Cyberscope

Audit Report

Xandao

December 2023

Repository https://github.com/xandao-xo/xandao-smart-contracts

Commit 207a0f4477a71f1d4ff600408eb580168b7b8d28

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/xandao-xo/xandao-smart-contracts
Commit	207a0f4477a71f1d4ff600408eb580168b7b8d28

Audit Updates

Initial Audit	30 Nov 2023

Source Files

Filename	SHA256
Trigonometry.sol	4591603d6a216772836b6b672ba28a8dd7de82c1c50bf75343d21e1ef605a874
PixelsV1.sol	94906ce3e373f66406e3b6b492e1c745a064bb516efca0079427a7c313b20128
PixelsTypesV1.sol	5ab5a21b5ada49f9c6e809387012a8e595a2ba3fdbf76361e53ae55179486690
PixelsMetadataUtils.sol	0ac9b4ca676040d46ef2f132b120da998348cb4bdd49938cd066b5b479981fa1

Overview

This document presents the overview of the smart contract audit conducted for the "PixelsV1" and "PixelsMetadataUtils" contracts. The purpose of this audit is to identify and address security vulnerabilities, provide recommendations for code improvements, and ensure the robustness of the codebase.

The audited contracts demonstrate a solid foundation with proper usage of ERC721 and OpenZeppelin libraries. Recommendations have been provided to enhance security and functionality.

PixelsV1 Contract

Manages the main functionality of the Pixels project, including minting and upgrading of NFTs.

Implements an ERC721 token for the Pixels project. Supports the minting and upgrading of NFTs based on a 36-character token ID. Handles ownership, burning, and token URI generation. Includes a price mechanism based on the length of the token ID.

PixelsMetadataUtils Contract

Provides utility functions for generating SVG and token URI metadata. Uses OpenZeppelin libraries for string manipulation and mathematical operations.

Functionality

Mint

The users can create new NFTs (pixels) by providing a valid tokenId, a description, and a creator name. The minting process requires payment in Ether, and the payment amount depends on the length of the tokenId.

Upgrades

The users can upgrade an existing token by burning it and minting a new one with a higher-level tokenId. The upgrade process also requires payment in Ether, and the payment

amount is based on the difference in minting prices between the original and upgraded tokens.

Pixel Metadata and SVG Generation

- The contract includes a library (`PixelsMetadataUtils`) for generating metadata and SVG content.
- Metadata includes details like creator's address, creator's name, grid size, background color, and composite color.
- SVG content is generated based on `tokenId` and a predefined color map.

Token URI and External URLs

- Functions (`tokenURIByTokenId` and `tokenURIByXN`) retrieve the token URI for a given token.
- External URL (`https://pixels.xandao.com/view?id=`) for viewing pixel art.

Ownership and Withdrawals

- The contract has an owner with special privileges.
- A percentage of Ether from minting and upgrading processes goes to a DAO address and an operations address.

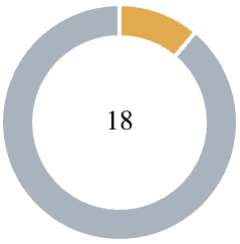
Token and User Information

- Users can query information about tokens, including the creator, availability status, and Ether balance of the contract.
- Mechanism to handle burn status of tokens, indicating whether a token has been upgraded.

External Libraries

The contract uses external libraries, such as OpenZeppelin's ERC-721 and utility libraries for string and math operations.

Findings Breakdown



Critical	0
Medium	2
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	2	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	SSDFM	Sequential Six Digit Free Mint	Unresolved
Medium	ITH	Inconsistent TokenId Handling	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	SLO	Sequencial Loop Optimization	Unresolved
Minor / Informative	AISRC	Ascending If Statements Redundant Condition	Unresolved
Minor / Informative	CCO	Color Calculation Optimizations	Unresolved
Minor / Informative	ISC	Inefficient String Comparison	Unresolved
Minor / Informative	RC	Redundant Call	Unresolved
Minor / Informative	XDFRE	XN Duplication From Reentrance Exploit	Unresolved
Minor / Informative	TPSMO	Token Price Storage Mapping Optimization	Unresolved
Minor / Informative	ECXH	Edge Case XN Handling	Unresolved
Minor / Informative	CNIV	Creator Name Input Validation	Unresolved
Minor / Informative	RSML	Redundant SafeMath Library	Unresolved

