thirdweb A-14

Security Audit

August 21st, 2023
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for thirdweb's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from August 8, 2023 to August 21, 2023.

The purpose of this audit is to review the source code of certain thirdweb Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	1	-	-	1
High	1	-	-	1
Medium	4	-	-	4
Low	5	1	-	4
Code Quality	10	1	-	9
Gas Optimization	1	-	-	1

thirdweb was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Slack with the thirdweb team.
- A audit handoff document provided through Notion.

Source Code

The following source code was reviewed during the audit:

- **Repository:** contracts
- **Commit Hash:** `cfc3b05e719941d7787263a897542bddb05a6017`

Specifically, we audited the following contracts within this repository:

Contract	SHA256
contracts/dynamic-contracts/extension/RulesEngine.sol	`5a29b078c40eb0c585775e7e5cc40265299023165cc969054b0c09e40781026a`
contracts/dynamic-contracts/extension/SharedMetadataBatch.sol	`6ae8b789f73e18584343bd6a2665f49896b13a534613e2115919cb349502f56a`
contracts/smart-wallet/dynamic/DynamicAccount.sol	`5e986de2977d9ecf9360931be97eaad976509acb7026ee1825b0b4e795842100`
contracts/smart-wallet/dynamic/DynamicAccountFactory.sol	`416d40d2f3aa0c8f0895705477ab23973b41c1002751628427ac6bcf5f0fa5a9`
contracts/smart-wallet/managed/ManagedAccount.sol	`c3b4c601c4106391d59a9b3dff6ae1c89b83c8992a51adaac89f73d8b7bb0e63`
contracts/smart-wallet/managed/ManagedAccountFactory.sol	`5b33e3ef1a0491147e24c1047e68f852cf7061e1671894a2e0e2a9242a80914d`
contracts/smart-wallet/non-upgradeable/Account.sol	`8bce0fb10cac41141478a228a96dad12afe39db384d1938f2529cc455d27081f`
contracts/smart-wallet/non-upgradeable/AccountFactory.sol	`ca1b595fe2f19497d4c1a01b76144ec3b2d5c635192171c00332ad258ec5fdec`
contracts/smart-wallet/utils/AccountCore.sol	`ae8078e8955b24c02c588a5b132813b9c9a08b32ae0db39b1cf7348e2c29ff2c`

Contract	SHA256
contracts/smart-wallet/utils/AccountExtension.sol	2311f64bd2875e63ff395cacb12083411597c3731d42a70e676f2fec925c065a
contracts/smart-wallet/utils/BaseAccount.sol	05841a1d7f05df8113f5e3c2e692121f69c190bdc3c6072c954a9925d44ab6a2
contracts/smart-wallet/utils/BaseAccountFactory.sol	d7181834e702d81b76569e722e342208fea45db7f21a38c28839fb427c2d0a35
contracts/unaudited/LoyaltyPoints.sol	b33c12660ea5b934875a66f992c228da0d7cd019fabe7feeaf81efac2fffcc72
contracts/unaudited/evolving-nfts/EvolvingNFT.sol	90c1072ed646669a51d7bcadaf92d6891d030739a2ea67123d143901443938b8
contracts/unaudited/evolving-nfts/EvolvingNFTLogic.sol	4fb77518d6c3134349bd6a420e35a1c30fb155892f1ae78b2c108ff9c134d503
contracts/unaudited/evolving-nfts/extension/RulesEngineExtension.sol	cbccc956811152c42e3804b848f387fb6bd9ac68d2576f10af61ec25cb380854

