CERTIK

Security Assessment

Sandbox - L2

Jul 5th, 2022

Table of Contents

Summary

This report has been prepared for Sandbox to discover issues and vulnerabilities in the source code of the Sandbox - L2 project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	Sandbox - L2
Platform	Polygon
Language	Solidity
Codebase	https://github.com/thesandboxgame/sandbox-smart-contracts-private
Commit	• 29be345b7c1a8c2c909da163a6286c250bbbb7ae • c698784244254ab912e5caef81c4ac04a53614c6

Audit Summary

Delivery Date	Jul 05, 2022 UTC
Audit Methodology	Static Analysis, Manual Review

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Mitigated	Partially Resolved	Resolved
● Critical	0	0	0	0	0	0	0
● Major	2	0	0	1	0	0	1
● Medium	0	0	0	0	0	0	0
● Minor	2	0	0	1	0	0	1
● Optimization	0	0	0	0	0	0	0
● Informational	5	0	0	2	0	0	3
● Discussion	0	0	0	0	0	0	0

Audit Scope

ID	File	SHA256 Checksum
IER	common/interfaces/IERC721MandatoryTokenReceiver.sol	9073b3da579f093123bf2419fef081f9aad801b46b6170d52dc49e0752fce800
IPL	common/interfaces/IPolygonLand.sol	cfafb021c4ac9c2e024d6164c946e42f25a30c4ab96972b865c53c424ddec6b2
ILT	common/interfaces/ILandToken.sol	aae5dcb417dec176726e5ade31275f71f5105ba336b8497ae7b9e7b88072c11f
ERB	common/BaseWithStorage/ERC721BaseToken.sol	166f9fc68a11515dbb0bcd0f3342f71b3816da737bd37c4317e142c9ad7f00bc
ERC	common/BaseWithStorage/ERC2771Handler.sol	1708c8a42025f3537fdc6a1f4ae74c4e1c537538bbd0f65896b822777186972c
LTC	polygon/root/land/LandTunnel.sol	6fa1e0429ef914e3ae5ae1968f9203963e641f38307642a95ab0e19d7da6f142
PLT	polygon/child/land/PolygonLandTunnel.sol	df65a08b6a83596d316cd70d8968c03ffcb333d5e90a18c806e34a0ad229ddf5
PLV	polygon/child/land/PolygonLandV1.sol	b7fd1e6f34f91ce892d48e366a6297732853d31d27b87e8455a4a89ef9c6b84f
PLB	polygon/child/land/PolygonLandBaseToken.sol	d97b15dd7b54afebc348b20c73d38e5b6466feb56291916ee9cd4a91095ae5f4

Overview

The Sandbox team has implemented the **Land Bridge** functionality. This feature allows the transfer of `LAND` tokens between the root chain (Ethereum) and the child chain (Polygon).

External Dependencies

The scope of the audit treats third-party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets.

There are a few dependent injection contracts or addresses in the current project:

- `AddressUpgradeable`, `IERC721ReceiverUpgradeable`, `IERC721Upgradeable`, `WithSuperOperators`, and `ERC2771Handler` for the contract `ERC721BaseToken`;
- `Initializable`, `ERC721BaseToken`, and `IPolygonLand` for the contract `PolygonLandBaseToken`;
- `PolygonLandBaseToken` and `
- `FxBaseChildTunnel`, `Ownable`, `Pausable`, and `childToken` for the contract `PolygonLandTunnel`;
- `FxBaseRootTunnel`, `Ownable`, `Pausable`, and `rootToken` for the contract `LandTunnel`.

We assume these contracts or addresses are valid and non-vulnerable actors and implement proper logic to collaborate with the current project.

Privileged Functions

In the contract `PolygonLandV1`, the role `_admin` has authority over the following functions:

- `setPolygonLandTunnel()`: Modify the `PolygonLandTunnel` address;
- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions.

In addition, the role `polygonLandTunnel` has authority over the following function:

- `mint()`: mints quads to an address.

The contract `PolygonLandV1` inherits the contract `PolygonLandBaseToken`, where the role `superOperator` has authority over the following functions:

- `batchTransferQuad()`: Transfer any user's quads to an address;
- `transferQuad()`: Transfer any user's quads to an address.

The contract `PolygonLandBaseToken` inherits the contract `ERC721BaseToken`, where `_admin` has authority over the following functions:

- `setSuperOperator()`: Give or remove the `superOperator` role to or from an address;
- `changeAdmin()`: Change the address of the role `_admin`.