●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved

SSDFM - Sequential Six Digit Free Mint

Criticality	Medium
Location	PixelsV1.sol#L271
Status	Unresolved

Description

During the smart contract audit assessment, it was identified that users can exploit the mintPixels function to mint NFTs with free tokenIds that contain only the digit 6 (e.g., 6, 66, 666, etc.). This vulnerability arises due to the behavior of the _checkTokenId method, which removes trailing 6's from the tokenId, and if the tokenId consists solely of the digit 6, it will result in a tokenId of 0. This allows users to mint NFTs with tokenId 0 without paying the required minting fee.

```
function mintPixels(
    uint256 tokenId,
    string memory description,
    string memory creatorName
) public payable returns (uint256) {
    // ...

    uint256 numDigits = _checkTokenId(tokenId);

    // ...

    _safeMint(msg.sender, tokenId);
    // ...

    // send eth to dao.xandao.eth and pixels.xandao.eth
    // ...
    emit PixelsMinted(msg.sender, tokenId, xn_str);
    return tokenId;
}

function _checkTokenId(uint256 tokenId) internal pure returns(uint256) {
    // ...

    // remove trailing 6's
    while (tokenId % 10 == 6) {
        tokenId /= 10;
    }

    // ...
    return numDigits;
}
```

Recommendation

To address this vulnerability, it is recommended to update the _checkTokenId method to handle the case where the modified tokenId becomes 0 due to the removal of trailing 6's. A simple adjustment can be made by introducing an additional check to ensure that the tokenId after removing trailing 6's is greater than zero. If the tokenId becomes zero, it implies an attempt to mint a free tokenId with only the digit 6, and the contract should revert. Otherwise, a special price should be intrduced for the zero index of `_tokenIdLenToMintPrice`.

ITH - Inconsistent TokenId Handling

Criticality	Medium
Location	PixelsMetadataUtils.sol#L94,99,152,154
Status	Unresolved

Description

The `mintPixels` and `upgradePixels` functions handle the `tokenId` differently compared to the `generateTokenURI` function, leading to inconsistencies in token processing and attribute assignment. Specifically, in `mintPixels`, the `tokenId` undergoes a transformation where trailing 6's are removed, influencing the mint price calculation. For example, a tokenId of 1666 effectively becomes 1, resulting in a mint price of 6 ether based on the `_tokenIdLenToMintPrice` mapping.

However, the `generateTokenURI` function does not implement this trailing 6's removal. As a result, when it calls `getLevelPrice` and `getLevel`, the original `tokenId` (e.g., 1666) is used, leading to different price and level calculations than those expected based on the minting price. This inconsistency can lead to significant confusion and potential financial discrepancies, as the perceived value and attributes of the NFT (as shown in `generateTokenURI`) do not align with the minting price.

```
function getLevel(uint256 tokenId) public pure returns(uint256) {
        uint256 tokenIdLength = bytes(tokenId.toString()).length;
        return (37 - tokenIdLength);
}

function getLevelPrice(uint256 tokenId) public view returns(string memory) {
        uint256 level = getLevel(tokenId);
        return _levelPrices[level];
}

uint256 level = getLevel(tokenId);
string memory price = getLevelPrice(tokenId);
```

Recommendation

Consider establishing a uniform approach by applying the same transformation logic to `tokenId` , by incorporating the trailing 6's removal logic in all relevant functions. Implementing this change will ensure that the transformed `tokenId` is consistently used throughout the contract, thereby aligning the attributes and pricing information associated with each token. This adjustment is crucial for maintaining the integrity and reliability of the smart contract's functionality.

