Security Review Report
NM-0156 BLOCKFRAME

NETHERMIND SECURITY

(Jan 29, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the Blockframe protocol. Blockframe is a marketplace for non-fungible tokens (NFTs) that supports on-chain royalties. Enabling users to buy, sell, and split royalty proceeds for NFT collections. The marketplace engine is an adapted version of the Seaport protocol, enabling royalty payments within the platform. The platform also introduces the Creator Studio, a user-friendly feature that simplifies NFT collection creation and customization for users.

The reviewed code comprises partially the two mentioned components. Nethermind reviewed the additions made to the Seaport marketplace, specifically the code enabling royalty payments. Additionally, the review encompassed the implementation of the Creator studio. Throughout this process, the Blockframe and Nethermind teams engaged in extensive discussions to deepen their understanding of the code and enhance the review's effectiveness. However, it was observed that there is a notable lack of adequate testing and documentation accompanying these developments.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contract. Along with this document, we report 18 points of attention, where one is classified as `Critical`, two are classified as `High`, three are classified as `Medium`, one are classified as `Low`, six are classified as `Info` and five are classified as `Best Practices`. The issues are summarized in Fig. 1.

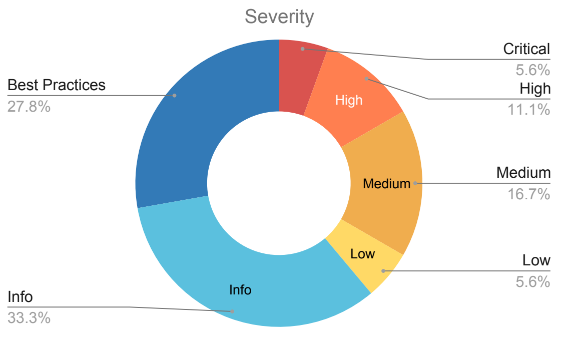
Remarks

Upon reviewing the source code of the project, to ensure the security and quality of the project, the Nethermind team suggests engaging in complementary security reviews and bug bounty programs.

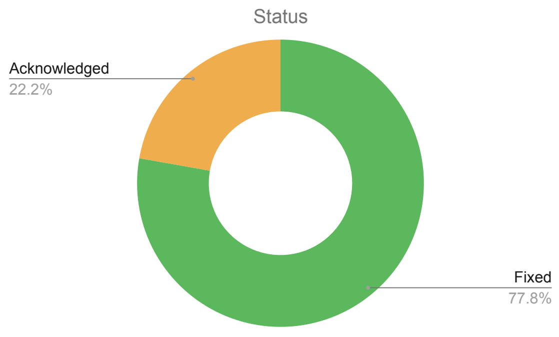
Remarks

All the fixes were reviewed in an isolated manner. No full version of the code was reviewed during the fixes-review phase.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation and test output. Section 9 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (1), **High** (2), **Medium** (3), **Low** (1), **Undetermined** (0), **Informational** (6), **Best Practices** (5). **(b) Distribution of status: Fixed** (14), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Nov 27, 2023
Final Report	Jan 29, 2024
Methods	Manual Review, Automated Analysis, Tests
Repository (Royalties, Creator-studio)	blockframeapp/audit
Commit Hash (Royalties, Creator-studio)	fd43ad326585a3438179aff2a090c79febc81745
Repository (Seaport modification)	blockframeapp/seaport
Commit Hash (Seaport modification)	2cf367d2c199c92c197cb98bfab1705e4d98a7e3
Documentation Assessment	Low
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/contracts/RoyaltyRegistrar.sol	25	1	4.0%	3	29
2	contracts/contracts/RoyaltyRegistrarImpl.sol	189	67	35.4%	40	296
3	contracts/contracts/creatorStudio/factory/GenerativeCollection.sol	147	21	14.3%	35	203
4	contracts/contracts/creatorStudio/factory/OnlyOwnerCollection.sol	26	19	73.1%	11	56
5	contracts/contracts/creatorStudio/factory/CollectionFactory.sol	39	20	51.3%	8	67
6	contracts/contracts/creatorStudio/factory/GenerativeCollectionMetadata.sol	19	1	5.3%	1	21
7	contracts/contracts/interfaces/ICurrencyManager.sol	8	1	12.5%	5	14
	Total	**453**	**130**	**28.7%**	**103**	**686**

Remarks

In addition to the files listed, the review also encompasses the modifications in the Seaport marketplace, particularly the changes facilitating royalty payments. The Seaport marketplace code was forked in commit a27eff9fa299a14c432666c8dc963aa072437dde

3 Summary of Issues

	Finding	Severity	Update
1	The `ERC20` royalties deposited in the `RoyaltyRegistrarImpl` contract cannot be withdrawn	Critical	Fixed
2	Marketplace cannot send ETH as Royalty to `RoyaltyRegistrarImpl.sol` , leading to reverting order fulfillment	High	Fixed
3	Possible deployment of multiple tokens for the same `(HostContract, Currency)` pair	High	Fixed
4	Rounding error in the `updateWithdrawn(...)` function may lead to reward exploitation	Medium	Fixed
5	The `reserve(...)` function will reserve more NFTs than it should	Medium	Fixed
6	Centralization Risks	Medium	Acknowledged
7	Forced excessive deposits for native ETH in the `depositRoyalties(...)` function, resulting in locked funds	Low	Fixed
8	Majority is incorrectly checked	Info	Fixed
9	Rebasing tokens are not supported as royalty payment currencies	Info	Acknowledged
10	Royalty token can be claimed with an invalid percentage	Info	Fixed
11	Total supply can be exceeded by the `reserve(...)` function	Info	Fixed
12	Unnecessary underflow checks in the `split(...)` function	Info	Fixed
13	`SafeERC20` is imported but not used	Info	Fixed
14	Avoid the usage of magic numbers	Best Practices	Fixed
15	Lack of input validation when creating a collection	Best Practices	Acknowledged
16	Redundant check for array length in the `getWithdrawn(...)` function	Best Practices	Fixed
17	Unused code	Best Practices	Fixed
18	Use `_safeMint(...)` instead of `_mint(...)` in `OnlyOwnerCollection`	Best Practices	Acknowledged

