QuillAudits

Audit Report
May, 2024

For

Zoniqx

Table of Content

Table of Content

Executive Summary

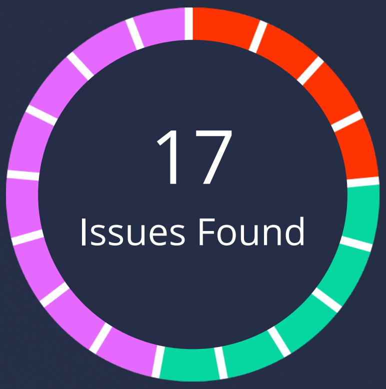
Project Name	Zoniqx
Project URL	https://www.zoniqx.com/
Overview	Zoniqx (formerly known as Tassets) is a pioneering fintech company based in Silicon Valley that specializes in transforming real-world assets into digital tokenized assets through its advanced Tokenized Asset Lifecycle Management (TALM) solutions
Timeline	06th May 2024 - 13th May 2024
Updated Code Received	28th May 2024
Second Review	29th May 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the Zoniqx Contracts for quality, security, and correctness.
Blockchain	EVM
Source Code	https://github.com/tassets/Tassets_ERC7518
Branch	Main

Executive Summary

Contracts In-Scope

```
niqx_ERC7518-main/cZoontracts/
  ├──── chainlink
  |     └──── AggregatorV3Interface.sol
  ├──── CompliTo.sol
  ├──── ERC1155Mod.sol
  ├──── ERC7518MarketPlace.sol
  ├──── ERC7518.sol
  ├──── interfaces
  |     ├──── ICompliTo.sol
  |     ├──── IERC7518.sol
  |     └──── IMarketPlace.sol
  ├──── module
  |     ├──── ERC1155AllowanceWrapper.sol
  |     ├──── ERC1155Permit.sol
  |     ├──── ERC2771Context.sol
  |     ├──── ERC2771ContextUpgradeable.sol
  |     ├──── FreezeAddress.sol
  |     ├──── IssuerFactory.sol
  |     ├──── Nonce.sol
  |     ├──── PaymentTracker.sol
  |     ├──── Payout.sol
  |     ├──── RestrictTransfer.sol
  |     └──── TokenLock.sol
  └──── STOBaseV2.sol
```

Number of Security Issues per Severity



🟥 High
🟨 Medium
🟩 Low
🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	1	0	0	0
Resolved Issues	3	0	5	8

Checked Vulnerabilities

✓ Re-entrancy	✓ Compiler version not fixed
✓ Arbitrary write to storage	✓ Address hardcoded
✓ Centralization of control	✓ Divide before multiply
✓ Ether theft	✓ Integer overflow/underflow
✓ Improper or missing events	✓ ERC's conformance
✓ Logical issues and flaws	✓ Dangerous strict equalities
✓ Arithmetic Correctness	✓ Tautology or contradiction
✓ Race conditions/front running	✓ Return values of low-level calls
✓ SWC Registry	✓ Missing Zero Address Validation
✓ Re-entrancy	✓ Private modifier
✓ Timestamp Dependence	✓ Revert/require functions
✓ Gas Limit and Loops	✓ Multiple Sends
✓ Exception Disorder	✓ Using suicide
✓ Gasless Send	✓ Using delegatecall
✓ Use of tx.origin	✓ Upgradeable safety
✓ Malicious libraries	✓ Using throw

Checked Vulnerabilities

- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of Zoniqx, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of various 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 Solana programs.

Structural Analysis

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

Static Analysis

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

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, and 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 Solana programs in production. Checks were done to know how much gas gets consumed and the possibilities of optimising code to reduce gas consumption.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

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 Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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 four 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.

High Severity Issues

1. Incorrect Allowance Verification and Handling

Path

ERC7518.sol

Function

safeTransferFrom

Description

The safeTransferFrom method incorrectly handles the allowance verification due to a parameter mismatch and a logical error in the associated _spendAllowance method. This issue arises because the function uses a hardcoded zero (0) for the token ID instead of the actual token ID provided to the function, and there's a mix-up between the amount and id parameters in _spendAllowance.