In addition, the `superOperator` role has authority over the following functions:

- `approve()`: Decide the allowance of any token;
- `approveFor()`: Decide the allowance of any token;
- `transferFrom()`: Transfer any user's tokens to an address;
- `safeTransferFrom()`: Transfer any user's tokens to an address;
- `batchTransferFrom()`: Transfer several of a user's tokens to an address;
- `safeBatchTransferFrom()`: Transfer several of a user's tokens to an address;
- `setApprovalForAllFor()`: Set the approval for an address to manage all of a user's tokens;
- `burnFrom()`: Burn any user's tokens.

In the contract `LandTunnel`, the role `_owner` has authority over the following functions:

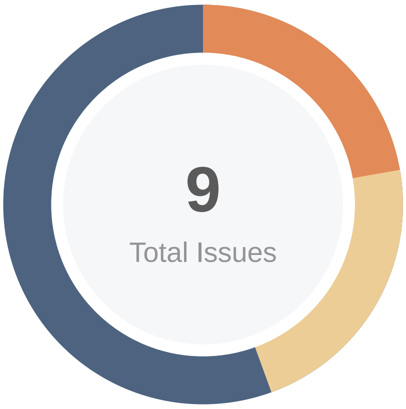
- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions;
- `pause()`: Pause the contract to disable transfers from L1 (Ethereum) to L2 (Polygon);
- `unpause()`: Unpause the contract, reenabling transfers from L1 (Ethereum) to L2 (Polygon).

In the contract `PolygonLandTunnel`, the role `_owner` has authority over the following functions:

- `setMaxLimitOnL1()`: Define the maximum amount of gas to spend for `batchTransferQuadToL1()` function;
- `setMaxAllowedQuads()`: Define the maximum amount of quads to transfer for `batchTransferQuadToL1()` function;
- `setLimit()`: Define the maximum amount of gas to spend on transfers for one quad type;
- `setupLimits()`: Define the maximum amount of gas to spend on transfers for each quad type;
- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions;
- `pause()`: Pause the contract, preventing transfers from L2 (Polygon) to L1 (Ethereum);
- `unpause()`: Unpause the contract, reenabling transfers from L2 (Polygon) to L1 (Ethereum).

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of `Timelock` contract.

Findings



🟥 **Critical**	**0** (0.00%)
🟧 **Major**	**2** (22.22%)
🟨 **Medium**	**0** (0.00%)
🟨 **Minor**	**2** (22.22%)
🟦 **Informational**	**5** (55.56%)
🟩 **Discussion**	**0** (0.00%)

ID	Title	Category	Severity	Status
GLOBAL-01	Centralization Related Risks	**Centralization / Privilege**	● **Major**	ⓘ Acknowledged
GLOBAL-02	Third Party Dependencies	Volatile Code	● Minor	ⓘ Acknowledged
CKP-01	Potential Denial-of-Service Attack	Logical Issue	● Informational	ⓘ Acknowledged
ERB-01	Possible To Approve A Burnt Token	Inconsistency	● Informational	✓ Resolved
ERB-02	Burning Logic Inconsistency On L1 And L2	Logical Issue	● Informational	ⓘ Acknowledged
PLB-01	Transferring Quads May Remove The Burn Status Of Tokens	Logical Issue	● Major	✓ Resolved
PLB-02	Possibly Incorrect Return For Function `exists()`	Volatile Code	● Minor	✓ Resolved
PLB-03	Inconsistent NatSpec For Minting Function	Inconsistency	● Informational	✓ Resolved
PLB-04	Redundant Code	Gas Optimization	● Informational	✓ Resolved

GLOBAL-01 | Centralization Related Risks

Category	Severity	Location	Status
Centralization / Privilege	● **Major**		ⓘ Acknowledged

Description

In the contract `PolygonLandV1`, the role `_admin` has authority over the following functions:

- `setPolygonLandTunnel()`: Modify the `PolygonLandTunnel` address;
- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions.

In addition, the role `polygonLandTunnel` has authority over the following function:

- `mint()`: mints quads to an address.

The contract `PolygonLandV1` inherits the contract `PolygonLandBaseToken`, where the role `superOperator` has authority over the following functions:

- `batchTransferQuad()`: Transfer any user's quads to an address;
- `transferQuad()`: Transfer any user's quads to an address.

The contract `PolygonLandBaseToken` inherits the contract `ERC721BaseToken`, where `_admin` has authority over the following functions:

- `setSuperOperator()`: Give or remove the `superOperator` role to or from an address;
- `changeAdmin()`: Change the address of the role `_admin`.

