Gryfyn DG Account Protocol Audit Report

by McToady

mctoady.dev

March 12, 2024

Gryfyn DG Account Protocol Audit Report

McToady

February 2nd, 2024

Prepared by: Mctoady

Table of Contents

Protocol Summary

The `dg-contract-factory` codebase is an ERC6551 (Non-fungible Token Bound Accounts) implementation that can be split into two parts:

The first part is related to the creation of ERC721 tokens (`DGTokenFactory.sol` and `DGIdentityTokenUpgradeable.sol`).

The second part is made up of the smart contract wallet implementation that the ERC721 tokens are linked to `DGAccountProxy.sol`, `DGAccountGuardian.sol` and `DGAccountUpgradeable.sol`.

The protocol team can mint it's users ERC721 tokens that use an external `ERC6551Registry` contract to deploy a smart contract wallet that will be linked to it's token id. The smart contract wallet implementation allows the tokens owner to execute calls via the smart contract wallet whilst also adopting the ERC4337 (Account Abstraction).

Disclaimer

All effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

The following 9 contracts were considered within the scope of the review.

```
src/tba/interfaces/IDGAccountGuardian.sol
src/tba/interfaces/IDGAccountProxy.sol
src/tba/interfaces/IDGAccountUpgradeable.sol
src/tba/DGAccountProxy.sol
src/tba/DGAccountGuardian.sol
src/tba/DGAccountUpgradeable.sol

src/tokens/interfaces/IDGTokenUpgradeable.sol
src/tokens/DGTokenFactory.sol
src/tokens/DGIdentityTokenUpgradeable.sol
```

Roles

The protocol has the following roles:

User

- Can Transfer/Trade their ERC721 `DGIdentityUpgradeable` token.
- Can sign ERC4337 transactions for `DGAccountUpgradeable` to be processed via `EntryPoint`.
- Can execute transactions for `DGAccountUpgradeable`.
- Can upgrade their implementation of `DGAccountUpgradeable` to one of the other trusted implementations.

Owner of `DGTokenFactory`

- Can create new `IDGTokenUpgradeable` instances.

Owner of `DGIdentityTokenUpgradeable`

- Can mint users tokens.
- Can transfer users tokens.
- Can set the token's baseURI
- Can refresh the metadata of one/many tokenIds

Owner of `DGAccountGuardian`

- Can set trusted implementations of `DGAccountUpgradeable`
- Can set a trusted signer
- Can set trusted executors

Trusted Signer

- Can sign ERC4337 transactions on behalf of a `DGAccountUpgradeable` owner.

Trusted Executor - Can execute transactions on behalf of a `DGAccountUpgradeable` owner.

Executive Summary

The following is high level overview of protocol's architecture, systematic risks the protocol should be aware of and other general advice for the protocol team.

Contract Overview

DGTokenFactory A simple factory implementation uses OpenZeppelin's Clones contract to clone an existing contract to deploy an ERC721 token.

DGIdentityTokenUpgradeable A simple upgradeable ERC721 token implementation that allows the protocol team to mint ERC721 tokens to users. The `mint` function also deploys the tokenId's related ERC6551 compliant smart contract account.

DGAccountGuardian A 'guardian' contract that can control which smart contract account logic implementations are 'trusted' for users to upgrade to. The guardian contract also has powers to assign addresses which can execute transactions on behalf of users as well as addresses that can be the signer for an accounts ERC4337 transactions.

DGAccountProxy A simple ERC1967Proxy contract. The contract is responsible for delegating calls to the smart contract account logic contract (here `DGAccountUpgradeable`).

DGAccountUpgradeable The logic implementation for the smart contract account which will be the default implementation that `DGAccountProxy` delegates it's calls to. The contract allows the owner of the paired ERC721 token to execute transactions via the smart contract wallet by calling `execute`. It also allows the owner to give permission to other accounts to execute transactions via this account by calling `setPermissions`. The owner will also be a valid signer for any ERC4337 transactions made by the account. Finally the owner is able to upgrade to another trusted implementation of the the smart contract account logic by calling `upgrade`.