4 Protocol Overview

4.1 Marketplace

The Blockframe protocol's Marketplace is a customized adaptation of the seaport protocol, tailored to support royalty payments within the system. At its core, the `RoyaltyRegistrarImpl` smart contract tracks royalties and enables their transfer, splitting, and withdrawal across diverse collections and currencies. Creators can claim royalty contracts for their collections, specifying precise royalty percentages. Furthermore, the contract owner possesses the authority to claim royalties for different collections, particularly for un-owned or revoked ownership instances.

Royalties are represented via an ERC20 token known as the Royalty Token. It is associated with each collection and its respective payment currency. These Royalty Tokens are tracked in the mapping `hostContractToRoyaltyTokens[_hostContract][currency]`. This mapping associates each host contract, representing a collection and its respective payment currency with the corresponding ERC20 Royalty Token. Users' stake within the contract royalties are denoted by their balances in these tokens. For example, possessing 10% of the total Royalty Tokens for a specific collection signifies ownership of 10% of the future royalty revenue for that particular collection.

Deposits into the royalty system are open to any participant. Additionally, Token holders can withdraw, transfer, or split their tokens with other users. The alteration to the seaport protocol includes the addition of two fields to all order items: `royaltyPaymentFor` and `frameIncentive`. Orders involving royalty payments should include the payment as a token itself within the order, designating the `RoyaltyRegistrarImpl` as the recipient and specifying the associated host contract in the `royaltyPaymentFor` field. It is essential to note that the current on-chain implementation does not enforce the royalty percentage set by the contract claimer during payments. Instead, the off-chain system retrieves these percentages from the `RoyaltyRegistrarImpl` contract to compute appropriate royalty payments, which are subsequently included in the corresponding orders as royalty payments.

4.2 Creator Studio

Blockframe's Creator Studio is designed to support users in effortlessly creating, customizing, and managing NFT collections. The `CollectionFactory` contract enables users to create a generative collection or an only owner collection by calling the respective functions `createGenerativeCollection(...)`, `createOnlyOwnerCollection(...)` from the `CollectionFactory` contract. The two types of collections differ in the functionality they offer:

- **OnlyOwnerCollection**: This collection type allows only the collection creator to mint the NFTs.
- **GenerativeCollection**: The `GenerativeCollection` is a more complex and flexible collection type. NFTs can be minted during the pre-sale and public sale periods. Only the whitelisted users can participate in the pre-sale. Apart from the whitelist, the collection creator specifies the currency in which he wants the users to pay for the NFTs. This can be either native Ether or ERC20 tokens. Funds gathered during these sales can be withdrawn by the owner by calling the `withdraw(...)` function. The 2% of these funds are sent to the protocol as fees. The collection owner can also reserve an arbitrary number of NFTs outside of the regular sale periods by calling the `reserve(...)` function.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] The ERC20 royalties deposited in the RoyaltyRegistrarImpl contract cannot be withdrawn

File(s): RoyaltyRegistrarImpl.sol

Description: The withdraw(...) function is used to withdraw the royalties deposited in the RoyaltyRegistrarImpl contract and transfer them to the recipient address. This function is supposed to handle both types of currencies that royalties can be denominated in - native tokens and ERC20 tokens. In the latter case, the tokens are transferred to the recipient by calling the transferFrom(...) function from the ERC20 standard. However, the problem with this function is that the spender requires the allowance to transfer the tokens successfully. The issue in the withdraw(...) function is that the spender - in this case, the RoyaltyRegistrarImpl contract does not have the necessary allowance to make the call to transferFrom(...) succeed. Whenever someone calls the withdraw(...) function to withdraw the deposited ERC20 royalties, the call will revert due to insufficient allowance. This failure causes the ERC20 royalties to be locked inside the RoyaltyRegistrarImpl contract. The snippet of the withdraw(...) function is presented below:

```
function withdraw(...) external override nonReentrant {
  // ...
  if (currency == address(0)) {
    // ...
  } else {
    // @audit A call to `transferFrom` will revert due to insufficient allowance
    require(IERC20(currency).transferFrom(address(this), recipient, availableRoyalties), "Transfer failed");
  }
  // ...
}
```

Recommendation(s): Consider using transfer(...) instead of transferFrom(...).

Status: Fixed.

Update from the client: Changed to transfer.

6.2 [High] Marketplace cannot send ETH as Royalty to RoyaltyRegistrarImpl.sol, leading to reverting order fulfillment

File(s): RoyaltyRegistrarImpl.sol

Description: Blockframe's marketplace incorporates royalties by including the royalty payment in the concerned order. The contract supports payments in both ERC20 tokens and native tokens. Upon order fulfillment, the royalty payment is sent to the RoyaltyRegistrarImpl contract.