Impact

This flaw allows users (e.g., delegated operators) to potentially transfer tokens beyond their authorized allowance, leading to unauthorized asset movement and breach of trust assumptions in the token management system. It undermines the ERC-1155 standard's security guarantees regarding token transfers and allowance management.

Code Affected

1. safeTransferFrom function in ERC7518.sol:

```
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) public virtual override whenNotPaused {
    canTransfer(from, to, id, amount, data);
    if (from != _msgSender()) {
        _spendAllowance(from, _msgSender(), amount, 0);  // Incorrect ID used here
    }
    _increaseNonceBasedOnSignature(from, data, id);
    return super._safeTransferFrom(from, to, id, amount, data);
}
```

2. _spendAllowance function in ERC1155AllowanceWrapper.sol:

```
function _spendAllowance(
    address owner,
    address spender,
    uint256 amount,
    uint256 id  // This parameter is mistakenly being treated as 'amount'
) internal virtual {
    uint256 currentAllowance = allowance(owner, spender, id);
    if (currentAllowance != type(uint256).max) {
        require(
            currentAllowance >= amount,
            "ERC1155: insufficient allowance"
        );
        unchecked {
            _approve(owner, spender, currentAllowance - amount, id);  // Parameter misuse
        }
    }
}
```

Code snippets above illustrate the incorrect implementation within the safeTransferFrom function where the token ID is passed as 0 and the subsequent logical errors in the _spendAllowance method where the token ID and amount parameters are misused. This misalignment allows for a potential exploitation scenario where allowances can be bypassed, posing a severe security risk.

Proof of Concept with Attack Scenario

Bob has been given a limited allowance to manage Alice's tokens. However, due to the bug, Bob is able to transfer more tokens than allowed from Alice's account to Charlie's account.

```
// Assume Alice has approved Bob to manage 100 tokens of id1.
// Bob uses the following function to exploit the vulnerability and transfer 1000 tokens
instead.
function exploitTransferFromAliceToCharlie() public {
   uint256 maliciousAmount = 1000;  // Amount greater than his actual allowance
   bytes memory data;  // Assume necessary data for a valid transaction
   erc1155Contract.safeTransferFrom(aliceAddress, charlieAddress, id1,
maliciousAmount, data);
}
```

Recommendation

1. Correctly implement the _spendAllowance to ensure that the correct token ID (id) and the actual amount (amount) are used for allowance verification and update.
2. Fix the safeTransferFrom function to use the passed token ID (id) instead of a hardcoded zero (0) when calling _spendAllowance. This aligns the allowance management with the token actually being transferred, preventing unauthorized operations.

Status

Resolved

2. Unauthorized Token Locking and Unlocking

Path

TokenLock.sol

Function

lockTokens

Description

The lockTokens method in the contract lacks necessary authorization checks to verify if the caller has the right to lock tokens for the specified account. This allows any user to lock any amount of tokens belonging to any account, leading to potential misuse where tokens can be frozen maliciously without the owner's permission.

Impact

This vulnerability allows unauthorized users to manipulate the token lock state, leading to possible denial of service for token holders as their tokens could be locked and made non-transferable without their approval. The impact is severe as it affects the fundamental rights of token holders to control their assets.

Code Affected

The following code snippet illustrates the vulnerable implementation where insufficient input validations make the contract's state vulnerable to unauthorized manipulation.

```
function lockTokens(
    address account,
    uint id,
    uint amount,
    uint releaseTime
) public override returns (bool) {
    require(
        transferableBalance(account, id) >= amount,
        "Insufficient balance"
    );
    _lockTokens(account, id, amount, releaseTime);
    return true;
}
```

Proof of Concept with Attack Scenario

An attacker, by exploiting this vulnerability, can lock a large amount of tokens of legitimate users, thereby freezing their assets.

```
// Assume the attacker knows the address and token ID of a victim
    function testUnauthorizedLockingAndUnlocking() public {
        startHoax(attacker);

        erc7518Token.lockTokens(victim, 1, 90, 1 days);
        skip(2 days);
        erc7518Token.unlockTokens(victim, 1);

        vm.stopPrank();
    }
```

Recommendation

1. Implement authorization checks within the lockTokens method to ensure that only the token owner or an authorized delegate (approved via a separate mechanism) can initiate locks on tokens. This could be implemented by checking if msg.sender == account or msg.sender has been granted a specific locking role or permission.
2. Consider introducing a role-based access control mechanism for sensitive functions such as token locking, which involves financial and operational risks.