Potential Risks

Protocol maintains a large amount of control over the project

There are a number of issues that users should be aware of when interacting with the protocol:

- The protocol team has power to execute transactions on behalf of users in `DGAccountUpgradeable` .
- The protocol team have the power to upgrade a users implementation of `DGAccountGuardian` .

- Users can only upgrade their account implementation to one whitelisted by `DGAccountGuardian` which is controlled by the protocol's team.
- `DGIdentityTokenUpgradable` has a function `adminTransfer` which allows the protocol team to transfer anyones token.

Even with the assumption the protocol team will not act in a malicious manner it is very important that they maintain very stringent security practices to ensure that access to the `owner` address by hostile parties is not possible. Such practices include but are not limited to: - Making the `owner` address a multisig that would require multiple private keys to be exposed for an attacker to take control. - Securely storing all necessary private keys somewhere accesible to the minimum necessary individuals. - Splitting admin powers between multiple "roles" to limit the damage an attacker could do if they gain access. - Having a contingency plan to quickly and safely generate a new `owner` private key and migrate all contracts "owned" by the current `owner` to this new address. See OpenZeppelin AccessControl

ERC4337 is still in its early phase of adoption

The ERC4337 contracts this project interacts with are still being actively worked on and regularly updated, therefore it's important to remain cautious until the related contracts have been thoroughly battle tested. While the contents of the ERC4337 contracts were out of the scope of this security review, lengths were taken to ensure that this projects implementation interacted with the ERC4337 contract as they should. However there is no guarantee that a future issue could be found within the ERC4337 contracts that could have an knock on effect to the functionality and safety of this project's contracts.

Architecture Comments

Documentation

The codebase is relatively well commented, especially when it comes to explaining things that may at first be unclear, it would be preferable to have full NatSpec comments to make things easier for interested users or projects looking to integrate with the codebase.

As well as this a more detailed README could be added that gives a brief overview of all the contracts, how they interact with each other and some general useage diagrams.

Testing

The codebase's test suite is relatively mature and covers most the expected use cases and the codes integration with things such as ERC4337 & ERC6551. As some of these integrations can become quite complicated it might be beneficial to set up an invariant test suite so various call combinations can be tested simultaneously rather than having to write individual tests for specific potential edge cases. RareSkills Foundry Invariant Testing

Function/Variable Naming

Overall the codebase does a very good job giving variables and functions clear and concise names where their useage is typically very clear on first read. One exception is that some of the contract's custom error names might fail to give users a clear understanding of what their problem is. For example in `DGAccountUpgradeable::setPermissions` the error `InvalidInput` is used when the two arrays given as arguments are not of the same length. Naming the error something like `MismatchedLengths` would make it clearer to any user who received this error what their problem was.

Overall Code Style

Overall the code is written and formatted very well. The functions are all lean and avoid doing anything unnecessary or anything you wouldn't expect based on the functions name alone.

Issues Found

The following issues were found during the security review of the project.

Severity	#
High	0
Medium	1
Low/Info	12
Gas	3

Findings

Medium

[M-01] Incrementing `EntryPoint` nonce not necessary for non-ERC4337 transactions

As the nonce is primarily to avoid potential double spends from signatures being reused, it is not necessary to increment it for calls that don't require signatures to ensure they're not replayed. The documentation suggests that the functional is only provided to allow the option to start an accounts nonce at 1 to absorb the larger gas cost of incrementing from 0 -> 1.

