⚠ Draft Standards Track: ERC

ERC-6551: Non-fungible Token Bound Accounts

An interface and registry for smart contract accounts owned by ERC-721 tokens

Authors	Jayden Windle (@jaydenwindle), Benny Giang <bg@futureprimitive.xyz>, Steve Jang, Druzy Downs (@druzydowns), Raymond Huynh (@huynhr), Alanah Lam <alanah@futureprimitive.xyz>, Wilkins Chung (@wwhchung) <wilkins@manifold.xyz>, Paul Sullivan (@sullivph) <paul.sullivan@manifold.xyz>

Created	2023-02-23
Discussion Link	https://ethereum-magicians.org/t/non-fungible-token-bound-accounts/13030
Requires	EIP-155, EIP-165, EIP-721, EIP-1167, EIP-1271

Table of Contents

- Abstract
- Motivation
- Specification
  - Overview
  - Registry
  - Account Interface
- Rationale
  - Counterfactual Account Addresses
  - Account Ambiguity
  - Proxy Implementation
  - EIP-155 Support
- Backwards Compatibility
- Reference Implementation
  - Example Account Implementation
  - Registry Implementation
- Security Considerations
  - Fraud Prevention

    - Ownership Cycles
- Copyright

Abstract

This proposal defines a system which gives every ERC-721 token a smart contract account. These token bound accounts allow ERC-721 tokens to own assets and interact with applications, without requiring changes to existing ERC-721 smart contracts or infrastructure.

Motivation

The ERC-721 standard enabled an explosion of non-fungible token applications. Some notable use cases have included breedable cats, generative artwork, and liquidity positions.

Non-fungible tokens are increasingly becoming a form of on-chain identity. This follows quite naturally from the ERC-721 specification - each non-fungible token has a globally unique identifier, and by extension, a unique identity.

Unlike other forms of on-chain identity, ERC-721 tokens cannot act as an agent or associate with other on-chain assets. This limitation stands in contrast with many real-world instances of non-fungible assets. For example:

- A character in a role-playing game that accumulates assets and abilities over time based on actions they have taken
- An automobile composed of many fungible and non-fungible components
- An automated investment portfolio composed of multiple fungible assets
- A punch pass membership card granting access to an establishment and recording a history of past interactions

Several proposals have attempted to give ERC-721 tokens the ability to own assets. Each of these proposals have defined an extension to the ERC-721 standard. This requires smart contract authors to include proposal support in their ERC-721 token contracts. As a result, these proposals are largely incompatible with previously deployed ERC-721 contracts.

This proposal grants every ERC-721 token the full capabilities of an Ethereum account while maintaining backwards compatibility with previously deployed ERC-721 token contracts. It does so by deploying unique, deterministically-addressed smart contract accounts for each ERC-721 token via a permissionless registry.

Each token bound account is owned by a single ERC-721 token, allowing the token to interact with the blockchain, record transaction history, and own on-chain assets. Control of each token bound account is delegated to the owner of the ERC-721 token, allowing the owner to initiate on-chain actions on behalf of their token.

Token bound accounts are compatible out of the box with nearly all existing infrastructure that supports Ethereum accounts, from on-chain protocols to off-chain indexers. Token bound accounts can own any type of on-chain asset, and can be extended to support new asset types created in the future.

