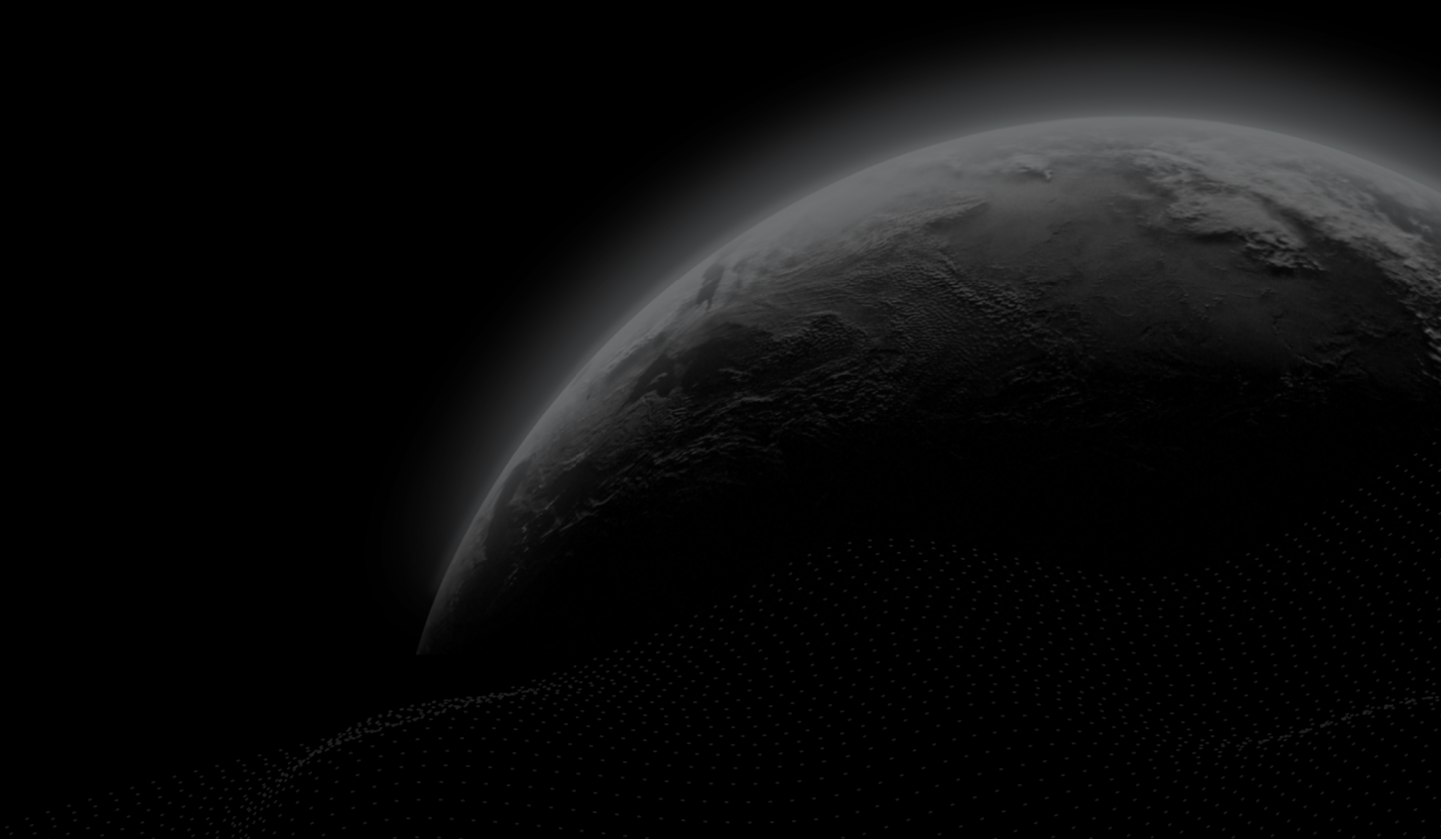
CERTIK

Security Assessment

Mindful Ocean Metaverse

CertiK Assessed on Oct 31st, 2023



CertiK Assessed on Oct 31st, 2023

Mindful Ocean Metaverse

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
NFT	Ethereum (ETH)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 10/31/2023	N/A

CODEBASE

DLL-Smart-Contracts

View All in Codebase Page

COMMITS

02c1b8b71cc3ca2b9151d232beb9721b5aa093f0

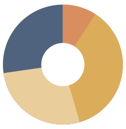
0cd040071509104914f95fe86fc142af678d24ab

View All in Codebase Page

Highlighted Centralization Risks

ⓘ Fees are unbounded

Vulnerability Summary



11	10	0	0	1	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

■ 0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
■ 1	Major	1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
■ 4	Medium	4 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
■ 3	Minor	3 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
■ 3	Informational	3 Resolved	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | MINDFUL OCEAN METAVERSE

CODEBASE | MINDFUL OCEAN METAVERSE

Repository

DLL-Smart-Contracts

Commit

02c1b8b71cc3ca2b9151d232beb9721b5aa093f0

0cd040071509104914f95fe86fc142af678d24ab

AUDIT SCOPE | MINDFUL OCEAN METAVERSE

2 files audited ● 2 files with Acknowledged findings

ID	Repo	File	SHA256 Checksum
● MNF	edwardtam919/DLL-Smart-Contracts	contracts/MindfulNFT.sol	86814683d3290b73166e4704b36f51b27a24f5c0e60d483cacd3bb6e85a64668
● MPD	edwardtam919/DLL-Smart-Contracts	contracts/MintPass.sol	f4c719317a82fa59ae0a5f3e04278a5f8f64f5e9fa7373535a2cafe609274930

APPROACH & METHODS | MINDFUL OCEAN METAVERSE

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

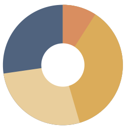
The auditing process pays special attention to the following considerations:

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

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

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

FINDINGS | MINDFUL OCEAN METAVERSE



11	0	1	4	3	3
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Mindful Ocean Metaverse. Through this audit, we have uncovered 11 issues ranging from different severity levels. Utilizing the techniques of Static Analysis & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

