DustLabs

Point Staking

SMART CONTRACT AUDIT

22.0.2023

Made in Germany by Chainsulting.de

Table of contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Dust Labs. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (20.03.2022)	Layout
0.4 (22.03.2022)	Automated Security Testing Manual Security Testing
0.5 (20.03.2023)	Verify Claims and Test Deployment
0.6 (21.03.2023)	Testing SWC Checks
0.9 (21.03.2023)	Summary and Recommendation
1.0 (22.03.2023)	Final document

2. Project Overview

DUST protocol is a decentralized protocol and SPL (Solana Program Library) token created on the Solana blockchain with a starting supply of zero, and a maximum supply of 33,300,000. DUST has an emission schedule with multiple halvenings and mining rewards that are earned via staking NFTs. Countless projects have independently adopted DUST within their own ecosystems making it the most used SPL token on the Solana blockchain.

DeGods and Y00ts are two popular nonfungible token (NFT) digital art collections on the Solana blockchain. Both of these projects have attracted attention from the NFT community, offering unique digital artwork and novel features for collectors.

Website:
https://degods.com
https://www.y00ts.com

Twitter:
https://twitter.com/DeGodsNFT
https://twitter.com/y00tsNFT

Discord:
https://discord.gg/dedao
https://discord.gg/y00ts

Instagram:
https://www.instagram.com/thedegods
https://instagram.com/they00ts

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i. Review of the specifications, sources, and instructions provided to Chainsulting to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Chainsulting describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

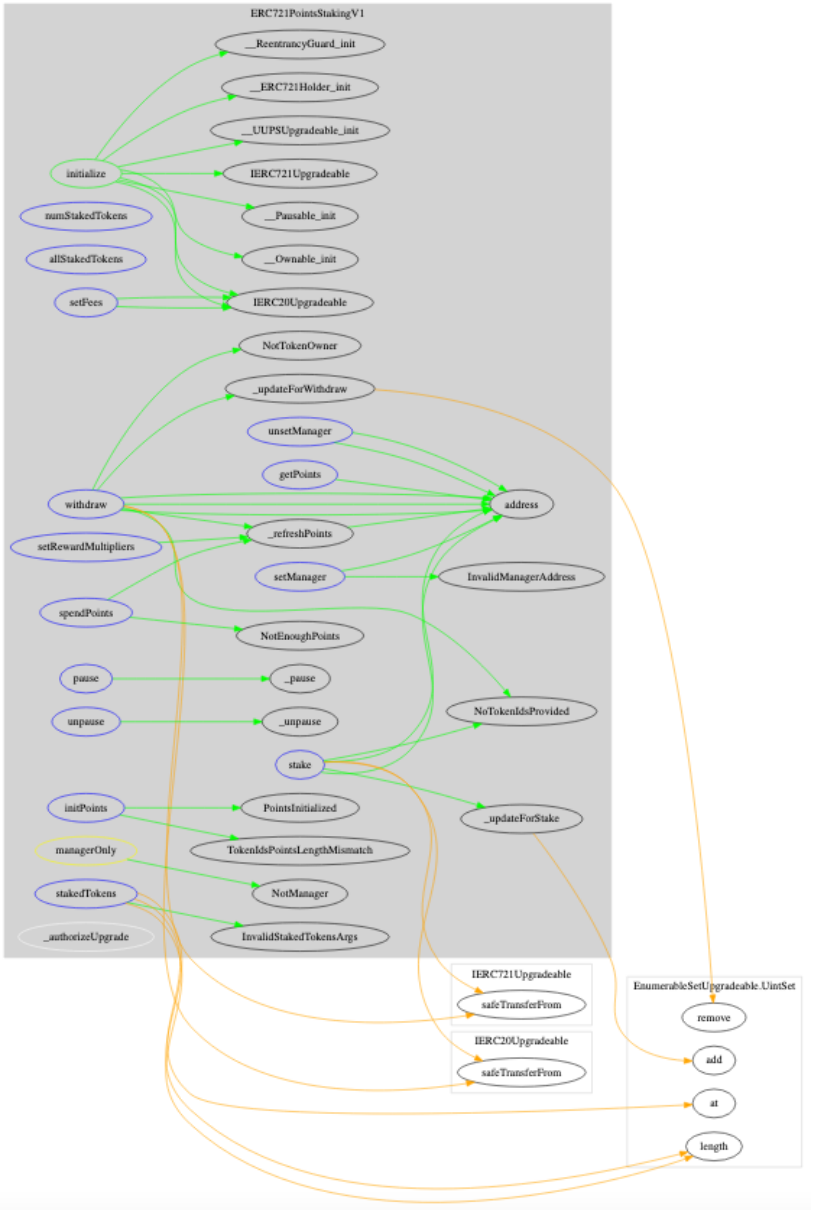
The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review