Specification

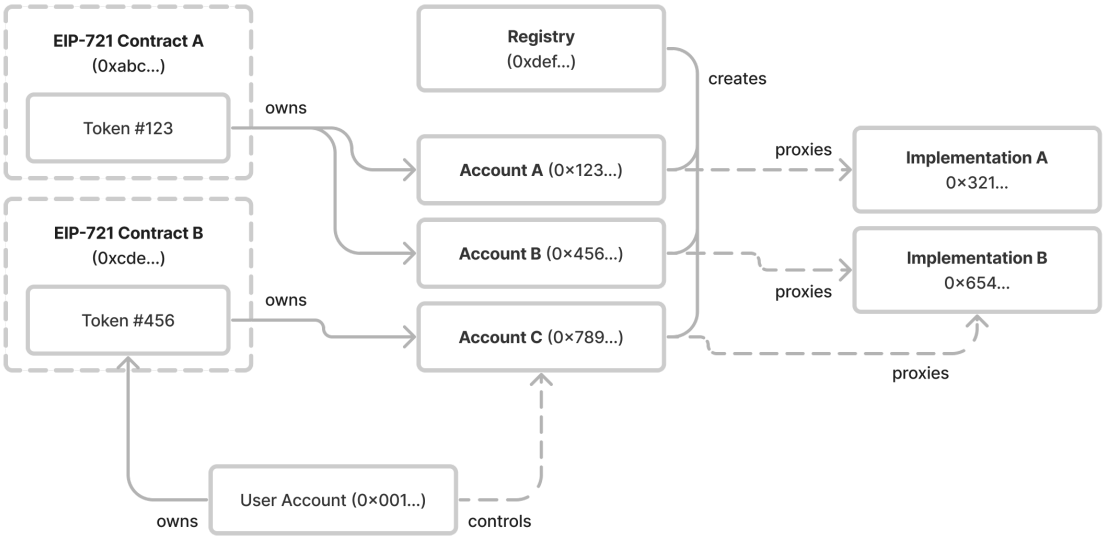
The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 and RFC 8174.

Overview

The system outlined in this proposal has two main components:

- A permissionless registry for deploying token bound accounts
- A standard interface for token bound account implementations

The following diagram illustrates the relationship between ERC-721 tokens, ERC-721 token owners, token bound accounts, and the Registry:



Registry

The registry serves as a single entry point for projects wishing to utilize token bound accounts. It has two functions:

- `createAccount` - deploys a token bound account for an ERC-721 token given an `implementation` address
- `account` - a read-only function that computes the token bound account address for an ERC-721 token given an `implementation` address

The registry SHALL deploy each token bound account as an ERC-1167 minimal proxy with immutable constant data appended to the bytecode.

The the deployed bytecode of each token bound account SHALL have the following structure:

```
ERC-1167 Header               (10 bytes)
<implementation (address)>    (20 bytes)
ERC-1167 Footer               (15 bytes)
<salt (uint256)>              (32 bytes)
<chainId (uint256)>           (32 bytes)
<tokenContract (address)>     (32 bytes)
<tokenId (uint256)>           (32 bytes)
```

For example, the token bound account with implementation address `0xbebebebebebebebebebebebebebebebebebebebe`, salt `0`, chain ID `1`, token contract `0xcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcf` and token ID `123` would have the following deployed bytecode:

```
363d3d373d3d3d363d73bebebebebebebebebebebebebebebebebebebebebebebebebebebebebe5af43d82803e903d916(
```

Each token bound account proxy SHALL delegate execution to a contract that implements the `IERC6551Account` interface.

The registry contract is permissionless, immutable, and has no owner. The complete source code for the registry can be found in the Registry Implementation section below. The registry SHALL be deployed at address `TBD` using Nick's Factory (`0x4e59b44847b379578588920cA78FbF26c0B4956C`) with salt `0x6551655165516551655165516551655165516551655165516551655165516551`.

The registry SHALL deploy all token bound account contracts using the `create2` opcode so that the account address for every ERC-721 token is deterministic. The account address for each ERC-721 token SHALL be derived from the unique combination of implementation address, token contract address, token ID, EIP-155 chain ID, and an optional salt.

The registry SHALL implement the following interface:

```
interface IERC6551Registry {
    /// @dev The registry SHALL emit the AccountCreated event upon successful a
```

```
    event AccountCreated(
        address account,
        address implementation,
        uint256 chainId,
        address tokenContract,
        uint256 tokenId,
        uint256 salt
    );

    /// @dev Creates a token bound account for an ERC-721 token.
    ///
    /// If account has already been created, returns the account address without
    ///
    /// If initData is not empty and account has not yet been created, calls acc
    /// provided initData after creation.
    ///
    /// Emits AccountCreated event.
    ///
    /// @return the address of the account
    function createAccount(
        address implementation,
        uint256 chainId,
        address tokenContract,
        uint256 tokenId,
        uint256 salt,
        bytes calldata initData
    ) external returns (address);

    /// @dev Returns the computed address of a token bound account
    ///
    /// @return The computed address of the account
    function account(
        address implementation,
        uint256 chainId,
        address tokenContract,
        uint256 tokenId,
        uint256 salt
    ) external view returns (address);
}
```

Account Interface

All token bound accounts SHOULD be created via the registry.

