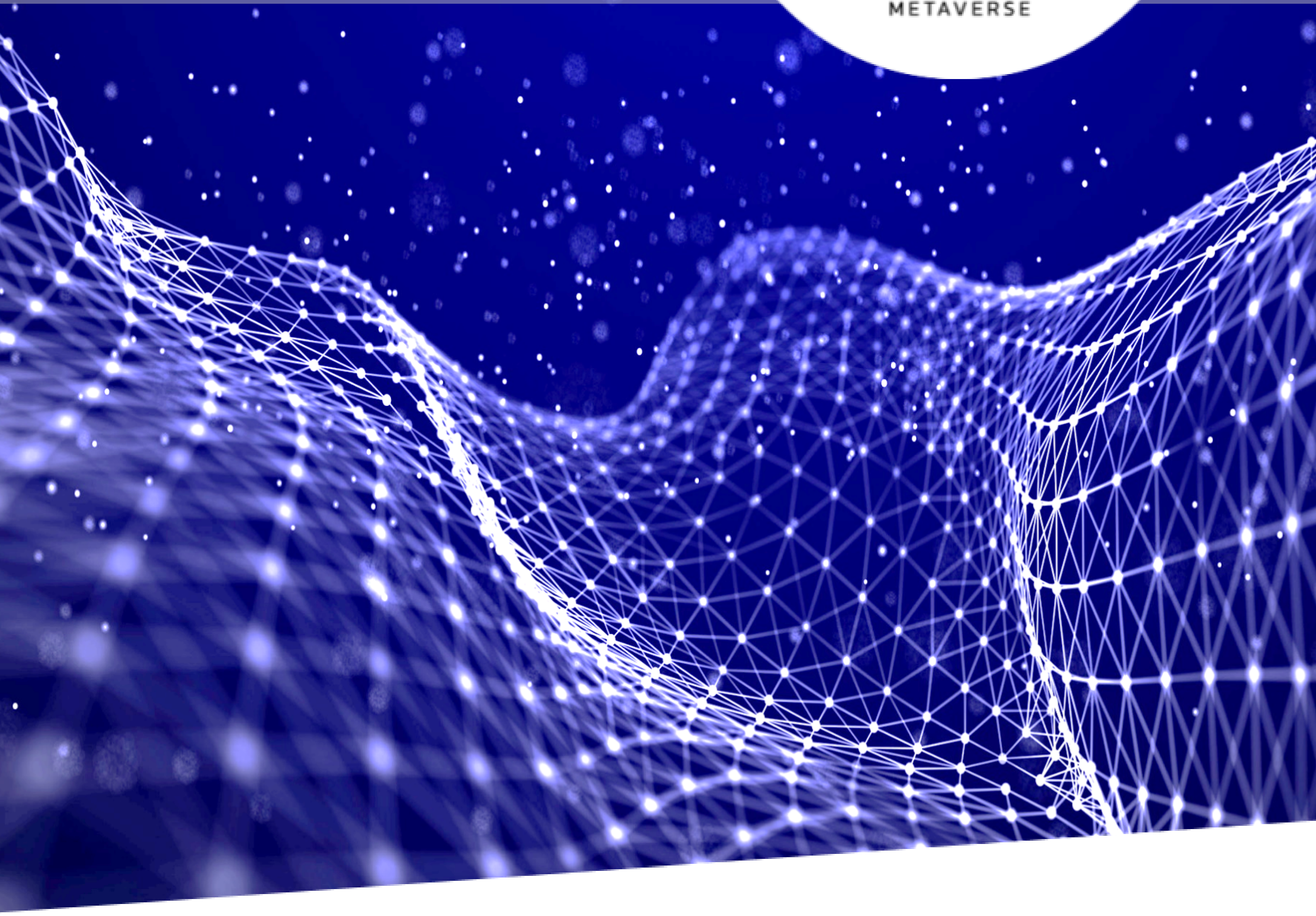
Aniverse

ANIV721Land

Smart Contract Audit Report



Date Issued: 21 Sep 2022

Version: Final v1.0

Public

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **ANIV721Land**. This audit report was published on `21 Sep 2022`. The audit scope is limited to the **ANIV721Land.** Our security best practices strongly recommend that the **Aniverse team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About ANIV721Land

Land of Aniverse is a land located on a Metaverse the land at Aniverse has a total of 250,000 blocks, which is the first map that focuses on the development of education that has divided the area for the large-scale study of many institutions.

Scope of Work

The security audit conducted does not replace the full security audit of the overall Aniverse protocol. The scope is limited to the **ANIV721Land** and its related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *ANIV721Land smart contract* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/CREATIVE-DIGITAL-LIVING-CO-LTD/SC_ERC721_LAND*
Audit Commit	▪ *f2412b75689d1187be208a291f31f7ca4e7aa61a (branch: dev)*
Reassessment Commit	▪ *134c5c5445ff08c8390918aea5cffe92710565e7 (branch: features/audit)*
Audited Files/Contracts	▪ *./contracts/ANIV721Land.sol*

	▪ *./contracts/Operator.sol* ▪ *./contracts/erc721/ERC721Tradable.sol* ▪ *./contracts/erc721/common/meta-transactions/ContextMixin.sol* ▪ *./contracts/erc721/common/meta-transactions/EIP712Base.sol* ▪ *./contracts/erc721/common/meta-transactions/Initializable.sol* ▪ *./contracts/erc721/common/meta-transactions/NativeMetaTransaction.sol* ▪ *ProxyRegistry contract (prototype implementation)* ▪ *Other imported associated Solidity files*
Excluded Files/Contracts	▪ *./contracts/test/MockProxyRegistry.sol* ▪ *ProxyRegistry contract (complete implementation)*

Remark: Our security best practices strongly recommend that the Aniverse team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	Anak Mirasing Atitawat Pol-in Kritsada Dechawattana Parichaya Thanawuthikrai Phuwanai Thummavet
Authors	Anak Mirasing Atitawat Pol-in Kritsada Dechawattana Parichaya Thanawuthikrai Phuwanai Thummavet
Reviewers	Sumedt Jitpukdebodin

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **ANIV721Land** has sufficient security protections to be safe for use.

The smart contracts PASS our security verification and qualification level.

Verified on 21 Sep 2022

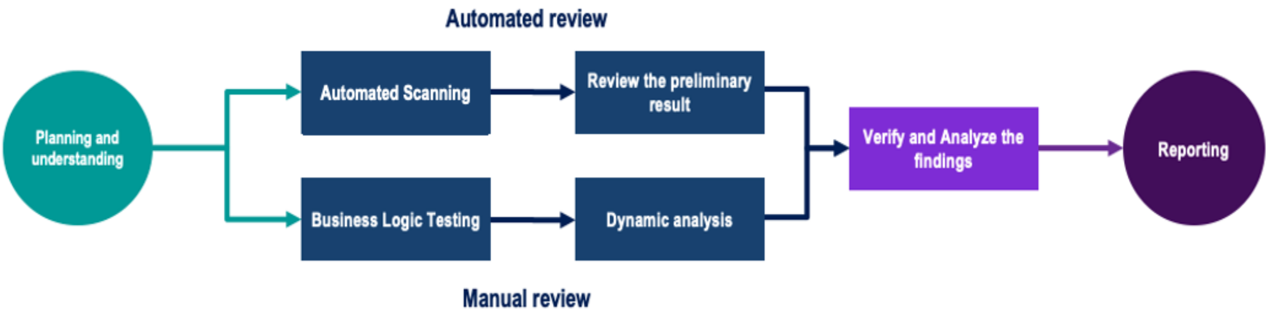
Initially, Valix was able to identify **22 issues** that were categorized from the “Critical” to “Informational” risk level in the given timeframe of the assessment. **For the reassessment, the *Aniverse* team fixed 18 issues. Other issues were partially fixed and acknowledged.** Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
ANIV721Land	-	2	8	8	4	-	1	2	1	1

