HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Bloqhouse Technologies
Date: March 24, 2023

Document

Name	Smart Contract Code Review and Security Analysis Report for Bloqhouse Technologies
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC721 token
Platform	EVM
Language	Solidity
Methodology	Link
Website	www.bloqhouse.com
Changelog	13.03.2023 - Initial Review 24.03.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Bloqhouse Technologies (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project includes review and security analysis of the following smart contracts from the provided repository:

Initial review scope

Repository	https://bitbucket.org/alfredpersson/token-shares-solidity/src/master
Commit	e6f6102b0d869590193143f07850d894a84125f9
Functional Requirements	https://bitbucket.org/alfredpersson/token-shares-solidity/src/master/docs/Documentation.pdf
Technical Requirements	https://bitbucket.org/alfredpersson/token-shares-solidity/src/master/docs/Documentation.pdf
Contracts	File: ./contracts/RWAT.sol SHA3:79ff9f8c3676e39c2709c1a47d60bff94e5c0d5ae94f825edd9370a3661872f1 File: ./interfaces/ICNR.sol SHA3 5257f637e1343d2aee061fd404ab50d78c45666ab12c4b69d148e7beb33f4af0

Second review scope

Repository	https://bitbucket.org/alfredpersson/token-shares-solidity/src/master
Commit	cbdc7c0d6162346b96cf62cb2ff93c15f416819e
Contracts	File: ./contracts/RWAT.sol SHA3: 81970eff160e050def2296aadf17d4b3b566ce8933800fecf79711fa434d0e5a File: ./interfaces/ICNR.sol SHA3: 60e1bcae2996ee150eac9903a8dfcefc181f9409e08c787f9694d7da671bbb74

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor gas optimization. These issues won't have a significant impact on code execution but affect code quality

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are comprehensive.
- Technical description is detailed.
- NatSpec is consistent.

Code quality

The total Code Quality score is **10** out of **10**.

- Solidity best practices are followed.
- Style guides are followed.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions with several users are tested thoroughly.

Security score

As a result of the audit, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10**.

1
2
3
4
5
6
7
8
9
10
The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
13 March 2023	3	1	3	0
24 March 2023	0	0	0	0

Risks

- The admin has comprehensive rights to user funds which makes the project **centralized**. These rights include forcing funds out of wallets without a user's permission, pausing transfers, changing name, symbol, and asset caps.
- The contracts are upgradable and the current audit covers only the implementation at the time of the review. Any further implementations are not covered by this audit.

System Overview

The project is a real world asset tokenization protocol, which aims to enable asset providers to fractionalize assets such as, but not limited to, real estate, land, property, on the blockchain as NFTs.

RWATP (Real World Asset Tokenization Protocol) is an ERC721 based tokenization protocol. One instance of RWATP can have multiple assets, this enables one asset provider to handle several assets in one contract. An asset is a set of NFTs, one NFT represents a share of the asset. E.g if an asset consists of 100 NFTs then one NFT represents 1% of the asset.

The contract admin (asset provider or someone else) can decide how to mint and distribute the NFTs, either through minting the NFTs to the asset provider, to a user, or delegating the minting to a separate contract that can then add additional logic to the distribution process, for example crypto payment.

The supply of an asset is not fixed, but controlled by the contract admin. The contract enables issuance of new NFTs for an asset, thus diluting the supply, at their discretion. Likewise the admin can enable burning of NFTs, concentrating the supply. However, these features can be locked in perpetuity to ensure a constant number of NFTs for an asset.

The admin of the contract has rights to the funds such as burning, minting, pausing transfers, setting whitelist functionality(only allowing whitelisted addresses to transfer their tokens), and transferring them from users. This functionality can be locked by the admin to never be enabled again.

The files in the scope:

- **RWAT.sol:** The ERC721 contract that represents the assets. Can be controlled by the admin.
- **ISNR.sol:** The interface for obtaining the Token URI.

Privileged roles

- DEFAULT_ADMIN_ROLE: The DEFAULT_ADMIN_ROLE is inherited from the OpenZeppelin AccessControl.sol contract. It is the role that can grant or revoke all other roles. Presumably this should be given to a secure multisig and not used for anything other than granting/revoking the other roles.
- Admin: The ADMIN role controls all issuance, burning and settings of the contract.
- Handler: HANDLER can be given to a separate contract adding additional logic to the minting process. This role can only control the minting of new NFTs.