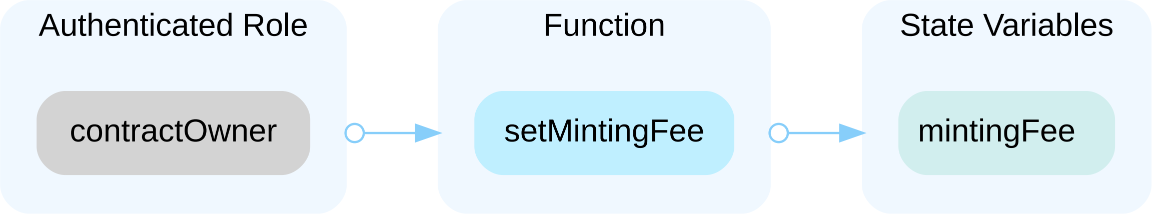
ID	Title	Category	Severity	Status
MPD-02	**Centralization Risks In MintPass.Sol**	**Centralization**	**Major**	● **Acknowledged**
DLL-04	Missing Check Of Pause Status In Minting Functions	Volatile Code, Logical Issue	Medium	● Resolved
DLL-05	Missing User Input Validation Of `TokenURI`	Logical Issue	Medium	● Resolved
MPD-05	Unhandled Overpayment In Native Token Transfers	Logical Issue	Medium	● Resolved
MPD-10	Lack Of Signature Replay Protection	Volatile Code, Logical Issue	Medium	● Resolved
DLL-03	Checks-Effects-Interactions Pattern Violated	Concurrency	Minor	● Resolved
MPD-07	Usage Of `transfer` / `send` For Sending Native Tokens	Volatile Code	Minor	● Resolved
MPD-08	Deprecated Usage Of `Counters.sol`	Logical Issue	Minor	● Resolved
MPD-01	Unnecessary Use Of SafeMath	Coding Issue	Informational	● Resolved
MPD-03	Missing Emit Events	Coding Style	Informational	● Resolved
MPD-09	Require Without Error Message	Coding Style	Informational	● Resolved

MPD-02 | CENTRALIZATION RISKS IN MINTPASS.SOL

Category	Severity	Location	Status
Centralization	● Major	contracts/MintPass.sol: 55	● Acknowledged

Description

In the contract `MintPassNFT` the role `contractOwner` has authority over the functions shown in the diagram below. Any compromise to the `contractOwner` account may allow the hacker to take advantage of this authority and set arbitrary minting fees.



Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

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

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

Long Term:

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

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

Permanent:

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

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

Alleviation

[Mindful Ocean Team, 10/31/2023:]

- will use Multisig solution (i.e. Gnosis) for managing crypto wallet
- added timelock feature for setMintingFee function (i.e. wait 2 days before minting fee can be changed again)
- before service launch, will write up some articles on Medium.com to explain about the Minting Fee issues.

DLL-04 | MISSING CHECK OF PAUSE STATUS IN MINTING FUNCTIONS

Category	Severity	Location	Status
Volatile Code, Logical Issue	● Medium	contracts/MindfulNFT.sol: 38; contracts/MintPass.sol: 6 6	● Resolved

Description

In the linked contracts the functions `mintMintPass()` and `onERC721Received()` should have `whenNotPaused` modifier to completely disable token minting when contracts are paused as the comments suggest.

```
    // pause minting action
    function pause() public onlyOwner {
        _pause();
    }

    // unpause minting action
    function unpause() public onlyOwner {
        _unpause();
    }
```

Recommendation

We recommend adding `whenNotPaused` modifier to the functions `mintMintPass()` and `onERC721Received()` .

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/756864d91d9bec62ed12c909629e1f0394ad96b7.

DLL-05 | MISSING USER INPUT VALIDATION OF `TokenURI`

Category	Severity	Location	Status
Logical Issue	● Medium	contracts/MindfulNFT.sol: 38~41; contracts/MintPass.sol: 66	● Resolved

Description

Function `mintMintPass()` does not validate user input `URI` in the signature, which allows users to set arbitrary token URI.

In the `onERC721Received()` function, the `_data` field can be leveraged to specify a new URI for a burnt `mintPass` token when performing a `mintPass.SafeTransferFrom()` call. This again is not validated and can be arbitrary data.

```
38      function onERC721Received(address, address _from, uint256 _tokenId, bytes calldata _data) external returns(bytes4) {
39
40          bool isBurned = false;
41          string memory url = string(_data);
42          ...
```

Recommendation

The auditors would like to discuss with the team whether users are allowed to set arbitrary token URIs when minting `mintPass` tokens, and whether modifying the token URI is permitted when minting `MindfulNFT` tokens.

Alleviation

[Mindful Ocean Team, 10/31/2023:]

URI checking is done in the backend before calling the minting function.

MPD-05 | UNHANDLED OVERPAYMENT IN NATIVE TOKEN TRANSFERS

Category	Severity	Location	Status
Logical Issue	● Medium	contracts/MintPass.sol: 72~73	● Resolved

Description

The function is marked as payable, but the surplus native token is not returned. In addition, the contract does not have any mechanism to extract the tokens. This would lead to the lock of the surplus tokens.

```
72      function mintMintPass(bytes32 hash, bytes memory signature, string memory uri) public payable returns(uint256){
73
74          // check signature
75          require(recoverSigner(hash, signature) == systemAddress, "Signature Failed");
76
77          // transfer minting fee to the defined wallet
78          require(msg.value >= mintingFee, "Not enough MATIC sent; check price!");
79          payable(address(mintingFeeRecipient)).transfer(mintingFee);
80
81          // set tokeID & recipient
82          uint256 tokenId = _tokenIdCounter.current();
83          _tokenIdCounter.increment();
84          _safeMint(msg.sender, tokenId);
85
86          // set loyalty fee
87          _setTokenRoyalty(tokenId, msg.sender, loyaltyFee);
88
89          // set token URI
90          _setTokenURI(tokenId, uri);
91
92          emit tokenIdMinted(tokenId);
93
94          return tokenId;
95      }
```

Recommendation

To mitigate this vulnerability, linked function should be modified to refund any excess native tokens sent by the user. This can be accomplished by sending back the difference between `msg.value` and `mintingFee` to the sender.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd040071509104914f95fe86fc142af678d24ab

MPD-10 | LACK OF SIGNATURE REPLAY PROTECTION

Category	Severity	Location	Status
Volatile Code, Logical Issue	● Medium	contracts/MintPass.sol: 69	● Resolved

Description

The linked contract does not enforce any checks to prevent the reuse of a previously used signature.

```
66      function mintMintPass(bytes32 hash, bytes memory signature, string memory uri) public payable returns(uint256){
67
68          // check signature
69          require(recoverSigner(hash, signature) == systemAddress, "Signature Failed");
70
71          // transfer minting fee to the defined wallet
72          require(msg.value >= mintingFee, "Not enough MATIC sent; check price!");
73          payable(address(mintingFeeRecipient)).transfer(mintingFee);
74
75          // set tokeID & recipient
76          uint256 tokenId = _tokenIdCounter.current();
77          _tokenIdCounter.increment();
78          _safeMint(msg.sender, tokenId);
79          ...
80      }
```

The user can use any previously used valid signature to mint a new NFT.