Note: *Risk Rating* C *Critical,* H *High,* M *Medium,* L *Low,* I *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application’s purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	**SEC01**	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	**FNC01**	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	**OPT01**	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	**DEV01**	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Likelihood Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Possibly Bypassing Token Transfer Verification Mechanism	High	Fixed	In use
2	Incorrect Logical Design Of Token Transfer Verification Mechanism	High	Acknowledged	In use
3	Denial-Of-Service On Operator Revoking Process	Medium	Fixed	In use
4	Possibly Permanent Ownership Removal	Medium	Fixed	In use
5	Unsafe Ownership Transfer	Medium	Fixed	In use
6	Recommended Adding A Setter Function For Proxy Registry Address	Medium	Partially Fixed	In use
7	Lack Of Deadline For Meta Transactions	Medium	Acknowledged	In use
8	Possibly Bypassing Token Disapproval Mechanism	Medium	Fixed	In use
9	Possible Cross-Chain Replay Attack Over Meta Transactions	Medium	Fixed	In use
10	Recommended Changing Visibility Of State Variables For Transparency	Medium	Fixed	In use
11	Potential Approval Of Duplicated Token IDs	Low	Fixed	In use
12	Lack Of Clearing Land Approval Array Of Revoked Operator	Low	Fixed	In use
13	Possibly Incorrect Token Disapproval	Low	Fixed	In use
14	Recommended Adding A Setter Function For Base Token URI	Low	Partially Fixed	In use
15	Recommended Event Emissions For Transparency And Traceability	Low	Fixed	In use
16	Possibly Minting Out-Of-Bound Token ID	Low	Fixed	In use
17	Lack Of Validating Existence Of Token ID	Low	Fixed	In use

18	**Recommended Removing Redundant Logic**	Low	Fixed	In use
19	**Inconsistent Error Message With The Code**	Informational	Fixed	In use
20	**Recommended Removing Unused State Variable**	Informational	Fixed	In use
21	**Inconsistent Contract Name**	Informational	Fixed	In use
22	**Depending On External Contract**	Informational	Acknowledged	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Possibly Bypassing Token Transfer Verification Mechanism		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 135 - 144*		

Detailed Issue

We discovered that the *_beforeTokenTransfer* function (L135 - 144 in code snippet 1.1) is vulnerable to bypassing a token transfer verification mechanism when an *Aniverse operator* transfers a token to itself. The root cause of this issue is that the function uses *msg.sender* to get a function caller. Since the *ANIV721Land* contract supports meta transactions, adopting the *msg.sender*, in this case, can cause the bypassing issue.

Consider the following scenario to understand this issue.

1. Assuming that an *Aniverse operator Eve* got approval to operate on the *TokenA*.
2. *Eve* signs a meta transaction for transferring the *TokenA* to herself.
3. *Eve* submits the signed meta transaction payload to the ***NativeMetaTransaction.executeMetaTransaction()*** function (L33 - 67 in code snippet 1.2).
4. The *executeMetaTransaction* function verifies the payload and executes the target ***ERC721.transferFrom(TokenA's owner address, Eve address, TokenA's id)*** function (L150 - 159 in code snippet 1.3).
5. The *transferFrom* function verifies a transfer approval and executes the internal ***ERC721._transfer(TokenA's owner address, Eve address, TokenA's id)*** function (L158 in code snippet 1.3).
6. The *_transfer* function invokes the ***ERC721Tradable._beforeTokenTransfer(TokenA's owner address, Eve address, TokenA's id)*** function (L339 in code snippet 1.3).

7. The *_beforeTokenTransfer* function's execution flow enters the operator's token transfer verification (L140 - 142 in code snippet 1.1) because the "***to***" variable is pointing to *Eve* who is an *Aniverse operator*.

   At this point, the operator's token transfer verification mechanism would be bypassed since the *msg.sender* (L141) would demonstrate that the function caller is the contract itself (i.e., ***this*** address), not the *operator Eve*.

8. The *_transfer* function transfers the *TokenA* to *Eve* without permission.

ERC721Tradable.sol

```
135 function _beforeTokenTransfer(
136     address from,
137     address to,
138     uint256 tokenId
139 ) internal virtual override {
140     if (isOperator(to)) {
141         require(msg.sender != to, "Operator can't transfer to itself");
142     }
143     super._beforeTokenTransfer(from, to, tokenId);
144 }
```

Listing 1.1 The vulnerable *_beforeTokenTransfer* function

NativeMetaTransaction.sol

```
function executeMetaTransaction(
    address userAddress,
    bytes memory functionSignature,
    bytes32 sigR,
    bytes32 sigS,
    uint8 sigV
) public payable returns (bytes memory) {
    MetaTransaction memory metaTx = MetaTransaction({
        nonce: nonces[userAddress],
        from: userAddress,
        functionSignature: functionSignature
    });

    require(
        verify(userAddress, metaTx, sigR, sigS, sigV),
        "Signer and signature do not match"
    );

    // increase nonce for user (to avoid re-use)
    nonces[userAddress] = nonces[userAddress].add(1);

    emit MetaTransactionExecuted(
        userAddress,
        payable(msg.sender),
        functionSignature
    );

    // Append userAddress and relayer address at the end to extract it from calling context
    (bool success, bytes memory returnData) = address(this).call(
        abi.encodePacked(functionSignature, userAddress)
    );
    require(success, "Function call not successful");

    return returnData;
}
```

Listing 1.2 The *executeMetaTransaction* function that allows anyone to submit a meta transaction to invoke *ANIV721Land* contract's functions

ERC721.sol

```
function transferFrom(
    address from,
    address to,
    uint256 tokenId
) public virtual override {
    //solhint-disable-next-line max-line-length
    require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: transfer caller is not owner nor approved");

    _transfer(from, to, tokenId);
}

// (...SNIPPED...)

function _transfer(
    address from,
    address to,
    uint256 tokenId
) internal virtual {
    require(ERC721.ownerOf(tokenId) == from, "ERC721: transfer from incorrect owner");
    require(to != address(0), "ERC721: transfer to the zero address");

    _beforeTokenTransfer(from, to, tokenId);

    // Clear approvals from the previous owner
    _approve(address(0), tokenId);

    _balances[from] -= 1;
    _balances[to] += 1;
    _owners[tokenId] = to;

    emit Transfer(from, to, tokenId);

    _afterTokenTransfer(from, to, tokenId);
}
```

Listing 1.3 The *transferFrom* and *_transfer* functions of the *ERC721* contract

Recommendations

We recommend calling the *_msgSender* function (L141 in the code snippet below) instead of using the *msg.sender* to get a legitimate function caller.

ERC721Tradable.sol

```
135 function _beforeTokenTransfer(
136     address from,
137     address to,
138     uint256 tokenId
139 ) internal virtual override {
140     if (isOperator(to)) {
141         require(_msgSender() != to, "Operator can't transfer to itself");
142     }
143     super._beforeTokenTransfer(from, to, tokenId);
144 }
```

Listing 1.4 The improved *_beforeTokenTransfer* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue according to our suggestion.

No. 2	Incorrect Logical Design Of Token Transfer Verification Mechanism		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 135 - 144*		

Detailed Issue

The *_beforeTokenTransfer* function was implemented to verify that an *Aniverse operator* would not be able to transfer any token to itself (L140 - 142 in code snippet 2.1). The *_beforeTokenTransfer* function would automatically be invoked every time when a token is being transferred by the *_transfer* function (L339 in code snippet 2.2).

Nonetheless, we noticed that this operator's token transfer verification mechanism is not practically effective. More specifically, **an *Aniverse operator* can easily bypass this mechanism by transferring a token to another operator and then making a transfer back to itself, or even transferring a token to its personal wallet.**

ERC721Tradable.sol

```
135 function _beforeTokenTransfer(
136     address from,
137     address to,
138     uint256 tokenId
139 ) internal virtual override {
140     if (isOperator(to)) {
141         require(msg.sender != to, "Operator can't transfer to itself");
142     }
143     super._beforeTokenTransfer(from, to, tokenId);
144 }
```

Listing 2.1 The *_beforeTokenTransfer* function that would not allow an *Aniverse operator* to transfer any token to itself

ERC721.sol

```
function _transfer(
    address from,
    address to,
    uint256 tokenId
) internal virtual {
    require(ERC721.ownerOf(tokenId) == from, "ERC721: transfer from incorrect owner");
    require(to != address(0), "ERC721: transfer to the zero address");

    _beforeTokenTransfer(from, to, tokenId);

    // Clear approvals from the previous owner
    _approve(address(0), tokenId);

    _balances[from] -= 1;
    _balances[to] += 1;
    _owners[tokenId] = to;

    emit Transfer(from, to, tokenId);

    _afterTokenTransfer(from, to, tokenId);
}
```

Listing 2.2 The *_transfer* function that calls the *_beforeTokenTransfer* function to verify the operator's token transfer

Recommendations

We recommend re-designing and re-implementing the logic for verifying a token transfer by an *Aniverse operator* by taking all possible bypassing cases into account.