See the following exerpt from `INonceManager.sol`:

```
    /**
     * Manually increment the nonce of the sender.
     * This method is exposed just for completeness..
     * Account does NOT need to call it, neither during validation, nor
         elsewhere,
     * as the EntryPoint will update the nonce regardless.
     * Possible use-case is call it with various keys to "initialize"
        their nonces to one, so that future
     * UserOperations will not pay extra for the first transaction with
         a given key.
     */
    function incrementNonce(uint192 key) external;
```

Impact As the ERC4337 specification suggests not manipulating the nonce except to absorb the initial gas price moving from 0 to 1 it's recommended that the protocol sticks to this recommendation to be as compatible with other protocols as possible.

To add to this it could also hamper user experience if they sign an ERC4337 transaction with their current nonce and then `execute` a transaction with their account before the first transaction goes through it will revert as the accounts nonce will have changed.

Finally incremementing the nonce on each call to `execute` or `setPermissions` uses a large amount of gas, see following differences after removing `_incrementNonce` from `DGAccountUpgradeable::execute` and `DGAccountUpgradeable::setPermissions`
:

Before:

```
| function name  | min  | avg   | median | max   | # calls
| execute        | 4371 | 22196 | 11708  | 66547 | 40
| setPermissions | 2897 | 23637 | 5648   | 53359 | 5
```

After:

```
| function name  | min  | avg   | median | max   | # calls
| execute        | 4371 | 18178 | 10871  | 41220 | 40
| setPermissions | 2897 | 13196 | 4676   | 27744 | 5
```

Low and Informational

[I-01] Currently possible to 'downgrade' implementation when calling `DGAccountUpgradeable:upgrade`

Assuming the current test setup will be replicated on launch, the `defaultImplmentation` will return false if calling `DGAccountGuardian.isTrustedImplementation(defaultImplementation)`. This means it's NOT possible to go from V1 -> V2 -> V1. However if V2 & V3 are trusted implementations it would be possible for a user to go from V2 -> V3 -> V2. Giving users the ability to 'downgrade' could mean there are values in storage that become unreachable when reverting to a previous version which may unexpectedly be accessed again when upgrading to a future version. Therefore it's recommended that users are only allowed to upgrade to a version number greater than their current one. Consider the following change in `DGAccountUpgradeable::_authorizeUpgrade`:

```
-    if (ERC1967Utils.getImplementation() == newImplementation) revert
    SameImplementation();
+    if (version() >= IDGAccountUpgradeable(newImplementation).version()
    ) revertOnlyUpgrade();
```

[I-02] DGAccountGuardian function has no zero address checks

Upon review of possible downsides should the owner of `DGAccountGuardian` accidentally set the zero address as an owner it appears that the only function presenting potential danger is `setTrustedImplementation` where if the zero address became a trusted implementation, users could brick their `DGAccountUpgradeable` accounts by changing the implementation to address zero, meaning they would no longer be able to call functions on the proxy contract other than the ones exposed in `DGAccountProxy`.

It should also be noted there are limited potential issues entering the zero address in `setTrustedExecutor` or `setTrustedSigner`.

[I-03] DGAccountUpgradeable::onlyOwner modifier is currently unused

The contract `DGAccountUpgradeable` declares an `onlyOwner` modifier but does not use it in the current implementation.

[I-04] Make bytes4 used in DGAccountUpgradeable::upgrade a constant state variable for better clarity

See snippet below:

```
    if (ERC1967Utils.getImplementation() == address(0)) {
            // if upgrading for the first time, "initialize()" first
            (success, result) = address(this).call(abi.encodePacked(
               bytes4(0x8129fc1c)));
            if (!success) revert UpgradeFailed();
        }
```

A constant state variable will compiled down to the same bytecode as the current implementation but will make it clearer to someone reading the code. Example:

```
    bytes4 constant INITIALIZE_SELECTOR = 0x8129fc1c;

    // snip

    function upgrade(address newImplementation) public virtual
        onlyAuthorized {
        // snip
        if (ERC1967Utils.getImplementation() == address(0)) {
            // if upgrading for the first time, "initialize()" first
            (success, result) = address(this).call(abi.encodePacked(
               INITIALIZE_SELECTOR));
            if (!success) revert UpgradeFailed();
        }
```

[I-05] DGAccountProxy::initialize can be called more than once

While in this instance repeated calls to `initialize` have no ill effects it's recommended to have a check that ensures your initializer can only be called once.

In this example if some future upgrade allowed implementation address to become `address(0)` someone would be able to call `initialize` on the proxy contract and reset the implementation address to `defaultImplementation` and it's unclear to the user that this is a possibility.

Consider using the OpenZeppelin initializer modifier or other means of ensuring `initialize` is called once on creation and not again after.

[I-06] DGAccountProxy::recieve function unnecessary

The code for contract `DGAccountProxy` has a payable receive function. This is not necessary as the contract inherits `ERC1967Proxy` which in turn inherits `Proxy` which implements it's own fallback function that is responsible of delegating the calls to this contract to the implementation address.

[I-07] Events should be emitted for state changing functions

All meaningful changes of state should have a related event emitted to make it easier for offchain indexers to keep track of the codes current (and previous) state.

Examples

- DGIdentityTokenUpgradeable::setBaseURI should emit a BaseURISet event
- DGTokenFactory::createNewCollection should emit a CollectionCreated event
- DGAccountGuardian::constructor should emit TrustedSignerUpdated & TrustedExecutorUpdated events

[I-08] Use preferred style for unchecked incrementing of iterator in for loops

The commonly accepted style for incrementing the iterator in a for loop is to isolate the `unchecked` block to only the iterator, even if the there are no other arithmetic operators inside the block.

Examples:

DGIdentityTokenUpgradeable::refreshMetadata:

```
    function refreshMetadata(uint256[] calldata _tokenIds) external
        onlyOwner {
        uint256 len = _tokenIds.length;
        unchecked {
            for (uint256 i; i < len; ++i) {
                if (_ownerOf(_tokenIds[i]) != address(0)) {
                    emit MetadataUpdate(_tokenIds[i]);
                }
            }
        }
    }
```

DGAccountUpgradeable::setPermissions:

```
    function setPermissions(
        address[] calldata callers,
        bool[] calldata _permissions
    ) external onlyOwner {
        uint256 length = callers.length;

        if (_permissions.length != length) revert InvalidInput();

        unchecked {
            for (uint256 i; i < length; ++i) {
                permissions[msg.sender][callers[i]] = _permissions[i];
                emit PermissionUpdated(msg.sender, callers[i],
                    _permissions[i]);
            }
        }

        _incrementNonce();
    }
```

Preferred style:

```
    for (uint256 i; i < length; ) {
        // loop logic
        unchecked {
            ++i;
        }
    }
```

[I-09] Name key, value pairs in mappings to improve code readability.

As of solidity version `0.8.18` it's possible to name the keys and values of mappings. This gets compiled down to the same bytecode so incurs no extra gas costs but improves the code's readability so is encouraged.

Cases: `DGAccountUpgradeable::permissions`
`DGAccountGuardian::isTrustedImplementation`

Example:

```
-    mapping(address => mapping(address => bool)) public permissions;
+    mapping(address owner => mapping(address caller => bool permitted)
    ) public permissions;
```

[I-10] Inconsistent way of returning 0 between `DGAccountUpgradeable::isValidSignature` and `DGAccountUpgradeable::isValidSigner`

Both `isValidSignature` and `isValidSigner` return bytes4 where anything except the `magicValue` is considered false.

See code below for inconsistency:

```

    function isValidSignature(
        bytes32 hash,
        bytes memory signature
    ) external view returns (bytes4 magicValue) {

        // snip

        return "";
    }


    function isValidSigner(
        address signer,
        bytes calldata context
    ) external view returns (bytes4 magicValue) {

        // snip

        return bytes4(0);
    }
```

It's recommended to keep the two methods for signifying zero the same for the sake of code clarity.