MU - Modifiers Usage

Criticality	Minor / Informative
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
uint256 length = bytes(description).length;
require(
    length < 36 && length > 0,
    "Pixels description must be between 1 and 36 characters"
);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

CR - Code Repetition

Criticality	Minor / Informative
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
uint256 daoAmount = msg.value * daoShare / 100;
payable(daoAddress).transfer(daoAmount);
payable(opsAddress).transfer(msg.value - daoAmount);
//
if (keccak256(abi.encodePacked(pixelsObj.xnVersion)) ==
keccak256(abi.encodePacked('a'))) {
    xn_str = pixelsObj.xn.toString();
} else {
    xn_str = string.concat(pixelsObj.xn.toString(), "-", pixelsObj.xnVersion);
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

SLO - Sequencial Loop Optimization

Criticality	Minor / Informative
Location	PixelsV1.sol#L127
Status	Unresolved

Description

The contract contains a redundant usage of two loops in the same function that can be optimized for improved contract performance. The current implementation consists of a while loop followed by a for loop, both serving the purpose of extracting individual digits from the tokenId and validating their range.

```
uint8 numDigits = 0;
uint256 tempTokenId = tokenId;
while (tempTokenId != 0) {
    numDigits++;
    tempTokenId /= 10;
}

for (uint8 i = 0; i < numDigits; i++) {
    uint256 digit = (tokenId / (10 ** i)) % 10;

    require(
        digit >= 1 && digit <= 6,
        "Invalid digit value pixels pattern"
    );
}
```

Recommendation

The loops can be efficiently merged into a single loop to enhance code readability and optimize gas consumption. In the optimized solution, the validation of each digit's range is incorporated within the same loop that counts the number of digits (numDigits). By doing so, the redundant second loop is eliminated, resulting in a more concise and performant implementation.

This adjustment maintains the intended logic of the code while improving efficiency by eliminating unnecessary iterations. Additionally, it contributes to a cleaner and more maintainable codebase.

AISRC - Ascending If Statements Redundant Condition

Criticality	Minor / Informative
Status	Unresolved

Description

The contract contains a redundant and unnecessary condition check in the if-else statements within the getViewBox, getGridSize, and getBackgroundColor functions. The conditions for the length of tokenId are ascending, starting from 1 and increasing sequentially. Therefore, it is only required to check if the length is less than the upper bound of each range, as the conditions are mutually exclusive.

```
if (tokenIdLength == 1) {
    // ...
} else if (tokenIdLength >= 2 && tokenIdLength <= 4) {
    // ...
} else if (tokenIdLength >= 5 && tokenIdLength <= 9) {
    // ...
} else if (tokenIdLength >= 10 && tokenIdLength <= 16) {
    // ...
} else if (tokenIdLength >= 17 && tokenIdLength <= 25) {
    // ...
} else {
    // ...
}
```

Recommendation

Simplify the if-else conditions by removing the first condition and retaining only the check for the upper bounds of each range. This enhances code readability and maintains the logical structure of the conditions. The modified conditions would look as follows. This adjustment does not affect the logic of the code but results in cleaner and more concise conditionals.

CCO - Color Calculation Optimizations

Criticality	Minor / Informative
Location	PixelsMetadataUtils.sol#L231
Status	Unresolved

Description

The `getColorHSL` function produces the same result for "black" and "transparent" colors. Both these conditions result in the same HSL values (0, 0, 0). However, this outcome is also the default return value specified in the final else statement of the function. Therefore, the explicit checks for "black" and "transparent" are redundant since these cases would be adequately handled by the default else condition.

The function's structure involves repeated computations of the hash of the color input within each if-else branch. This repetitive computation is inefficient, especially considering that the input color remains constant throughout the function's execution. A more gas-efficient approach would be to compute the hash of the color once at the beginning of the function and use it in conditional checks.

```
function getColorHSL(string memory color) public pure returns (uint256 h,
uint256 s, uint256 l) {
        if (keccak256(abi.encodePacked(color)) == keccak256("cyan")) {
            return (uint256(180), uint256(100), uint256(50));
            ...
        } else {
            return (uint256(0), uint256(0), uint256(0));
        }
}
```

Recommendation

To enhance the function's efficiency, three modifications are recommended. Firstly, remove the explicit checks for "black" and "transparent" colors. The default else condition is sufficient for returning the HSL values (0, 0, 0) for these colors, as well as any other undefined colors.

Secondly, condider using precomputed hashes as constant. Define constant values for the color hashes. This change will avoid the need for hash computations within the function, saving gas.

Lastly, to optimize gas consumption, calculate the hash of the input `color` once at the start of the function. Use this single calculated hash in the conditional checks to avoid repeated calculations. Implementing these changes will lead to a more efficient execution of the `getColorHSL` function, aligning with best practices for smart contract development and optimizing for gas cost efficiency.