Recommendation

To mitigate the risk of replay attacks, it is recommended to implement a mechanism to track used signatures. For instance, you could use a mapping to store used signatures and check against this mapping whenever the `mintMintPass()` function is called. If a signature is found in the mapping, the function should revert to prevent the double-mint.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049 14f95fe86fc142af678d24ab

DLL-03 | CHECKS-EFFECTS-INTERACTIONS PATTERN VIOLATED

Category	Severity	Location	Status
Concurrency	● Minor	contracts/MindfulNFT.sol: 60~66; contracts/MintPass.sol: 78, 81, 84	● Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call (`_safeMint` will trigger the `onERC721Received()` function on the receiver's contract) to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

External call(s) in `MindfulNFT.sol`

```
60          _safeMint(_from, _tokenId);
```

- This function call executes the following external call(s).
- In `ERC721._checkOnERC721Received` ,
    - `retval = IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data)`

State variables written after the call(s)

```
63          _setTokenRoyalty(_tokenId, _from, loyaltyFee);
```

- This function call executes the following assignment(s).
- In `ERC2981._setTokenRoyalty` ,
    - `_tokenRoyaltyInfo[tokenId] = RoyaltyInfo(receiver,feeNumerator)`

```
66          _setTokenURI(_tokenId, uri);
```

- This function call executes the following assignment(s).
- In `ERC721URIStorage._setTokenURI` ,
    - `_tokenURIs[tokenId] = _tokenURI`

External call(s) in `MintPass.sol`

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

- This function call executes the following external call(s).
- In `ERC721._checkOnERC721Received` ,
    - `retval = IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data)`

State variables written after the call(s)

```
        _setTokenRoyalty(tokenId, msg.sender, loyaltyFee);
```

- This function call executes the following assignment(s).
- In `ERC2981._setTokenRoyalty` ,
    - `_tokenRoyaltyInfo[tokenId] = RoyaltyInfo(receiver,feeNumerator)`

```
        _setTokenURI(tokenId, uri);
```

- This function call executes the following assignment(s).
- In `ERC721URIStorage._setTokenURI` ,
    - `_tokenURIs[tokenId] = _tokenURI`

It is recommended to always first change the state before doing external calls.

Recommendation

It's recommended using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd040071509104914f95fe86fc142af678d24ab

MPD-07 USAGE OF `transfer` / `send` FOR SENDING NATIVE TOKENS

Category	Severity	Location	Status
Volatile Code	● Minor	contracts/MintPass.sol: 73	● Resolved

Description

It is not recommended to use Solidity's `transfer()` and `send()` functions for transferring native tokens, since some contracts may not be able to receive the funds. Those functions forward only a fixed amount of gas (2300 specifically) and the receiving contracts may run out of gas before finishing the transfer. Also, EVM instructions' gas costs may increase in the future. Thus, some contracts that can receive now may stop working in the future due to the gas limitation.

```
73          payable(address(mintingFeeRecipient)).transfer(mintingFee);
```

- `MintPassNFT.mintMintPass` uses `transfer()`.

Recommendation

We recommend using the `Address.sendValue()` function from OpenZeppelin.

Since `Address.sendValue()` may allow reentrancy, we also recommend guarding against reentrancy attacks by utilizing the Checks-Effects-Interactions Pattern or applying OpenZeppelin ReentrancyGuard.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049 14f95fe86fc142af678d24ab

MPD-08 | DEPRECATED USAGE OF `Counters.sol`

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/MintPass.sol: 16	● Resolved

Description

The linked contracts import and use OpenZeppelin's `Counters` contract. OpenZeppelin has deprecated the usage of the `Counters` contract: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/4233

Recommendation

Consider removing the usage of deprecated 3rd party contracts.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd040071509104914f95fe86fc142af678d24ab

MPD-01 | UNNECESSARY USE OF SAFEMATH

Category	Severity	Location	Status
Coding Issue	● Informational	contracts/MintPass.sol: 19	● Resolved

Description

The `SafeMath` library is used unnecessarily. With Solidity compiler versions 0.8.0 or newer, arithmetic operations will automatically revert in case of integer overflow or underflow.

```
19      using SafeMath for uint256;
```

- `SafeMath` library is used for `uint256` type in `MintPassNFT` contract.

Recommendation

We advise removing the usage of `SafeMath` library and using the built-in arithmetic operations provided by the Solidity programming language.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049144f95fe86fc142af678d24ab

MPD-03 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	● Informational	contracts/MintPass.sol: 55~58	● Resolved

Description

Functions that update state variables should emit relevant events as notifications.

```
55      function setMintingFee(uint256 _mintingFee) public  {
56          require(contractOwner == msg.sender);
57          mintingFee = _mintingFee;
58      }
```

Recommendation

It is recommended to add events for state-changing actions, and emitting them in their relevant functions.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd040071509104914f95fe86fc142af678d24ab

MPD-09 | REQUIRE WITHOUT ERROR MESSAGE

Category	Severity	Location	Status
Coding Style	● Informational	contracts/MintPass.sol: 56	● Resolved

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

```
  function setMintingFee(uint256 _mintingFee) public  {
        require(contractOwner == msg.sender);
        mintingFee = _mintingFee;
    }
```

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049l4f95fe86fc142af678d24ab

OPTIMIZATIONS | MINDFUL OCEAN METAVERSE

ID	Title	Category	Severity	Status
DLL-01	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	● Resolved
MNF-03	Dead Code	Coding Style	Optimization	● Acknowledged
MPD-04	Inefficient Memory Parameter	Gas Optimization	Optimization	● Resolved
MPD-11	Imports Are Not Used	Code Optimization	Optimization	● Resolved

DLL-01 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	● Optimization	contracts/MindfulNFT.sol: 20, 21; contracts/MintPass.sol: 21, 22, 24, 25	● Resolved

Description

The linked variables assigned in the constructor can be declared as `immutable`. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

We recommend declaring these variables as immutable. Please note that the `immutable` keyword only works in Solidity version `v0.6.5` and up.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049144f95fe86fc142af678d24ab

MNF-03 | DEAD CODE

Category	Severity	Location	Status
Coding Style	● Optimization	contracts/MindfulNFT.sol: 72~74	● Acknowledged

Description

The linked internal function is not used.

```
function _burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
```

Recommendation

We recommend removing those unused functions for gas optimization purpose.