Status

Resolved

3. Ineffective freezing operation leads to multiple issues

Path

contracts/ERC7518.sol :: freeze()

Description

According to the specifications of the ERC7518 smart contract, the freeze functionality is designed to freeze the operations of any user to restrict the transfers and usage of tokens. However the current implementation has certain limitations that cause hindrance for normal and power users;

1. A frozen account can still have its funds force transfered by an authorized role which goes against the specifications of the ERC7518 token standard as stated below (an excerpt from the erc7518 specs): > MUST revert if to address is Frozen.
2. A frozen account can stil recieve payouts by a payout role power user for its profits generated from locked funds which should not be possible given the specifications of the ERC7518 token standard as stated below (an excerpt from the erc7518 specs for freezeAddress()): > MUST prevent account to transfer and payout.
3. The authorized roles like the default admin, payout role, minter role etc can also be frozen to perform any operations concerning the tokens. This functionality can prove to be vulnerable even if the freezing limitations are made applicable according to their correct definitions when certain normal or power users are restricted for the token standard's functionalities like transfer and usage of tokens as stated below (an excerpt from the erc7518 specs): > MUST revert if the from address is not Frozen.

Proof of Concept

1. A frozen account can have its funds force transfered as the following test confirms.

~~~
```solidity
function testFrozenAccountTokenForceTransfer() public {
    erc7518Token.transferableBalance(acc1, 1);
    startHoax(owner);
    erc7518Token.freeze(owner, "");
    erc7518Token.freeze(acc1, "");
    erc7518Token.forceTransfer(acc1, acc3, 1, 50, data);
    vm.stopPrank();
}
~~~

2. Since the freezing functionality does not work correctly according to the specs of the protocol, a frozen account is still allowed to receive the payouts which should not be possible given the nature of freezing operation.

```solidity
function testFrozenAccountPayout() public {
    erc7518Token.transferableBalance(acc1, 1);
    startHoax(owner);
    erc7518Token.freeze(acc1, "");
    erc7518Token.payout(acc1, 10);
    vm.stopPrank();
}
```

3. Similarly, the test below proves the attack scenario where an escalation of privilege can backfire to the authorized roles.

```solidity
function testFreezeAndUnfreeze() public {
    startHoax(owner);
    erc7518Token.freeze(owner, "");
    erc7518Token.freeze(acc1, ""); // tokenHolderUser
    erc7518Token.freeze(attacker, "");
    erc7518Token.unFreeze(owner, "");
    erc7518Token.unFreeze(acc1, "");
    erc7518Token.unFreeze(attacker, "");
    vm.stopPrank();
}
```

Recommendation

Correctly implement appropriate access control measures to ensure that

1. the specs are respected across all the functionalities available especially for transfers and payouts,
2. only the authorized addresses can remain unfrozen.

Status

Partially Resolved

4. Locked Tokens can be transferred before release duration

Path

contracts/ERC7518.sol :: safeBatchTransferFrom()

Description

The erc7518 specification states that only unlocked assets can be transferred or used by users who own them, however the safeBatchTransferFrom() allows the users to move their funds as this method is directly inherited from the ERC1155Upgradeable contract and is not overridden like safeTransferFrom() that implements additional checks like canTransfer to ensure correct restrictions and signatures are provided before transfers are made.

Proof of Concept

The following test case confirms that locked tokens can be moved using the safeBatchTransferFrom() method without providing any specific signatures for verification.