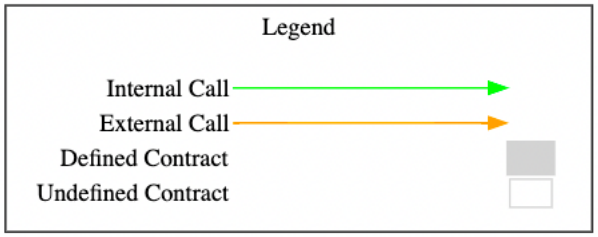
File	Fingerprint (MD5)
./ERC721PointsStakingV1.sol	941034bcf23327b1c8f091a2a248bb49

5.2 Used Code from other Frameworks/Smart Contracts (direct imports)

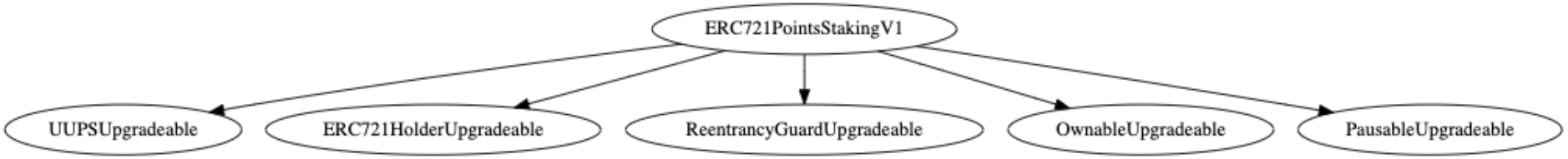
Dependency / Import Path	Source
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/access/OwnableUpgradeable.sol
@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/proxy/utils/UUPSUpgradeable.sol
@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/security/PausableUpgradeable.sol
@openzeppelin/contracts-upgradeable/security/ReentrancyGuardUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/security/ReentrancyGuardUpgradeable.sol
@openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/token/ERC20/IERC20Upgradeable.sol
@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/token/ERC20/utils/SafeERC20Upgradeable.sol
@openzeppelin/contracts-upgradeable/token/ERC721/IERC721Upgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/token/ERC721/IERC721Upgradeable.sol
@openzeppelin/contracts-upgradeable/token/ERC721/utils/ERC721HolderUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/token/ERC721/utils/ERC721HolderUpgradeable.sol
@openzeppelin/contracts-upgradeable/utils/structs/EnumerableSetUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/tree/v4.8.2/contracts/utils/structs/EnumerableSetUpgradeable.sol