For payments made in native tokens, the _updateRoyaltiesIfNeeded(...) function is called, which updates the accounting of royalties in the RoyaltyRegistrarImpl contract by invoking the updateRoyalties(...) function and using a currency of address(0) to denote native tokens. Subsequently, the tokens are transferred to the contract within the _transferNativeTokens(...) function. Below is the code segment handling this process within the BasicOrderFullfiller contract:

```
// @audit calling `updateRoyalties(...)` function to account royalties
_updateRoyaltiesIfNeeded(token, address(0), additionalRecipientAmount, recipient);
// ...
// @audit Sending native tokens to the registrar contract
_transferNativeTokens(recipient, additionalRecipientAmount);
```

The _transferNativeTokens(...) function sends native tokens to the RoyaltyRegistrarImpl contract via a low-level call. However, the latter contract does not implement the necessary functionality to receive native tokens. **This results in every order fulfillment reverting when the order sends royalties in native tokens.**

Recommendation(s): Add the necessary functionality to the RoyaltyRegistrarImpl contract to receive native tokens.

Status: Fixed.

Update from the client: Added receive() external payable .

Update from Nethermind: As an additional comment, any eth sent by an address different from seaport will be locked.

6.3 [High] Possible deployment of multiple tokens for the same (HostContract, Currency) pair

File(s): RoyaltyRegistrarImpl.sol

Description: The RoyaltyRegistrarImpl contract allows users to claim Royalty contracts associated with their host contracts and specific currencies. It defines a mapping called hostContractToRoyaltyTokens, which stores the token address for each unique (host, currency) pair. However, the following issues in the code allow the deployment of multiple Royalty tokens pointing to the same (host, currency) pair.

1. In the mintRoyaltyToken(...) function, there is no check to ensure that a token hasn't been minted before for a specific host contract and currency pair. This lack of validation allows the deployment of multiple tokens for the same pair;

```
function mintRoyaltyToken(address _hostContract, address currency) external virtual override {
  // ...
  // @audit deploys a new token for the host contract/currency pair
  address token = createERC20("Royalty Token", "RT", claimed[_hostContract], currency, _hostContract);

  // @audit If the token was deployed previously, its value will be overridden
  hostContractToRoyaltyTokens[_hostContract][currency] = token;
  // ...
}
```

2. The claimRoyaltyForRevoked(...) function doesn't verify whether the recipient address differs from zero. If a zero address is set, both claimRoyalty(...) and claimRoyaltyForRevoked(...) functions could be called as we will have claimed[_hostContract] == address(0), resulting in the deployment of another token with the same host contract and currency. The corresponding value in the hostContractToRoyaltyTokens mapping will be overridden.

```
function claimRoyaltyForRevoked(address _hostContract, uint256 _percentage, address recipient)
  external
  virtual
  override
  onlyOwner
{
  require(claimed[_hostContract] == address(0), "Token already claimed.");
  IContract hostContract = IContract(_hostContract);
  address token = createERC20("Royalty Token", "RT", recipient, address(0), _hostContract);
  // ...
  // @audit recipient can be zero address
  claimed[_hostContract] = recipient;
  hostContractToRoyaltyTokens[_hostContract][address(0)] = token;
  // ...
}
```

Deploying multiple tokens for a single (host contract, currency) pair causes erroneous updates in the withdrawn mapping within the updateWithdrawn(...) function. These incorrect updates directly affect royalty computation and user withdrawals. Furthermore, this flaw enables malicious behavior by the host contract owner. Consider a scenario where users purchase royalty tokens for a specific host and currency pair. The host owner can maliciously deploy another token for the same pair, overriding the initial token in the hostContractToRoyaltyTokens mapping. Consequently, users cannot withdraw their earned royalties as their balances are zero in the new token.

Recommendation(s): Consider adding the relevant checks in both the mintRoyaltyToken(...) and the claimRoyaltyForRevoked(...) functions to ensure that only one token is deployed per host contract and currency.

Status: Fixed.

Update from the client: Added missing `require(hostContractToRoyaltyTokens[_hostContract][currency] == address(0), "Royalty token minted");` in mintRoyaltyToken.

6.4 [Medium] Rounding error in the `updateWithdrawn(...)` function may lead to reward exploitation

File(s): `RoyaltyRegistrarImpl.sol`

Description: The `updateWithdrawn(...)` function is invoked during the transfer of `RoyaltyTokens`. This function is critical for updating the `withdrawn` mapping for both the `sender` and the `receiver`. The intent is to prevent the double-claiming of assets associated with the transferred `RoyaltyTokens`. However, an issue arises when calculating the `transferredWithdrawn` value within the function.

```
function updateWithdrawn(...) external onlyAuthorizedTokens {
  uint256 senderBalance = IERC20(hostContractToRoyaltyTokens[hostContract][currency]).balanceOf(sender);
  uint256 senderWithdrawn = withdrawn[hostContract][currency][sender];
  uint256 recipientWithdrawn = withdrawn[hostContract][currency][recipient];

  // @audit This division rounds down, causing the value `transferredWithdrawn`
  // to be lower than it should be in certain cases.
  uint256 transferredWithdrawn = (senderWithdrawn * amount) / senderBalance;

  withdrawn[hostContract][currency][sender] = senderWithdrawn - transferredWithdrawn;
  withdrawn[hostContract][currency][recipient] = recipientWithdrawn + transferredWithdrawn;
}
```

The division used to compute `transferredWithdrawn` rounds down. This behavior can result in the `transferredWithdrawn` value being lower than it should be, particularly with small transfer amounts. For example, if user `A` transfers a fraction of their `RoyaltyTokens` to user `B`, the calculated `transferredWithdrawn` might round to zero. Consequently, the withdrawn mapping remains unchanged, allowing user `B` to claim rewards on the received tokens, even if those rewards were already claimed once. User `A` can repeat the process multiple times, claiming the already claimed rewards.