In addition, the `superOperator` role has authority over the following functions:

- `approve()`: Decide the allowance of any token;
- `approveFor()`: Decide the allowance of any token;
- `transferFrom()`: Transfer any user's tokens to an address;
- `safeTransferFrom()`: Transfer any user's tokens to an address;
- `batchTransferFrom()`: Transfer several of a user's tokens to an address;
- `safeBatchTransferFrom()`: Transfer several of a user's tokens to an address;
- `setApprovalForAllFor()`: Set the approval for an address to manage all of a user's tokens;
- `burnFrom()`: Burn any user's tokens.

In the contract `LandTunnel`, the role `_owner` has authority over the following functions:

- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions;
- `pause()`: Pause the contract to disable transfers from L1 (Ethereum) to L2 (Polygon);
- `unpause()`: Unpause the contract, reenabling transfers from L1 (Ethereum) to L2 (Polygon).

In the contract `PolygonLandTunnel`, the role `_owner` has authority over the following functions:

- `setMaxLimitOnL1()`: Define the maximum amount of gas to spend for `batchTransferQuadToL1()` function;
- `setMaxAllowedQuads()`: Define the maximum amount of quads to transfer for `batchTransferQuadToL1()` function;
- `setLimit()`: Define the maximum amount of gas to spend on transfers for one quad type;
- `setupLimits()`: Define the maximum amount of gas to spend on transfers for each quad type;
- `setTrustedForwarder()`: Modify the Trusted Forwarder for the Meta Transactions;
- `pause()`: Pause the contract, preventing transfers from L2 (Polygon) to L1 (Ethereum);
- `unpause()`: Unpause the contract, reenabling transfers from L2 (Polygon) to L1 (Ethereum).

Any compromise to the aforementioned privileged accounts may allow a hacker to take advantage of this authority and manipulate the reward system.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[Sandbox]: In the near future, Sandbox will introduce a DAO to decentralize the governance. In the long run, the admin role can also be renounced. A Multisig will be used on L2 too.

GLOBAL-02 | Third Party Dependencies

Category	Severity	Location	Status
Volatile Code	● Minor		ⓘ Acknowledged

Description

The contract is serving as the underlying entity to interact with third-party **fx-portal** protocols. The **fx-portal** protocol performs the cross-chain functionality to exchange messages between Ethereum and Polygon. Land Bridge interacts with the following contracts:

- FxBaseChildTunnel
- FxBaseRootTunnel

The scope of the audit treats 3rd party entities as black boxes and assumes their functional correctness. However, in the real world, 3rd parties can be compromised, which may lead to lost or stolen assets.

Recommendation

It is understandable that the business logic of Land Bridge requires interaction with **fx-portal**. It is recommended to constantly monitor the statuses of **fx-portal** to mitigate the side effects when unexpected activities are observed.

Alleviation

[Sandbox]: The team is aware of the trust that the team provides to the fx-portal library.

CKP-01 | Potential Denial-of-Service Attack

Category	Severity	Location	Status
Logical Issue	● Informational	polygon/child/land/PolygonLandBaseToken.sol: 118; LandBaseToken.sol (1e920d0): 75	ⓘ Acknowledged

Description

When the contract tries to mint a quad via `_mintQuad()`, there is a check to ensure that no quads containing the quad to mint and no quads (or LANDS) within the quad to mint have already been minted by calling the function `exists()`.

This leads to a possible denial-of-service attack where the attacker mints 1x1 LANDS at specific locations to prevent the minting of larger quads. For example, out of the possible 166,464 LAND placements, only 289 LANDS need to be minted to prevent 24x24 quads from occurring.

Proof of Concept

The following test shows that if 289 1x1 Land is minted, no one can mint 24x24 land anymore.

```
    it('Mint 289 land to prevent others mint 24x24', async function () {
      const {landOwners} = await setupTest();
      const bytes = '0x3333';

      let totalMint = 0;
      for (let x = 0; x < GRID_SIZE; x = x + 24) {
        for (let y = 0; y < GRID_SIZE; y = y + 24 ){
          await waitFor(
            landOwners[0].MockLandWithMint.mintQuad(
              landOwners[0].address,
              1,
              x,
              y,
              bytes
            ));
          //console.log('Minted a land on (%d,%d)', x , y)
          totalMint = totalMint + 1;
        }
      }
      console.log('Step1: Mint lands in each 24x24 land. In total %d 1x1 land is minted',
totalMint);

      let totalTry = 0;
      for (let x = 0; x < GRID_SIZE; x = x + 24) {
        for (let y = 0; y < GRID_SIZE; y = y + 24) {
          await expect(
            landOwners[0].MockLandWithMint.mintQuad(
              landOwners[0].address,
              24,
              x,
              y,
              bytes
            )
          ).to.be.revertedWith('Already minted');
          totalTry = totalTry + 1;
        }
      }
      console.log('Step2: Cannot Mint 24x24 land Anymore. In total %d times tried to mint
24x24 land', totalTry)
    });
```