5.3 CallGraph

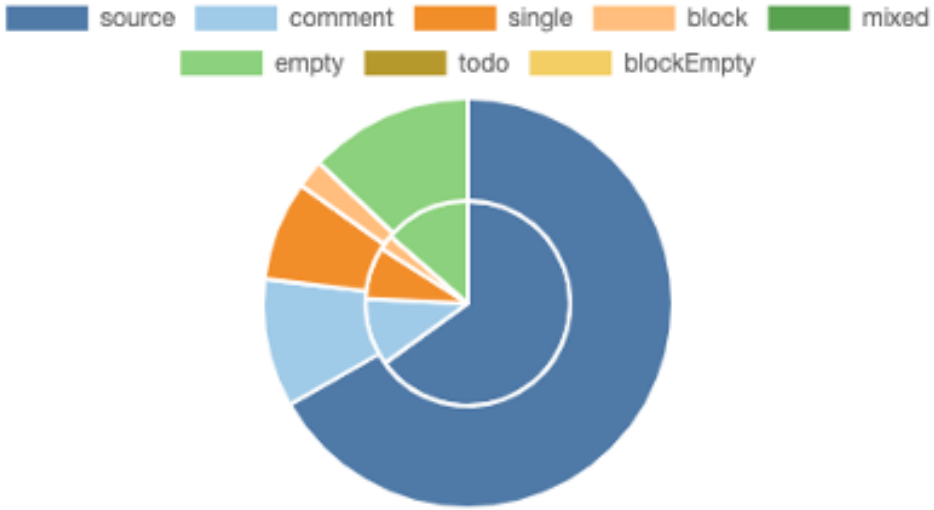


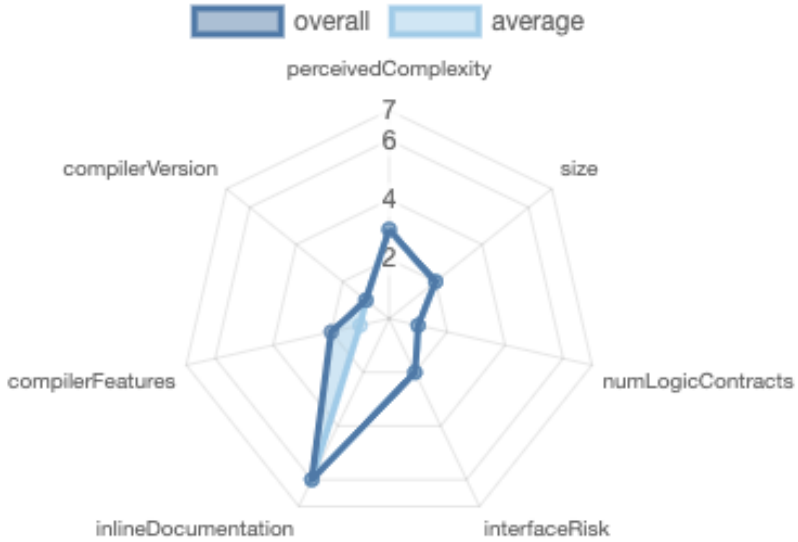


5.4 Inheritance Graph



5.5 Source Lines & Risk





5.6 Capabilities

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥️ Uses Assembly	💣 Has Destroyable Contracts
0.8.19				

📫 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧱 Uses Hash Functions	🪚 ECRecover	🌀 New/Create/Create2

Exposed Functions
This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

🌐Public	💰Payable
15	0

External	Internal	Private	Pure	View
14	19	0	0	4

StateVariables

Total	🌐Public
10	8

5.7 Source Unites in Scope

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
	contracts/ERC721PointsStakingV1.sol	1		319	302	220	36	195	
	Totals	**1**		**319**	**302**	**220**	**36**	**195**	

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments
- **Complexity Score**: a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...)