Recommendation(s): Consider rounding up when computing the `transferredWithdrawn` value.

```diff
- uint256 transferredWithdrawn = (senderWithdrawn * amount) / senderBalance;
+ uint256 transferredWithdrawn = (senderWithdrawn * amount + (senderBalance - 1)) / senderBalance;
```

Status: Fixed.

Update from the client: Fixed exactly as recommended.

Update from Nethermind: It was noticed that the applied fix introduces an error similar to the one described. With the added code, the sender may decrease its `withdrawn` value more than it should, which could allow the sender to re-claim some rewards that were already claimed. Consider modifying the scheme used for accounting rewards. One possible solution would be to apply a model similar to the one used by Synthetix to track rewards.

Update from the client: Refactored to use a discrete staking rewards methodology:

```
  function withdraw(address recipient, address hostContract, address currency) nonReentrant external override {
      accrueUserRewardsInternal(recipient, hostContract, currency);

      uint256 withdrawableAmount = rewards[recipient][hostContract][currency];
      require(withdrawableAmount > 0, "No royalties available for withdrawal.");

      rewards[recipient][hostContract][currency] = 0;

      if (currency == address(0)) {
          (bool success,) = recipient.call{value: withdrawableAmount}("");
          require(success, "Transfer failed");
      } else {
          IERC20(currency).safeTransfer(recipient, withdrawableAmount);
      }

      emit Withdraw(recipient, hostContract, currency, withdrawableAmount);
  }
```

```
  function accrueUserRewardsInternal(address account, address hostContract, address currency) internal {
    uint256 userBalance = IERC20(hostContractToRoyaltyTokens[hostContract][currency]).balanceOf(account);
    uint256 owed = (userBalance * (globalIndex[hostContract][currency] - userIndex[account][hostContract][currency])) /
      ↪ PRECISION_FACTOR;
    rewards[account][hostContract][currency] += owed;
    userIndex[account][hostContract][currency] = globalIndex[hostContract][currency];
  }
```

6.5 [Medium] The `reserve(...)` function will reserve more NFTs than it should

File(s): `GenerativeCollection.sol`

Description: The `reserve(...)` function is used by the collection owner to reserve some NFTs aside from the sale/presale process. The owner specifies an `amount` of the NFTs to be minted to his address. The `_safeMint(...)` function from the `ERC721A` is called for each NFT to be minted. However, the problem with this function is that it mints multiple NFTs simultaneously, unlike OpenZeppelin's `ERC721` implementation, which mints only one token at a time. In the code snippet below, `supply + i` tokens will be minted for each loop iteration.

```
function reserve(...) public onlyOwner {
  uint256 supply = totalSupply();
  uint256 i;
  for (i = 0; i < amount; i++) {
    // @audit Will mint supply + i NFTs for each loop iteration
    _safeMint(msg.sender, supply + i);
  }
  // ...
}
```

The number of tokens minted depends on the `amount` specified by the owner and the already existing `totalSupply` of tokens. The `reserve(...)` function can be called anytime. It is not bound to the sale status. This means that the total number of tokens reserved by the collection owner will be higher if there was a sale already and the total supply was increased.

A numerical example: There were `10` NFTs sold in the last sale. The collection owner decides to reserve `5` additional NFTs. A call to `reserve(...)` with an `amount` of `5` will mint `10` NFTs in the first loop iteration, `11` in the second, `12` in the third, and so on...

Recommendation(s): Consider removing the loop and mint an appropriate amount of NFTs by calling the `ERC721A`'s `_safeMint(...)` function with the desired amount of tokens to be reserved.

Status: Fixed.

Update from the client: Fixed:

```
    function reserve(uint256 amount) public onlyOwner {
        uint256 supply = totalSupply();
        require(supply + amount <= metadata.totalSupply, "Purchase would exceed max supply of NFTs");
        _safeMint(msg.sender, amount);
        address seller = owner();
        uint256 totalCost = 0;

        emit GenerativeCollectionMint(msg.sender, seller, amount, totalCost, supply, MARKETPLACE_HASH);
    }
```

6.6 [Medium] Centralization Risks

File(s): `RoyaltyRegistrarImpl.sol`

Description: There are multiple situations that can be caused in the Blockframe systems by users with enough privileges. These situations could occur, for example, due to a privilege key being compromised or similar. Below we list examples of those situations:

- The setSeaport(...) function permits users with sufficient privileges to assign any address to the seaport variable. The designated address can then execute the updateRoyalties(...) function, which alters the royalty mapping. Such changes potentially lead to the misappropriation of all tokens within the contract;
- Another risk is related to the function `claimRoyaltyForRevoked()`. This function is expected to be used when the NFT host contract does not implement the `owner()` function. The owners of the NFT host contract can then ask Blockframe to call the function and update the contract's royalties mapping. That is where centralization risk lies, as it can be used by the owner to create royalty for any NFT contract in the system without any checks;

Recommendation(s): Consider setting multi-sig accounts as owner when possible and clearly stating the risks of using the system.

Status: Acknowledged.