ISC - Inefficient String Comparison

Criticality	Minor / Informative
Location	PixelsMetadataUtils.sol#L231
Status	Unresolved

Description

The `getColorHSL` function is designed to determine Hue, Saturation, and Lightness (HSL) values based on given color names. This function matches color names, provided as strings, against a set of predefined colors using keccak256 for string hashing. While this method is effective in achieving the intended functionality, it entails a higher computational resource usage than necessary. Each string comparison is executed through a hashing operation, which is generally more resource-intensive compared to integer comparisons. In EVM smart contracts, where computational resources directly translate to transaction costs (gas fees), such resource-intensive operations, even if minor, can lead to increased costs over time, particularly if the `getColorHSL` function is frequently invoked.

```
if (keccak256(abi.encodePacked(color)) == keccak256("cyan")) {
    ...
} else if (keccak256(abi.encodePacked(color)) == keccak256("magenta")) {
    ...
} else if (keccak256(abi.encodePacked(color)) == keccak256("yellow")) {
    ...
} else if (keccak256(abi.encodePacked(color)) == keccak256("black")) {
```

Recommendation

Given that the color names are fixed and known, adopting an enumeration (enum) type for representing these colors is advised. Enums in Solidity are used to define a type with a restricted and predefined set of constants and are internally treated as integers. This internal representation of enums as integers facilitates more efficient comparisons, significantly reducing the computational resource usage of the `getColorHSL` function. Implementing this change involves defining an enum for the colors, updating the `colorMap` to use this enum, and adjusting the `getColorHSL` function to work with enum values. This modification will optimize the smart contract's performance by minimizing the

computational steps involved, thus potentially lowering the associated transaction costs in line with smart contract optimization practices.

RC - Redundant Call

Criticality	Minor / Informative
Location	PixelsMetadataUtils.sol#L151
Status	Unresolved

Description

In the `generateTokenURI` function of the contract, the `averageHSL` function is called twice, with the first call appearing to be redundant. `averageHSL` is executed without utilizing its return value, which suggests that it might be an unnecessary computation within the function's scope. Such redundant operations, particularly in functions that may be called frequently, can result in increased gas costs and inefficient contract execution.

```
averageHSL(digitCounts);
```

Recommendation

Review the necessity of the first call of the `averageHSL` function within `generateTokenURI`. If it's confirmed to be redundant and does not contribute to the contract's intended functionality, it should be removed to optimize the function's efficiency. Ensuring that each line of code serves a purpose, especially in public and external functions, is crucial for optimizing gas usage and overall contract performance.

XDFRE - XN Duplication From Reentrance Exploit

Criticality	Minor / Informative
Location	PixelsV1.sol#L241,281
Status	Unresolved

Description

The `mintPixels` and `upgradePixels` functions are susceptible to a potential reentrance exploit due to the sequence of operations where external calls are made before updating the contract's state. The functions execute `_safeMint`, which could interact with an external contract, followed by state changes and Ether transfers. If `_safeMint` interacts with a malicious contract, this contract could invoke a fallback function containing untrusted code that re-enters `mintPixels`. During the reentrance phase, an NFT with the same `xn` number will be produced. As a result, the entire business logic of the contract might be violated since the uniqueness of the `xn` and `xn_str` variables will be broken.

```
//mint
uint256 xn = _xn.current();
string memory xn_str = xn.toString();
_safeMint(msg.sender, tokenId);
pixelsCount++;
pixels[tokenId] = pixelsInfo(description, msg.sender, creatorName, xn, "a",
false);
_xnToTokenId[xn_str] = tokenId;
_xn.increment();
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

TPSMO - Token Price Storage Mapping Optimization

Criticality	Minor / Informative
Location	PixelsV1.sol#L66
Status	Unresolved

Description

The `_tokenIdLenToMintPrice` mapping is currently using `uint256 -> uint256` for its values. However, a close analysis reveals an opportunity for optimization by using a smaller integer type. The highest value in this mapping is 6.0 ether, which is equivalent to 6 * 10^18 wei in Solidity. Since Solidity handles ether and tokens at the base unit of wei, 1 ether is equal to 10^18 wei. Additionally, the maximum index is 36 so a uint8 could be used.

A key aspect of Solidity is the efficiency in storage and gas usage. Smaller data types are generally more cost-effective in terms of gas consumption. Therefore, determining the smallest integer type that can handle the highest value in the mapping without overflow is crucial for optimization.