6. Scope of Work

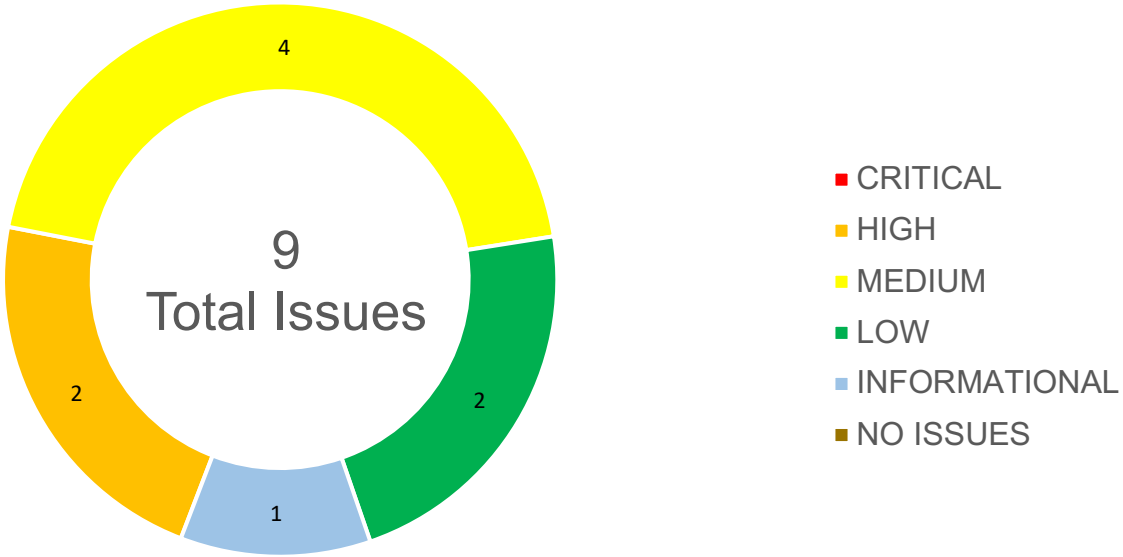
The DustLabsTeam provided us with the files that needs to be tested. The scope of the audit is the point staking contract.

The team put forward the following assumptions regarding the security, usage of the contracts:

- Staking and unstaking of NFTs is working as expected.
- Calculation of points is working as expected.
- Privileged functions for owner and manager are working as expected.
- The smart contract is coded according to the newest standards and in a secure way.

The main goal of this audit was to verify these claims. The auditors can provide additional feedback on the code upon the client's request.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Overpowered Manager/Owner Rights	HIGH	OPEN
6.2.2	Owner Can Permanently Lock Staked NFTs	HIGH	OPEN
6.2.3	Points Calculation Vulnerability	MEDIUM	OPEN
6.2.4	Owner Can Disable Staking	MEDIUM	OPEN
6.2.5	Potential Loss Of Ownership	MEDIUM	OPEN
6.2.6	Missing Function to Withdraw Collected Fees	MEDIUM	OPEN
6.2.7	State Variable Could Be Marked As Constant	LOW	OPEN
6.2.8	Not Emitting Events for Fee Updates	LOW	OPEN
6.2.9	Unexplicit uint Types	INFORMATIONAL	OPEN

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES
During the audit, Chainsulting's experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES
During the audit, Chainsulting's experts found **2 High issues** in the code of the smart contract.