Update from the client:

- `setSeaport` mitigated by the fact it can only be set once and provably so;
- yes, but a necessity;

6.7 [Low] Forced excessive deposits for native ETH in the `depositRoyalties(...)` function, resulting in locked funds

File(s): `RoyaltyRegistrarImpl`

Description: The `depositRoyalties(...)` function allows users to deposit royalties for a specific host contract and currency. However, in the case where the currency is native `ETH`, the code enforces a condition where the transferred `ETH` amount `msg.value` must be **strictly** greater than the specified `amount`. This requirement forces users to send at least one extra wei, which is unnecessary. Furthermore, the function only accounts for the specified `amount` in the `royaltyBalances` mapping, leaving any surplus `ETH` (`msg.value - amount`) locked in the contract as there is no way to utilize or withdraw it.

```
function depositRoyalties(...) nonReentrant external override payable {
  // ...
  if (currency != address(0)) {
    require(IERC20(currency).transferFrom(msg.sender, address(this), amount), "Transfer failed");
  } else {
    // @audit user is enforced to transfer more than the accounted amount
    require(msg.value > amount, "Insufficient funds");
  }
  royaltyBalances[hostContract][currency] += amount; // @audit only `amount` is accounted for
  // ...
}
```

Recommendation(s): In order to align the expected deposit amount with the required value, consider updating the condition to accept the exact expected amount:

```
require(msg.value == amount, "Insufficient funds");
```

Status: Fixed.

Update from the client: Resolved exactly as recommended.

6.8 [Info] Majority is incorrectly checked

File(s): `RoyaltyRegistrarImpl.sol`

Description: The `setPercentage (...)` function changes the expected royalties percentage from the sale of an asset from the `_hostContract` collection. The caller of this function must own the majority of the tokens representing rights over these royalties.

```
function setPercentage(address _hostContract, uint256 _newPercentage) external virtual override {
  require(_newPercentage <= 10 ** 20, "Maximum royalty is 100%");
  address royaltyToken = hostContractToRoyaltyTokens[_hostContract][address(0)];
  uint256 totalSupply = IERC20(royaltyToken).totalSupply();
  uint256 callerBalance = IERC20(royaltyToken).balanceOf(msg.sender);
  // @audit The majority is not checked correctly
  require(callerBalance >= totalSupply / 2, "Caller does not own a majority of the token supply");
  require(_newPercentage <= percentages[_hostContract], "New royalty cannot be more than current royalty");

  percentages[_hostContract] = _newPercentage;
  emit NewPercentage(_hostContract, _newPercentage);
}
```

However, the condition that checks the majority is not implemented correctly. The operator used is >= instead of =, allowing the owner of `half` of the tokens to modify the percentage.

Recommendation(s): Consider using > instead of >=.

Status: Fixed.

Update from the client: Fixed exactly as recommended.

6.9 [Info] Rebasing tokens are not supported as royalty payment currencies

File(s): `RoyaltyRegistrarImpl.sol`

Description: The collection owner can specify any currency(token) for his royalty payments. This can lead to potential problems when the currency is a rebasing token, such as `sETH`.

The rebasing tokens are changing their supply based on the token's price, posing a problem during royalties withdrawal. When a certain amount of royalties is deposited and accounted for in the contract (using the `depositRoyalties(...)` or `updateRoyalties(...)` functions), it can be decreased with time, resulting in the `withdraw(...)` function potentially reverting as it will compute the royalties to be withdrawn based on the initially deposited amount.

Recommendation(s): To avoid potential losses related to the behavior of rebasing tokens, consider explicitly stating in the documentation that Blockframe does not support the use of rebasing tokens for royalty payments.

Status: Acknowledged.

Update from the client: OK

6.10 [Info] Royalty token can be claimed with an invalid percentage

File(s): `RoyaltyRegistrarImpl.sol`

Description: `claimRoyalty(...)` and `claimRoyaltyForRevoked(...)` functions allow for claiming the royalty token for a specific host contract with a certain royalty percentage. However, these functions are missing a check for the validity of the provided percentage, making it possible to pass invalid values as a percentage higher than 100%.

```
function claimRoyalty(address _hostContract, uint256 _percentage) external virtual override {
  // ...
  // @audit missing check that `_percentage` is within a valid range
  percentages[_hostContract] = _percentage;
  // ...
}
```

Recommendation(s): Consider adding a check for both `claimRoyalty(...)` and `claimRoyaltyForRevoked(...)` functions so that the provided percentage is within a valid range.

Status: Fixed.

Update from the client: Added `require(_percentage <= MAX_PERCENTAGE, "Maximum royalty is 100%");`

6.11 [Info] Total supply can be exceeded by the `reserve(...)` function

File(s): `GenerativeCollection.sol`

Description: The `generativeCollection` contract includes a `totalSupply` field in its metadata, which specifies the maximum supply of the NFT. This value is enforced during every mint action to prevent surpassing the defined limit.

The contract defines the `reserve(...)` function, allowing the owner to reserve some tokens for themselves. However, This function mints the desired amount to be reserved without checking whether this will result in exceeding the supply limit defined in `metadata.totalSupply`.

```
function reserve(uint256 amount) public onlyOwner {
  // @audit `metadata.totalSupply` can be exceeded after minting `amount`
  uint256 supply = totalSupply();
  uint256 i;
  for (i = 0; i < amount; i++) {
    _safeMint(msg.sender, supply + i);
  }
  emit GenerativeCollectionMint(address(this), amount);
}
```

Recommendation(s): Implement a check within the `reserve(...)` function to comply with the `totalSupply` limit specified in the metadata.

Status: Fixed.

Update from the client: Decided not to fix it as it's the owner's collection. They can do whatever they want. Increase the supply, update the base URI.

Update from Nethermind: The mentioned issue was actually fixed in the received commit.