The test case passed, meaning 24x24 land can no longer be minted

```
Step1: Mint lands in each 24x24 land. In total 289 1x1 land is minted
Step2: Cannot Mint 24x24 land Anymore. In total 289 times tried to mint 24x24 land
      ✓ Mint 289 land to prevent others mint 24x24 (18107ms)
```

Recommendation

We recommend only allowing mints of larger quads if this is not intended.

Alleviation

[Sandbox]: The team acknowledged this finding and decided to not change the codebase at this time.

ERB-01 | Possible To Approve A Burnt Token

Category	Severity	Location	Status
Inconsistency	● Informational	common/BaseWithStorage/ERC721BaseToken.sol: 51	✓ Resolved

Description

The `approveFor()` and `approve()` functions are used to approve an operator to spend tokens on a user's behalf, but they are inconsistent with each other as it is possible to approve a burnt token using `approveFor()` but not `approve()`.

When `approve()` is used, the owner of the token is acquired by calling `_ownerOf()`, which will return the zero address for a burnt token, and a check is done to ensure that the owner is not the zero address.

```
35      function approve(address operator, uint256 id) external override {
36          uint256 ownerData = _owners[_storageId(id)];
37          address owner = _ownerOf(id);
38          address msgSender = _msgSender();
39          require(owner != address(0), "NONEXISTENT_TOKEN");
```

Hence it is not possible to approve a burnt token. However, for the function `approveFor()`, the owner is acquired from the `_owners` mapping and the check on this owner is that the least significant 20 bytes is the same as the `sender`.

```
  51      function approveFor(
  52          address sender,
  53          address operator,
  54          uint256 id
  55      ) external {
  56          uint256 ownerData = _owners[_storageId(id)];
  57          address msgSender = _msgSender();
  58          require(sender != address(0), "ZERO_ADDRESS_SENDER");
  59          require(
  60              msgSender == sender || _superOperators[msgSender] ||
_operatorsForAll[sender][msgSender],
  61              "UNAUTHORIZED_APPROVAL"
  62          );
  63          require(address(uint160(ownerData)) == sender, "OWNER_NOT_SENDER");
```

As a burnt token retains information about the previous owner, it is possible to pass all of these checks. A consequence of this is that after approving the burnt token, its associated value in the `_owners` mapping will be `BURNED_FLAG = 2**160`, losing the information of the previous owner.

Note: the detailed proof of concept can be found in *Appendix - Supplementary Tests: Test 3*.

Recommendation

We recommend changing `appoveFor()` to not allow approval of burnt tokens.

Alleviation

[Sandbox]: The issue is resolved in commit 349f7227bc81d42299f7c641638d87e6c397b24e by only allowing approval for tokens with non-zero owners.

ERB-02 | Burning Logic Inconsistency On L1 And L2

Category	Severity	Location	Status
Logical Issue	● Informational	common/BaseWithStorage/ERC721BaseToken.sol: 337	ⓘ Acknowledged

Description

Sandbox has implemented Land token protocols on both L1 (Ethereum) and L2 (Polygon). The burning logic of the two protocols is different.

On L2, the `_burn()` function sets the 160st bit of the token's owner record to be 1 (as an identification for burnt tokens) yet maintains the previous owner's address.

```
    //Burning Logic on L2
    function _burn(
        address from,
        address owner,
        uint256 id
    ) internal {
        require(from == owner, "NOT_OWNER");
        uint256 storageId = _storageId(id);
        _owners[storageId] = (_owners[storageId] & NOT_OPERATOR_FLAG) | BURNED_FLAG;
// record as non owner but keep track of last owner
        _numNFTPerAddress[from]--;
        emit Transfer(from, address(0), id);
    }
```

On L1, the `_burn()` function sets the token's owner record as 2**160 and removes the previous owner's address.

```
    //Burning Logic on L1
    function _burn(address from, address owner, uint256 id) internal {
        require(from == owner, "not owner");
        _owners[id] = 2**160; // cannot mint it again
        _numNFTPerAddress[from]--;
        emit Transfer(from, address(0), id);
    }
```

If tokens on L1 and L2 are designed to be equal, it is recommended to maintain the same logic to avoid unexpected consequences resulting from protocol incompatibility.