6.2.1 Overpowered Manager/Owner Rights
Severity: HIGH
Status: OPEN
Code: CWE 264
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The contract manager can deduct any staking rewards by calling the spendPoints function. The manager is allowed to set any token reward point to zero. Additionally, the manager can set the token reward multiplier for each token to any desired value. This can lead to having no staking to very high staking rewards for any targeted token. Furthermore, the manager is able to set an unlimited amount of initial rewards for any token, that has not been staked before. In the current implementation, the owner can set and change the manager at will, thus the overpowered rights issue implicitly holds for the owner role as well.
Code	Line 245 – 288 (ERC721PointsStakingV1.sol) `function setRewardMultipliers(` `uint[] calldata _tokenIds,` `uint _newMultiplier` `) external nonReentrant managerOnly {` `for (uint i = 0; i < _tokenIds.length; i += 1) {` `_refreshPoints(_tokenIds[i]);`

```
            stakingMetadata[_tokenIds[i]].multiplier = uint16(_newMultiplier);
        }

        emit MultiplierUpdated(msg.sender, _newMultiplier, _tokenIds);
    }

    // @notice Initialize points
    // @param tokenIds The tokenIds of the NFTs to initialize points for
    function initPoints(
        uint[] calldata _tokenIds,
        uint[] calldata _points
    ) external nonReentrant managerOnly {
        if (_tokenIds.length != _points.length) {
            revert TokenIdsPointsLengthMismatch();
        }
        for (uint i = 0; i < _tokenIds.length; i += 1) {
            StakingMetadata storage metadata = stakingMetadata[_tokenIds[i]];
            if (metadata.lastUpdated != 0) {
                revert PointsInitialized();
            }
            metadata.points = uint32(_points[i]);
            metadata.lastUpdated = uint40(block.timestamp);
            emit SetPoints(msg.sender, _tokenIds[i], metadata.points);
        }
    }

    // @notice Spend points
    // @param tokenIds The tokenIds of the NFTs which will be spent
    function spendPoints(uint _tokenId, uint _points) external nonReentrant
managerOnly {
        _refreshPoints(_tokenId);

        StakingMetadata storage metadata = stakingMetadata[_tokenId];
```

	`` if (_points > metadata.points) { revert NotEnoughPoints(); } metadata.points -= uint32(_points); emit SpentPoints(msg.sender, _tokenId, _points); } ```
Result/Recommendation	It is recommended to remove overpowered owner rights such as deducting staking rewards at will. If there is a wish for adjustable staking multiplier rates, it is recommended to set a max and min value to ensure a specific range of reward multipliers. The same recommendation holds for setting an initial reward amount for unstaked tokens.

6.2.2 Owner Can Permanently Lock Staked NFTs

Severity: HIGH
Status: OPEN
Code: NA
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The users can unstake their staked NFTs by calling the withdraw function. This function charges a unstakeFee paid with the related unstakeFeeToken (ERC20-Token). The owner has the right to change the unstakeFeeToken to any address and the unstakeFee to any uint96 amount by calling the setFees function. Changing the unstakeFeeToken to an invalid address leads to a function call revert on any withdraw and thus permanent locking of the NFT. Additionally, if the owner set the unstakeFee to a high amount (max uint96), the users are very likely not able to unstake their NFT for a valid unstakeFeeToken.
Code	Line 168 – 172 (ERC721PointsStakingV1.sol) ``` // Transfer unstake fee to the staking contract unstakeFeeToken.safeTransferFrom(msg.sender, address(this), ```

	unstakeFee); Line 223 – 233 (ERC721PointsStakingV1.sol) function setFees(address _stakeFeeToken, uint _stakeFee, address _unstakeFeeToken, uint _unstakeFee) external onlyOwner { stakeFeeToken = IERC20Upgradeable(_stakeFeeToken); stakeFee = uint96(_stakeFee); unstakeFeeToken = IERC20Upgradeable(_unstakeFeeToken); unstakeFee = uint96(_unstakeFee); }
Result/Recommendation	It is highly recommended to set the *unstakeFeeToken* to a constant address, supporting the IERC20 interface. Otherwise, it is recommended to check if the new token address implements the IERC20 interface by implementing ERC165 checks. Additionally, it is highly recommended to limit the possible values for *unstakeFee,* by setting a max value. Having these two parameters in combination changeable, is not a good approach, because a maximum value for one token is not comparable to a maximum for a newly set token (i.e. 5 WETH are not comparable to 5 WBTC). Subsequently, it is highly recommended to set the *unstakeFeeToken* to a constant address.