6.12 [Info] Unnecessary underflow checks in the `split(...)` function

File(s): `RoyaltyRegistrarImpl.sol`

Description: The `split(...)` function performs unnecessary checks for underflows. Solidity v0.8.0 introduced default overflow checks for arithmetic operations, making the function revert in these cases. Hence, the checks after the computation are ineffective.

```
function split(...) external virtual override {
  // ...
  for (uint256 i = 0; i < addresses.length; i++) {
    amount = totalAmount * weights[i] / 1e20;
    // @audit Computation reverts if `remainingAmount < amount`, the following check is unnecessary
    remainingAmount = remainingAmount - amount;
    require(remainingAmount <= totalAmount, "Integer overflow");
    require(IERC20(token).transferFrom(msg.sender, addresses[i], amount), "Transfer failed");
  }
  // @audit Unnecessary check as the function reverts sooner when an underflow/overflow occurs
  require(remainingAmount <= totalAmount, "Integer overflow");
}
```

Recommendation(s): Consider removing unnecessary checks.

Status: Fixed.

Update from the client: Fixed exactly as recommended.

6.13 [Info] `SafeERC20` is imported but not used

File(s): `RoyaltyRegistrarImpl.sol`, `GenerativeCollection.sol`

Description: The `SafeERC20` wrapper for ERC20 tokens from Openzeppelin is imported but not used. A list of files where this issue is present is shown below:

```
RoyaltyRegistrarImpl.sol
    function split
    function withdraw
    function depositRoyalties

GenerativeCollections.sol
    function withdraw
```

Recommendation(s): Consider using the `SafeERC20` wrapper for ERC20 operations or remove the import if it won't be used.

Status: Fixed.

Update from the client: Using `SafeERC20` now.

Update from Nethermind: `transfer(...)` is still used in `GenerativeCollection.sol` contract instead of `safeTransfer(...)`.

Update from the client: Fixed.

6.14 [Best Practices] Avoid the usage of magic numbers

File(s): `RoyaltyRegistrarImpl`

Description: A magic number is a numerical value hard-coded in the source code, which, without the context, does not give away the developer's intent in using that number. A proposed solution to this anti-pattern is to define a `constant` with a meaningful name instead of the magic number.

An example of this behavior is the number `10^20` in the `setPercentage(...)` function. Since it represents the maximum royalty percentage, it could be replaced with a constant `MAX_ROYALTY_PERCENTAGE`, which is much easier to understand and change in the future.

Another occurrence is in the `withdraw(...)` function, which defines a hardcoded variable `totalSupply`, representing the total supply of the Royalty token. This value is also hardcoded in the `RoyaltyToken` contract. Maintaining this value in multiple places increases the risk of inconsistencies, as one value can be updated without the other, potentially leading to inaccuracies in royalty computations. This can be fixed by storing this value in a constant that both functions will use.

Recommendation(s): Consider implementing the fixes to the abovementioned issues.

Status: Fixed.

6.15 [Best Practices] Lack of input validation when creating a collection

File(s): CollectionFactory.sol

Description: The function `createGenerativeCollection(...)` is used by users to create and deploy their own NFT collections. To do so, the user must provide the collection metadata, which, among others, contains information about the sale/presale start and end times and maximum token mints in each period. The lack of input validation makes the following scenarios possible:

- Both the sale and presale can be started in the past because there is no check that the start time is greater than the current `block.timestamp`;
- The presale period can overlap the sale period;
- The lack of `address(0)` check on the `feeRecipient` might result in a loss of fees if the collection creator forgets to set this address correctly, as it is not possible to update the `feeRecipient` address after the collection is created;
- The presale that starts in the past (timestamp of `0`) can be started with a `presaleMaxMint` of `0`;

Recommendation(s): Consider improving the input validation on the user-provided input to prevent the above scenarios from happening.

Status: Acknowledged.

6.16 [Best Practices] Redundant check for array length in the `getWithdrawn(...)` function

File(s): RoyaltyRegistrarImpl.sol

Description: The `getWithdrawn(...)` function within the `RoyaltyRegistrarImpl` contract returns the withdrawn amounts by the `msg.sender` for a provided list of host contracts with their respective currencies. The function checks that the provided arrays have the same length and that both lengths differ from zero. The check of the second array's length is redundant and can be safely removed as the two arrays have the same size.

```
function getWithdrawn(address[] calldata hostContracts, address[] calldata currencies)
  external
  view
  override
  returns (uint256[] memory)
{
  require(hostContracts.length == currencies.length, "Different length input");
  // @audit checking the second array length is unnecessary
  require(hostContracts.length != 0 && currencies.length != 0, "Zero length");
  // ...
}
```

Recommendation(s): Remove the redundant check for `currencies.length != 0`.

Status: Fixed.

Update from the client: Fixed exactly as recommended.

6.17 [Best Practices] Unused code

File(s): *

Description: The codebase contains some unused code, affecting overall readability and code quality. A list of files and their unused declarations are shown below:

```
GenerativeCollection.sol
    Counters.Counter _tokenIds

RoyaltyRegistrar.sol
    event WithdrawAll

RoyaltyRegistrarImpl.sol
    function claimRoyaltyForRevoked(...) -> IContract hostContract

GenerativeCollection.sol
    function _setTokenURI(uint256 tokenId, string memory _tokenURI)
```

Recommendation(s): Consider removing the unused code shown above.

Status: Fixed.

Update from the client: Removed.

6.18 [Best Practices] Use `_safeMint(...)` instead of `_mint(...)` in `OnlyOwnerCollection`

File(s): `OnlyOwnerCollection.sol`

Description: The `mintNFT(...)` function mints an NFT of `OnlyOwnerCollection`. However, when the function is called, it mints an NFT via the internal function `_mint(...)`. This is not a safe approach, as the contract should ensure that the recipient can manage NFTs.