[I-11] Consider storing an array of created collections in DGTokenFactory

Currently the contract `DGTokenFactory` has no easy way on chain of knowing the tokens that have been created using this factory contract. This could be done by pushing the deployed address to an array during `createNewCollection`. As the use of `createNewCollection` has `onlyOwner` access control it seems that theintention is that the deployed contracts will only be created by team itself, so it could be a useful means of provenance to have it's deployed addresses stored within the contract.

[I-12] Add all public/external functions to the contract interfaces as well as the necessary documentation

To provide a more complete documentation for users as well as other projects that may want to integrate with the protocol it would be beneficial if the project had more complete interfaces which expose all the of their related contracts external functionality. It's also recommended to add the contracts NatSpec comments in the interfaces and then use `@inheritdoc` in the contracts themselves. See NatSpec

Gas

[G-01] Make use of `DGAccountUpgradeable::onlyOwner` modifier for `DGAccountUpgradeable::setPermissions` for a small gas saving

See the following snippet below:

```

    function setPermissions(
        address[] calldata callers,
        bool[] calldata _permissions
    ) external {
        address _owner = owner();
        if (msg.sender != _owner) revert NotAuthorized();

        ...
    }
```

This check is the same as would be performed by using the `onlyOwner` modifier. Then once you have confirmed `msg.sender == owner()` it's safe to replace the following lines:

```
permissions[_owner][callers[i]] = _permissions[i];
emit PermissionUpdated(_owner, callers[i], _permissions[i]);
```

with:

```
permissions[msg.sender][callers[i] = _permissions[i];
emit PermissionUpdated(msg.sender, callers[i], _permissions[i]);
```

Impact: This change improves gas usage in two ways. 1. Reduces the deployment size of the contract

Prechange:

```
| src/tba/DGAccountUpgradeable.sol:DGAccountUpgradeable contract |
|----------------------------------------------------------------|
| Deployment Cost | Deployment Size
| 1744119          | 9130
```

Postchange:

```
| src/tba/DGAccountUpgradeable.sol:DGAccountUpgradeable contract |
|----------------------------------------------------------------|
| Deployment Cost | Deployment Size
| 1740512         | 9112
```

2. Reduces gas for users when calling setPermissions

Prechange:

```
| function name  | min  | avg   | median | max   | # calls
| setPermissions | 2890 | 23642 | 5660   | 53374 | 5
```

Postchange:

```
| function name  | min  | avg   | median | max   | # calls
| setPermissions | 2897 | 23637 | 5648   | 53359 | 5
```

This is because the opcode `CALLER` (msg.sender) is cheaper than `MLOAD` (owner stored in memory).

[G-02] Ignore `result` in `DGAccountUpgradeable::upgrade` if it won't be used

The function implementation currently defines a memory variable `result` which is set as the result returned by an external call. However as this value is never used it would be better to ignore it.

See below:

```
    function upgrade(address newImplementation) public virtual
        onlyAuthorized {
        bool success;
-       bytes memory result;

        if (ERC1967Utils.getImplementation() == address(0)) {
            // if upgrading for the first time, "initialize()" first
-           (success, result) = address(this).call(abi.encodePacked(
    bytes4(0x8129fc1c)));
+           (success, ) = address(this).call(abi.encodePacked(bytes4(0
    x8129fc1c)));

            if (!success) revert UpgradeFailed();
        }
        (success, result) = address(this).delegatecall(
            abi.encodeWithSignature("upgradeToAndCall(address,bytes)",
                newImplementation, "")
        );
```

```
            if (!success) revert UpgradeFailed();
        }
```

[G-03] Potential to reorder signature to check first in `DGAccountUpgradeable::isValidSignature`

Currently the function `isValidSignature` (which is called during ERC4337 transactions by `EntryPoint` first checks if the message signer is `DGAccountGuardian::trustedSigner` before checking if the signer is the account `owner`. It's recommended to consider whether `trustedSigner` or `owner` are more likely to be the the signer as it will result in one less signature check and thus save gas overall for ERC4337 transactions.