MEDIUM ISSUES

During the audit, Chainsulting's experts found **4 Medium issues** in the code of the smart contract

6.2.3 Points Calculation Vulnerability

Severity: MEDUM
Status: OPEN
Code: CWE-682
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The current points calculation in the getPoints function may lead to incorrect results due to integer overflow.
Code	Line 86 – 94 (ERC721PointsStakingV1.sol) `function getPoints(uint32 _tokenId) external view returns (uint32) {` `StakingMetadata memory metadata = stakingMetadata[_tokenId];` `if (metadata.owner == address(0)) {` `return metadata.points;` `}` `uint16 multiplier = metadata.multiplier == 0 ? 100 : metadata.multiplier;` `return uint32(metadata.points + (block.timestamp - metadata.lastUpdated) * multiplier / 6000);` `}`
Result/Recommendation	Use the SafeMath library for arithmetic operations to prevent integer overflows. Update the getPoints function as follows: import "@openzeppelin/contracts-upgradeable/utils/math/SafeMathUpgradeable.sol"; ... using SafeMathUpgradeable for uint; ... function getPoints(uint32 _tokenId) external view returns (uint32) { ... uint16 multiplier = metadata.multiplier == 0 ? 100 : metadata.multiplier;

	return uint32(metadata.points.add((block.timestamp.sub(metadata.lastUpdated)).mul(multiplier).div(6000))); }

6.2.4 Owner Can Disable Staking

Severity: MEDUM
Status: OPEN
Code: NA
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The users can stake their NFTs by calling the *stake* function. This function charges a *stakeFee,* paid with the related *stakeFeeToken* (ERC20-Token). The owner has the right to change the *stakeFeeToken* to any address and the *stakeFee* to any uint96 amount by calling the *setFees* function. Changing the *stakeFeeToken* to an invalid address lead to a function call revert, on staking any token even if the contract is unpaused. Additionally, if the owner set the *stakeFee* to a high amount (max uint96), the users are very likely not able to stake their NFT for a valid *stakeFeeToken.*
Code	Line 136 – 137 (ERC721PointsStakingV1.sol) `// Transfer stake fee to the staking contract` `stakeFeeToken.safeTransferFrom(msg.sender, address(this), stakeFee);` Line 223 – 233 (ERC721PointsStakingV1.sol) `function setFees(` `address _stakeFeeToken,` `uint _stakeFee,` `address _unstakeFeeToken,` `uint _unstakeFee` `) external onlyOwner {` `stakeFeeToken = IERC20Upgradeable(_stakeFeeToken);`

	stakeFee = uint96(_stakeFee); unstakeFeeToken = IERC20Upgradeable(_unstakeFeeToken); unstakeFee = uint96(_unstakeFee); }
Result/Recommendation	It is highly recommended to set the *stakeFeeToken* to a constant address supporting the IERC20 interface. Otherwise, it is recommended to check if the new token address implements the IERC20 interface by implementing ERC165 checks. Additionally, it is recommended to limit the possible values for *stakeFee* by setting a max value. Having these two parameters in combination changeable is not a good approach, because a maximum value for one token is not comparable to a maximum for a newly set token (i.e. 5 WETH are not comparable to 5 WBTC). Subsequently, it is highly recommended to set the *stakeFeeToken* to a constant address.

6.2.5 Potential Loss Of Ownership
Severity: MEDUM
Status: OPEN
Code: NA
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The owner of the contract can be changed by calling transferOwnership function of OpenZeppelin Ownable implementation. This function directly sets the owner to the given address if the address is not the zero address. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.
Code	Line 10 (ERC721PointsStakingV1.sol) import {OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

Result/Recommendation	It is recommended to use the two-step pattern, by setting the new owner as potential owner in the first step and approving the ownership transfer by calling an approve function, by the new owner. OpenZeppelin provides such a Ownable2Step implementation. This prevents transferring ownership to an invalid address.