Recommendation

We recommend using the same logic on both L1 and L2.

Alleviation

[Sandbox]: The team acknowledged this finding and decided to not change the codebase at this time.

PLB-01 | Transferring Quads May Remove The Burn Status Of Tokens

Category	Severity	Location	Status
Logical Issue	● Major	polygon/child/land/PolygonLandBaseToken.sol: 281	✓ Resolved

Description

Both functions `batchTransferQuad()` and `transferQuad()` call the internal function `_transferQuad()` to transfer quads. If the quad being transferred is 1x1, then the token is checked to ensure that it is not burned. However, when the quad is of a larger size, no such check is made, meaning that tokens with the burnt flag can have the flag reset.

Each token is associated to a 256 bit `uint` in the mapping `_owners`. The least significant 160 bits are used for the token owner's address while the least significant 161st bit is used to decide if the token is burned or not. When using `batchTransferQuad()` or `transferQuad()` to transfer a quad of size larger than 1x1, the internal function `_regroup()` is called to perform checks and execute the transfer.

The function `_regroup()` first checks if each 1x1 in the quad is owned by the correct owner for the transfer via `_checkAndClear()`, but this check only considers if the owner is non-zero and if so, the least significant 160 bits.

```
function _checkAndClear(address from, uint256 id) internal returns (bool) {
    uint256 owner = _owners[id];
    if (owner != 0) {
        require(address(uint160(owner)) == from, "not owner");
        _owners[id] = 0;
        return true;
    }
    return false;
}
```

Note that `_checkAndClear()` sets the owner of the 1x1 to 0 if the checks are passed. As burned tokens may retain information about the previous owner, such as if a user transferred a 1x1 LAND to themselves and then burned the 1x1, they would be able to clear the burn flag by calling `transferQuad()` to transfer the 1x1 to themselves. However, this process decreases the user's balance due to the burn, so resetting the burn flag can only be done if the user has an excess balance amount.

Another issue of this is that it is possible for users to be unable to transfer quads. The reason is that the burning process decreases their balance. For example, if a user minted themselves a 6x6 quad, their current balance is 36. Suppose the user transfers a 1x1 LAND to themselves and then burns it, bringing their balance to 35. The user may forget this occurred and if they sell a 3x3 quad containing the burnt land, their new balance would be 26, even though they are the owner of 27 pieces of LAND. Hence they will be unable to sell all of their tokens due to an underflow revert.

Note: the detailed proof of concept can be found in *Appendix - Supplementary Tests: Test 1 & Test 2*.

Recommendation

We recommend also checking for the burn flag when transferring quads.

Alleviation

[Sandbox]: The issue is resolved in commit 335bc482fc3a29f280e980d672f47aa19cfa00d8 by now checking if the token has been burnt or not.

PLB-02 | Possibly Incorrect Return For Function `exists()`

Category	Severity	Location	Status
Volatile Code	● Minor	polygon/child/land/PolygonLandBaseToken.sol: 230	✓ Resolved

Description

The `exists()` function is used to see if an input quad of size `size` starting at coordinates `(x,y)` is contained in or contains an already minted quad. It returns a bool value:

- `true`: the input quad contains an already minted quad (meaning you cannot mint the quad)
- `false`: the input quad does not contain an already minted quad (meaning you may mint the quad)

However, this function does not ensure that `size` is a parameter supported by the project.

Although this function is only called in `_mintQuad()` where `size` is ensured to be an input supported by the project, `exists()` is a `public` function, meaning a user may use this on unsupported parameters for `size`, which may lead to incorrect return values.

For example, suppose a user wants enough LAND to cover a 13x13 square starting at (13,0). Assume (12,0) is owned as a 12x12 quad and no one owns a 24x24 quad at (0,0). Now suppose the user calls `exists(13,13,0)` to see if all LANDs and quads are free to mint. The 24x24 check passes since no one owns the 24x24. As `size` is larger than 12, the function checks if anyone owns a 12x12 starting at (13,0), which is not possible. Similarly, the 6x6, 3x3, and 1x1 checks can be passed so it returns `false` even though pieces of the 13x13 grid are owned.