For that purpose, the `_safeMint(...)` should be used to check if the receiver can handle NFTs. This is a good practice to mitigate any possible lock of NFT in some contracts.

Recommendation(s): Use `_safeMint` instead of `_mint` to mitigate any possible lock of NFT in the recipient contract.

Status: Acknowledged.

Update from the client: Pending solution, might delete this contract completely soon.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Blockframe documentation

Documentation, mainly in the form of Natspec and comments, was found within the contracts. Additionally, the Blockframe team participated in several meetings to facilitate our understanding of the protocol. However, the documentation for the reviewed contracts lacks completeness and details, falling short of providing a comprehensive understanding.

8 Test Suite Evaluation

8.1 Compilation Output

```
npx hardhat compile
Downloading zksolc 1.3.17
zksolc version 1.3.17 successfully downloaded
Compiling contracts for zkSync Era with zksolc v1.3.17 and solc v0.8.19
Compiling 122 Solidity files
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪ "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪ non-open-source code. Please see https://spdx.org for more information.
--> contracts/accountAbstraction/AAFactory.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪ "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪ non-open-source code. Please see https://spdx.org for more information.
--> contracts/sealed-bids/Bid.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪ "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪ non-open-source code. Please see https://spdx.org for more information.
--> contracts/sealed-bids/BidFactory.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/accountAbstraction/Paymaster.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/ERC6551Registry.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/SimpleERC6551Account.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/interfaces/IERC6551Account.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/interfaces/IERC6551Executable.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/interfaces/IERC6551Registry.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/lib/ERC6551AccountLib.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/creatorStudio/ERC6551/lib/ERC6551BytecodeLib.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/currency/CurrencyManager.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/governance/BlockFrameCollections.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/governance/BlockFrameDAO.sol
```

```
Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/governance/BlockFrameToken.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/governance/TimeLock.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/interfaces/IBlockFrameToken.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/interfaces/ICurrencyManager.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/BlockFrameAirdrop.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/Incentives.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/IncentivesWithPaymaster.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/TokenDistributor.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/TokenSplitter.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/TradingRewardsDistributor.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/VestingContract.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.19;"
--> contracts/tokenDistribution/VotingRewardsDistributor.sol


Warning: This declaration shadows an existing declaration.
  --> @matterlabs/zksync-contracts/l2/system-contracts/libraries/Utils.sol:85:9:
   |
85 |         uint256 bytecodeLenInWords = _bytecode.length / 32;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> @matterlabs/zksync-contracts/l2/system-contracts/libraries/Utils.sol:43:5:
   |
43 |     function bytecodeLenInWords(bytes32 _bytecodeHash) internal pure returns (uint256 codeLengthInWords) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: This declaration shadows an existing declaration.
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:92:47:
   |
92 |     function setBaseURI(bytes32 _baseUriHash, string memory _baseURI) public {
   |                                               ^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:81:5:
   |
81 |     function _baseURI() internal view override returns (string memory) {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

```
Warning: Unused local variable.
   --> contracts/RoyaltyRegistrarImpl.sol:161:9:
    |
161 |         IContract hostContract = IContract(_hostContract);
    |         ^^^^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:304:31:
    |
304 |     function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)
    |                               ^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:304:49:
    |
304 |     function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)
    |                                                 ^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:304:63:
    |
304 |     function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)
    |                                                               ^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:304:80:
    |
304 |     function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)
    |                                                                                ^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:312:32:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |                                ^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:312:50:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |                                                  ^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:312:64:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |                                                                ^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:312:76:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |                                                                            ^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:312:91:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |                                                                                           ^^^^^^^^^^^^^^^^^^^
```

```
Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:321:9:
    |
321 |         address operator,
    |         ^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:322:9:
    |
322 |         address from,
    |         ^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:323:9:
    |
323 |         uint256[] calldata ids,
    |         ^^^^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:324:9:
    |
324 |         uint256[] calldata values,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> contracts/sealed-bids/DRA.sol:325:9:
    |
325 |         bytes calldata data
    |         ^^^^^^^^^^^^^^^^^^^


Warning: Function state mutability can be restricted to pure
   --> contracts/sealed-bids/DRA.sol:304:5:
    |
304 |     function onERC721Received(address operator, address from, uint256 tokenId, bytes calldata data)
    |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
   --> contracts/sealed-bids/DRA.sol:312:5:
    |
312 |     function onERC1155Received(address operator, address from, uint256 id, uint256 value, bytes calldata data)
    |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
   --> contracts/sealed-bids/DRA.sol:320:5:
    |
320 |     function onERC1155BatchReceived(
    |     ^ (Relevant source part starts here and spans across multiple lines).



----------------------------------------------------------------------------------------------------
| Warning: It looks like you are using 'ecrecover' to validate a signature of a user account.      |
| zkSync Era comes with native account abstraction support, therefore it is highly recommended NOT |
| to rely on the fact that the account has an ECDSA private key attached to it since accounts might|
| implement other signature schemes.                                                               |
| Read more about Account Abstraction at https://v2-docs.zksync.io/dev/developer-guides/aa.html    |
----------------------------------------------------------------------------------------------------
--> @openzeppelin/contracts/utils/cryptography/ECDSA.sol
```

```
---------------------------------------------------------------------------------------------
| Warning: It looks like you are using 'ecrecover' to validate a signature of a user account.     |
| zkSync Era comes with native account abstraction support, therefore it is highly recommended NOT |
| to rely on the fact that the account has an ECDSA private key attached to it since accounts might|
| implement other signature schemes.                                                               |
| Read more about Account Abstraction at https://v2-docs.zksync.io/dev/developer-guides/aa.html    |
---------------------------------------------------------------------------------------------