Reassessment

The *Aniverse* team acknowledged this issue and decided to retain the original code and design. However, the *Aniverse* team would enforce a law on all *Aniverse* operators to prevent them from such abusing transactions.

No. 3	**Denial-Of-Service On Operator Revoking Process**		
Risk	**Medium**	**Likelihood**	**Low**
		Impact	**High**
Functionality is in use	**In use**	**Status**	**Fixed**
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 119 - 125*		

Detailed Issue

The *ERC721Tradable* contract has the *revokeOperator* function (L119 - 125 in the code snippet below) for revoking an *Aniverse* operator.

We noticed that the *revokeOperator* function would disapprove all token approvals of a revoking operator. At this point, we are concerned that the token disapproval process could consume gas beyond the block gas limit, leading to a denial-of-service issue.

To elaborate, the *revokeOperator* function uses the *for-loop* (L122 - 124) to disapprove all token approvals. Imagine the case that the length of the *_tokenId* array is too large; the function would consume gas beyond the block gas limit.

As a result, the revoking transaction would be reverted. In other words, the contract owner would not be able to revoke that operator anyhow.

ERC721Tradable.sol

```
119  function revokeOperator(address to) public onlyOwner {
120      _revokeOperator(to);
121      uint256[] memory _tokenId = _operartorLandApproval[to];
122      for (uint256 i = 0; i < _tokenId.length; i++) {
123          _approve(address(0), _tokenId[i]);
124      }
125  }
```

Listing 3.1 The *revokeOperator* function

Recommendations

We recommend re-designing and re-implementing the *revokeOperator* function by taking the denial-of-service issue into consideration.

Reassessment

The *Aniverse* team remediated this issue by limiting the length of the token approval array for each operator on the *_addLandToOperator* function (L137 in the code snippet below). The approval limit is controlled by the *maxOperatorLand* variable and this variable can be updated by way of invoking the *setMaxOperatorLand* function (L181 - 186).

Note that, the default value of the token length limit is 600 (L46) whereas the maximum value is 1000 (L47). These values have been tested and confirmed by the Aniverse team that they are not too large to exceed the block gas limit of the blockchain network they would like to deploy the contract to.

ERC721Tradable.sol

```
 46  uint256 public maxOperatorLand = 600;
 47  uint256 public immutable MAX_VALUE_OPERATOR_LAND = 1000;

     // (...SNIPPED...)

132  function _addLandToOperator(address to, uint256 tokenId) internal virtual {
133      require(isOperator(to), "Address is not operator");
134      require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
135      require(!_operatorTokenApproval[to][tokenId], "Token id was approved");
136      uint256[] storage _tokenId = _operatorLandApproval[to];
137      require(_tokenId.length < maxOperatorLand, "Current operator has maxed
     land");
138      if (getApproved(tokenId) != address(0)) {
139          _operatorTokenApproval[getApproved(tokenId)][tokenId] = false;
140      }
141      _tokenId.push(tokenId);
142      _operatorTokenApproval[to][tokenId] = true;
143      emit AddLandToOperator(tokenId, to);
144
145  }

     // (...SNIPPED...)

181  function setMaxOperatorLand(uint256 _newMaxOperatorLand) external onlyOwner {
182      require(_newMaxOperatorLand > 0 && _newMaxOperatorLand <=
     MAX_VALUE_OPERATOR_LAND, "Operator must be operate lands between 1 - 1000");
183      uint256 _oldMaxOperatorLand = maxOperatorLand;
184      maxOperatorLand = _newMaxOperatorLand;
```

```
    emit SetMaxOperatorLand(_oldMaxOperatorLand, _newMaxOperatorLand);
}
```

Listing 3.2 Limiting the length of the token approval array for each operator

No. 4	Possibly Permanent Ownership Removal		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*@openzeppelin/contracts/access/Ownable.sol*		
Locations	*Ownable.sol L: 54 - 56*		

Detailed Issue

The *ERC721Tradable* contract inherits from the *Ownable* abstract contract. The *Ownable* contract implements the *renounceOwnership* function (L54 - 56 in the code snippet below), which can remove the contract's ownership permanently.

If the contract owner mistakenly invokes the *renounceOwnership* function, they will immediately lose ownership of the contract, and this action cannot be undone.

Ownable.sol

```
54  function renounceOwnership() public virtual onlyOwner {
55      _transferOwnership(address(0));
56  }

    // (...SNIPPED...)

71  function _transferOwnership(address newOwner) internal virtual {
72      address oldOwner = _owner;
73      _owner = newOwner;
74      emit OwnershipTransferred(oldOwner, newOwner);
75  }
```

Listing 4.1 The *renounceOwnership* function
that can remove the ownership of the contract permanently

Recommendations

We consider the *renounceOwnership* function risky, and the contract owner should use this function with extra care.

If possible, we recommend removing or disabling this function from the contract. The code snippet below shows an example solution to disabling the associated *renounceOwnership* function.

ERC721Tradable.sol

```
function renounceOwnership() external override onlyOwner {
    revert("Ownable: renounceOwnership function is disabled");
}
```

Listing 4.2 The disabled *renounceOwnership* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue by disabling the *renounceOwnership* function according to our recommendation.

No. 5	Unsafe Ownership Transfer		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*@openzeppelin/contracts/access/Ownable.sol*		
Locations	*Ownable.sol L: 62 - 65*		

Detailed Issue

The *ERC721Tradable* contract inherits from the *Ownable* abstract contract. The *Ownable* contract implements the *transferOwnership* function (L62 - 65 in the code snippet below), which can transfer the ownership of the contract from the current owner to another owner.

Ownable.sol

```
62 function transferOwnership(address newOwner) public virtual onlyOwner {
63     require(newOwner != address(0), "Ownable: new owner is the zero address");
64     _transferOwnership(newOwner);
65 }

   // (...SNIPPED...)

71 function _transferOwnership(address newOwner) internal virtual {
72     address oldOwner = _owner;
73     _owner = newOwner;
74     emit OwnershipTransferred(oldOwner, newOwner);
75 }
```

Listing 5.1 The *transferOwnership* function that has the unsafe ownership transfer

From the code snippet above, the address variable *newOwner* (L62) may be incorrectly specified by the current owner by mistake; for example, an address that a new owner does not own was inputted. Consequently, the new owner loses ownership of the contract immediately, and this action is unrecoverable.

Recommendations

We recommend applying the two-step ownership transfer mechanism as shown in the code snippet below.

ERC721Tradable.sol

```
146 function transferOwnership(address _candidateOwner) public override onlyOwner {
147     require(_candidateOwner != address(0), "Ownable: candidate owner is the zero
    address");
148     candidateOwner = _candidateOwner;
149     emit NewCandidateOwner(_candidateOwner);
150 }
151
152 function claimOwnership() external {
153     require(candidateOwner == _msgSender(), "Ownable: caller is not the
    candidate owner");
154     _transferOwnership(candidateOwner);
155     candidateOwner = address(0);
156 }
```

Listing 5.2 The recommended two-step ownership transfer mechanism

This mechanism works as follows.

1. The current owner invokes the *transferOwnership* function by specifying the candidate owner address *_candidateOwner* (L146).

2. The candidate owner proves access to his account and claims the ownership transfer by invoking the *claimOwnership* function (L152)

The recommended mechanism ensures that the ownership of the contract would be transferred to another owner who can access his account only.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue by adopting the two-step ownership transfer mechanism as per our suggestion.

No. 6	Recommended Adding A Setter Function For Proxy Registry Address		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 49 and 83 - 96*		

Detailed Issue

The *proxyRegistryAddress* state variable (L49 in the code snippet below) would point to an external *ProxyRegistry* contract, and this variable is used in the *isApprovedForAll* function (L90).

However, we found that there is no setter function that can update the value of the *proxyRegistryAddress* variable. Hence, if the address of the *ProxyRegistry* contract has to be changed in the future, the *ANIV721Land*'s contract owner would have no approach to updating this variable, and this issue might impact the function of the *ANIV721Land* contract.