Proof of Concept

The following test is derivated from the example above:

```
it('Exists function can be misleading', async function () {
  const {landOwners} = await setupTest();
  const bytes = '0x3333';

  await waitFor(
    landOwners[0].MockLandWithMint.mintQuad(
      landOwners[0].address,
      12,
      12,
      0,
      bytes
    ));
```

```
      const owner = await landOwners[0].MockLandWithMint.ownerOf(12);
      await expect(owner).to.be.equal(landOwners[0].address);

      const existResult = await landOwners[0].MockLandWithMint.exists(13,13,0);
      await expect(existResult).to.be.equal(false);
    });
  });
```

The result shows it returns `false`, which could be misleading to front end/users.

```
    Possibly Incorrect Return for Function exists()
Nothing to compile
      ✓ Exists function can be misleading (6426ms)
```

Recommendation

We recommend either placing restrictions on the variable `size` or making `exists()` an internal function.

Alleviation

[Sandbox]: The issue is resolved in commit 7581baa0536b23844e735e18e6bed9e3e845446c by now validating all parameters.

PLB-03 | Inconsistent NatSpec For Minting Function

Category	Severity	Location	Status
Inconsistency	● Informational	polygon/child/land/PolygonLandBaseToken.sol: 111	Resolved

Description

The NatSpec of the `_mintQuad()` function states that mints a quad of size 3, 6, 12, or 24 only.

```
111        * @notice Mint a new quad (aligned to a quad tree with size 3, 6, 12 or 24 only)
```

However, the function implementation allows mints of size 1.

```
117      function _mintQuad(
118          address to,
119          uint256 size,
120          uint256 x,
121          uint256 y,
122          bytes memory data
123      ) internal {
124          require(to != address(0), "to is zero address");
125          require(!exists(size, x, y), "Already minted");
126
127          uint256 quadId;
128          uint256 id = x + y * GRID_SIZE;
129
130          if (size == 1) {
131              quadId = id;
```

Proof of Concept

```
describe('Can mint 1x1 Land with mintQuad function', function () {
  it('Invoke mintQuad() to mint 1x1 land', async function() {
    const {landOwners} = await setupTest();
    const bytes = '0x3333';

    await waitFor(
      landOwners[0].MockLandWithMint.mintQuad(
        landOwners[1].address,
        1,
        0,
        0,
        bytes
      )
    );
    const owner = await landOwners[0].MockLandWithMint.ownerOf(0)
    await expect(owner).to.equal(landOwners[1].address);
    console.log('Successfully mint a 1x1 land at (0,0)')
  });
});
```

result:

```
Successfully mint a 1x1 land at (0,0)
      ✓ Invoke mintQuad() to mint 1x1 land (6578ms)
```

Recommendation

We recommend changing either the NatSpec or the code of `_mintQuad()` so that both are consistent with each other.

Alleviation

[Sandbox]: The issue is resolved in commit 0d27b7f3c9502950c26f95fecb63bbda308d79a6 by changing the NatSpec to be consistent with the function.

PLB-04 | Redundant Code

Category	Severity	Location	Status
Gas Optimization	● Informational	polygon/child/land/PolygonLandBaseToken.sol: 477	✓ Resolved

Description

The internal function `_ownerOfQuad()` is used to find the owner or parent owner of a quad and can only be called by itself or one of the regroup functions. As none of these will call `_ownerOfQuad()` with a value of 1 for the input variable `size`, the `if` code branch `size == 1` will never be reached.

Recommendation

We recommend removing the branch `size == 1`.

Alleviation

[Sandbox]: The issue is resolved in commit c698784244254ab912e5caef81c4ac04a53614c6 by removing the redundant code.

Appendix

Supplementary Tests

The following tests serve as detailed proof of concepts for findings **PLB-01**, **PLB-05** and **ERC-01**

Test 1: Burnt Tokens Can Be Transferred

Description

A user mints a 6x6 land token but burns a 1x1 land token in it. However, the attacker can still transfer lands that contain the burnt 1x1 land token.