```
function testUnsafeTransferAndUseOfTokens() public {
    erc7518Token.lockTokens(acc1, 1, 50, 3 days);
    erc7518Token.lockTokens(acc1, 3, 45, 3 days);
    erc7518Token.lockTokens(acc2, 2, 50, 3 days);
    erc7518Token.lockTokens(acc2, 0, 50, 3 days);
    erc7518Token.lockTokens(acc3, 1, 40, 3 days);
    erc7518Token.lockTokens(acc4, 3, 5, 3 days);

    uint256[] memory ids = new uint256[](2);
    ids[0] = 2;
    ids[1] = 0;
    uint256[] memory amounts = new uint256[](2);
    amounts[0] = 100;
    amounts[1] = 100;

    startHoax(acc2);
    erc7518Token.safeBatchTransferFrom(acc2, acc3, ids, amounts, data);
    vm.stopPrank();
}
```

Recommendation

It is highly suggested to ensure no tokens can be transferred out of the users wallet without updating the contract's state for the user's locked token mapping as in locked.

Status

Resolved

Low Severity Issues

1. Insufficient Input Validations

Path

contracts/ERC7518.sol :: changePayoutAddress
contracts/STOBaseV2.sol :: constructor()
contracts/ERC1155Mod.sol :: setFloorPrice()

Description

The methods listed above do not check for the validity of the incoming function arguments which might result in setting invalid values such as for the payout address or an excessively large floor price, potentially leading to loss of funds or failed transactions.

Recommendation

Implement validation to ensure that the new payout address is a non-zero address. For example, add a check like below before setting the new payout address;

require(newAddress != address(0), “Payout address cannot be zero”);

Status

Resolved

2. Missing non-reentrant modifier

Path

contracts/ERC7518.sol :: batchPayout()

Description

The batchPayout() function lacks a non-reentrant modifier, potentially allowing reentrancy attacks which could lead to unexpected behaviors or exploits.

Recommendation

Add the nonReentrant modifier to the batchPayout() function to prevent reentrancy attacks.

Status

Resolved

3. Substandard args order may cause confusion for users

Path

contracts/interfaces/ICompliTo.sol :: verifySignature()
contracts/ERC1155Mod.sol :: _transferTokensToInvestor()

Description

The order of arguments in verifySignature() _transferTokensToInvestor() is unconventional and might cause confusion, potentially leading to incorrect implementations or usage.

Recommendation

Reorder the parameters of both methods to follow a more intuitive sequence, ensuring that related parameters are grouped together in order, for instance, as follows;

```
function verifySignature(
    address from,
    address to,
    uint id,
    uint value, // use consistent naming; either `value` or `amount` across the smart
contract complex
    uint nonce,
    uint8 v // standard order as practiced in the industry, helps in data encoding and
decoding
    bytes32 r,
    bytes32 s,
) external view returns (bool) {}
```

Status

Resolved

4. Unsanitized freezing, restriction and whitelist removal operations

Path

contracts/module/IssuerFactory.sol :: _removeFromWhiteList()
contracts/module/FreezeAddress.sol :: _freeze()
_unfreeze(), contracts/module/RestrictTransfer.sol :: _restrict()
_removeRestriction()

Description

Methods such as _freeze(), _unfreeze() in FreezeAddress.sol, _removeFromWhiteList() in IssuerFactory.sol and _restrict(), _removeRestriction() in RestrictTransfer.sol are performed without checking the previous state of the account address. This oversight can lead to redundant operations and event spamming, which may impact the efficiency and reliability of the system. For instance, the ERC7518 token standard specs state the following about _removeRestriction() > MUST check if id is previously restricted.

Recommendation

Implement state checks to ensure that operations such as whitelisting, freezing, unfreezing, and restriction alterations only occur if the state change is necessary. For example, in _freeze(), verify the account is not already frozen before applying the freeze:

```
function _freeze(address account) internal {
    if (isFrozen(account)) revert(“Account is already frozen.”);
    frozen[account] = true;
    emit Frozen(account);
}
```

Status

Resolved

5. Remove redundant checks for to Address

Path

contracts/module/Payout.sol :: _payout()

Description

Redundant checks for the to address in the _payout() function can lead to unnecessary gas consumption and code complexity. This check is already placed inside ERC20Upgradeable contract that handles the payout's transfer call underneath.

Recommendation

Ensure that all address checks are necessary and remove any redundant conditions.

Status

Resolved

Informational Issues

1. Insufficient NatSpec

Path

contracts/module/RestrictTransfer.sol
contracts/module/PaymentTracker.sol
contracts/module/Payout.sol
contracts/ERC7518.sol :: lockTokens()
contracts/ERC1155Mod.sol :: _tokenSettlement()

Description

The contracts and methods listed above perform critical functionality to manage users' assets and funds. However, there is no supporting documentation provided for NatSpec that explains the business needs and limitations of the applications which can cause misunderstandings and threaten users' operations.

Recommendation

It is highly suggested that supporting documentation is provided for all the critical contracts and functions listed above.

Status

Resolved

2. Unordered methods according to Solidity style doc

Path

contracts/CompliTo.sol, contracts/ERC1155Mod.sol

Description

The methods in CompliTo.sol and ERC1155Mod.sol are not organized according to the Solidity style guide which recommends ordering methods by visibility and functionality so that gas consumption for all users facing external or public methods are optimized.

Recommendation

Rearrange the methods in CompliTo.sol to align with the Solidity style guide, improving readability and maintainability.

Status

Resolved

3. Inconsistent function naming may cause confusion

Path

contracts/ERC7518.sol :: forceTokenUnlock()
_forceUnlockTokens
contracts/ERC7518.sol :: forceTransfer()
_safeTransferFrom()

Description

The naming of the forceTokenUnlock() and _forceUnlockTokens functions in ERC7518.sol are inconsistent, which could lead to confusion about their functionality.

Recommendation

Standardize function names to consistently reflect their actions and distinguish between public and internal functions clearly.

Status

Resolved

4. Remove unused fn args and ret vars

Path

contracts/ERC7518.sol :: freeze()
unFreeze()
contracts/module/IssuerFactory.sol :: getDealsCount()

Description

Some functions in the listed contracts include arguments or return variables that are never used, leading to unnecessary code complexity and gas costs. For instance, the data variable in freeze() and unfreeze() is never utilized in the method for operation. Similarly, in the getDealsCount(), the return value deals is never used.

Recommendation

Review and remove unused parameters and return variables from functions to optimize contract efficiency and clarity.

Status

Resolved

5. Unconformed context fetching

Path

contracts/module/ERC2771ContextUpgradeable.sol :: _msgSender(), _msgData()

Description

The _msgSender and _msgData methods in the ERC2771ContextUpgradeable contract do not conform to a set standard while fetching the data or caller from the context as the sender is extracted using assembly code block which has proven manipulatable in past due to either delegatecalls or by calldata stuffing.

Recommendation

It is recommended to implement the standard while fetching the msgSender and msgData using the ERC2771ContextUpgradeable contract such that both methods conform to the industry best practices.

Status

Resolved

6. Ungraceful Event Emission

Path

contracts/module/Payout.sol :: _changePayoutAddress()

Description

The _changePayoutAddress() method does not emit the previous payout address when it is changed, which can obscure tracking of changes in the contract state.

Recommendation

Modify _changePayoutAddress() to emit an event that includes both the old and the new payout addresses to enhance transparency and auditability.

Status

Resolved

7. Use Block Letters for enum variables

Path

contracts/module/PaymentTracker.sol :: InvestmentStatus, InvestmentType

Description

The enumerable structs InvestmentStatus and InvestmentType in the PaymentTracker contract do not implement the variables inside in uppercase format, which is the standard practice for enums to differentiate them from other variables and mark the use of uint8 variable for enums.

Recommendation

Refactor enum names in PaymentTracker to use uppercase letters, enhancing code readability and adhering to common coding standards.

Status

Resolved

8. Unimplemented Interface for ERC7518Marketplace contract

Path

contracts/interfaces/IMarketPlace.sol :: create()

Description

There is a typo in the interface at changeComplainceAddress which could lead to implementation errors or issues when interfacing with other contracts. Additionally, the interface is not implemented in the ERC7518Marketplace contract, specifically for create() method the contract implements a different function selector with additional arguments that deviate from the interface.

Recommendation

Correctly implement the interface in the smart contract and ensure there are no typos across the codebase.

Status

Resolved

Closing Summary

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

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Zoniqx smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Zoniqx smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Zoniqx to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
May, 2024

For

Zoniqx

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com