MPD-04 | INEFFICIENT MEMORY PARAMETER

Category	Severity	Location	Status
Gas Optimization	● Optimization	contracts/MintPass.sol: 66, 66	● Resolved

Description

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
66      function mintMintPass(bytes32 hash, bytes memory signature, string memory
 uri) public payable returns(uint256){
```

`mintMintPass` has memory location parameters: `signature` , `uri` .

Recommendation

We recommend changing the parameter's data location to `calldata` to save gas.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd0400715091049l4f95fe86fc142af678d24ab

MPD-11 | IMPORTS ARE NOT USED

Category	Severity	Location	Status
Code Optimization	● Optimization	contracts/MintPass.sol: 13	● Resolved

Description

The linked contract imports a contract that is never used.

Recommendation

We advise to remove the imports from the aforementioned lines to increase the legibility and quality of the codebase.

Alleviation

Fixed in https://github.com/edwardtam919/DLL-Smart-Contracts/commit/0cd040071509104914f95fe86fc142af678d24ab

FORMAL VERIFICATION | MINDFUL OCEAN METAVERSE

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied automated formal verification (symbolic model checking) to prove that well-known functions in the smart contracts adhere to their expected behavior.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of Compliance with Pausable ERC-721

We verified the properties of the public interface of those token contracts that implement the pausable ERC-721 interface.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc721pausable-supportsinterface-correct-erc721	`supportsInterface` Signals Support for `ERC721`
erc721pausable-supportsinterface-erc721-receiver	`supportsInterface` Signals Support for `ERC721 Token Receiver`
erc721pausable-balanceof-succeed-normal	`balanceOf` Succeeds on Admissible Inputs
erc721pausable-balanceof-correct-count	`balanceOf` Returns the Correct Value
erc721pausable-balanceof-revert	`balanceOf` Fails on the Zero Address
erc721pausable-transferfrom-succeed-normal	`transferFrom` Succeeds on Admissible Inputs
erc721pausable-balanceof-no-change-state	`balanceOf` Does Not Change the Contract's State
erc721pausable-ownerof-succeed-normal	`ownerOf` Succeeds For Valid Tokens
erc721pausable-ownerof-correct-owner	`ownerOf` Returns the Correct Owner
erc721pausable-ownerof-revert	`ownerOf` Fails On Invalid Tokens
erc721pausable-ownerof-no-change-state	`ownerOf` Does Not Change the Contract's State
erc721pausable-getapproved-succeed-normal	`getApproved` Succeeds For Valid Tokens
erc721pausable-transferfrom-revert-pause	`transferFrom` Fails when Paused

Property Name	Title
erc721pausable-getapproved-correct-value	`getApproved` Returns Correct Approved Address
erc721pausable-getapproved-revert-zero	`getApproved` Fails on Invalid Tokens
erc721pausable-getapproved-change-state	`getApproved` Does Not Change the Contract's State
erc721pausable-isapprovedforall-succeed-normal	`isApprovedForAll` Always Succeeds
erc721pausable-isapprovedforall-correct	`isApprovedForAll` Returns Correct Approvals
erc721pausable-isapprovedforall-change-state	`isApprovedForAll` Does Not Change the Contract's State
erc721pausable-approve-succeed-normal	`approve` Returns for Admissible Inputs
erc721pausable-approve-set-correct	`approve` Sets Approval
erc721pausable-approve-revert-invalid-token	`approve` Fails For Calls with Invalid Tokens
erc721pausable-approve-revert-not-allowed	`approve` Prevents Unpermitted Approvals
erc721pausable-setapprovalforall-succeed-normal	`setApprovalForAll` Returns for Admissible Inputs
erc721pausable-approve-change-state	`approve` Has No Unexpected State Changes
erc721pausable-setapprovalforall-set-correct	`setApprovalForAll` Approves Operator
erc721pausable-setapprovalforall-multiple	`setApprovalForAll` Can Set Multiple Operators
erc721pausable-setapprovalforall-change-state	`setApprovalForAll` Has No Unexpected State Changes
erc721pausable-transferfrom-correct-increase	`transferFrom` Transfers the Complete Token in Non-self Transfers
erc721pausable-transferfrom-correct-one-token-self	`transferFrom` Performs Self Transfers Correctly
erc721pausable-transferfrom-correct-approval	`transferFrom` Updates the Approval Correctly
erc721pausable-transferfrom-correct-owner-from	`transferFrom` Removes Token Ownership of From
erc721pausable-transferfrom-correct-owner-to	`transferFrom` Transfers Ownership
erc721pausable-transferfrom-correct-balance	`transferFrom` Sum of Balances is Constant
erc721pausable-transferfrom-correct-state-balance	`transferFrom` Keeps Balances Constant Except for From and To

Property Name	Title
erc721pausable-transferfrom-correct-state-owner	`transferFrom` Has Expected Ownership Changes
erc721pausable-transferfrom-correct-state-approval	`transferFrom` Has Expected Approval Changes
erc721pausable-transferfrom-revert-invalid	`transferFrom` Fails for Invalid Tokens
erc721pausable-transferfrom-revert-from-zero	`transferFrom` Fails for Transfers From the Zero Address
erc721pausable-transferfrom-revert-to-zero	`transferFrom` Fails for Transfers To the Zero Address
erc721pausable-supportsinterface-metadata	`supportsInterface` Signals that ERC721Metadata is Implemented
erc721pausable-supportsinterface-succeed-always	`supportsInterface` Always Succeeds
erc721pausable-transferfrom-revert-not-owned	`transferFrom` Fails if `From` Is Not Token Owner
erc721pausable-supportsinterface-correct-erc165	`supportsInterface` Signals Support for ERC165
erc721pausable-supportsinterface-correct-false	`supportsInterface` Returns `False` for Id 0xffffffff
erc721pausable-transferfrom-revert-exceed-approval	`transferFrom` Fails for Token Transfers without Approval
erc721pausable-supportsinterface-no-change-state	`supportsInterface` Does Not Change the Contract's State

Verification Results

In the remainder of this section, we list all contracts where model checking of at least one property was not successful. There are several reasons why this could happen:

- Model checking reports a counterexample that violates the property. Depending on the counterexample,this occurs if
  - The specification of the property is too generic and does not accurately capture the intended behavior of the smart contract. In that case, the counterexample does not indicate a problem in the underlying smart contract. We report such instances as being "inapplicable".
  - The property is applicable to the smart contract. In that case, the counterexample showcases a problem in the smart contract and a correspond finding is reported separately in the Findings section of this report. In the following tables, we report such instances as "invalid". The distinction between spurious and actual counterexamples is done manually by the auditors.

- The model checking result is inconclusive. Such a result does not indicate a problem in the underlying smart contract. An inconclusive result may occur if
  - The model checking engine fails to construct a proof. This can happen if the logical deductions necessary are beyond the capabilities of the automated reasoning tool. It is a technical limitation of all proof engines and cannot be avoided in general.

- The model checking engine runs out of time or memory and did not produce a result. This can happen if automatic abstraction techniques are ineffective or of the state space is too big.

Detailed Results For Contract MindfulNFT (contracts/MindfulNFT.sol) In Commit 02c1b8b71cc3ca2b9151d232beb9721b5aa093f0

Verification of Compliance with Pausable ERC-721

Detailed results for function `supportsInterface`

Property Name	Final Result	Remarks
erc721pausable-supportsinterface-correct-erc721	🟢 True	
erc721pausable-supportsinterface-erc721-receiver	🔴 False	
erc721pausable-supportsinterface-metadata	🟢 True	
erc721pausable-supportsinterface-succeed-always	🟢 True	
erc721pausable-supportsinterface-correct-erc165	🟢 True	
erc721pausable-supportsinterface-correct-false	🟢 True	
erc721pausable-supportsinterface-no-change-state	🟢 True	

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc721pausable-balanceof-succeed-normal	🟢 True	
erc721pausable-balanceof-correct-count	🟢 True	
erc721pausable-balanceof-revert	🟢 True	
erc721pausable-balanceof-no-change-state	🟢 True	

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc721pausable-transferfrom-succeed-normal	● True	
erc721pausable-transferfrom-revert-pause	● False	
erc721pausable-transferfrom-correct-increase	● True	
erc721pausable-transferfrom-correct-one-token-self	● True	
erc721pausable-transferfrom-correct-approval	● True	
erc721pausable-transferfrom-correct-owner-from	● True	
erc721pausable-transferfrom-correct-owner-to	● True	