6.2.6 Missing Function to Withdraw Collected Fees

Severity: MEDUM
Status: OPEN
Code: CWE-384 (Session Fixation)
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The contract currently collects stake and unstake fees, but there is no functionality for the owner or manager to withdraw these collected fees. This could lead to a lock-up of funds in the contract with no way to access them.
Code	Line 136 – 137 (ERC721PointsStakingV1.sol) `// Transfer stake fee to the staking contract` `stakeFeeToken.safeTransferFrom(msg.sender, address(this), stakeFee);` Line 167 – 172 (ERC721PointsStakingV1.sol) `// Transfer unstake fee to the staking contract` `unstakeFeeToken.safeTransferFrom(` `msg.sender,` `address(this),` `unstakeFee` `);`
Result/Recommendation	It is recommended to implement a restricted fee withdraw function, to claim all collected staking and unstaking fees. Alternatively, it is recommended to implement a fee receiving address to send the tokens directly to an address of choice.

LOW ISSUES

During the audit, Chainsulting's experts found **2 Low issues** in the code of the smart contract

6.2.7 State Variable Could Be Marked As Constant

Severity: LOW
Status: OPEN
Code: NA
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The *stakingToken* variable is defined as normal state variable although it is set once during contract initialization and is never changed.
Code	Line 44 (ERC721PointsStakingV1.sol) `IERC721Upgradeable public stakingToken;`
Result/Recommendation	It is recommended to mark the *stakingToken* state variable as *immutable,* to symbolize that it is not changeable the staking contract is permanently valid for one ERC721 contract.

6.2.8 Not Emitting Events for Fee Updates

Severity: LOW
Status: OPEN
Code: CWE-778 (Insufficient Logging)
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The contract does not emit any events when updating stake and unstake fees. This makes it difficult for external parties, such as users or third-party services, to track changes in the fee structure.
Code	Line 223 - 233 (ERC721PointsStakingV1.sol) `function setFees(` `address _stakeFeeToken,` `uint _stakeFee,` `address _unstakeFeeToken,` `uint _unstakeFee` `) external onlyOwner {` `stakeFeeToken = IERC20Upgradeable(_stakeFeeToken);` `stakeFee = uint96(_stakeFee);` `unstakeFeeToken = IERC20Upgradeable(_unstakeFeeToken);` `unstakeFee = uint96(_unstakeFee);` `}`
Result/Recommendation	Implement events for updating stake and unstake fees and emit them in the setFees function. This will provide transparency to users and third-party services about changes in the fee structure of the contract. Example of events to add: event StakeFeeUpdated(address indexed _stakeFeeToken, uint256 _stakeFee); event UnstakeFeeUpdated(address indexed _unstakeFeeToken, uint256 _unstakeFee); Example of emitting events in the setFees function: function setFees(address _stakeFeeToken, uint _stakeFee, address _unstakeFeeToken, uint _unstakeFee) external onlyOwner {

	stakeFeeToken = IERC20Upgradeable(_stakeFeeToken); stakeFee = uint96(_stakeFee); unstakeFeeToken = IERC20Upgradeable(_unstakeFeeToken); unstakeFee = uint96(_unstakeFee); emit StakeFeeUpdated(_stakeFeeToken, _stakeFee); emit UnstakeFeeUpdated(_unstakeFeeToken, _unstakeFee); }

INFORMATIONAL ISSUES

During the audit, Chainsulting's experts found **1 Informational issue** in the code of the smart contract

6.2.9 Unexplicit uint Types

Severity: INFORMATIONAL
Status: OPEN
Code: NA
File(s) affected: ERC721PointsStakingV1.sol

Attack / Description	The current implementation uses the term uint for defining unsigned integers. This term is implicitly using the uint256.
Code	NA
Result/Recommendation	It is recommended to use uint256 explicitly to indicate that the largest sized unsigned integer is intended to be used. This is a recommendation best on best practices.

