Audit Report

Xandao

December 2023

Network	Sepolia
Commit	0x1E0181ceD1f6E4DE42769Ba696C6df9757cf9f28
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://sepolia.etherscan.io/address/0x1e0181ced1f6e4de42769ba696c6df9757cf9f28

Audit Updates

Initial Audit	30 Nov 2023
Corrected Phase 2	16 Dec 2023

Source Files

Filename	SHA256
Trigonometry.sol	2931db4219fa0676e5a11716e3e6ee6043e585d97e321feb611b28c25b50684b
Strings.sol	d81a4a55da7145b070af1100e65d0437bee08606024266d5c39f116c1fd6ca47
SignedMath.sol	420a5a5d8d94611a04b39d6cf5f02492552ed4257ea82aba3c765b1ad52f77f6
PixelsV1.sol	61f42fb9289108b5f82cea75810601a578cc80ac4949841d872ce015ad8fde82
PixelsTypesV1.sol	ea8bf376b3ad41ac523f9f6c6947ec355479098a1fe504432ac3687134f04340
PixelsMetadataUtils.sol	3a3cccfbb0a94d26a1ef9a677d2ddc65ea47e45be8800e2735d475c5a001071f
Ownable2Step.sol	3e3bdb084bc14ade54e8259e710287956a7dbf2b2b4ad1e4cd8899d2293c7241
Ownable.sol	33422e7771fefe5fbfe8934837515097119d82a50eda0e49b38e4d6a64a1c25d

Math.sol	85a2caf3bd06579fb55236398c1321e15fd524a8fe140dff748c0f73d7a52345
IERC721Receiver.sol	77f0f7340c2da6bb9edbc90ab6e7d3eb8e2ae18194791b827a3e8c0b11a09b43
IERC721Metadata.sol	fc0ee82753767d429965338d8e79830f2350908ed6c51b538f7b0f8b15b4224c
IERC721.sol	7a265b34318bc9845f06fb9a80f1d7ac32155db0ec0357ed8bf08f09743011ed
IERC165.sol	701e025d13ec6be09ae892eb029cd83b3064325801d73654847a5fb11c58b1e5
ERC721Burnable.sol	c0f4a5e7df8396227b907fe60c6ed8ed472557d833e99993d4b1ad7cb8a4fdac
ERC721.sol	49704a47f37fc5ce0b85bc3a26d161fd0a6f6c9768c35da86f67bf331672ad54
ERC165.sol	8806a632d7b656cadb8133ff8f2acae4405b3a64d8709d93b0fa6a216a8a6154
Counters.sol	2fdcb1343e5621385b62e57b5c7775607c272122b6f2dc77da8f84828aa40cd0
Context.sol	b2cfee351bcafd0f8f27c72d76c054df9b571b62cfac4781ed12c86354e2a56c
Base64.sol	9fbd7a4462f54bbb6b0bd03231738e5f081a092e9a8fd789fb4d1aeca3758aec
Address.sol	8b85a2463eda119c2f42c34fa3d942b61aee65df381f48ed436fe8edb3a7d602

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.

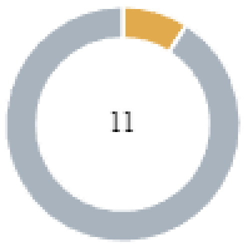
Token and User Information

- 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	1
Minor / Informative	10

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	10	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	ITH	Inconsistent TokenId Handling	Unresolved
	MU	Modifiers Usage	Unresolved
	CR	Code Repetition	Unresolved
	SLO	Sequencial Loop Optimization	Unresolved
	AISRC	Ascending If Statements Redundant Condition	Unresolved
	CCO	Color Calculation Optimizations	Unresolved
	ISC	Inefficient String Comparison	Unresolved
	XDFRE	XN Duplication From Reentrance Exploit	Unresolved
	ECXH	Edge Case XN Handling	Unresolved
	L17	Usage of Solidity Assembly	Unresolved
	L19	Stable Compiler Version	Unresolved

ITH - Inconsistent TokenId Handling

Criticality	Medium
Location	PixelsMetadataUtils.sol#L91,96,148,150
Status	Unresolved