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

- C-1 Uninitialized `EvolvingNFT` implementation contract can `selfdestruct` and brick delegated proxies
- ~~H-1~~ `isValidSignature` accepts signatures all active signers, potentially allowing funds to be lost
- ~~M-1~~ `EntryPoint` contract can change
- ~~M-2~~ Use of forbidden `TIMESTAMP` op-code
- ~~M-3~~ Scores using multiplicative rules for ERC20s can be inflated
- ~~M-4~~ Shared metadata will get out of order when deleting metadata
- ~~L-1~~ Unable to upgrade `receive()`
- ~~L-2~~ Cannot remove upgradability without revoking all default admins
- ~~L-3~~ `isValidSignature` should be upgradable
- ~~L-4~~ Invalid accounts can register with Account factories
- L-5 Payable transfer and approvals can lead to native tokens stuck in contract
- ~~Q-1~~ Duplicate code
- ~~Q-2~~ Duplicate comment
- ~~Q-3~~ Constant `MAX_BPS` is not used
- ~~Q-4~~ Mistyped functions
- Q-5 Missing NatSpec documentation
- ~~Q-6~~ Inaccurate comment
- ~~Q-7~~ `_setPlatformFeeType()` is not used
- ~~Q-8~~ Spelling mistakes
- ~~Q-9~~ Duplicate Import

~~Q-10~~ Unused contract

~~G-1~~ `platformFeeType` can share a storage slot

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

C-1 Uninitialized `EvolvingNFT` implementation contract can `selfdestruct` and brick delegated proxies

TOPIC	STATUS	IMPACT	LIKELIHOOD
Upgradability	Fixed	Critical	High

`EvolvingNFT.sol` inherits from `BaseRouter.sol` , which allows extensions to be added by a permissioned account, which allows the contract to `delegatecall` to these set contracts, extending its functionality. `EvolvingNFT` is also intended to be a implementation contract that proxy contracts delegate to in order to save on deployment costs. The state of a implementation contract usually doesn't matter, however if there is any way to cause it to `selfdestruct` , it would destroy the contract, and cause any proxies delegating to it to lose all of their functionality.

In the case of `EvolvingNFT` , it's initializers are not disabled in its constructor, allowing anyone to call its `initialize()` function and set themselves as the implementation contract's `defaultAdmin` .

```
constructor(Extension[] memory _extensions) BaseRouter(_extensions) {}

/// @dev Initiliazes the contract, like a constructor.
function initialize(
    address _defaultAdmin,
    string memory _name,
    string memory _symbol,
    string memory _contractURI,
    address[] memory _trustedForwarders,
    address _saleRecipient,
    address _royaltyRecipient,
    uint128 _royaltyBps
) external initializer initializerERC721A {
    bytes32 _transferRole = keccak256("TRANSFER_ROLE");

    // Initialize inherited contracts, most base-like -> most derived.
    __ERC2771Context_init(_trustedForwarders);
    __ERC721A_init(_name, _symbol);
```

```
        _setupContractURI(_contractURI);
        _setupOwner(_defaultAdmin);
        _setupOperatorFilterer();

        _setupRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
        _setupRole(keccak256("MINTER_ROLE"), _defaultAdmin);
        _setupRole(_transferRole, _defaultAdmin);
        _setupRole(_transferRole, address(0));

        _setupDefaultRoyaltyInfo(_royaltyRecipient, _royaltyBps);
        _setupPrimarySaleRecipient(_saleRecipient);
    }
```

Reference: EvolvingNFT.sol#L44-L74

This allows them to set themselves as the `EXTENSION_ROLE` , which they then can make calls to `BaseRouter` 's `addExtension()` . Doing so can allow an extension to be added that calls `selfdestruct` , which would cause the implementation contact to be destroyed.

This issue is also present in `BurnToClaimDropERC721.sol`

Remediations to Consider

Add a call to `_disableInitializers()` in `EvolvingNFT.sol` and `BurnToClaimDropERC721` 's constructor to prevent a malicious user from taking control of the implementation contract and potentially causing it to `selfdestruct` .

~~H-1~~ `isValidSignature` accepts signatures all active signers, potentially allowing funds to be lost

TOPIC	STATUS	IMPACT	LIKELIHOOD
Loss of Funds	Fixed	High	Medium

As found initially and described in their report, signers set by the account admin can sign valid signatures for target contracts that they are not authorized to interact with. Signers should be restricted to only interact with contracts that have been explicitly set as a `approvedTarget` by a admin of the wallet. However in `isValidSignature()` , it is only checked if the signer of the signature is active, and not if the sender is an `approvedTarget` of the signer.

```
/// @notice See EIP-1271
function isValidSignature(bytes32 _hash, bytes memory _signature)
    public
    view
    virtual
    override
    returns (bytes4 magicValue)
{
    address signer = _hash.recover(_signature);

    if (isAdmin(signer) || isActiveSigner(signer)) {
        magicValue = MAGICVALUE;
    }
}
```