ERC721Tradable.sol

```
49  address proxyRegistryAddress;

    // (...SNIPPED...)

83  function isApprovedForAll(address owner, address operator)
84      public
85      view
86      override
87      returns (bool)
88  {
89      // Whitelist OpenSea proxy contract for easy trading.
90      ProxyRegistry proxyRegistry = ProxyRegistry(proxyRegistryAddress);
91      if (address(proxyRegistry.proxies(owner)) == operator) {
92          return true;
93      }
94
95      return super.isApprovedForAll(owner, operator);
96  }
```

Listing 6.1 The *isApprovedForAll* function calling the external contract pointed by the state variable *proxyRegistryAddress*

Recommendations

We recommend implementing a setter function for updating the *proxyRegistryAddress* state variable as shown in the below code snippet. And, this setter function should be under the control of the *Timelock* mechanism.

ERC721Tradable.sol

```
event SetProxyRegistryAddress(address indexed _oldAddress, address indexed _newAddress);

// (...SNIPPED...)

function setProxyRegistryAddress(address _newProxyRegistryAddress) external onlyOwner {
    require(_newProxyRegistryAddress != address(0), "Set proxy registry address to zero address");
    address _oldAddress = proxyRegistryAddress;
    proxyRegistryAddress = _newProxyRegistryAddress;
    emit SetProxyRegistryAddress(_oldAddress, _newProxyRegistryAddress);
}
```

Listing 6.2 The recommended *setProxyRegistryAddress* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team partially fixed this issue by implementing the *setProxyRegistryAddress* function as per our recommendation. However, the *setProxyRegistryAddress* function would not be controlled under the *Timelock* mechanism.

No. 7	Lack Of Deadline For Meta Transactions		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/erc721/common/meta-transactions/NativeMetaTransaction.sol*		
Locations	*NativeMetaTransaction.sol L: 10 - 14, 27 - 31, and 69 - 83*		

Detailed Issue

The *ANIV721Land* contract supports the meta-transaction feature allowing relayers such as *OpenSea*'s relayers to execute a transaction signed by a user and pay gas for a user.

We noticed that, however, the process of proving the meta transaction does not include a *deadline* which is an important property in the process (code snippet below). Specifically, the *deadline* property would restrict an expired timestamp of each signed meta transaction. The signed transaction payload would be invalid if its *deadline* property is reached.

Lacking the *deadline* property, a signed meta-transaction payload might be submitted anytime without any control from a user.

Since the *ANIV721Land* contract must be interacting with *OpenSea*'s meta-transaction features, changing the way to prove the signed payload might break the compatibility with *OpenSea*. For this reason, we would like to raise this issue as ***acknowledgment*** only.

NativeMetaTransaction.sol

```
10  bytes32 private constant META_TRANSACTION_TYPEHASH = keccak256(
11      bytes(
12          "MetaTransaction(uint256 nonce,address from,bytes functionSignature)"
13      )
14  );

    // (...SNIPPED...)

27  struct MetaTransaction {
28      uint256 nonce;
29      address from;
30      bytes functionSignature;
31  }
```

```
// (...SNIPPED...)

function hashMetaTransaction(MetaTransaction memory metaTx)
    internal
    pure
    returns (bytes32)
{
    return
        keccak256(
            abi.encode(
                META_TRANSACTION_TYPEHASH,
                metaTx.nonce,
                metaTx.from,
                keccak256(metaTx.functionSignature)
            )
        );
}
```

Listing 7.1 The *deadline* property was not included
in the process of proving a meta transaction

Recommendations

Since the *ANIV721Land* contract must be interacting with *OpenSea*'s meta-transaction features, changing the way to prove the signed payload might break the compatibility with *OpenSea*. For this reason, we would like to raise this issue as ***acknowledgment*** only.

Reassessment

The *Aniverse* team acknowledged this issue.

No. 8	Possibly Bypassing Token Disapproval Mechanism		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 108 - 113*		

Detailed Issue

We noticed that the *approve* function (L108 - 113 in code snippet 8.1) is vulnerable to bypassing a token disapproval mechanism when an *Aniverse operator* gets revoked. The root cause of this issue is that the function uses *msg.sender* to get a function caller. Since the *ANIV721Land* contract supports meta transactions, adopting the *msg.sender*, in this case, can cause the bypassing issue.

Consider the following scenario to understand this issue.

1. A *contract owner* (also the *TokenA owner*) signs a meta transaction to approve the *TokenA* to an *Aniverse operator*.

2. *Anyone* (including the *contract owner* itself) submits the signed meta-transaction payload to the ***NativeMetaTransaction.executeMetaTransaction()*** function (L33 - 67 in code snippet 8.2).

3. The *executeMetaTransaction* function verifies the payload and executes the target ***ERC721Tradable.approve(AniverseOperator's address, TokenA's id)*** function (L61 - 63 in code snippet 8.2)

4. The *approve* function's execution flow would not execute the *_addLandToOperator* function (L110 in code snippet 8.1) since the *msg.sender* would demonstrate that the function caller is the contract itself (i.e., ***this*** address), not the *contract owner*.

   Consequently, the *approved Aniverse operator* would not track the approval of the *TokenA*.

5. The *contract owner* executes the ***ERC721Tradable.revokeOperator(AniverseOperator's address)*** function to revoke the *Aniverse* operator (L119 - 125 in code snippet 8.3). At this step, the approval of the *TokenA* to the *revoking operator* would not be disapproved.

6. The *revoked operator* has the full right to operate on the *TokenA*, even transfer the token to itself, since it is not an *Aniverse operator* anymore.

ERC721Tradable.sol

```
function approve(address to, uint256 tokenId) public override {
    if (msg.sender == owner()) {
        _addLandToOperator(to, tokenId);
    }
    super.approve(to, tokenId);
}

// (...SNIPPED...)

function _addLandToOperator(address to, uint256 tokenId) internal virtual {
    require(isOperator(to), "Address is not operator");
    require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
    uint256[] storage _tokenId = _operartorLandApproval[to];
    _tokenId.push(tokenId);
    _operartorLandApproval[to] = _tokenId;
}
```

Listing 8.1 The vulnerable *approve* function

NativeMetaTransaction.sol

```
function executeMetaTransaction(
    address userAddress,
    bytes memory functionSignature,
    bytes32 sigR,
    bytes32 sigS,
    uint8 sigV
) public payable returns (bytes memory) {
    MetaTransaction memory metaTx = MetaTransaction({
        nonce: nonces[userAddress],
        from: userAddress,
        functionSignature: functionSignature
    });

    require(
        verify(userAddress, metaTx, sigR, sigS, sigV),
        "Signer and signature do not match"
    );

    // increase nonce for user (to avoid re-use)
    nonces[userAddress] = nonces[userAddress].add(1);

    emit MetaTransactionExecuted(
        userAddress,
        payable(msg.sender),
        functionSignature
    );

    // Append userAddress and relayer address at the end to extract it from calling context
    (bool success, bytes memory returnData) = address(this).call(
        abi.encodePacked(functionSignature, userAddress)
    );
    require(success, "Function call not successful");

    return returnData;
}
```

Listing 8.2 The *executeMetaTransaction* function that allows anyone to submit a meta transaction to invoke *ANIV721Land* contract's functions

ERC721Tradable.sol

```
function revokeOperator(address to) public onlyOwner {
    _revokeOperator(to);
    uint256[] memory _tokenId = _operartorLandApproval[to];
    for (uint256 i = 0; i < _tokenId.length; i++) {
        _approve(address(0), _tokenId[i]);
    }
}
```

Listing 8.3 The *revokeOperator* function that revokes an *Aniverse operator* and disapproves all the *operator*'s (tracked) approved tokens

Recommendations

We recommend calling the *_msgSender* function (L109 in the code snippet below) instead of using the *msg.sender* to get a legitimate function caller.

ERC721Tradable.sol

```
101 function _msgSender() internal view override returns (address sender) {
102     return ContextMixin.msgSender();
103 }

    // (...SNIPPED...)

108 function approve(address to, uint256 tokenId) public override {
109     if (_msgSender() == owner()) {
110         _addLandToOperator(to, tokenId);
111     }
112     super.approve(to, tokenId);
113 }
```

Listing 8.4 The improved *approve* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed according to our recommendation.