All token bound account implementations MUST implement ERC-165 interface detection.

All token bound account implementations MUST implement ERC-1271 signature validation.

All token bound account implementations MUST implement the following interface:

```
/// @dev the ERC-165 identifier for this interface is `0x400a0398`
interface IERC6551Account {
    /// @dev Token bound accounts MUST implement a `receive` function.
    ///
    /// Token bound accounts MAY perform arbitrary logic to restrict conditions
    /// under which Ether can be received.
    receive() external payable;

    /// @dev Executes `call` on address `to`, with value `value` and calldata
    /// `data`.
    ///
    /// MUST revert and bubble up errors if call fails.
    ///
    /// By default, token bound accounts MUST allow the owner of the ERC-721 to
    /// which owns the account to execute arbitrary calls using `executeCall`.
    ///
    /// Token bound accounts MAY implement additional authorization mechanisms
    /// which limit the ability of the ERC-721 token holder to execute calls.
    ///
    /// Token bound accounts MAY implement additional execution functions which
    /// grant execution permissions to other non-owner accounts.
    ///
    /// @return The result of the call
    function executeCall(
        address to,
        uint256 value,
        bytes calldata data
    ) external payable returns (bytes memory);

    /// @dev Returns identifier of the ERC-721 token which owns the
    /// account
    ///
    /// The return value of this function MUST be constant - it MUST NOT change
    /// over time.
    ///
    /// @return chainId The EIP-155 ID of the chain the ERC-721 token exists on
    /// @return tokenContract The contract address of the ERC-721 token
    /// @return tokenId The ID of the ERC-721 token
    function token()
        external
        view
        returns (
            uint256 chainId,
            address tokenContract,
```

```
            uint256 tokenId
        );

    /// @dev Returns the owner of the ERC-721 token which controls the account
    /// if the token exists.
    ///
    /// This is value is obtained by calling `ownerOf` on the ERC-721 contract.
    ///
    /// @return Address of the owner of the ERC-721 token which owns the accoun
    function owner() external view returns (address);

    /// @dev Returns a nonce value that is updated on every successful transact
    ///
    /// @return The current account nonce
    function nonce() external view returns (uint256);
}
```

Rationale

Counterfactual Account Addresses

By specifying a canonical account registry, applications wishing to support this proposal can compute the address for a given token bound account using a certain implementation prior to the deployment of the contract for that account. This allows assets to be sent securely to the owner of a token without needing to know the owner's address. A canonical registry also allows client side applications to query for assets owned by a token from a single entry point.

Account Ambiguity

The specification proposed above allows ERC-721 tokens to have multiple token bound accounts, one per implementation address. During the development of this proposal, alternative architectures were considered which would have assigned a single token bound account to each ERC-721 token, making each token bound account address an unambiguous identifier.

However, these alternatives present several trade offs.

First, due to the permissionless nature of smart contracts, it is impossible to enforce a limit of one token bound account per ERC-721 token. Anyone wishing to utilize multiple token bound accounts per ERC-721 token could do so by deploying an additional registry contract.

Second, limiting each ERC-721 token to a single token bound account would require a static, trusted account implementation to be included in this proposal. This implementation would inevitably impose specific constraints on the capabilities of token bound accounts. Given the number of unexplored use cases this proposal enables and the benefit that diverse account

implementations could bring to the non-fungible token ecosystem, it is the authors' opinion that defining a canonical and constrained implementation in this proposal is premature.

Finally, this proposal seeks to grant ERC-721 tokens the ability to act as agents on-chain. In current practice, on-chain agents often utilize multiple accounts. A common example is individuals who use a "hot" account for daily use and a "cold" account for storing valuables. If on-chain agents commonly use multiple accounts, it stands to reason that ERC-721 tokens ought to inherit the same ability.