```
mapping(uint256 => uint256) private _tokenIdLenToMintPrice;
```

Recommendation

To find the smallest necessary integer type, we calculate the smallest n such that 2^n is greater than or equal to 6 * 10^18. The result of this calculation is 63, indicating that a 64-bit unsigned integer (uint64) would be sufficient to store the highest value without risk of overflow.

Based on this calculation, switching the value type in the `_tokenIdLenToMintPrice` mapping from `uint256 -> uint256` to `uint8 -> uint64` is recommended. This change will reduce the storage space required for each entry in the mapping, potentially leading to reduced gas costs during contract execution. The implementation of this optimization aligns with best practices, focusing on efficiency and cost-effectiveness.

ECXH - Edge Case XN Handling

Criticality	Minor / Informative
Status	Unresolved

Description

The `upgradePixels` function displays a potential flaw in handling the edge case for `xnVersion` updates. Specifically, the function increments an index to determine the next version of `xnVersion` for a token upgrade. However, this logic does not account for the scenario where the current `xnVersion` is the last element in the `_xnVersions` array. In such a case, incrementing the index would result in an out-of-bounds reference, potentially leading to unexpected behavior or a contract revert due to invalid array access.

Moreover, the comment " 'a' " within the code suggests a misunderstanding or an incorrect annotation, as it implies that an index of 0 should correspond to 'a'. In reality, an index of 0 would indicate that the current `xnVersion` is not found in the array, given that the initial index is set to -1 and only updated upon finding a match.

This issue highlights a crucial gap in the function's logic, particularly in handling the transition of xnVersion when upgrading tokens that have reached the end of the defined version sequence.

```
for (int256 i = 0; i < 36; i++) {
    if (keccak256(abi.encodePacked(_xnVersions[uint256(i)])) == keccak256(abi.
encodePacked(originPixelsObj.xnVersion))) {
        idx = i;
        break;
    }
}

uint256 xn = originPixelsObj.xn; // keep origin token XN
uint256 newIdx = uint256(idx + 1);

string memory xnVersion;  // change XN version
string memory xn_str;

if (newIdx == 0) { // 'a'
    xnVersion = '';
    xn_str = xn.toString();
} else {
    xnVersion = _xnVersions[newIdx];
    xn_str = string.concat(xn.toString(), "-", xnVersion);
}
```

Recommendation

To address the edge case in the `upgradePixels` function, it is recommended to revise the logic that updates the `xnVersion` for token upgrades. The current implementation should be enhanced to handle the scenario where the `xnVersion` is at the last element of the `_xnVersions` array. Specifically, the function should include a mechanism to reset the `xnVersion` back to the first element in the array when the end is reached. This adjustment would ensure that the function does not attempt to access an out-of-bounds index in the array, thereby preventing unexpected behavior or potential contract reverts. Implementing this change will align the function's behavior with the intended cyclic progression through `xnVersion` values, ensuring reliability and consistency in the token upgrade process.

CNIV - Creator Name Input Validation

Criticality	Minor / Informative
Location	PixelsV1.sol#L179,259
Status	Unresolved

Description

`MintPixels` and `upgradePixels` functions currently exhibit a notable oversight in input validation, specifically concerning the `creatorName` parameter. Unlike the description parameter, where a careful check is implemented to ensure a length constraint, the `creatorName` parameter lacks such validation. This discrepancy in input handling can lead to a variety of issues.

Firstly, the absence of a length constraint on `creatorName` allows for the input of excessively long names. This situation poses a concern for storage efficiency on the blockchain. Longer strings demand more storage space, which translates to increased gas requirements. In blockchain systems where storage space is at a premium and directly correlates with transaction costs, such unchecked inputs can lead to inflated gas consumption.

Secondly, there is potential for misuse. In the current state, the functions could be exploited to insert large data payloads into the `creatorName` field. Such exploitation could be part of a larger malicious strategy, potentially aiming to overload the system, execute spam attacks, or even probe for vulnerabilities in the contract or the platforms interacting with it.