Proof of Concept

```
    it('Quad (>=3x3) contains burnt 1x1 Land is possible to be transferred', async
function() {
      const {landOwners} = await setupTest();
      const bytes = '0x3333';

      console.log("Step1: Mint a 6x6 Quad (x=0,y=0)");
      await landOwners[0].MockLandWithMint.mintQuad(landOwners[0].address,6,0,0,bytes);

      console.log("Step2: Owner transfers each LAND of the Quad to himself");
      for(let i = 0; i<6; i++){
        for(let j = 0; j<6; j++){
          await landOwners[0].MockLandWithMint.transferQuad(
            landOwners[0].address,
            landOwners[0].address,
            1,
            i,
            j,
            bytes
          )
        }
      }

      const beforeBurn = await
landOwners[0].MockLandWithMint.balanceOf(landOwners[0].address);
      console.log('The balance before burnning is ', beforeBurn)

      console.log("Step 3: Owner burns (0,0) LAND of the Quad");
      await landOwners[0].MockLandWithMint.burn(
        0x0000000000000000000000000000000000000000000000000000000000000000 +
          (0 + 0 * 408)
      )
      const afterBurn = await
landOwners[0].MockLandWithMint.balanceOf(landOwners[0].address);
      console.log('The balance after burnning is ', afterBurn)
      const beforeTransfer = await
landOwners[0].MockLandWithMint.balanceOf(landOwners[0].address);
      console.log('The balance of Sender before transfer is ', beforeTransfer)

      console.log("Step 4: Owner attempts to transfer the 3x3 Quad containing the burnt
token to others");
      await expect(landOwners[0].MockLandWithMint.transferQuad(
        landOwners[0].address,landOwners[1].address,3,0,0,bytes
      ));

      const afterTransfer = await
landOwners[0].MockLandWithMint.balanceOf(landOwners[0].address);
      console.log('The balance of Sender after transfer is ', afterTransfer)
      const afterTransfer_receiver = await
landOwners[0].MockLandWithMint.balanceOf(landOwners[1].address);
      console.log('The balance of Receiver after transfer is ', afterTransfer_receiver)
    });
```

Result & Explanation

```
Step1: Mint a 6x6 Quad (x=0,y=0)
Step2: Owner transfers each LAND of the Quad to himself
The balance before burnning is  BigNumber { _hex: '0x24', _isBigNumber: true }
Step 3: Owner burns (0,0) LAND of the Quad
The balance after burnning is  BigNumber { _hex: '0x23', _isBigNumber: true }
The balance of Sender before transfer is  BigNumber { _hex: '0x23', _isBigNumber: true }
Step 4: Owner attempts to transfer the 3x3 Quad containing the burnt token to others
The balance of Sender after transfer is  BigNumber { _hex: '0x1a', _isBigNumber: true }
The balance of Receiver after transfer is  BigNumber { _hex: '0x09', _isBigNumber: true }
      ✓ Quad (>=3x3) contains burnt 1x1 Land is possible to be transferred (7452ms)
```

In step 3, a 1x1 land token (0,0) is burnt. However, the user can still transfer the burnt token by calling `transferQuad()` to transfer a 3x3 land token that contains the burnt token.

One consequence of this is that the balance is not updated as expected. When the burnt token is transferred along with the 3x3 quad, the sender's balance will decrease by 9 and the receiver's balance will increase by 9. However, since a token is actually burnt in the 3x3 token, the update of the balance could be inaccurate.

Test 2: Burnt Tokens Can Regain Ownership

Description

This test could be seen as a consequence of the previous test (Test 1). After transferring a 3x3 land token that contains a burnt token, the recipient can regain ownership of the burnt token.

Proof of Concept

```
    it('Burnt Land is possible to regain ownership by transferQuad', async function() {
      const {landOwners} = await setupTest();
      const bytes = '0x3333';

      console.log("Step 1: Mint a 6x6 Quad (x=0,y=0)");
      await landOwners[0].MockLandWithMint.mintQuad(landOwners[0].address,6,0,0,bytes);

      console.log("Step 2: Owner transfers each LAND of the Quad to himself");
      for(let i = 0; i<6; i++){
        for(let j = 0; j<6; j++){
          await landOwners[0].MockLandWithMint.transferQuad(
            landOwners[0].address,
            landOwners[0].address,
            1,
            i,
            j,
            bytes
          )
        }
      }

      const ownerBeforeBurn = await landOwners[0].MockLandWithMint.ownerOf(0);
      console.log("Step 3: Owner (%s) burns (0,0) LAND of the Quad", ownerBeforeBurn);
      await landOwners[0].MockLandWithMint.burn(
        0x0000000000000000000000000000000000000000000000000000000000000000 +
          (0 + 0 * 408)
      )

      //Ensure the token is burnt
      await expect(landOwners[0].MockLandWithMint.ownerOf(0)).to.be.reverted;

      console.log("Step 4: Owner attempts to transfer the 3x3 Quad containing the burnt
token to others");
      await landOwners[0].MockLandWithMint.transferQuad(
        landOwners[0].address,landOwners[1].address,3,0,0,bytes
      )

      const ownerAfterTransfer = await landOwners[0].MockLandWithMint.ownerOf(0);
      console.log('The receiver (%s) regain the ownership of the burnt land',
ownerAfterTransfer)
      await expect(ownerAfterTransfer).to.be.equal(landOwners[1].address);

      console.log('Now the receiver fully owned all the 3x3 land and the burnt 1x1 can be
transferred to others')
      await expect(landOwners[1].MockLandWithMint.transferQuad(
        landOwners[1].address,landOwners[2].address,1,0,0,bytes
      ));
    });
```