No. 9	Possible Cross-Chain Replay Attack Over Meta Transactions		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/common/meta-transactions/EIP712Base.sol*		
Locations	*EIP712Base.sol L: 67 - 76*		

Detailed Issue

In the *EIP712Base* contract, the *_initializeEIP712* function (L27 - 34 in code snippet 9.1) would be executed only once at a contract construction. The *_initializeEIP712* function would invoke the *_setDomainSeperator* function (L33) to compute the state variable *domainSeperator* (L37).

One of the integral components of the *domainSeperator* is the *chainId* (L43) that would be used to prevent a replay attack across the blockchain networks.

We found that the computed *domainSeperator* would be used to calculate a typed message hash in the *toTypedMessageHash* function (L74 in code snippet 9.2). Since the *domainSeperator* would be initialized only once at a contract construction, the *chainId* variable would not be updated if the hard fork of the chain occurs. This issue opens room for a cross-chain replay attack, as a signed message payload from a user/signer would be executable on both the forked chains.

As a result, an attacker can use a valid signed message executed on one forked chain to replay and execute a transaction on behalf of a user/signer on another forked chain.

EIP712Base.sol

```
27  function _initializeEIP712(
28      string memory name
29  )
30      internal
31      initializer
32  {
33      _setDomainSeperator(name);
34  }
35
36  function _setDomainSeperator(string memory name) internal {
37      domainSeperator = keccak256(
38          abi.encode(
            EIP712_DOMAIN_TYPEHASH,
            keccak256(bytes(name)),
            keccak256(bytes(ERC712_VERSION)),
            address(this),
            bytes32(getChainId())
        )
    );
}

// (...SNIPPED...)

function getChainId() public view returns (uint256) {
    uint256 id;
    assembly {
        id := chainid()
    }
    return id;
}
```

Listing 9.1 The *domainSeperator* would be constructed only once by the *_initializeEIP712* function

EIP712Base.sol

```
function getDomainSeperator() public view returns (bytes32) {
    return domainSeperator;
}

// (...SNIPPED...)

function toTypedMessageHash(bytes32 messageHash)
    internal
    view
    returns (bytes32)
{
    return
        keccak256(
            abi.encodePacked("\x19\x01", getDomainSeperator(), messageHash)
        );
}
```

Listing 9.2 The *domainSeperator* would be reused every time to compute a typed message hash in the *toTypedMessageHash* function

Recommendations

We recommend computing the *domainSeperator* every time when calculating a typed message hash. In other words, we compute the *domainSeperator* in the *getDomainSeperator* function (L48 - 59) as presented in the below code snippet.

However, the suggested code may consume more gas when compared to the original code. For the gas optimization solution, please consider the *EIP712* contract of *OpenZeppelin* as a reference, link: *https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/EIP712.sol* .

EIP712Base.sol

```
48  function getDomainSeperator() public view returns (bytes32) {
49      return
50          keccak256(
51              abi.encode(
52                  EIP712_DOMAIN_TYPEHASH,
53                  keccak256(bytes(name)),
54                  keccak256(bytes(ERC712_VERSION)),
55                  address(this),
56                  bytes32(getChainId())
57              )
58          );
59  }

    // (...SNIPPED...)

76  function toTypedMessageHash(bytes32 messageHash)
77      internal
78      view
79      returns (bytes32)
80  {
81      return
82          keccak256(
83              abi.encodePacked("\x19\x01", getDomainSeperator(), messageHash)
84          );
85  }
```

Listing 9.3 Computing the *domainSeperator* every time when calculating a typed message hash

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed in accordance with our suggestion.

No. 10	**Recommended Changing Visibility Of State Variables For Transparency**		
Risk	**Medium**	**Likelihood**	**Medium**
		Impact	**Medium**
Functionality is in use	**In use**	**Status**	**Fixed**
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 41 and 49*		

Detailed Issue

We found that the ***_operartorLandApproval*** was declared a ***private*** state variable (L41 in the below code snippet) whereas the ***proxyRegistryAddress*** was declared an ***internal*** state variable (L49).

The current visibilities would not allow platform users to examine the variables' state via a blockchain explorer which may raise concerns in the community about transparency and traceability issues.

For this reason, we consider that the visibility of the state variables ***_operartorLandApproval*** and ***proxyRegistryAddress*** should be declared ***public*** to improve transparency and traceability issues.

ERC721Tradable.sol

```
    // (...SNIPPED...)

29  abstract contract ERC721Tradable is
30      ERC721,
31      ContextMixin,
32      NativeMetaTransaction,
33      Operator,
34      Ownable
35  {
36      using SafeMath for uint256;
37      using Counters for Counters.Counter;
38  
39      bool IS_USE_OPENSEA_PROXY;
40  
41      mapping(address => uint256[]) private _operartorLandApproval;
42  
43      /**
44       * We rely on the OZ Counter util to keep track of the next available ID.
45       * We track the nextTokenId instead of the currentTokenId to save users on
    gas costs.
```

```
46        * Read more about it here:
     https://shiny.mirror.xyz/OUampBbIz9ebEicfGnQf5At_ReMHlZy0tB4glb9xQ0E
47        */
48
49       address proxyRegistryAddress;

     // (...SNIPPED...)
```

Listing 10.1 The associated _*operartorLandApproval* and *proxyRegistryAddress* state variables

Recommendations

We recommend changing the visibility of the state variables ***_operartorLandApproval*** (L41) and ***proxyRegistryAddress*** (L49) to improve transparency and traceability issues as presented in the code snippet below.

ERC721Tradable.sol

```
     // (...SNIPPED...)

29   abstract contract ERC721Tradable is
30       ERC721,
31       ContextMixin,
32       NativeMetaTransaction,
33       Operator,
34       Ownable
35   {
36       using SafeMath for uint256;
37       using Counters for Counters.Counter;
38
39       bool IS_USE_OPENSEA_PROXY;
40
41       mapping(address => uint256[]) public _operartorLandApproval;
42
43       /**
44        * We rely on the OZ Counter util to keep track of the next available ID.
45        * We track the nextTokenId instead of the currentTokenId to save users on
     gas costs.
46        * Read more about it here:
     https://shiny.mirror.xyz/OUampBbIz9ebEicfGnQf5At_ReMHlZy0tB4glb9xQ0E
47        */
48
49       address public proxyRegistryAddress;

     // (...SNIPPED...)
```

Listing 10.2 The public _*operartorLandApproval* and *proxyRegistryAddress* state variables

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue as per our suggestion.

No. 11	Potential Approval Of Duplicated Token IDs		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 127 - 133*		

Detailed Issue

The *ERC721Tradable* contract keeps track of all permitted operators using the *_operartorLandApproval* mapping. A contract owner can approve their land token (i.e., *tokenId*) to an operator (i.e., *to*) by invoking the *_addLandToOperator* function (L110 in the code snippet below) through the *approve* function (L108 - 113).

However, we detected the possibility of approving a duplicated *tokenId* to an operator since the *_addLandToOperator* function does not check for a duplicated *tokenId* before pushing it into the operator's approval tracking array, *_tokenId* (L131).

Subsequently, on a contract owner invoking the *revokeOperator* function to revoke a specific operator, the duplicated *tokenIds* make the *revokeOperator* function consume more unnecessary gas.

ERC721Tradable.sol

```
108  function approve(address to, uint256 tokenId) public override {
109      if (msg.sender == owner()) {
110          _addLandToOperator(to, tokenId);
111      }
112      super.approve(to, tokenId);
113  }

     // (...SNIPPED...)

127  function _addLandToOperator(address to, uint256 tokenId) internal virtual {
128      require(isOperator(to), "Address is not operator");
129      require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
130      uint256[] storage _tokenId = _operartorLandApproval[to];
131      _tokenId.push(tokenId);
132      _operartorLandApproval[to] = _tokenId;
133  }
```

Listing 11.1 The *_addLandToOperator* function that does not check for duplicated *tokenIds*

Recommendations

We recommend updating the *ERC721Tradable* contract to check for duplicated *tokenIds* as shown in the below code snippet. More specifically, the mapping *_operatorTokenApproval* was added to track the approval of a specific *tokenId* to a particular operator (L43).