```
function upgradePixels(uint256 tokenId, string memory description, string
memory creatorName, uint256 originTokenId)

function mintPixels(uint256 tokenId, string memory description, string memory
creatorName)
```

Recommendation

It is recommended to implement a character length constraint for the `creatorName` parameter in both the `mintPixels` and `upgradePixels` functions. A sensible limit, similar to the one applied to description, should be enforced. This change will enhance the

contract's robustness, reduce potential storage inefficiencies, and improve the overall user experience. Additionally, clear error messages should be provided when the input does not meet the length requirement, guiding users to comply with the set constraints.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	PixelsMetadataUtils.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	PixelsV1.sol#L48,51,53PixelsMetadataUtils.sol#L20
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 daoShare = 90;
address private daoAddress = 0x074926F6527353437C59d43A076098F71385057a;
address private opsAddress = 0x2c04DbEaA7B545aBF3a30600A7522d7f521b8A5b;
string public basePart2 = "</g></svg>"
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Trigonometry.sol#L54,55,56,69,139
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint8 constant entry_bytes = 4
uint256 constant entry_mask = ((1 << (8 * entry_bytes)) - 1)

bytes constant sin_table =
        hex"0000000000...ffffff"
uint256 _angle
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	Trigonometry.sol#L100
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
                // mload will grab one word worth of bytes (32), as that is
the minimum size in EVM
                x1_2 := mload(add(table, offset1_2))
            }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Trigonometry.sol#L2PixelsV1.sol#L29PixelsTypesV1.sol#L2PixelsMetadataUtils.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.12;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

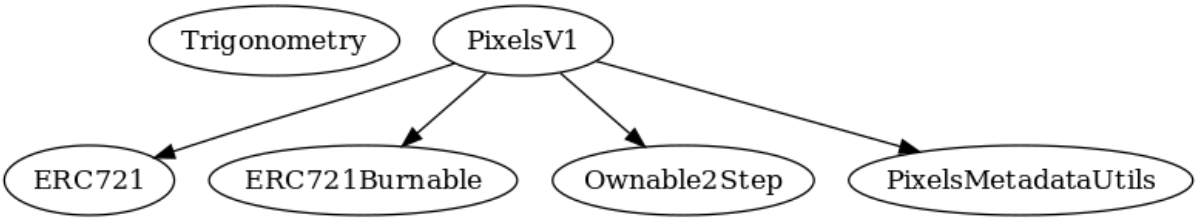
Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
Trigonometry	Library			
	sin	Internal		
	cos	Internal		
	atan2	Internal		
	abs	Internal		
	max	Internal		
	min	Internal		
PixelsV1	Implementation	ERC721, ERC721Burnable, Ownable2Step, PixelsMetadataUtils		
		Public	✓	ERC721
	_checkTokenId	Internal		
	getXN	Public		-
	getAllTokenIds	Public		-