Description

The `mintPixels` and `upgradePixels` functions handle the `rawTokenId` differently compared to how the `generateTokenURI` function handles the tokenId, leading to inconsistencies in token processing and attribute assignment. Essentially, it is the same argument in all three functions with different declaration. Specifically, in `mintPixels`, the `rawTokenId` 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 `rawTokenId` and `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 descLen = bytes(description).length;
require(
    descLen < 36 && descLen > 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 * DAO_SHARE / 100;
payable(DAO_ADDRESS).transfer(daoAmount);
payable(OPS_ADDRESS).transfer(msg.value - daoAmount);
//
if (keccak256(abi.encodePacked(pixelsObj.xnVersion)) ==
keccak256(abi.encodePacked('a'))) {
    xnStr = pixelsObj.xn.toString();
} else {
    xnStr = 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 - Sequential 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.

```
//get number of digits
uint8 tokenLen = 0;
uint256 tempTokenId = tokenId;
while (tempTokenId != 0) {
    tokenLen++;
    tempTokenId /= 10;
}

// verify digits are between 1-6
for (uint8 i = 0; i < tokenLen; i++) {
    // Get the i-th digit of pixelPattern
    uint256 digit = (tokenId / (10 ** i)) % 10;

    // Check if the digit is between 1-6
    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 lenth of the token (tokenLen). 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 (tokenLen == 1) {
    // ...
} else if (tokenLen >= 2 && tokenLen <= 4) {
    // ...
} else if (tokenLen >= 5 && tokenLen <= 9) {
    // ...
} else if (tokenLen >= 10 && tokenLen <= 16) {
    // ...
} else if (tokenLen >= 17 && tokenLen <= 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#L227
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.

XDFRE - XN Duplication From Reentrance Exploit

Criticality	Minor / Informative
Location	PixelsV1.sol#L250,293
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 `xnStr` variables will be broken.

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

ECXH - Edge Case XN Handling

Criticality	Minor / Informative
Location	PixelsV1.sol#L229
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(XN_VERSIONS[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 xnStr;

if (newIdx == 0) { // 'a'
    xnVersion = '';
    xnStr = xn.toString();
} else {
    xnVersion = XN_VERSIONS[newIdx];
    xnStr = 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 `XN_VERSIONS` 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.

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.

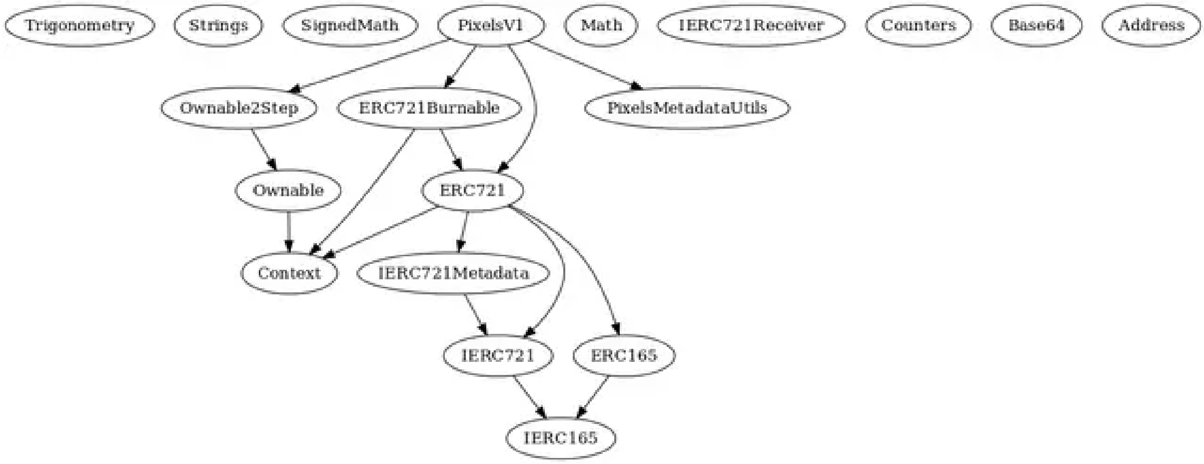
Functions Analysis

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		-

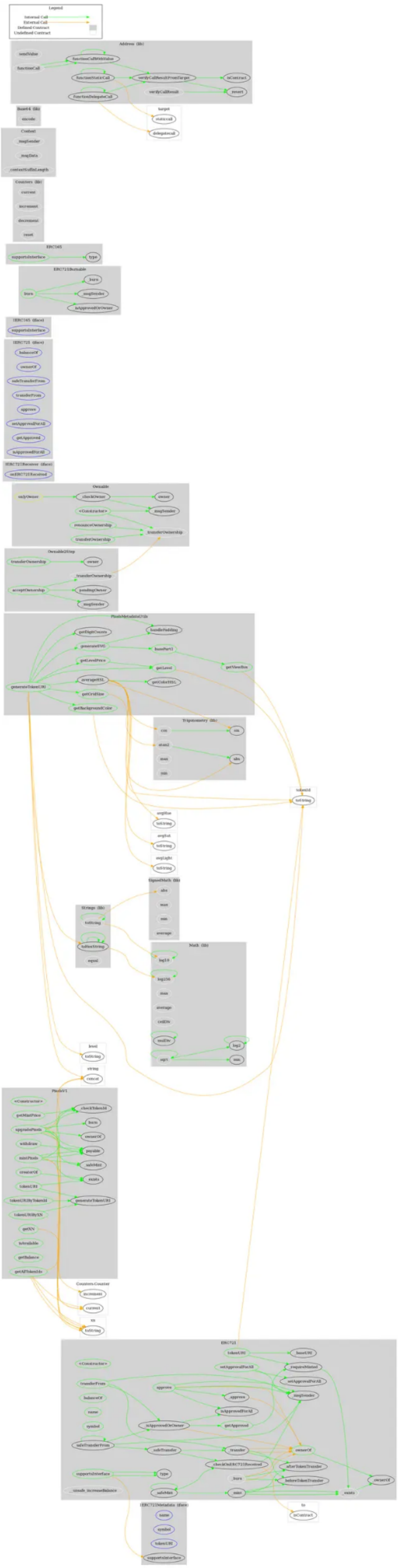
	upgradePixels	Public	Payable	-
	mintPixels	Public	Payable	-
	tokenURI	Public		-
	tokenURIByTokenId	Public		-
	tokenURIByXN	Public		-
	creatorOf	Public		-
	isAvailable	Public		-
	getBalance	Public		-
	withdraw	Public	✓	onlyOwner
PixelsMetadata Utils	Implementation			
	getViewBox	Public		-
	getGridSize	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



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