The *_addLandToOperator* function was improved to detect if a *tokenId* was already approved for the given operator or not (L133). The function would allow the approval if and only if the specified *tokenId* was not approved before.

ERC721Tradable.sol

```
 43 mapping(address => mapping(uint256 => bool)) public _operatorTokenApproval;

    // (...SNIPPED...)

121 function revokeOperator(address to) public onlyOwner {
122     _revokeOperator(to);
123     uint256[] memory _tokenId = _operartorLandApproval[to];
124     for (uint256 i = 0; i < _tokenId.length; i++) {
125         _approve(address(0), _tokenId[i]);
126         _operatorTokenApproval[to][_tokenId[i]] = false;
127     }
128 }

    // (...SNIPPED...)

130 function _addLandToOperator(address to, uint256 tokenId) internal virtual {
131     require(isOperator(to), "Address is not operator");
132     require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
133     require(!_operatorTokenApproval[to][tokenId], "tokenId was approved");
134     uint256[] storage _tokenId = _operartorLandApproval[to];
135
136     if (getApproved(tokenId) != address(0)) {
137         _operatorTokenApproval[getApproved(tokenId)][tokenId] = false;
138     }
139
140     _tokenId.push(tokenId);
141     _operartorLandApproval[to] = _tokenId;
142     _operatorTokenApproval[to][tokenId] = true;
143 }
```

Listing 11.2 The improved *revokeOperator* and *_addLandToOperator* functions that check for duplicated *tokenIds*

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue according to our suggestion.

No. 12	Lack Of Clearing Land Approval Array Of Revoked Operator		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 119 - 125*		

Detailed Issue

The *revokeOperator* function revokes an *Aniverse operator* (L120 in code snippet 12.1) and then disapproves all tokens ever approved to the *operator* (L122 - 124).

However, we discovered that the *revokeOperator* function revokes an operator without clearing the land approval array of that operator (*_operartorLandApproval[to]*), resulting in the possibility of disapproving an address other than the revoked operator's address in the future.

Let's consider the following scenario to learn more about this issue.

1. A *contract owner* (also the *TokenA owner*) calls the ***approve(AniverseOperatorBob's address, TokenA's id)*** function to approve the *TokenA* to the *Aniverse operator, Bob* (L108 - 113 in code snippet 12.2).
2. The *contract owner* executes the ***revokeOperator(AniverseOperatorBob's address)*** function to revoke the *operator Bob*. At this step, *Bob* is revoked (L120 in code snippet 12.1) and the approval of the *TokenA* is disapproved (L122 - 124).

   Nonetheless, the *revokeOperator* function does not clear the land approval array of the *operator Bob* (*_operartorLandApproval[AniverseOperatorBob's address]*) at this step.
3. The *contract owner* executes the ***approve(AniverseOperatorAlice's address, TokenA's id)*** function to approve the *TokenA* to *another operator, Alice* (L108 - 113 in code snippet 12.2).
4. The *contract owner* invokes the ***addOperator(AniverseOperatorBob's address)*** function to add the *operator Bob* back to work again (L115 - 117 in code snippet 12.3).
5. The *contract owner* executes the ***revokeOperator(AniverseOperatorBob's address)*** function to revoke the *operator Bob* again.

At this step, *Bob* is revoked but the approval of the *TokenA* to the *operator Alice* gets disapproved unexpectedly since the land approval array of the *operator Bob* (*_operartorLandApproval[AniverseOperatorBob's address]*) was not previously cleared in Step 2.

ERC721Tradable.sol

```
function revokeOperator(address to) public onlyOwner {
    _revokeOperator(to);
    uint256[] memory _tokenId = _operartorLandApproval[to];
    for (uint256 i = 0; i < _tokenId.length; i++) {
        _approve(address(0), _tokenId[i]);
    }
}
```

Listing 12.1 The *revokeOperator* function that
does not clear the land approval array of a revoked operator

ERC721Tradable.sol

```
function approve(address to, uint256 tokenId) public override {
    if (msg.sender == owner()) {
        _addLandToOperator(to, tokenId);
    }
    super.approve(to, tokenId);
}

// (...SNIPPED...)

function _addLandToOperator(address to, uint256 tokenId) internal virtual {
    require(isOperator(to), "Address is not operator");
    require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
    uint256[] storage _tokenId = _operartorLandApproval[to];
    _tokenId.push(tokenId);
    _operartorLandApproval[to] = _tokenId;
}
```

Listing 12.2 The *approve* and *_addLandToOperator* functions

ERC721Tradable.sol

```
function addOperator(address to) public onlyOwner {
    _addOperator(to);
}
```

Listing 12.3 The *addOperator* function

Recommendations

We recommend clearing the land approval array after revoking any operator like L125 in the code snippet below.

ERC721Tradable.sol

```
119 function revokeOperator(address to) public onlyOwner {
120     _revokeOperator(to);
121     uint256[] memory _tokenId = _operartorLandApproval[to];
122     for (uint256 i = 0; i < _tokenId.length; i++) {
123         _approve(address(0), _tokenId[i]);
124     }
125     delete _operartorLandApproval[to];
126 }
```

Listing 12.4 The improved *revokeOperator* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue in accordance with our suggestion.

No. 13	Possibly Incorrect Token Disapproval		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 119 - 125*		

Detailed Issue

The *revokeOperator* function would typically revoke an *Aniverse operator* (L120 in code snippet 13.1) and then disapprove all tokens ever approved to the *operator* (L122 - 124).

Nevertheless, we found the case that the *revokeOperator* function can operate incorrectly. Specifically, the *revokeOperator* function can disapprove an address other than the revoking *Aniverse operator*.

To elaborate on the issue, let's consider the following scenario.

1. A *contract owner* (also the *TokenA owner*) calls the ***ERC721Tradable.approve(AniverseOperator's address, TokenA's id)*** function to approve the *TokenA* to the *Aniverse operator*.
2. The *contract owner* executes the ***ERC721.setApprovalForAll(Bob's address, true)*** function (L136 - 138 in code snippet 13.2) to approve *Bob* as an *external operator* (the *operator* tracked by the *ERC721* contract, not the *Aniverse operator*) to operate on all of the *owner*'s tokens, including the *TokenA*.
3. The *external operator Bob* invokes the ***ERC721.approve(Alice's address, TokenA's id)*** function (L112 - 122 in code snippet 13.3) to approve the *TokenA* to *Alice*. At this step, the approval of *TokenA* has been changed from the *Aniverse operator* to *Alice* now.
4. The *contract owner* executes ***ERC721Tradable.revokeOperator(AniverseOperator's address)*** function to revoke the *Aniverse* operator (L119 - 125 in code snippet 13.1). At this step, *Alice*'s approval for the *TokenA* would be disapproved unexpectedly since the *TokenA*'s id was still tracked by the *revoking operator*.

ERC721Tradable.sol

```solidity
function revokeOperator(address to) public onlyOwner {
    _revokeOperator(to);
    uint256[] memory _tokenId = _operartorLandApproval[to];
    for (uint256 i = 0; i < _tokenId.length; i++) {
        _approve(address(0), _tokenId[i]);
    }
}
```

Listing 13.1 The *revokeOperator* function that can disapprove an address other than the revoking *Aniverse operator*

ERC721.sol

```solidity
function setApprovalForAll(address operator, bool approved) public virtual override {
    _setApprovalForAll(_msgSender(), operator, approved);
}

// (...SNIPPED...)

function _setApprovalForAll(
    address owner,
    address operator,
    bool approved
) internal virtual {
    require(owner != operator, "ERC721: approve to caller");
    _operatorApprovals[owner][operator] = approved;
    emit ApprovalForAll(owner, operator, approved);
}
```

Listing 13.2 The *setApprovalForAll* and *_setApprovalForAll* functions of the *ERC721* contract

ERC721.sol

```solidity
function approve(address to, uint256 tokenId) public virtual override {
    address owner = ERC721.ownerOf(tokenId);
    require(to != owner, "ERC721: approval to current owner");

    require(
        _msgSender() == owner || isApprovedForAll(owner, _msgSender()),
        "ERC721: approve caller is not owner nor approved for all"
    );

    _approve(to, tokenId);
}
```

Listing 13.3 The *approve* function of the *ERC721* contract

Recommendations

We recommend updating the *revokeOperator* function as the code snippet below. The function would check whether or not the currently approved address for each token equals a *revoking operator* (L123), and the function would disapprove a token if and only if the currently approved address is the *revoking operator* (L124).