	getMintPrice	Public		-

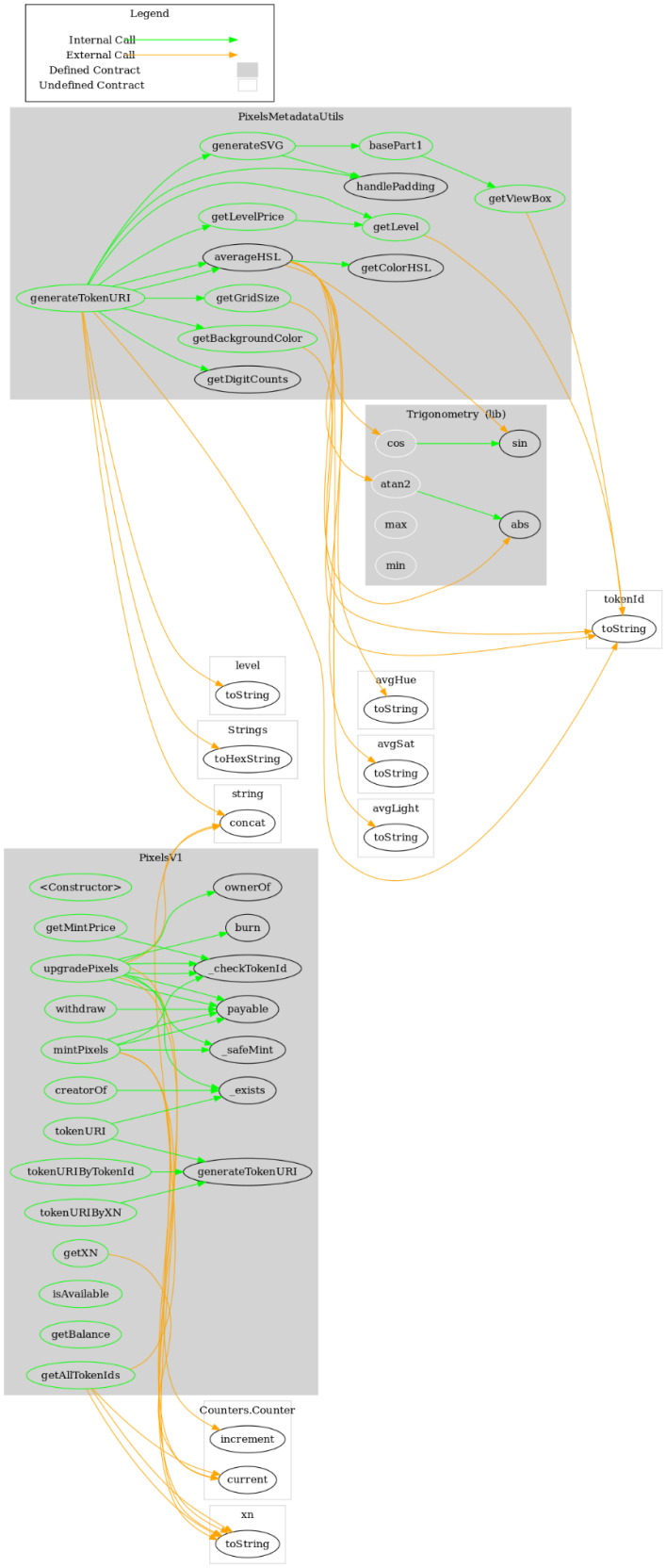
	upgradePixels	Public	Payable	-
	mintPixels	Public	Payable	-
	tokenURI	Public		-
	tokenURIByTokenId	Public		-
	tokenURIByXN	Public		-
	creatorOf	Public		-
	isAvailable	Public		-
	getBalance	Public		-
	withdraw	Public	✓	onlyOwner
PixelsMetadata Utils	Implementation			
	getViewBox	Public		-
	getGridSize	Public		-
	getBackgroundColor	Public		-
	getLevel	Public		-
	getLevelPrice	Public		-
	basePart1	Public		-
	generateSVG	Public		-
	generateTokenURI	Public		-
	handlePadding	Public		-
	getDigitCounts	Public		-

	getColorHSL	Public		-
	averageHSL	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
PixelsMetadataUtils
generateSVG
basePart1
handlePadding
getViewBox
getLevelPrice
getLevel
averageHSL
getColorHSL
generateTokenURI
getGridSize
getBackgroundColor
getDigitCounts
Trigonometry (lib)
cos
sin
atan2
max
abs
min
tokenId
toString
level
toString
avgHue
toString
Strings
toHexString
avgSat
toString
string
concat
avgLight
toString
PixelsV1
<Constructor>
ownerOf
getMintPrice
burn
upgradePixels
checkTokenId
withdraw
payable
mintPixels
_safeMint
creatorOf
_exists
tokenURI
tokenURIByTokenId
generateTokenURI
tokenURIByXN
getXN
isAvailable
getBalance
getAllTokenIds
Counters.Counter
increment
current
xn
toString

Summary

The Xandao contract smart contract serves as the backbone for a collaborative, on-chain, minimalist pixel art NFT project known as Xandao pixels. Enabling users to mint and upgrade pixel art NFTs. Overall, the contract fosters a community-driven exploration of the intersection between art and technology through the lens of pixel art. This audit investigates security issues, business logic concerns, and potential improvements

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. 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 Cyberscope's prior written consent. 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 regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document 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 represents an extensive assessment 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 Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope'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 Cyberscope are subject to dependencies and are 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.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io