Proxy Implementation

ERC-1167 minimal proxies are well supported by existing infrastructure and are a common smart contract pattern. This proposal deploys each token bound account using a custom ERC-1167 proxy implementation that stores the salt, implementation address, chain id, token contract address, and token ID as ABI-encoded constant data appended to the contract bytecode. This allows token bound account implementations to easily query this data while ensuring it remains constant. This approach was taken to maximize compatibility with existing infrastructure while also giving smart contract developers full flexibility when creating custom token bound account implementations.

EIP-155 Support

This proposal uses EIP-155 chain IDs to identify ERC-721 tokens along with their contract address and token ID. ERC-721 token identifiers are globally unique on a single Ethereum chain, but may not be unique across multiple Ethereum chains. Using chain IDs to uniquely identify ERC-721 tokens allows smart contract authors wishing to implement this proposal to optionally support multi-chain token bound accounts.

Backwards Compatibility

This proposal seeks to be maximally backwards compatible with existing non-fungible token contracts. As such, it does not extend the ERC-721 standard.

Additionally, this proposal does not require the registry to perform an ERC-165 interface check for ERC-721 compatibility prior to account creation. This is by design in order to maximize backwards compatibility with non-fungible token contracts that pre-date the ERC-721 standard, such as Cryptokitties. Smart contract authors implementing this proposal may optionally choose to enforce interface detection for ERC-721.

Non-fungible token contracts that do not implement an `ownerOf` method, such as Cryptopunks, are not compatible with this proposal. The system outlined in this proposal could be used to support such collections with minor modifications, but that is outside the scope of this proposal.

Reference Implementation

Example Account Implementation

```
pragma solidity ^0.8.13;

import "openzeppelin-contracts/utils/introspection/IERC165.sol";
import "openzeppelin-contracts/token/ERC721/IERC721.sol";
import "openzeppelin-contracts/interfaces/IERC1271.sol";
import "openzeppelin-contracts/utils/cryptography/SignatureChecker.sol";
import "sstore2/utils/Bytecode.sol";

contract ExampleERC6551Account is IERC165, IERC1271, IERC6551Account {
    receive() external payable {}

    function executeCall(
        address to,
        uint256 value,
        bytes calldata data
    ) external payable returns (bytes memory result) {
        require(msg.sender == owner(), "Not token owner");

        bool success;
        (success, result) = to.call{value: value}(data);

        if (!success) {
            assembly {
                revert(add(result, 32), mload(result))
            }
        }
    }

    function token()
        external
        view
        returns (
            uint256 chainId,
            address tokenContract,
            uint256 tokenId
        )
    {
        uint256 length = address(this).code.length
        return
            abi.decode(
                Bytecode.codeAt(address(this), length - 0x60, length),
                (uint256, address, uint256)
```

```
            );
    }

    function owner() public view returns (address) {
        (uint256 chainId, address tokenContract, uint256 tokenId) = this
            .token();
        if (chainId != block.chainid) return address(0);

        return IERC721(tokenContract).ownerOf(tokenId);
    }

    function supportsInterface(bytes4 interfaceId) public pure returns (bool) {
        return (interfaceId == type(IERC165).interfaceId ||
            interfaceId == type(IERC6551Account).interfaceId);
    }

    function isValidSignature(bytes32 hash, bytes memory signature)
        external
        view
        returns (bytes4 magicValue)
    {
        bool isValid = SignatureChecker.isValidSignatureNow(
            owner(),
            hash,
            signature
        );

        if (isValid) {
            return IERC1271.isValidSignature.selector;
        }

        return "";
    }
}
```