Reference: AccountCore.sol#L136-L149 and Account.sol#L158-L171

This can allow signers set by the wallet admin to interact with protocols that validate signatures with contracts using `isValidSignature()`, like contracts that uses permit, permit2, or any protocol that follows ERC-1271 for handling contract signatures. This can allow set signers to interact with contracts that the wallet admin may not have intended them to be able to, potentially allowing assets within the wallet to be drained.

Remediations to Consider

Check to ensure that the caller is an `approvedTarget` for the signer, if the signature doesn't belong to an admin.

~~M-1~~ `EntryPoint` contract can change

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec	Fixed	Medium	Medium

The smart wallet contracts currently set an immutable value for the `entrypointContract`. However as expressed in the ERC-4337, it is possible that the entryPoint contract could change "to add new functionality, improve gas efficiency, or fix a critical security bug". Since there is no way currently to set a new entryPoint contract, it would require a new wallet to be deployed and assets to be migrated in order to use the updated entryPoint.

Remediations to Consider

Add a function that allows the admin to update the entryPoint contract, allowing users to adapt to a changing entryPoint, and continue to utilize the full functionality of ERC-4337.

~~M-2~~ Use of forbidden `TIMESTAMP` op-code

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec	Fixed	Medium	Low

As described in ERC-4337, the block.timestamp opcode is forbidden when validating user operations, such as with the `validateUserOp` function. However, in each smart wallet contract, in order to validate if a signer is valid, block.timestamp is used to check if the current time fits the time range set by the admin for the signer. This can cause opperations to be valid when simulated, but when executed at a later time could become invalidated and revert. Having wallets revert this way can effect bundlers reputation and potentially cause bundlers to not include operations from these wallets.

Remediations to Consider

As mentioned in the ERC, pack the timestamps `validUntil` and `validAfter` into the returned `validationData` of `validateUserOp()` , this can be done using `Helpers.sol` ’s `packValidationData()` function. This allows a bundler to simulate an operation and check if it would expire before it would likely execute, allowing them to reject these nearly expired operations.

~~M-3~~ Scores using multiplicative rules for ERC20s can be inflated

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec	Fixed	Medium	Medium

In `RulesEngine.sol` scores are calculated for users that have balances that meet the sets rules criteria, and are either calculated as a flat score for `Threshold` rules, or the score is multiplied by the users balance for `Multiplicative` rules. However, `ERC20` tokens balance is returned as a large number with a set `Decimals`, unlike `ERC721` or `ERC1155` tokens. This can lead to scores being multiplied by large values that may be unintended.

Remediations to Consider

Consider using the ERC20 tokens `decimals()` and converting the balance in terms of full tokens before multiplying by the score to receive a score more in line with the scores returned from `ERC721` and `ERC1155`.

~~M-4~~ Shared metadata will get out of order when deleting metadata

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Fixed	Medium	Medium

The shared metadata is stored as an `EnumerableSet`, meaning the order of metadata is not guaranteed, as described in the OpenZeppelin documentation. This means that when adding or removing the metadata, its order will sometimes be altered.

For example, take the following scenario:

- Shared metadata is set sequentially for 0, 10, 50, and 150 target scores.
- The user score is 150, and the correct token URI is returned.
- The metadata for a target score of 10 is removed.
- Now the order of shared metadata is 0, 150, and 50.
- The user scores the same at 150, yet the token URI that gets returned is 50.