erc721pausable-transferfrom-correct-balance	● True	
erc721pausable-transferfrom-correct-state-balance	● True	
erc721pausable-transferfrom-correct-state-owner	● True	
erc721pausable-transferfrom-correct-state-approval	● True	
erc721pausable-transferfrom-revert-invalid	● True	
erc721pausable-transferfrom-revert-from-zero	● True	
erc721pausable-transferfrom-revert-to-zero	● True	
erc721pausable-transferfrom-revert-not-owned	● True	
erc721pausable-transferfrom-revert-exceed-approval	● True	

Detailed results for function `ownerOf`

Property Name	Final Result	Remarks
erc721pausable-ownerof-succeed-normal	● True	
erc721pausable-ownerof-correct-owner	● True	
erc721pausable-ownerof-revert	● True	
erc721pausable-ownerof-no-change-state	● True	

Detailed results for function `getApproved`

Property Name	Final Result	Remarks
erc721pausable-getapproved-succeed-normal	● True	
erc721pausable-getapproved-correct-value	● True	
erc721pausable-getapproved-revert-zero	● True	
erc721pausable-getapproved-change-state	● True	

Detailed results for function `isApprovedForAll`

Property Name	Final Result	Remarks
erc721pausable-isapprovedforall-succeed-normal	● True	
erc721pausable-isapprovedforall-correct	● True	
erc721pausable-isapprovedforall-change-state	● True	

Detailed results for function `approve`

Property Name	Final Result	Remarks
erc721pausable-approve-succeed-normal	● True	
erc721pausable-approve-set-correct	● True	
erc721pausable-approve-revert-invalid-token	● True	
erc721pausable-approve-revert-not-allowed	● True	
erc721pausable-approve-change-state	● True	

Detailed results for function `setApprovalForAll`

Property Name	Final Result	Remarks
erc721pausable-setapprovalforall-succeed-normal	● True	
erc721pausable-setapprovalforall-set-correct	● True	
erc721pausable-setapprovalforall-multiple	● True	
erc721pausable-setapprovalforall-change-state	● True	

Detailed Results For Contract MintPassNFT (contracts/MintPass.sol) In Commit 02c1b8b71cc3ca2b9151d232beb9721b5aa093f0

Verification of Compliance with Pausable ERC-721

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc721pausable-balanceof-succeed-normal	● True	
erc721pausable-balanceof-revert	● True	
erc721pausable-balanceof-correct-count	● True	
erc721pausable-balanceof-no-change-state	● True	

Detailed results for function `supportsInterface`

Property Name	Final Result	Remarks
erc721pausable-supportsinterface-correct-erc721	● True	
erc721pausable-supportsinterface-metadata	● True	
erc721pausable-supportsinterface-succeed-always	● True	
erc721pausable-supportsinterface-correct-erc165	● True	
erc721pausable-supportsinterface-correct-false	● True	
erc721pausable-supportsinterface-no-change-state	● True	

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc721pausable-transferfrom-succeed-normal	● True	
erc721pausable-transferfrom-revert-pause	● False	
erc721pausable-transferfrom-correct-increase	● True	
erc721pausable-transferfrom-correct-one-token-self	● True	
erc721pausable-transferfrom-correct-approval	● True	
erc721pausable-transferfrom-correct-owner-from	● True	
erc721pausable-transferfrom-correct-owner-to	● True	
erc721pausable-transferfrom-correct-state-balance	● True	
erc721pausable-transferfrom-correct-balance	● True	
erc721pausable-transferfrom-correct-state-owner	● True	
erc721pausable-transferfrom-correct-state-approval	● True	
erc721pausable-transferfrom-revert-invalid	● True	
erc721pausable-transferfrom-revert-from-zero	● True	
erc721pausable-transferfrom-revert-to-zero	● True	
erc721pausable-transferfrom-revert-not-owned	● True	
erc721pausable-transferfrom-revert-exceed-approval	● True	

Detailed results for function `ownerOf`

Property Name	Final Result	Remarks
erc721pausable-ownerof-succeed-normal	● True	
erc721pausable-ownerof-correct-owner	● True	
erc721pausable-ownerof-revert	● True	
erc721pausable-ownerof-no-change-state	● True	

Detailed results for function `getApproved`

Property Name	Final Result	Remarks
erc721pausable-getapproved-succeed-normal	● True	
erc721pausable-getapproved-correct-value	● True	
erc721pausable-getapproved-revert-zero	● True	
erc721pausable-getapproved-change-state	● True	

Detailed results for function `isApprovedForAll`

Property Name	Final Result	Remarks
erc721pausable-isapprovedforall-succeed-normal	● True	
erc721pausable-isapprovedforall-correct	● True	
erc721pausable-isapprovedforall-change-state	● True	

Detailed results for function `approve`

Property Name	Final Result	Remarks
erc721pausable-approve-succeed-normal	● True	
erc721pausable-approve-set-correct	● True	
erc721pausable-approve-revert-invalid-token	● True	
erc721pausable-approve-revert-not-allowed	● True	
erc721pausable-approve-change-state	● True	

Detailed results for function `setApprovalForAll`

Property Name	Final Result	Remarks
erc721pausable-setapprovalforall-succeed-normal	● True	
erc721pausable-setapprovalforall-set-correct	● True	
erc721pausable-setapprovalforall-multiple	● True	
erc721pausable-setapprovalforall-change-state	● True	

APPENDIX | MINDFUL OCEAN METAVERSE

Finding Categories

Categories	Description
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.

Checksum Calculation Method

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

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

Details on Formal Verification

Technical description

Some Solidity smart contracts from this project have been formally verified using symbolic model checking. Each such contract was compiled into a mathematical model which reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The model also formalizes a simplified execution environment of the Ethereum blockchain and a verification harness that performs the initialization of the contract and all possible interactions with the contract. Initially, the contract state is initialized non-deterministically (i.e. by arbitrary values) and over-approximates the reachable state space of the contract throughout any actual deployment on chain. All valid results thus carry over to the contract's behavior in arbitrary states after it has been deployed.

Assumptions and simplifications

The following assumptions and simplifications apply to our model:

- Gas consumption is not taken into account, i.e. we assume that executions do not terminate prematurely because they run out of gas.
- The contract's state variables are non-deterministically initialized before invocation of any of those functions. That ignores contract invariants and may lead to false positives. It is, however, a safe over-approximation.
- The verification engine reasons about unbounded integers. Machine arithmetic is modeled as operations on the congruence classes arising from the bit-width of the underlying numeric type. This ensures that over- and underflow characteristics are faithfully represented.
- Certain low-level calls and inline assembly are not supported and may lead to an ERC-20 token contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property definitions

All properties are expressed in linear temporal logic (LTL). For that matter, we treat each invocation of and each return from a public or an external function as a discrete time steps. Our analysis reasons about the contract's state upon entering and upon leaving public or external functions.

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `started(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond`.
- `willSucceed(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond` and considers only those executions that do not revert.
- `finished(f, [cond])` Indicates that execution returns from contract function `f` in a state satisfying formula `cond`. Here, formula `cond` may refer to the contract's state variables and to the value they had upon entering the function (using the `old` function).
- `reverted(f, [cond])` Indicates that execution of contract function `f` was interrupted by an exception in a contract state satisfying formula `cond`.

The verification performed in this audit operates on a harness that non-deterministically invokes a function of the contract's public or external interface. All formulas are analyzed w.r.t. the trace that corresponds to this function invocation.

Description of ERC-20 Properties

The specifications are designed such that they capture the desired and admissible behaviors of the ERC-20 functions `transfer` , `transferFrom` , `approve` , `allowance` , `balanceOf` , and `totalSupply` .

In the following, we list those property specifications.

Properties for ERC-20 function `transfer`

erc20-transfer-revert-zero

Function `transfer` Prevents Transfers to the Zero Address.

Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address.

Specification:

```
[](started(contract.transfer(to, value), to == address(0))
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return)))
```

erc20-transfer-succeed-normal

Function `transfer` Succeeds on Admissible Non-self Transfers.

All invocations of the form `transfer(recipient, amount)` must succeed and return `true` if

- the `recipient` address is not the zero address,
- `amount` does not exceed the balance of address `msg.sender` ,
- transferring `amount` to the `recipient` address does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transfer(to, value), to != address(0)
          && to != msg.sender && value >= 0 && value <= _balances[msg.sender]
          && _balances[to] + value <= type(uint256).max && _balances[to] >= 0
          && _balances[msg.sender] <= type(uint256).max)
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-succeed-self

Function `transfer` Succeeds on Admissible Self Transfers.

All self-transfers, i.e. invocations of the form `transfer(recipient, amount)` where the `recipient` address equals the address in `msg.sender` must succeed and return `true` if

- the value in `amount` does not exceed the balance of `msg.sender` and

- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transfer(to, value), to != address(0)
          && to == msg.sender && value >= 0 && value <= _balances[msg.sender]
          && _balances[msg.sender] >= 0
          && _balances[msg.sender] <= type(uint256).max)
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-correct-amount

Function `transfer` Transfers the Correct Amount in Non-self Transfers.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address.

Specification:

```
[](willSucceed(contract.transfer(to, value), to != msg.sender
   && _balances[to] >= 0 && value >= 0
   && _balances[to] + value <= type(uint256).max
   && _balances[msg.sender] >= 0 && _balances[msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transfer(to, value), return
             ==> _balances[msg.sender] == old(_balances[msg.sender]) - value
                 && _balances[to] == old(_balances[to]) + value)))
```

erc20-transfer-correct-amount-self

Function `transfer` Transfers the Correct Amount in Self Transfers.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` and where the `recipient` address equals `msg.sender` (i.e. self-transfers) must not change the balance of address `msg.sender`.

Specification:

```
[](willSucceed(contract.transfer(to, value), to == msg.sender
  && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
     ==> <>(finished(contract.transfer(to, value), return
         ==> _balances[to] == old(_balances[to]))))
```

erc20-transfer-change-state

Function `transfer` Has No Unexpected State Changes.

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must only modify the balance entries of the `msg.sender` and the `recipient` addresses.

Specification:

```
[](willSucceed(contract.transfer(to, value), p1 != msg.sender && p1 != to)
  ==> <>(finished(contract.transfer(to, value), return
      ==> (_totalSupply == old(_totalSupply) && _allowances == old(_allowances)
           && _balances[p1] == old(_balances[p1])  ))))
```

erc20-transfer-exceed-balance

Function `transfer` Fails if Requested Amount Exceeds Available Balance.

Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail.

Specification:

```
[](started(contract.transfer(to, value), value > _balances[msg.sender]
   && _balances[msg.sender] >= 0 && value <= type(uint256).max)
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return)))
```

erc20-transfer-recipient-overflow

Function `transfer` Prevents Overflows in the Recipient's Balance.

Any invocation of `transfer(recipient, amount)` must fail if it causes the balance of the `recipient` address to overflow.

Specification:

```
[](started(contract.transfer(to, value), to != msg.sender
   && _balances[to] + value > type(uint256).max
   && _balances[to] >= 0 && _balances[to] <= type(uint256).max
   && _balances[msg.sender] <= type(uint256).max
   && value > 0 && value <= _balances[msg.sender])
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return) || finished(contract.transfer(to, value), _balances[to]
                      > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transfer-false

If Function `transfer` Returns `false`, the Contract State Has Not Been Changed.

If the `transfer` function in contract `contract` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
[](willSucceed(contract.transfer(to, value))
   ==> <>(finished(contract.transfer(to, value), !return]
   ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
                                    && _allowances == old(_allowances)  ))))
```

erc20-transfer-never-return-false

Function `transfe` Never Returns `false`.

The transfer function must never return `false` to signal a failure.

Specification:

```
[](!(finished(contract.transfer, !return)))
```

Properties for ERC-20 function `transferFrom`

erc20-transferfrom-revert-from-zero

Function `transferFrom` Fails for Transfers From the Zero Address.

All calls of the form `transferFrom(from, dest, amount)` where the `from` address is zero, must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), from == address(0))
   ==> <>(reverted(contract.transferFrom) || finished(contract.transferFrom,
          !return)))
```

erc20-transferfrom-revert-to-zero

Function `transferFrom` Fails for Transfers To the Zero Address.

All calls of the form `transferFrom(from, dest, amount)` where the `dest` address is zero, must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), to == address(0))
   ==> <>(reverted(contract.transferFrom) || finished(contract.transferFrom,
          !return)))
```

erc20-transferfrom-succeed-normal

Function `transferFrom` Succeeds on Admissible Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` must succeed and return `true` if

- the value of `amount` does not exceed the balance of address `from`,

- the value of `amount` does not exceed the allowance of `msg.sender` for address `from` ,
- transferring a value of `amount` to the address in `dest` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0)
   && to != address(0) && from != to && value <= _balances[from]
   && value <= _allowances[from][msg.sender]
   && _balances[to] + value <= type(uint256).max
   && value >= 0 && _balances[to] >= 0 && _balances[from] >= 0
   && _balances[from] <= type(uint256).max
   && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-succeed-self

Function `transferFrom` Succeeds on Admissible Self Transfers.

All invocations of `transferFrom(from, dest, amount)` where the `dest` address equals the `from` address (i.e. self-transfers) must succeed and return `true` if:

- The value of `amount` does not exceed the balance of address `from` ,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from` , and
- the supplied gas suffices to complete the call.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0)
   && from == to && value <= _balances[from]
   && value <= _allowances[from][msg.sender]
   && value >= 0 && _balances[from] <= type(uint256).max
   && _allowances[from][msg.sender] <= type(uint256).max)
     ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-correct-amount

Function `transferFrom` Transfers the Correct Amount in Non-self Transfers.

All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest` .

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from != to && value >= 0
&& _balances[from] >= 0 && _balances[from] <= type(uint256).max
&& _balances[to] >= 0 && _balances[to] + value <= type(uint256).max)
   ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> _balances[from] == old(_balances[from]) - value
              && _balances[to] == old(_balances[to] + value))))
```

erc20-transferfrom-correct-amount-self

Function `transferFrom` Performs Self Transfers Correctly.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` and where the address in `from` equals the address in `dest` (i.e. self-transfers) do not change the balance entry of the `from` address (which equals `dest`).

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from == to
   && value >= 0 && value <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
             ==> _balances[from] == old(_balances[from]))))
```

erc20-transferfrom-correct-allowance

Function `transferFrom` Updated the Allowance Correctly.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`.

Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), value >= 0
   && value <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max && _balances[to] >= 0
   && _balances[to] <= type(uint256).max && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> ((_allowances[from][msg.sender]
                == old(_allowances[from][msg.sender]) - value)
                 || (_allowances[from][msg.sender]
                     == old(_allowances[from][msg.sender])
                        && (from == msg.sender
                             || old(_allowances[from][msg.sender])
                                == type(uint256).max))))))
```

erc20-transferfrom-change-state

Function `transferFrom` Has No Unexpected State Changes.

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` may only modify the following state variables:

- The balance entry for the address in `dest` ,
- The balance entry for the address in `from` ,
- The allowance for the address in `msg.sender` for the address in `from` . Specification:

```
[](willSucceed(contract.transferFrom(from, to, amount), p1 != from && p1 != to
   && (p2 != from || p3 != msg.sender))
    ==> <>(finished(contract.transferFrom(from, to, amount), return
     ==> (_totalSupply == old(_totalSupply) && _balances[p1] == old(_balances[p1])
         && _allowances[p2][p3] == old(_allowances[p2][p3])  ))))
```

erc20-transferfrom-fail-exceed-balance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Balance.