Registry Implementation

```
pragma solidity ^0.8.13;

import "openzeppelin-contracts/utils/Create2.sol";

contract ERC6551Registry is IERC6551Registry {
    error InitializationFailed();

    function createAccount(
        address implementation,
        uint256 chainId,
```

```
        address tokenContract,
        uint256 tokenId,
        uint256 salt,
        bytes calldata initData
    ) external returns (address) {
        bytes memory code = _creationCode(implementation, chainId, tokenContrac

        address _account = Create2.computeAddress(
            bytes32(salt),
            keccak256(code)
        );

        if (_account.code.length != 0) return _account;

        _account = Create2.deploy(0, bytes32(salt), code);

        if (initData.length != 0) {
            (bool success, ) = _account.call(initData);
            if (!success) revert InitializationFailed();
        }

        emit AccountCreated(
            _account,
            implementation,
            chainId,
            tokenContract,
            tokenId,
            salt
        );

        return _account;
    }

    function account(
        address implementation,
        uint256 chainId,
        address tokenContract,
        uint256 tokenId,
        uint256 salt
    ) external view returns (address) {
        bytes32 bytecodeHash = keccak256(
            _creationCode(implementation, chainId, tokenContract, tokenId, salt
        );

        return Create2.computeAddress(bytes32(salt), bytecodeHash);
    }
```

```
    function _creationCode(
        address implementation_,
        uint256 chainId_,
        address tokenContract_,
        uint256 tokenId_,
        uint256 salt_
    ) internal pure returns (bytes memory) {
        return
            abi.encodePacked(
                hex"3d60ad80600a3d3981f3363d3d373d3d3d363d73",
                implementation_,
                hex"5af43d82803e903d91602b57fd5bf3",
                abi.encode(salt_, chainId_, tokenContract_, tokenId_)
            );
    }
}
```

Security Considerations

Fraud Prevention

In order to enable trustless sales of token bound accounts, decentralized marketplaces will need to implement safeguards against fraudulent behavior by malicious account owners.

Consider the following potential scam:

- Alice owns an ERC-721 token X, which owns token bound account Y.
- Alice deposits 10ETH into account Y
- Bob offers to purchase token X for 11ETH via a decentralized marketplace, assuming he will receive the 10ETH stored in account Y along with the token
- Alice withdraws 10ETH from the token bound account, and immediately accepts Bob's offer
- Bob receives token X, but account Y is empty

To mitigate fraudulent behavior by malicious account owners, decentralized marketplaces SHOULD implement protection against these sorts of scams at the marketplace level. Contracts which implement this EIP MAY also implement certain protections against fraudulent behavior.

Here are a few mitigations strategies to be considered:

- Attach the current token bound account nonce to the marketplace order. If the nonce of the account has changed since the order was placed, consider the offer void. This functionality would need to be supported at the marketplace level.

- Attach a list of asset commitments to the marketplace order that are expected to remain in the token bound account when the order is fulfilled. If any of the committed assets have been removed from the account since the order was placed, consider the offer void. This would also need to be implemented by the marketplace.
- Submit the order to the decentralized market via an external smart contract which performs the above logic before validating the order signature. This allows for safe transfers to be implemented without marketplace support.
- Implement a locking mechanism on the token bound account implementation that prevents malicious owners from extracting assets from the account while locked

Preventing fraud is outside the scope of this proposal.

Ownership Cycles

All assets held in an token bound account may be rendered inaccessible if an ownership cycle is created. The simplest example is the case of an ERC-721 token being transferred to it's own token bound account. If this occurs, both the ERC-721 token and all of the assets stored in the token bound account would be permanently inaccessible, since the token bound account is incapable of executing a transaction which transfers the ERC-721 token.

Ownership cycles can be introduced in any graph of n>0 token bound accounts. On-chain prevention of these cycles is difficult to enforce given the infinite search space required, and as such is outside the scope of this proposal. Application clients and account implementations wishing to adopt this proposal are encouraged to implement measures that limit the possibility of ownership cycles.

Copyright

Citation

Please cite this document as:

Jayden Windle (@jaydenwindle), Benny Giang <bg@futureprimitive.xyz>, Steve Jang, Druzy Downs (@druzydowns), Raymond Huynh (@huynhr), Alanah Lam <alanah@futureprimitive.xyz>, Wilkins Chung (@wwhchung) <wilkins@manifold.xyz>, Paul Sullivan (@sullivph) <paul.sullivan@manifold.xyz>, "ERC-6551: Non-fungible Token Bound Accounts [DRAFT]," *Ethereum Improvement Proposals*, no. 6551, February 2023. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-6551.