Result & Explanation

```
Step 1: Mint a 6x6 Quad (x=0,y=0)
Step 2: Owner transfers each LAND of the Quad to himself
Step 3: Owner (0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266) burns (0,0) LAND of the Quad
Step 4: Owner attempts to transfer the 3x3 Quad containing the burnt token to others
The receiver (0x14dC79964da2C08b23698B3D3cc7Ca32193d9955) regain the ownership of the
burnt land
Now the receiver fully owned all the 3x3 land and the burnt 1x1 can be transferred to
others
      ✓ Burnt Land is possible to regain ownership by transferQuad (7019ms)
```

According to the result, after Step 4, the recipient fully owned the 3x3 land token (including the burnt 1x1 land token) and is able to transfer the burnt token.

Test 3: Burnt tokens Can Be Approved Through `approveFor()`

Description

The `approveFor()` function is able to approve a burnt token. However, even if the token is approved, it cannot be transferred via `transferFrom()`.

Proof of Concept

```
    it('Burnt token can be approved', async function () {
      const {landOwners} = await setupTest();
      const bytes = '0x3333';

      console.log("Step 1: Mint a 6x6 Quad (x=0,y=0)");
      await landOwners[0].MockLandWithMint.mintQuad(landOwners[0].address,6,0,0,bytes);

      console.log("Step 2: Owner transfers each LAND of the Quad to himself");
      for(let i = 0; i<6; i++){
        for(let j = 0; j<6; j++){
          await landOwners[0].MockLandWithMint.transferQuad(
            landOwners[0].address,
            landOwners[0].address,
            1,
            i,
            j,
            bytes
          )
        }
      }

      const ownerBeforeBurn = await landOwners[0].MockLandWithMint.ownerOf(0);
      console.log("Step 3: Owner (%s) burns (0,0) LAND of the Quad", ownerBeforeBurn);
      await landOwners[0].MockLandWithMint.burn(
        0x0000000000000000000000000000000000000000000000000000000000000000 +
          (0 + 0 * 408)
      )

      console.log('Step 4: Calling approveFor() will succeed')
      await expect(landOwners[0].MockLandWithMint.approveFor(
        landOwners[0].address, landOwners[1].address, 0
        ));

      console.log("Step 5: Approved burnt token cannot be transferred by 1x1")
      await expect(landOwners[1].MockLandWithMint.transferFrom(
          landOwners[0].address, landOwners[1].address, 0
        )).to.be.revertedWith('NONEXISTENT_TOKEN');
    });
```

Result & Explanation

```
Step 1: Mint a 6x6 Quad (x=0,y=0)
Step 2: Owner transfers each LAND of the Quad to himself
Step 3: Owner (0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266) burns (0,0) LAND of the Quad
Step 4: Calling approveFor() will succeed
Step 5: Approved burnt token cannot be transferred by 1x1
      ✓ Burnt token can be approved (6935ms)
```

According to the result, the `approveFor()` invocation succeeded, but the burnt token cannot be transferred via `transferFrom()`.

Test 4: Burnt Tokens Cannot Be Minted Again

Description

Once a token is burnt, the same token (with the same coordinates) cannot be minted again.

Proof of Concept

```
    it('Burnt token should be minted again', async function() {
      const {landOwners} = await setupTest();
      const bytes = '0x3333';

      console.log("Step 1: A 1x1 Quad (x=0,y=0) is minted");
      await landOwners[0].MockLandWithMint.mintQuad(landOwners[0].address,1,1,0,bytes);
      console.log("Step 2: Owner burns (1,0) LAND of the Quad");
      await landOwners[0].MockLandWithMint.burn(
        0x0000000000000000000000000000000000000000000000000000000000000000 +
          (1 + 0 * 408)
      )

      console.log('Step 3: Attempt to mint the burnt token again')
      await expect(landOwners[0].MockLandWithMint.mintQuad(
        landOwners[0].address,1,1,0,bytes)).to.be.revertedWith('Already minted');
    });
```

Result & Explanation

```
Step 1: A 1x1 Quad (x=0,y=0) is minted
Step 2: Owner burns (1,0) LAND of the Quad
Step 3: Attempt to mint the burnt token again
      ✓ Burnt token cannot be minted again (6160ms)
```

According to the result, a 1x1 token is minted with coordinates (1,0). After burning this token, the token with the same coordinates can never be minted again.

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND

“AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK’S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER’S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED “AS IS” AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK’S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

CERTIK