTokenTransfer	modifier	PASS
✓ _currentBlockTimestamp	function	PASS

Automated Tests

No major issues were found. Some false positive errors were reported by the tools.
All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the xEVMToken by SpaceFi. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low Severity and Informational nature were found in this audit. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the SpaceFi Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the SpaceFi Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
February, 2023

For

SPACE

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com