The above happens due to the efficient implementation of `remove()` in the `EnumerableSet`. However, it also disrupts the shared metadata order. As a result, the `tokenUri()` in EvolvingNFTLogic.sol returns an incorrect token URI due to how the iteration is performed until the `uint256(ids[i]) <= score` condition is satisfied, as shown below, and it expects the shared metadata to be stored in sequential order:

```
function tokenURI(...) {
    // ...

    for (uint256 i = 0; i < ids.length; i += 1) {
        if (uint256(ids[i]) <= score) {
            targetId = ids[i];
        } else {
            break;
        }
    }

    // ...
}
```

Reference: EvolvingNFTLogic.sol#L75-L99

Remediations to Consider

- Do not assume shared metadata order, and set `targetId` by identifying the shared metadata with the largest target score. Then use that as the final result to be used for displaying the token URI.
- Add a way to alter the existing shared metadata so that the ordering of shared metadata is kept intact.
- Use another mechanism for storing shared mechanism such as a custom implementation of `EnumerableSet` which does not alter the ordering.

~~L-1~~ Unable to upgrade `receive()`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Feature	Fixed	Low	Low

One interesting aspect of a ERC-4337 wallet, or smart contract wallets in general, is the ability to react to receiving native tokens, like ETH. For `ManagedAccount.sol` and `DynamicAccount.sol` adding the ability to update the `receive()` function may be desired for user.

Remediations to Consider

Refactor the code to allow the `receive()` function to be set in an extension, allowing custom receive fallbacks.

~~L-2~~ Cannot remove upgradability without revoking all default admins

TOPIC	STATUS	IMPACT	LIKELIHOOD
Upgradability	Fixed	Low	Low

In `EvolvingNFT.sol` the ability to add or update extensions to the contract can be called by an account with the `EXTENSION_ROLE`. This role can only be granted and revoked by any account with the `DEFAULT_ADMIN_ROLE`, since there is no role admin set for the `EXTENSION_ROLE`. In the case where a project using these contracts wants to turn off the ability to add/update extensions, they would have to revoke all users with the `EXTENSION_ROLE` as well as users with the `DEFAULT_ADMIN_ROLE`, since they can grant the `EXTENSION_ROLE` to another user at a later time.

Revoking all accounts with the `DEFAULT_ADMIN_ROLE` may be undesirable as it also manages other roles like the `TRANSFER_ROLE` and `MINTER_ROLE`, as well as setting multiple other values defined in `EvolvingNFTLogic.sol`, all of which may be needed by the protocol.

Remediations to Consider

Set the `EXTENSION_ROLE` as it's own role admin in the initializer and set an initial account with the `EXTENSION_ROLE`, this will allow it so the contract can no longer be upgraded when there are no account with the `EXTENSION_ROLE`.

~~L-3~~ `isValidSignature` should be upgradable

TOPIC	STATUS	IMPACT	LIKELIHOOD
Upgradability	Fixed	Low	Low

`DynamicAccount.sol` and `ManagedAccount.sol` are upgradable ERC-4337 wallets that inherit base immutable functionality from `AccountCore.sol`, and functions like those found in `AccountExtension.sol` can be added or updated as extensions. Since `isValidSignature()` isn't necessary for the spec of ERC-4337, and there is the possibility that users may want custom functionality for verifying valid signatures, it could be added to `AccountExtension.sol` to allow users to update it as desired.

Remediations to Consider

Move `isValidSignature()` to `AccountExtension.sol` to allow users to have the ability to customize how the contract validates signatures.

~~L-4~~ Invalid accounts can register with Account factories

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed	Low	Low

In `BaseAccountFactory.sol`, accounts are allowed to be registered to the factory by calling `onRegister()`.

```
/// @notice Callback function for an Account to register itself on the factory.
function onRegister() external {
    address account = msg.sender;
    require(_isAccountOfFactory(account), "AccountFactory: not an account.");

    require(allAccounts.add(account), "AccountFactory: account already registered");
}
```

Reference: BaseAccountFactory.sol#L74-L80

However, there are no checks to ensure the caller is an account created by this factory. A contract could potentially call `onRegister()` and become registered with this factory, if it passes the checks of `_isAccountOfFactory`, which does not guarantee the account was created by the factory, as it only checks if the factory's implementation address is in the bytecode of the calling contract at the expected location.

```
/// @dev Returns whether the caller is an account deployed by this factory.
function _isAccountOfFactory(address _account) internal view virtual returns (bool) {
    address impl = _getImplementation(_account);
    return _account.code.length > 0 && impl == accountImplementation;
}

function _getImplementation(address cloneAddress) internal view returns (address) {
    bytes memory code = cloneAddress.code;
    return BytesLib.toAddress(code, 10);
}
```

Reference: BaseAccountFactory.sol#L134-143

Setting an invalid address for an account can lead to inaccurate book keeping, and if any contract or protocol were to query the factory to check if an address is a account created by the factory, it may not be accurate.