6.3 SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✅
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✅
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✅
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✅
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✅
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✅
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✅
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✅

ID	Title	Relationships	Test Result
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✅
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✅
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✅
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✅
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✅
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✅
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✅
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✅
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✅

ID	Title	Relationships	Test Result
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✅
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✅
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✅
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✅
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✅
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✅
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✅
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✅
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✅
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✅

ID	Title	Relationships	Test Result
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✅
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✅
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✅
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✅

6.4 Verify Claims

6.4.1 Staking and unstaking of NFTs is working as expected
Status: tested and verified ✅

6.4.2 Calculation of points is working as expected
Status: tested and verified ✅

6.4.3 Privileged functions for owner and manager are working as expected
Status: tested and verified ✅

6.4.4 The smart contract is coded according to the newest standards and in a secure way.
Status: tested and verified ✅

6.5 Unit Tests

ERC721 Staking
- ✓ Should initialize properly with correct configuration

Owner
- ✓ Should set owner to deployer
- ✓ Should allow owner to pause
- ✓ Should allow owner to unpause
- ✓ Should not allow nonOnwer to pause
- ✓ Should not allow nonOnwer to unpause
- ✓ Should allow owner to set a manager
- ✓ Should not allow nonOwner to set a manager
- ✓ Should not allow nonOwner to unset a manager
- ✓ Should allow owner to set fees
- ✓ Should not allow nonOwner to set fees

Manager
- ✓ Should allow manager to set a reward multiplier
- ✓ Should not allow nonManager to set a reward multiplier
- ✓ Should allow manager to init points
- ✓ Should not allow nonManager to init points
- ✓ Should allow manager to spend points
- ✓ Should not allow spending points if not manager
- ✓ Should emit events correctly

Staking
- ✓ Should stake 1 NFT successfully
- ✓ Should stake multiple NFTs successfully
- ✓ Should update fields correctly on second time staking
- ✓ Should revert on staking non-existing tokens

```
    ✓ Should revert on staking non-owned tokens
    ✓ Should not allow staking of no tokens
    ✓ Should not allow staking when paused
    ✓ Should not allow staking if not enough stake fee
    ✓ Should emit events correctly
  Withdrawal
    ✓ Should withdraw staked NFTs successfully
    ✓ Should withdraw when paused
    ✓ Should emit events correctly on Withdraw
    ✓ Should not be able to withdraw NFTs staked by other person
    ✓ Should not allow withdraw of no tokens
    ✓ Should not allow withdraw if not enough unstaking fee
  Points
    ✓ Should accrue points correctly for 1 NFT staked
    ✓ Should get points correctly if time elapsed is not divisible by minutes
    ✓ Should accrue points correctly when staked and multiplier is set
    ✓ Should not accrue points when unstaked
    ✓ Should correctly set points after withdrawal
    ✓ Should init points correctly
    ✓ Should not allow setting points if already set
    ✓ Should not allow setting points if array lengths do not match
    ✓ Should spend points correctly
    ✓ Should not allow spending points if not enough
43 passing (3s)
```

Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	95	93.33	98.68	
ERC721PointsStakingV1.sol	100	95	93.33	98.68	168
All files	100	95	93.33	98.68	

7. Executive Summary

Two (2) independent Chainsulting experts performed an unbiased and isolated audit of the smart contract codebase.

The main goal of the audit was to verify the claims regarding the security and functions of the smart contract. During the audit, no critical, two high, four medium, two low and one informational issues have been found, after the manual and automated security testing.

We advise the DustLabs team to implement the recommendations to further enhance the code's security and readability.

8. About the Auditor

Chainsulting is a professional software development firm, founded in 2017 and based in Germany. They show ways, opportunities, risks and offer comprehensive Web3 solutions. Their services include Web3 development, security and consulting.

Chainsulting conducts code audits on market-leading blockchains such as Solana, Tezos, Ethereum, Binance Smart Chain, and Polygon to mitigate risk and instil trust and transparency into the vibrant crypto community. They have also reviewed and secure the smart contracts of many top DeFi projects.

Chainsulting currently secures $100 billion in user funds locked in multiple DeFi protocols. The team behind the leading audit firm relies on their robust technical know-how in the web3 sector to deliver top-notch smart contract audit solutions, tailored to the clients' evolving business needs.

Check our website for further information: https://chainsulting.de