- Whitelister: There is one role for adding users to the whitelist, this is separate from the admin role to add an extra level of security.
- User: A user has no special privileges unless the whitelist is enabled, then the user must be on the whitelist in order to do anything other than holding the NFT.

Recommendations

- Consider using a multisig wallet for admin functionality to introduce a safer form of centralization for the users.
- Using the same internalNonce mapping to check the signature in the `userMintUnits()` and `claimUnits()` functions may lead to invalid signatures if the functions were configured for the same user address with the same nonce parameters. This would cause the owner to create a new signature. Consider using different mappings for the two functions.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Not Relevant
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Not Relevant
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Passed
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed

Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Not Relevant
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Not Relevant
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Passed
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
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.	Not Relevant
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Not Relevant
Presence of Unused Variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP Standards Violation	EIP	EIP standards should not be violated.	Passed
Assets Integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
User Balances Manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed

Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Not Relevant
Token Supply Manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer.	Passed
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style Guide Violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. EIP Standard Violation

Signatures do not include chain-specific parameters like chain id as stated in the EIP-712 standard.

This may lead to signature replay attacks if the contract is deployed multiple times and the verifier address is the same.

Path: ./contracts/RWAT.sol : userMintUnits(), claimUnits()

Recommendation: follow the EIP-712 standard when creating and verifying signatures.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

H02. Data Consistency

The project uses the tokens for every asset to determine the percentage of the asset the token holder holds. The contract RWAT.sol allocates 1.000.000.000 ids for every asset. The *_tokenCap* is not checked according to this allocation.

If the *_tokenCap* is larger than the max allocation of 1.000.000.000, the *tokenIds* might override the next asset allocation.

Path: ./contracts/RWAT.sol : createAsset(), updateAssetCap()

Recommendation: verify that the *_tokenCap* is smaller than 1.000.000.000 when setting or updating asset caps.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

H03. Double-Spending

In the *claimUnits()* function, there is no internal nonce check for the signature verification.

In the scenario that:

- the *reclaimUnitsDisabled* parameter is *true*,
- The admin gets all tokens back (specifically the ones that were claimed previously), and transfers them back into the contract;

the user can successfully call the *claimUnits()* function with the same signature and transfer the tokens to their address again.

This would lead to fund loss since the admin cannot reclaim the tokens.

Path: ./contracts/RWAT.sol : claimUnits()

Recommendation: add a nonce mechanism for this function specific for the claiming users into the signature and increment it after the call so that the same signature cannot be used.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

■■ Medium

M01. Contradiction

The project intends to override the ERC721 *_name* and *_symbol* parameters; however since it is using different names which are *name_* and *symbol_* for RWAT.sol, the parameters are not actually overwritten. The functions to view them are overwritten, which handles the issue but the function *setNameAndSymbol()* should be called before it is actually active.

This function is not called in the *initializer()* of the contract, only the ERC721Upgradeable parameters are set. This will cause the contract to be initialized without any name or symbol.

Path: ./contracts/RWAT.sol : initialize()

Recommendation: call the *setNameAndSymbol()* function in the *initializer()*.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

■ Low

L01. Floating Pragma

The project uses floating pragma *^0.8.4* in contracts RWAT.sol and ICNR.sol.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Paths: ./contracts/RWAT.sol

./interfaces/ICNR.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

L02. Shadowing State Variable

In RWA.sol contracts' *initialize()*, *setTransfersPaused()*, *setAssetTransfersPaused()*, and the *setNameAndSymbol()* function, variables *_name*, *_symbol*, and *_paused* are shadowed from the ERC721Upgradeable contract.

Path: ./contracts/RWAT.sol : initialize(), setTransfersPaused(), setAssetTransfersPaused(), setNameAndSymbol()

Recommendation: Rename related variables/arguments.

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

L03. Unindexed Events

Having indexed parameters in the events makes it easier to search for these events using *indexed* parameters as filters.

Path: ./contracts/RWAT.sol: UnitsClaimed()

Recommendation: Use the *"indexed"* keyword to the event parameters

Found in: e6f6102b0d869590193143f07850d894a84125f9

Status: Fixed (Revised commit: cbdc7c0)

Disclaimers

Hacken Disclaimer

Technical Disclaimer