Since these contracts are generated using `create2` , if the initial seed to generate the account is provided, the generated address that the factory would have deployed can be predicted using `Clones.predictDeterministicAddress()` and checked with the calling contract to verify the account was created by the factory.

Remediations to Consider

When registering an account with a factory, accept the initial admin and data parameter used to generate the account address and verify that the caller is the same as the predicted address generated by those values. This will ensure only accounts created by the factory can be registered.

L-5 Payable transfer and approvals can lead to native tokens stuck in contract

TOPIC	STATUS	IMPACT	LIKELIHOOD
Error Recovery	Acknowledged	Medium	Low

`EvolvingNFTLogic.sol` 's `approve()` , `transferFrom()` and `safeTransferFrom()` functions are set to `payable` which allows native tokens to be sent into the contract when making these function calls. However, native tokens sent in via these function calls are not used, and there is no way to withdraw these tokens without a permissioned user adding an extension to do so. In cases where extension permissions have been revoked, there would be no way to withdraw these sent in funds. There is a chance that funds could be sent to the contract accidentally, especially when interacting with third party protocols like etherscan to transfer tokens or set approvals as it would prompt the user to enter a value of native tokens to send, which may get confused for other function parameters. It is understood that using the `payable` keyword reduces gas costs as there is no check to ensure that `msg.value == 0`, but the added gas cost is negligible compared to the potential downsides.

Remediations to Consider

Remove the payable keywords from `EvolvingNFTLogic.sol`'s `approve()`, `transferFrom()` and both `safeTransferFrom()` functions, in order to prevent native tokens from accidentally getting stuck in the contract.

RESPONSE BY THIRDWEB

> These functions are payable since they override from the ERC721AUpgradeable contract, where these functions are payable ref.
>
> We will fix this issue later on, where we do a sweep of our external dependencies.

~~Q-1~~ Duplicate code

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Medium

`Account.sol` and `AccountCore.sol` share a lot of the same functions, and the code is identical, and both inherit `BaseAccount.sol`. Duplicate code can cause errors as changes to the code has to be done in multiple places.

Remediations to Consider

Refactor Account.sol and AccountCore.sol to prevent duplicate code, and reducing the chance of errors when updating these contracts.

~~Q-2~~ Duplicate comment

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

In `AccountCore.sol` there is a duplicate comment.

```
// We use the underlying storage instead of high level view functions to save gas.
// We use the underlying storage instead of high level view functions to save gas.
```

Reference: Account.sol#L81-L82

Remediations to Consider

Remove the duplicate comment.

~~Q-3~~ Constant `MAX_BPS` is not used

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

Constant `MAX_BPS` is declared in EvolvingNFTLogic.sol, yet it’s not used in the contract.

Remediations to Consider

Remove `MAX_BPS` .

~~Q-4~~ Mistyped functions

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

`RulesEngine.sol` contains mistyped functions which can be corrected as follows:

- `_canOverrieRulesEngine()` → `_canOverrideRulesEngine()`
- `createRuleMulitiplicative*()` →* `createRuleMultiplicative()`

Q-5 Missing NatSpec documentation

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Low

Functions in these contracts use NatSpec documentation, but they tend to not include the `@param` and `@return` tags, which give more information about the intent of the function, and is used by some protocols like etherscan to make improve the user experience when making these calls. Additionally `RulesEngine.sol` does not have any NatSpec documentation.

Remediations to Consider

Add missing NatSpec documentation.

RESPONSE BY THIRDWEB

> We're planning a sweep of the repository to add proper Natspec documentation across all Solidity files.