ERC721Tradable.sol

```
119 function revokeOperator(address to) public onlyOwner {
120     _revokeOperator(to);
121     uint256[] memory _tokenId = _operartorLandApproval[to];
122     for (uint256 i = 0; i < _tokenId.length; i++) {
123         if (getApproved(_tokenId[i]) == to) {
124             _approve(address(0), _tokenId[i]);
125         }
126     }
127 }
```

Listing 13.4 The improved *revokeOperator* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue according to our suggestion.

No. 14	Recommended Adding A Setter Function For Base Token URI		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/ANIV721Land.sol*		
Locations	*ANIV721Land.sol L: 22 - 24*		

Detailed Issue

The *ANIV721Land* contract has the *baseTokenURI* function (L22 - 24 in the code snippet below) indicating the base token URI for each land token of the platform.

However, we noticed that the base token URI is hard coded in the current implementation (L23) which cannot be changed after the contract deployment. If the base token URI has to be updated somehow, the developer would have no solution to updating this base URI. This issue can render all land tokens' metadata to be inaccessible.

ANIV721Land.sol

```
22  function baseTokenURI() public pure override returns (string memory) {
23      return "https://api-asset-dev.aniv.io/OpenSeaLand/by_token/";
24  }
```

Listing 14.1 The *baseTokenURI* function of the *ANIV721Land* contract

Recommendations

We recommend adding a setter function for updating the base token URI. However, the setter function should be under the control of the *Timelock* mechanism.

If possible, furthermore, all land tokens' metadata should be hosted on a decentralized storage system, such as *IPFS*, to ensure the availability and integrity of the metadata.

Reassessment

The *Aniverse* team fixed this issue by adding a setter function for updating the base token URI according to our recommendation. Nonetheless, the setter function would not be under the control of the *Timelock* mechanism.

No. 15	**Recommended Event Emissions For Transparency And Traceability**		
Risk	**Low**	**Likelihood**	**Medium**
		Impact	**Low**
Functionality is in use	**In use**	**Status**	**Fixed**
Associated Files	*contracts/Operator.sol* *contracts/erc721/ERC721Tradable.sol*		
Locations	*Operator.sol L: 7 - 11 and 13 - 17* *ERC721Tradable.sol L: 51 - 59 and 127 - 133*		

Detailed Issue

We consider operations of the following state-changing functions important and require proper event emissions for improving transparency and traceability.

- ***_addOperator*** function *(L7 - 11 in code snippet 15.1)*
- ***_revokeOperator*** function *(L13 - 17 in code snippet 15.1)*
- ***constructor*** *(L51 - 59 in code snippet 15.2)*
- ***_addLandToOperator*** function *(L127 - 133 in code snippet 15.2)*

Operator.sol

```
 4  contract Operator {
 5     mapping(address => bool) private _operators;
 6
 7     function _addOperator(address operatorAddr) internal virtual {
 8         require(operatorAddr != address(0), "Operator can't be address zero");
 9         require(!_operators[operatorAddr], "Duplicate operator");
10         _operators[operatorAddr] = true;
11     }
12
13     function _revokeOperator(address operatorAddr) internal virtual {
14         require(operatorAddr != address(0), "Operator can't be address zero");
15         require(_operators[operatorAddr], "operator not found");
16         delete _operators[operatorAddr];
17     }

       // (...SNIPPED...)
}
```

Listing 15.1 The *_addOperator* and *_revokeOperator* functions

ERC721Tradable.sol

```
constructor(
    string memory _name,
    string memory _symbol,
    address _proxyRegistryAddress
   ) ERC721(_name, _symbol) {
    IS_USE_OPENSEA_PROXY = false;
    proxyRegistryAddress = _proxyRegistryAddress;
    _initializeEIP712(_name);
}

// (...SNIPPED...)

function _addLandToOperator(address to, uint256 tokenId) internal virtual {
    require(isOperator(to), "Address is not operator");
    require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
    uint256[] storage _tokenId = _operartorLandApproval[to];
    _tokenId.push(tokenId);
    _operartorLandApproval[to] = _tokenId;
}
```

Listing 15.2 The *constructor* and *_addLandToOperator* functions

Recommendations

We recommend emitting relevant events in the associated functions to improve transparency and traceability like the code snippets 15.3 and 15.4 below.

Operator.sol

```
contract Operator {
   mapping(address => bool) private _operators;

   event AddOperator(address indexed operatorAddr);
   event RevokeOperator(address indexed operatorAddr);

   function _addOperator(address operatorAddr) internal virtual {
       require(operatorAddr != address(0), "Operator can't be address zero");
       require(!_operators[operatorAddr], "Duplicate operator");
       _operators[operatorAddr] = true;
       emit AddOperator(operatorAddr);
   }
```

```

    function _revokeOperator(address operatorAddr) internal virtual {
        require(operatorAddr != address(0), "Operator can't be address zero");
        require(_operators[operatorAddr], "operator not found");
        delete _operators[operatorAddr];
        emit RevokeOperator(operatorAddr);
    }

    // (...SNIPPED...)
}
```

Listing 15.3 The improved *_addOperator* and *_revokeOperator* functions

ERC721Tradable.sol

```
event SetIsUseOpenseaProxy(bool indexed isUseOpenseaProxy);
event SetProxyRegistryAddress(address indexed proxyRegistryAddress);
event AddLandToOperator(uint256 indexed tokenId, address indexed operatorAddr);

constructor(
    string memory _name,
    string memory _symbol,
    address _proxyRegistryAddress
   ) ERC721(_name, _symbol) {
    IS_USE_OPENSEA_PROXY = false;
    proxyRegistryAddress = _proxyRegistryAddress;
    _initializeEIP712(_name);

    emit SetIsUseOpenseaProxy(IS_USE_OPENSEA_PROXY);
    emit SetProxyRegistryAddress(proxyRegistryAddress);
}

// (...SNIPPED...)

function _addLandToOperator(address to, uint256 tokenId) internal virtual {
    require(isOperator(to), "Address is not operator");
    require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
    uint256[] storage _tokenId = _operartorLandApproval[to];
    _tokenId.push(tokenId);
    _operartorLandApproval[to] = _tokenId;
    emit AddLandToOperator(tokenId, to);
}
```

Listing 15.4 The improved *constructor* and *_addLandToOperator* functions

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed by emitting proper events on all associated functions.

No. 16	Possibly Minting Out-Of-Bound Token ID		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/ANIV721Land.sol*		
Locations	*ANIV721Land.sol L: 30 - 34*		

Detailed Issue

The *ANIV721Land* contract has a function for minting a land token named the *mint* function (L30 - 34 in the code snippet below). The function is restricted to a contract owner to invoke only. This function validates the total supply to limit the total amount of tokens that can be minted (L31).

However, we found that there are no bounds checking for the *tokenId* parameter before minting which could allow an owner to mint a land token with an out-of-bound *tokenId* mistakenly.

ANIV721Land.sol

```
30  function mint(address _to, uint256 tokenId) public onlyOwner {
31      require(_totalSupply.current() < MAX_LANDS, "tokenId is out of bounds");
32      _safeMint(_to, tokenId);
33      _totalSupply.increment();
34  }
```

Listing 16.1 The *mint* function that lacks of bounds checking for the *tokenId* parameter

Recommendations

We recommend adding the *require* statement to check whether the `tokenId` is exceeding the *MAX_LANDS* or not like L32 in the code snippet below.

ANIV721Land.sol

```
30  function mint(address _to, uint256 tokenId) public onlyOwner {
31      require(_totalSupply.current() < MAX_LANDS, "tokenId is out of bounds");
32      require(tokenId < MAX_LANDS, "tokenId must be less than MAX_LANDS");
33      _safeMint(_to, tokenId);
34      _totalSupply.increment();
}
```

Listing 16.2 The improved *mint* function adding the *tokenId* validation

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team remediated this issue by validating that the range of the inputted *tokenId* must be between 1 to *MAX_LANDS* (including the lower and upper bounds).