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), value > _balances[from]
   && _balances[from] >= 0 && _balances[from] <= type(uint256).max)
       ==> <>(reverted(contract.transferFrom)
              || finished(contract.transferFrom, !return)))
```

erc20-transferfrom-fail-exceed-allowance

Function `transferFrom` Fails if the Requested Amount Exceeds the Available Allowance.

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail.

Specification:

```
  [](started(contract.transferFrom(from, to, value), value > _allowances[from]
[msg.sender]
     && _allowances[from][msg.sender] >= 0 && value <= type(uint256).max)
     ==> <>(reverted(contract.transferFrom)
           || finished(contract.transferFrom(from, to, value), !return)
           || finished(contract.transferFrom(from, to, value), return
              && (msg.sender == from
                  || _allowances[from][msg.sender] == type(uint256).max))))
```

erc20-transferfrom-fail-recipient-overflow

Function `transferFrom` Prevents Overflows in the Recipient's Balance.

Any call of `transferFrom(from, dest, amount)` with a value in `amount` whose transfer would cause an overflow of the balance of address `dest` must fail.

Specification:

```
[](started(contract.transferFrom(from, to, value), from != to
   && _balances[to] + value > type(uint256).max && value <= type(uint256).max
   && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
      ==> <>(reverted(contract.transferFrom)
          || finished(contract.transferFrom(from, to, value), !return)
          || finished(contract.transferFrom(from, to, value), _balances[to]
              > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transferfrom-false

If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed.

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

```
[](willSucceed(contract.transfer(to, value))
   ==> <>(finished(contract.transfer(to, value), !return
   ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
                                    && _allowances == old(_allowances)  ))))
```

erc20-transferfrom-never-return-false

Function `transferFrom` Never Returns `false`.

The `transferFrom` function must never return `false`.

Specification:

```
[](!(finished(contract.transferFrom, !return)))
```

Properties related to function `totalSupply`

erc20-totalsupply-succeed-always

Function `totalSupply` Always Succeeds.

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
[](started(contract.totalSupply) ==> <>(finished(contract.totalSupply)))
```

erc20-totalsupply-correct-value

Function `totalSupply` Returns the Value of the Corresponding State Variable.

The `totalSupply` function must return the value that is held in the corresponding state variable of contract contract.

Specification:

```
[](willSucceed(contract.totalSupply)
   ==> <>(finished(contract.totalSupply, return == _totalSupply)))
```

erc20-totalsupply-change-state

Function `totalSupply` Does Not Change the Contract's State.

The `totalSupply` function in contract contract must not change any state variables.

Specification:

```
[](willSucceed(contract.totalSupply)
   ==> <>(finished(contract.totalSupply, _totalSupply == old(_totalSupply)
          && _balances == old(_balances) && _allowances == old(_allowances)  )))
```

Properties related to function `balanceOf`

erc20-balanceof-succeed-always

Function `balanceOf` Always Succeeds.

Function `balanceOf` must always succeed if it does not run out of gas.

Specification:

```
[](started(contract.balanceOf) ==> <>(finished(contract.balanceOf)))
```

erc20-balanceof-correct-value

Function `balanceOf` Returns the Correct Value.

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner` .

Specification:

```
[](willSucceed(contract.balanceOf)
   ==> <>(finished(contract.balanceOf(owner), return == _balances[owner])))
```

erc20-balanceof-change-state

Function `balanceOf` Does Not Change the Contract's State.

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
[](willSucceed(contract.balanceOf)
   ==> <>(finished(contract.balanceOf(owner), _totalSupply == old(_totalSupply)
          && _balances == old(_balances)
          && _allowances == old(_allowances)  )))
```

Properties related to function `allowance`

erc20-allowance-succeed-always

Function `allowance` Always Succeeds.

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
[](started(contract.allowance) ==> <>(finished(contract.allowance)))
```

erc20-allowance-correct-value

Function `allowance` Returns Correct Value.

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
[](willSucceed(contract.allowance(owner, spender))
   ==> <>(finished(contract.allowance(owner, spender),
           return == _allowances[owner][spender])))
```

erc20-allowance-change-state

Function `allowance` Does Not Change the Contract's State.

Function `allowance` must not change any of the contract's state variables.

Specification:

```
[](willSucceed(contract.allowance(owner, spender))
   ==> <>(finished(contract.allowance(owner, spender),
   _totalSupply == old(_totalSupply) && _balances == old(_balances)
                && _allowances == old(_allowances)  )))
```

Properties related to function `approve`

erc20-approve-revert-zero

Function `approve` Prevents Giving Approvals For the Zero Address.

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
[](started(contract.approve(spender, value), spender == address(0))
   ==> <>(reverted(contract.approve)
         || finished(contract.approve(spender, value), !return)))
```

erc20-approve-succeed-normal

Function `approve` Succeeds for Admissible Inputs.

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
[](started(contract.approve(spender, value), spender != address(0))
   ==> <>(finished(contract.approve(spender, value), return)))
```

erc20-approve-correct-amount

Function `approve` Updates the Approval Mapping Correctly.

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0)
   && value >= 0 && value <= type(uint256).max)
       ==> <>(finished(contract.approve(spender, value), return
              ==> _allowances[msg.sender][spender] == value)))
```

erc20-approve-change-state

Function `approve` Has No Unexpected State Changes.

All calls of the form `approve(spender, amount)` must only update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount` and incur no other state changes.

Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0)
   && (p1 != msg.sender || p2 != spender))
       ==> <>(finished(contract.approve(spender, value), return
              ==> _totalSupply == old(_totalSupply) && _balances == old(_balances)
                  && _allowances[p1][p2] == old(_allowances[p1][p2])  )))
```

erc20-approve-false

If Function `approve` Returns `false`, the Contract's State Has Not Been Changed.

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
[](willSucceed(contract.approve(spender, value))
   ==> <>(finished(contract.approve(spender, value), !return
          ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
               && _allowances == old(_allowances)  ))))
```

erc20-approve-never-return-false

Function `approve` Never Returns `false`.

The function `approve` must never returns `false`.

Specification:

```
 [](!(finished(contract.approve, !return)))
```

DISCLAIMER | CERTIK

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

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

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

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

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

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

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

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

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

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

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

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

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

CertiK | **Securing** the **Web3** World

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