~~Q-6~~ Inaccurate comment

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

In `EvolvingNFT.sol` there is a comment above the `EXTENSION_ROLE` that mentions the `MINTER_ROLE`.

```
/// @dev Only MINTER_ROLE holders can sign off on `MintRequest`s.
bytes32 private constant EXTENSION_ROLE = keccak256("EXTENSION_ROLE");
```

Reference: EvolvingNFT.sol#L41-L42

Remediations to Consider

Update the comment to accurately describe the `EXTENSION_ROLE`.

~~Q-7~~ `_setPlatformFeeType()` is not used

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

In `PlatformFee.sol`, the internal function `_setPlatformFeeType()` is not used, and similar logic is found in the external function `setPlatformFeeType()`. Other functions within this contract have a pattern of an external call with checks, to a internal call that changes state and emits the event, but these functions break that pattern.

```
/// @notice Lets a module admin set platform fee type.
function setPlatformFeeType(PlatformFeeType _feeType) external {
    if (!_canSetPlatformFeeInfo()) {
        revert("Not authorized");
    }
    platformFeeType = _feeType;

    emit PlatformFeeTypeUpdated(_feeType);
}

/// @dev Sets platform fee type.
function _setupPlatformFeeType(PlatformFeeType _feeType) internal {
    platformFeeType = _feeType;

    emit PlatformFeeTypeUpdated(_feeType);
}
```

Reference: PlatformFee.sol#L88-L103

Remediations to Consider

Consider having `setPlatformFeeType()` call `_setPlatformFeeType()` after the checks to prevent duplicate code and follow to the set style pattern.

~~Q-8~~ Spelling mistakes

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

In `LoyaltyPoints.sol`, in the comment above `mintWithSignature()`, "recipient" is spelled incorrectly

```
/// @notice Mints tokens to a recipeint using a signature from an authorized party.
```

Reference: LoyaltyPoints.sol#L119

Additionally, in most contracts with an initializer, there is a comment that misspells the word “initializes”.

```
/// @dev Initiliazes the contract, like a constructor.
```

Reference: EvolvingNFT.sol#L48

Remediations to Consider

Fix these spelling mistakes.

~~Q-9~~ Duplicate Import

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Low

In LoyaltyPoints.sol, PrimarySale.sol is imported twice.

```
import "./extension/PrimarySale.sol";
import "./extension/PrimarySale.sol";
```

Reference: LoyaltyPoints.sol#L27-L28

Remediations to Consider

Remove the duplicate import.

~~Q-10~~ Unused contract

TOPIC	STATUS	QUALITY IMPACT
code Quality	Fixed	Low

In the utils directory of smartWallet, there is a `BaseRouter.sol` contract that is not used or referenced.

Remediations to Consider

Remove `BaseRouter.sol` .

~~G-1~~ `platformFeeType` can share a storage slot

TOPIC	STATUS	GAS SAVINGS
Gas Optimization	Fixed	Low

In `PrimarySale.sol` , the storage values `platformFeeRecipient` and `platformFeeBps` are defined next to each other, and since `platformFeeRecipient` is an `address` , taking 20 bytes of space, and `platformFeeBps` is a `uint16` taking 2 bytes of space, there is 10 bytes left over in the first storage slot. The next defined value is `flatPlatformFee` , a `uint256` which needs its own storage slot, and `platformFeeType` is defined after, and is a `PlatformFeeType` `enum` which takes up 1 byte of space, and since the second slot above it is full, it will take up a the 3rd storage slot on its own. If `platformFeeType` is defined before `flatPlatformFee` , it will share the first storage slot with `platformFeeRecipient` and `platformFeeBps` , saving a new storage write when set, and since these values are typically read together it would benefit from warm `SLOAD` 's whenever they are read together.

```
/// @dev The address that receives all platform fees from all sales.
address private platformFeeRecipient;

/// @dev The % of primary sales collected as platform fees.
uint16 private platformFeeBps;

/// @dev The flat amount collected by the contract as fees on primary sales.
uint256 private flatPlatformFee;

/// @dev Fee type variants: percentage fee and flat fee
PlatformFeeType private platformFeeType;
```

Reference: PrimarySale.sol#L16-L26

Remediations to Consider

Swap the positions of `platformFeeType` and `flatPlatformFee` to save users gas on storage reads and writes.

Disclaimer