ANIV721Land.sol

```
function mint(address _to, uint256 tokenId) public onlyOwner {
    require(_totalSupply.current() < MAX_LANDS, "Total supply is Maxed");
    require(tokenId > 0 && tokenId <= MAX_LANDS, "Token Id must be more than 0 AND less than or equal to MAX_LANDS");
    _safeMint(_to, tokenId);
    _totalSupply.increment();
}
```

Listing 16.3 The fixed *mint* function

No. 17	Lack Of Validating Existence Of Token ID		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 68 - 78*		

Detailed Issue

The *ERC721Tradable* contract implements the *tokenURI* function (the code snippet below) to encode and return the token URI in accordance with the inputted *_tokenId* parameter (L76).

We discovered that the *tokenURI* function does not verify the existence of the inputted *_tokenId* parameter. Specifically, if the parameter *_tokenId* represents a non-existent token id, the *tokenURI* function would return an invalid token URI.

ERC721Tradable.sol

```
68  function tokenURI(uint256 _tokenId)
69          public
70          pure
71          override
72          returns (string memory)
73  {
74      return
75          string(
76              abi.encodePacked(baseTokenURI(), Strings.toString(_tokenId))
77          );
78  }
```

Listing 17.1 The *tokenURI* function that
does not verify the existence of the inputted *_tokenId* parameter

Recommendations

We recommend verifying the existence of the inputted *_tokenId* parameter before computing the token URI as shown in L74 in the following code snippet.

ERC721Tradable.sol

```
68 function tokenURI(uint256 _tokenId)
69         public
70         view
71         override
72         returns (string memory)
73 {
74     require(_exists(_tokenId), "_tokenId does not exist");
75     return
76         string(
77             abi.encodePacked(baseTokenURI(), Strings.toString(_tokenId))
78         );
79 }
```

Listing 17.2 The improved *tokenURI* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue as per our recommendation.

No. 18	Recommended Removing Redundant Logic		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 132*		

Detailed Issue

We detected a redundant logic in the *_addLandToOperator* function (L132 in the following code snippet). Since the array *_tokenId* would be loaded by reference (L130), the “***_operartorLandApproval[to] = _tokenId***” statement in L132 is not necessary and can be removed for gas savings.

ERC721Tradable.sol

```
127 function _addLandToOperator(address to, uint256 tokenId) internal virtual {
128     require(isOperator(to), "Address is not operator");
129     require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
130     uint256[] storage _tokenId = _operartorLandApproval[to];
131     _tokenId.push(tokenId);
132     _operartorLandApproval[to] = _tokenId;
133 }
```

Listing 18.1 The *_addLandToOperator* function that contains a redundant logic

Recommendations

We recommend removing the redundant logic from the *_addLandToOperator* function for saving gas as shown in the code snippet below.

ERC721Tradable.sol

```
function _addLandToOperator(address to, uint256 tokenId) internal virtual {
    require(isOperator(to), "Address is not operator");
    require(ERC721.ownerOf(tokenId) == owner(), "Land not owned by owner");
    uint256[] storage _tokenId = _operartorLandApproval[to];
    _tokenId.push(tokenId);
}
```

Listing 18.2 The improved *_addLandToOperator* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Aniverse* team fixed this issue by removing the redundant logic as per our suggestion.

<table>
<tr><th>No. 19</th><th colspan="3">Inconsistent Error Message With The Code</th></tr>
<tr><td rowspan="2">Risk</td><td rowspan="2">Informational</td><td>Likelihood</td><td>Low</td></tr>
<tr><td>Impact</td><td>Low</td></tr>
<tr><td>Functionality is in use</td><td>In use</td><td>Status</td><td>Fixed</td></tr>
<tr><td>Associated Files</td><td colspan="3"><i>contracts/ANIV721Land.sol</i></td></tr>
<tr><td>Locations</td><td colspan="3"><i>ANIV721Land.sol L: 31</i></td></tr>
</table>

Detailed Issue

We found an error message inconsistent with the code in the function *mint* (L31 in the code snippet below). This inconsistency can lead to misunderstanding among users or developers when maintaining the source code.

ANIV721Land.sol

```
30  function mint(address _to, uint256 tokenId) public onlyOwner {
31      require(_totalSupply.current() < MAX_LANDS, "tokenId is out of bounds");
32      _safeMint(_to, tokenId);
33      _totalSupply.increment();
34  }
```

Listing 19.1 The *mint* function with an inconsistent error message

Recommendations

We recommend revising the associated error message to reflect the actual code.

Reassessment

The *Aniverse* team revised the error message to fix this issue.

No. 20	Recommended Removing Unused State Variable		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 39*		

Detailed Issue

We found that the *ERC721Tradable* contract declares an unused state variable named *IS_USE_OPENSEA_PROXY* (L39 in the code snipped below). This unused variable can be removed to save contract deployment gas and improve code readability.

ERC721Tradable.sol

```
 29  abstract contract ERC721Tradable is
 30      ERC721,
 31      ContextMixin,
 32      NativeMetaTransaction,
 33      Operator,
 34      Ownable
 35  {
 36      using SafeMath for uint256;
 37      using Counters for Counters.Counter;
 38
 39      bool IS_USE_OPENSEA_PROXY;

         // (...SNIPPED...)
145  }
```

Listing 20.1 The unused state variable *IS_USE_OPENSEA_PROXY*

Recommendations

We recommend removing the unused state variable *IS_USE_OPENSEA_PROXY* to save contract deployment gas and improve code readability.

Reassessment

The *Aniverse* team removed the unused state variable *IS_USE_OPENSEA_PROXY* according to our recommendation.

No. 21	Inconsistent Contract Name		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/erc721/common/meta-transactions/ContentMixin.sol*		
Locations	*ContentMixin.sol L: 5*		

Detailed Issue

We found inconsistency between the ***file name*** (***ContentMixin***) and the ***contract name*** (***ContextMixin***) as presented in the below code snippet, which can confuse the users and developers.

ContentMixin.sol

```
   // (...SNIPPED...)

 5 abstract contract ContextMixin {
 6     function msgSender()
 7         internal
 8         view
 9         returns (address payable sender)
10     {

   // (...SNIPPED...)
```

Listing 21.1 The contract name *ContextMixin*

Recommendations

We recommend renaming the associated contract and file names to be consistent.

Reassessment

The associated file name was renamed from ***ContentMixin.sol*** to ***ContextMixin.sol*** to be consistent with the contract name.

No. 22	Depending On External Contract		
Risk	Informational	Likelihood	Low
		Impact	Undetermined
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/erc721/ERC721Tradable.sol*		
Locations	*ERC721Tradable.sol L: 21 - 23 and 83 - 96*		

Detailed Issue

The *isApprovedForAll* function of the *ERC721Tradable* contract (code snippet 22.1) relies on an external contract named *ProxyRegistry* (L90 - 91). Considering the implementation of the *ProxyRegistry* contract (code snippet 22.2), we noticed that the contract is just a prototype (incomplete) implementation.

In the deployment time, a complete implementation of the *ProxyRegistry* contract must be required. We, therefore, recommend the *Aniverse* team do a full security audit for the complete version of the *ProxyRegistry* contract to guarantee the security of the contract.

ERC721Tradable.sol

```
83  function isApprovedForAll(address owner, address operator)
84      public
85      view
86      override
87      returns (bool)
88  {
89      // Whitelist OpenSea proxy contract for easy trading.
90      ProxyRegistry proxyRegistry = ProxyRegistry(proxyRegistryAddress);
91      if (address(proxyRegistry.proxies(owner)) == operator) {
92          return true;
93      }
94
95      return super.isApprovedForAll(owner, operator);
96  }
```

Listing 22.1 The *isApprovedForAll* function
that depends on an external *ProxyRegistry* contract

ERC721Tradable.sol

```
contract ProxyRegistry {
    mapping(address => OwnableDelegateProxy) public proxies;
}
```

Listing 22.2 A prototype implementation of the *ProxyRegistry* contract

Recommendations

A complete implementation of the *ProxyRegistry* contract must be required in the deployment time. We, therefore, recommend the *Aniverse* team do a full security audit for the complete version of the *ProxyRegistry* contract to guarantee the security of the contract.

Reassessment

The *Aniverse* team acknowledged this issue.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information

✉ info@valix.io

https://www.facebook.com/ValixConsulting

https://twitter.com/ValixConsulting

https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/

ValiX