--> contracts/accountAbstraction/Account.sol



---------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                             |
| This might be a false positive if you are using an interface (like IERC20) instead of the        |
| native Solidity `send/transfer`.                                                                 |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas       |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                          |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                  |
---------------------------------------------------------------------------------------------
--> contracts/creatorStudio/factory/GenerativeCollection.sol



---------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                             |
| This might be a false positive if you are using an interface (like IERC20) instead of the        |
| native Solidity `send/transfer`.                                                                 |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas       |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                          |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                  |
---------------------------------------------------------------------------------------------
--> contracts/creatorStudio/factory/GenerativeCollection.sol



---------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                             |
| This might be a false positive if you are using an interface (like IERC20) instead of the        |
| native Solidity `send/transfer`.                                                                 |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas       |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                          |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                  |
---------------------------------------------------------------------------------------------
--> contracts/creatorStudio/factory/GenerativeCollection.sol



---------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                             |
| This might be a false positive if you are using an interface (like IERC20) instead of the        |
| native Solidity `send/transfer`.                                                                 |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas       |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                          |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                  |
---------------------------------------------------------------------------------------------
--> contracts/creatorStudio/factory/GenerativeCollection.sol

---------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                             |
| This might be a false positive if you are using an interface (like IERC20) instead of the        |
| native Solidity `send/transfer`.                                                                 |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas       |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                          |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                  |
---------------------------------------------------------------------------------------------
--> contracts/sealed-bids/Bid.sol
```

```
----------------------------------------------------------------------------------------------------
| Warning: It looks like you are using '<address payable>.send/transfer(<X>)' without providing    |
| the gas amount. Such calls will fail depending on the pubdata costs.                              |
| This might be a false positive if you are using an interface (like IERC20) instead of the         |
| native Solidity `send/transfer`.                                                                  |
| Please use 'payable(<address>).call{value: <X>}("")' instead, but be careful with the reentrancy |
| attack. `send` and `transfer` send limited amount of gas that prevents reentrancy, whereas        |
| `<address>.call{value: <X>}` sends all gas to the callee. Learn more on                           |
| https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy                   |
----------------------------------------------------------------------------------------------------
--> contracts/sealed-bids/Bid.sol

----------------------------------------------------------------------------------------------------
| Warning: Your code or one of its dependencies uses the 'extcodesize' instruction, which is       |
| usually needed in the following cases:                                                            |
|   1. To detect whether an address belongs to a smart contract.                                    |
|   2. To detect whether the deploy code execution has finished.                                    |
| zkSync Era comes with native account abstraction support (so accounts are smart contracts,       |
| including private-key controlled EOAs), and you should avoid differentiating between contracts   |
| and non-contract addresses.                                                                       |
----------------------------------------------------------------------------------------------------
--> erc721a-upgradeable/contracts/ERC721A__Initializable.sol

Successfully compiled 122 Solidity files
```

```
forge build
[] Compiling...
[] Compiling 158 files with 0.8.20
[] Solc 0.8.20 finished in 7.50s
Compiler run successful with warnings:
Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/accountAbstraction/TokenPaymaster.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/ERC6551/ERC6551Registry.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/ERC6551/SimpleERC6551Account.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/EditableGenerativeCollection.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/Operation.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/Traits.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/tokenDistribution/IncentivesWithPaymaster.sol

Warning (2519): This declaration shadows an existing declaration.
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:84:47:
   |
84 |     function setBaseURI(bytes32 _baseUriHash, string memory _baseURI) public {
   |                                               ^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:73:5:
   |
73 |     function _baseURI() internal view override returns (string memory) {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2072): Unused local variable.
   --> contracts/RoyaltyRegistrarImpl.sol:131:7:
    |
131 |       IContract hostContract = IContract(_hostContract);
    |       ^^^^^^^^^^^^^^^^^^^^^^
```

```
Warning (2072): Unused local variable.
  --> contracts/tokenDistribution/Incentives.sol:52:23:
   |
52 |           (bool sent, bytes memory data) = caller.call{value: ethReward}("");
   |                       ^^^^^^^^^^^^^^^^^
```

8.2 Tests Output

```
forge test
[] Compiling...
[] Compiling 158 files with 0.8.20
[] Solc 0.8.20 finished in 7.50s
Compiler run successful with warnings:
Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/accountAbstraction/TokenPaymaster.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/ERC6551/ERC6551Registry.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/ERC6551/SimpleERC6551Account.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/EditableGenerativeCollection.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/Operation.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/creatorStudio/factory/Traits.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> contracts/tokenDistribution/IncentivesWithPaymaster.sol

Warning (2519): This declaration shadows an existing declaration.
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:84:47:
   |
84 |     function setBaseURI(bytes32 _baseUriHash, string memory _baseURI) public {
   |                                               ^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/creatorStudio/factory/GenerativeCollection.sol:73:5:
   |
73 |     function _baseURI() internal view override returns (string memory) {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2072): Unused local variable.
   --> contracts/RoyaltyRegistrarImpl.sol:131:7:
    |
131 |       IContract hostContract = IContract(_hostContract);
    |       ^^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> contracts/tokenDistribution/Incentives.sol:52:23:
   |
52 |           (bool sent, bytes memory data) = caller.call{value: ethReward}("");
   |                       ^^^^^^^^^^^^^^^^^


Running 1 test for test/RoyaltyToken.t.sol:RoyaltyTokenTest
[PASS] testClaimRoyalty() (gas: 884554)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.38ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer