SeaDrop Security Review

Auditors

Harikrishnan Mulackal, Lead Security Researcher

Sawmon and Natalie, Lead Security Researcher

Dravee, Security Researcher

Devansh Bantham, Apprentice

Parth Patel, Apprentice

Report prepared by: Pablo Misirov

November 8, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

SeaDrop is a contract to perform primary drops on evm-compatible blockchains. The types of drops supported are public drops, allow list stages, token gated drops, and server-side signed mints. An implementing token contract should contain the methods to interface with SeaDrop through an authorized user such as an Owner or Administrator.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of seadrop according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 5 days in total, OpenSea engaged with Spearbit to review SeaDrop. In this period of time a total of 51 issues were found.

Summary

Project Name	OpenSea
Repository	SeaDrop
Commit	284e8075...28adb
Type of Project	Marketplace, NFT
Audit Timeline	Aug 24th - Aug 29th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	3
Medium Risk	8
Low Risk	8
Gas Optimizations	10
Informational	22
Total Issues	51

5 Findings

5.1 High Risk

5.1.1 An allowed signer can sign mints with malicious parameters

Severity: *High Risk*

Context: SeaDrop.sol#L259-L266, SeaDrop.sol#L318-L319

Description: An allowed signer (SeaDrop.sol#L318-L319) can sign mints that have either:

- mintParams.feeBps equal to 0.
- A custom feeRecipient with mintParams.restrictFeeRecipients equal to false to circumvent the check at SeaDrop.sol#L469.

And thus avoid the protocol fee being paid or allow the protocol fee to be sent to a desired address decided by the signer.

Note that the ERC721SeaDrop owner can allow signers by calling ERC721SeaDrop.updateSigner. Therefore, the owner can allow an address they control as a signer and sign mints that have either one of the above features.

OpenSea: This is correct; currently any signer would have ultimate control around the parameters of a mint, and this should be understood by parties who wish to use a centralized signer, ie, self-hosted or in a legal agreement with a marketplace

However, we could make it slightly less "trustful" by storing a struct of validation params rather than a simple bool in the mapping

```
struct SignedMintParams {
    uint80 minMintPrice;
    uint24 maxMaxTotalMintableByWallet;
    uint48 minStartTime;
    uint48 maxEndTime;
    uint40 maxMaxTokenSupplyForStage;
    uint16 maxFeeBps;
}
```

and always assume restrictFeeRecipients == true.

Spearbit: That could work. If this solution is implemented, all the instances of mintParams.<FIELDS> would need to be replaced by the stored (storage) parameters in this function.

Also, a question comes up as to who would have the authority to set SignedMintParams based on the current architecture.

Is there a reason you didn't include dropStageIndex in the SignedMintParams struct?

In the above SignedMintParams struct, the last field is named maxFeeBps. Was that intentional or did you meant to name it feeBps?

OpenSea dropStageIndex is purely informational for metrics-purposes as part of the SeaDropMint event (we want to be able to see which addresses redeem allow-lists at which stage, etc)

In the case of SignedMintParams, the Owner would set it, though for partnered drops, the fee-setting pattern would likely be the same as elsewhere. (Pending confirmation from legal) OpenSea would initialize a signer with a maxFeeBps (which still requires trust that we don't set it to a higher-than-agreed-upon value), and the Owner can then submit the rest of the parameters.

maxFeeBps would allow variable feeBps - which is probably a rare use-case, but was a requirement from product for allow-list tiers, which we applied to the other mint methods. Enforcing a maxFeeBps, of course, includes the caveat that it would not prevent a malicious signer from always specifying the maximum fee rate. The Owner should specify maxFeeBps to ensure that a signer cannot specify a feeBps larger than the largest acceptable feeBps. (The signer would be free to specify a lower feeBps, which I'm sure a creator would appreciate)

In the general case, if the `Owner` changes an allowed signer's `maxFeeBps` (or any other mint parameter) to a value that is no longer acceptable, the signer can refuse to sign mints.

5.1.2 `ERC721SeaDrop`'s modifier `onlyOwnerOrAdministrator` would allow either the `owner` or the `admin` to override the other person's config parameters.

Severity: *High Risk*

Context:

- ERC721SeaDrop.sol#L106,
- ERC721SeaDrop.sol#L212,
- ERC721SeaDrop.sol#L289,
- ERC721SeaDrop.sol#L345

Description: The following 4 `external` functions in `ERC721SeaDrop` have the `onlyOwnerOrAdministrator` modifier which allows either one to override the other person's work.

- `updateAllowedSeaDrop`
- `updateAllowList`
- `updateDropURI`
- `updateSigner`

That means there should be some sort of off-chain trust established between these 2 entities. Otherwise, there are possible vectors of attack.

Here is an example of how the `owner` can override `AllowListData.merkleRoot` and the other fields within `AllowListData` to generate proofs for any allowed `SeaDrop`'s `mintAllowList` endpoint that would have `MintParams.feeBps` equal to `0`:

1. The `admin` calls `updateAllowList` to set the Merkle root for an allowed `SeaDrop` implementation for this contract and emit the other parameters as logs. The `SeaDrop` endpoint being called by `ERC721SeaDrop.updateAllowList`: SeaDrop.sol#L827
2. The `owner` calls `updateAllowList` but this time with new parameters, specifically a new Merkle root that is computed from leaves that have `MintParams.feeBps == 0`.
3. Users/minters use the generated proof corresponding to the latest allow list update and pass their `mintParams.feeBps` as `0`. And thus avoiding the protocol fee deduction for the `creatorPaymentAddress` (SeaDrop.sol#L187-L194).

Recommendation: Only use this implementation of `IERC721SeaDrop` if there is already a legal off-chain contract and level of trust between the different parties. Otherwise, a different implementation with a stricter separation of roles is recommended.

OpenSea: This is related to specific legal/BD requirements - we need to be able to administer the contract for Partners (some may choose to administer it themselves), but for legal clarity, they also need to unambiguously be the "owner" of the contract, in that they have the power to administer it as well.

In practice, in this implementation of the contract, both parties should be considered trusted, but also ideally shouldn't have privileges that overstep their bounds (in particular, fee and creator payouts)

This contract is intended to be used as the basis for our first few partnered primary mints. As such, there are some assumptions and particular tailored logic to meet our and our partners' needs. (In hindsight, it might have made more sense to split out into a more-generic ERC721SeaDrop, and more-specific ERC721PartnerSeaDrop)

Assumptions:

- OpenSea will be collecting a fee
- There is a good deal of trust (ie, legal contracts) established between the two parties

- Some Partners will prefer (or require) us configure drop mechanics and metadata
  - This is why some functions are `onlyOwnerOrAdministrator`

Requirements, passed down from legal:

- OpenSea is the "Administrator"
- The Partner is the "Owner"
- The Partner is the only entity in control of the pricing of the general drop and the creator payout address
- OpenSea is the only entity that can update fees and fee recipients

You are correct that this requires trust between the two parties. As mentioned elsewhere, in general, an administrator will not be necessary for all token contracts.

In practice, a marketplace (OpenSea) will have to decide whether or not to provide a proof for a mint transaction depending on the allowed fee recipients and specified feeBps off-chain.

Spearbit: Acknowledged.

5.1.3 Reentrancy of fee payment can be used to circumvent max mints per wallet check

Severity: *High Risk*

Context: SeaDrop.sol#L586

Description: In case of a `mintPublic` call, the function `_checkMintQuantity` checks whether the minter has exceeded the parameter `maxMintsPerWallet`, among other things. However, re-entrancy in the above fee dispersal mechanism can be used to circumvent the check.

The following is an example contract that can be employed by the `feeRecipent` (assume that `maxMintsPerWallet` is 1):

```
contract MaliciousRecipient {
    bool public startAttack;
    address public token;
    SeaDrop public seaDrop;

    fallback() external payable {
        if (startAttack) {
            startAttack = false;
            seaDrop.mintPublic{value: 1 ether}({
                nftContract: token,
                feeRecipient: address(this),
                minterIfNotPayer: address(this),
                quantity: 1
            });
        }
    }

    // Call `attack` with at least 2 ether.
    function attack(SeaDrop _seaDrop, address _token) external payable {
        token = _token;
        seaDrop = _seaDrop;
        startAttack = true;

        _seaDrop.mintPublic{value: 1 ether}({
            nftContract: _token,
            feeRecipient: address(this),
            minterIfNotPayer: address(this),
            quantity: 1
        });

        token = address(0);
        seaDrop = SeaDrop(address(0));
    }
}
```

This is especially bad when the parameter `PublicDrop.restrictFeeRecipients` is set to `false`, in which case, anyone can circumvent the max mints check, making it a high severity issue. In the other case, only privileged users, i.e., should be part of `_allowedFeeRecipients[nftContract]` mapping, would be able to circumvent the check--lower severity due to needed privileged access.

Also, `creatorPaymentAddress` can use re-entrancy to get around the same check. See SeaDrop.sol#L571.

Recommendation: There are two ways to fix the above issue:

1. Code paths that disperse the ETH as fees should have reentrancy locks set.
2. Change `safeTransferETH` to use `.transfer` that only forwards "call stipend" amount of gas to the sub-call. This may break some smart contracts wallets from receiving the ETH.

OpenSea: Added reentrancy lock (+ test), and (before this commit) mint was re-arranged to be before payment. See commit 160c034.

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 Cross SeaDrop reentrancy

Severity: *Medium Risk*

Context: SeaDrop.sol#L586

Description: The contract that implements `IERC721SeaDrop` can work with multiple Seadrop implementations, for example, a Seadrop that accepts ETH as payment as well as another Seadrop contract that accepts USDC as payment at the same time. This introduces the risk of cross contract re-entrancy that can be used to circumvent the `maxMintsPerWallet` check.

Here's an example of the attack:

1. Consider an ERC721 token that that has two allowed `SeaDrop`, one that accepts ETH as payment and the other that accepts USDC as payment, both with public mints and `restrictedFeeRecipients` set to `false`.
2. Let `maxMintPerWallet` be `1` for both these cases.
3. A malicious fee receiver can now do the following:
   - Call `mintPublic` for the Seadrop with ETH fees, which does the `_checkMintQuantity` check and transfers the fees in ETH to the receiver.
   - The receiver now calls `mintPublic` for Seadrop with USDC fees, which does the `_checkMintQuantity` check that still passes.
   - The mint succeeds in the Seadrop-USDC case.
   - The mint succeeds in the Seadrop-ETH case.
   - The minter has 2 NFTs even though it's capped at 1.

Even if a re-entrancy lock is added in the SeaDrop, the same issue persists as it only enters each Seadrop contract once.

Recommendation: Consider adding a reentrancy lock in the ERC-721 contract. Also see the related issue *Reentrancy of fee payment can be used to circumvent max mints per wallet check* about reentrancy.

5.2.2 Lack of replay protection for `mintAllowList` and `mintSigned`

Severity: *Medium Risk*

Context: SeaDrop.sol#L227, SeaDrop.sol#L318

Description: In the case of `mintSigned` (minting via signatures) and `mintAllowList` (minting via merkle proofs) there are no checks that prevent re-using the same signature or Merkle proof multiple times. This is indirectly enforced by the `_checkMintQuantity` function that checks the mint statistics using `IERC721SeaDrop(nftContract).getMintStats(minter)` and reverting if the quantity exceeds `maxMintsPerWallet`.

Replays can happen if a wallet does not claim all of `maxMintsPerWallet` in one transaction. For example, assume that `maxMintsPerWallet` is set to 2. A user can call `mintSigned` with a valid signature and `quantity = 1` twice.

Typically, contracts try to avoid any forms of signature replays, i.e., a signature can only be used once. This simplifies the security properties. In the current implementation of the `ERC721Seadrop` contract, we couldn't see a way to exploit replay protection to mint beyond what could be minted in a single initial transaction with the maximum value of `quantity` supplied. However, this relies on the contract correctly implementing `IERC721SeaDrop.getMintStats`.

Recommendation: We recommend implementing replay protection for both cases. Here are some ideas to do this:

1. Consider also including the `tokenId` for the signature and passing that along in `mintSeaDrop` call. This way, even if the signature is replayed, minting the same `tokenId` should not be possible--most ERC-721 libraries prevent this. However, some care should be made to check the following case: mint a fixed token id using the signature, then burn the token id, and resurrecting the same token id by replaying the signature.

2. Consider storing the digest and if a digest is used once, then it shouldn't be able to use again.
3. Do not use signature as a way to check if something was consumed. They are malleable.

OpenSea: We discussed replay-protection here, and decided it was a more or less acceptable risk for the following reasons:

1. Allow-lists, which also `_checkMintQuantity` are likewise not redeemed, so Merkle proofs can be re-used in the same way, up to the maximum mint quantity
2. Also like allow-lists, the supplied MintParams specify a `startTime` and `endTime`; a signature can supply a short window (minutes) for consumption before a new signature needs to be generated
3. A broken `_checkMintQuantity` or unreasonably large `maxTokensMintable` quantity is likely (though not always) exploitable in the first time a signature (or Merkle proof) is used

However! Riffing off of the `tokenId` suggestion (We don't think it's possible to know exactly which starting token ID a given tx will mint), since we're already checking `minterNumMinted`; we could include that as part of the signature to prevent re-use.

Spearbit: 2. A malicious user can always get around the `startTime` and `endTime` limits, using some automation. 3. We think that most ERC-721 contracts would assume that Opensea would handle the signature verification and replay protection--the burden of the sale mechanism should be on the Seadrop contract. Also, because this requires the ERC-721 contract to keep track of the number of the number of tokens minted by an address. ERC721A tracks this, but neither Solmate, nor Openzeppelin does this currently. We'd expect some user errors because of this problem.

5.2.3 The `digest` in `SeaDrop.mintSigned` is not calculated correctly according to `EIP-712`

Severity: *Medium Risk*

Context: SeaDrop.sol#L308

Description: `mintParams` in the calculation of the `digest` in `mintSigned` is of `struct` type, so we would need to calculate and use its `hashStruct` , not the actual variable on its own.

Recommendation: According to EIP-712 the correct `digest` would be:

```
// include this typehash at the top of the contract
bytes32 internal constant _MINT_PARAMS_TYPEHASH = keccak256(
    "MintParams("
        "uint256 mintPrice,"
        "uint256 maxTotalMintableByWallet,"
        "uint256 startTime,"
        "uint256 endTime,"
        "uint256 dropStageIndex,"
        "uint256 maxTokenSupplyForStage,"
        "uint256 feeBps,"
        "bool restrictFeeRecipients"
    ")"
);

...
// hashStruct for mintParams
bytes32 mintParamsHashStruct = keccak256(
    abi.encode(
        _MINT_PARAMS_TYPEHASH,
        mintParams.mintPrice,
        mintParams.maxTotalMintableByWallet,
        mintParams.startTime,
        mintParams.endTime,
        mintParams.dropStageIndex,
        mintParams.maxTokenSupplyForStage,
        mintParams.feeBps,
        mintParams.restrictFeeRecipients
    )
);

bytes32 digest = keccak256(
    abi.encodePacked(
        // EIP-191: `0x19` as set prefix, `0x01` as version byte
        bytes2(0x1901),
        _domainSeparator(),
        keccak256(
            abi.encode(
                _SIGNED_MINT_TYPEHASH,
                nftContract,
                minter,
                feeRecipient,
                mintParamsHashStruct  // <--- correction
            )
        )
    )
);
```

This wasn't caught in the test because the test re-uses the same digest calculation. It would be nice to also test it against an external `EIP-712` signature calculation.

OpenSea: Thanks, digest was fixed and ethers EIP-712 `signTypedData` has been used to verify in added unit tests here and here.

Spearbit: Acknowledged.

5.2.4 Token gated drops with a self-allowed `ERC721SeaDrop` or a variant of that can lead to the drop getting drained by one person.

Severity: *Medium Risk*

Context: SeaDrop.sol#L345

Description: There are scenarios where an actor with only 1 token from an allowed NFT can drain Token Gated Drops that are happening simultaneously or back to back.

- Scenario 1 - An `ERC721SeaDrop` is registered as an `allowedNftToken` for itself

This is a simple example where an `ERC721SeaDrop` N is registered by the `owner` or the `admin` as an `allowedNftToken` for its own token gated drop and during or before this drop (let's call this drop D) there is another token gated drop (D') for another `allowedNftToken` N', which does not need to be necessarily an `IERC721SeaDrop` token. Here is how an actor can drain the self-registered token gated drop:

1. The actor already owns or buys an N' token t' with wallet w_0.
2. During D', the actor mints an N token t_0 with wallet w_0 passing N', t' to `mintAllowedTokenHolder` and transfer t_0 to another wallet if necessary to avoid the max mint per wallet limit (call this wallet w_1 which could still be w_0).
3. Once D starts or if it is already started, the actor mints another N token t_1 with w_1 passing N, t_0 to `mintAllowedTokenHolder` and transfers t_1 to another wallet if necessary to avoid the max mint per wallet limit (call this wallet w_2 which could still be w_1)
4. Repeat step 3 with the new parameters till we hit the `maxTokenSupplyForStage` limit for D.

```
# during token gated drop D'
t = seaDrop.mintAllowedTokenHolder(N, f, w, {N', [t']})

# during token gated drop D
while ( have not reached maxTokenSupplyForStage ):
   if ( w has reached max token per wallet ):
      w' = newWallet()
      N.transfer(w, w', t)
      w = w'
   t = seaDrop.mintAllowedTokenHolder(N, f, w, {N, [t]})
```

- Scenario 2 - Two `ERC721SeaDrop` tokens are doing a simultaneous token gated drop promotion

In this scenario, there are 2 `ERC721SeaDrop` tokens N_1, N_2 where they are running simultaneous token gated drop promotions. Each is allowing a wallet/bag holder from the other token to mint a token from their project. So if you have an N_1 token you can mint an N_2 token and vice versa. Now if an actor already has an N_1 or N_2 token maybe from another token gated drop or from an allow list mint, they can drain these 2 drops till one of them hits `maxTokenSupplyForStage` limit.

```
# wallet <w> already holds token <t> from N1

while ( have not reached N1.maxTokenSupplyForStage or  N2.maxTokenSupplyForStage):

   w = newWalletIfMaxMintReached(N1, w, t) # this also transfers t to the new wallet
   w = newWalletIfMaxMintReached(N2, w, t) # this also transfers t to the new wallet

   t = seaDrop.mintAllowedTokenHolder(N2, f, w, {N1, [t]})
   t = seaDrop.mintAllowedTokenHolder(N1, f, w, {N2, [t]})
```

This scenario can be extended to more complex systems, but the core logic stays the same.

Also, it's good to note that in general `maxTotalMintableByWallet` for token gated drops and `maxMintsPerWallet` for public mints are not fully enforceable since actors can either distribute their allowed tokens between multiple wallets to mint to their full potential for the token gated drops. And for public mints, they would just use different wallets. It does add extra gas for them to mint since they can't batch mint. That said these limits are enforceable for the signed and allowed mints (or you could say the enforcing has been moved to some off-chain mechanism)

OpenSea: In scenario 1, I think a check against allowing a token to register itself as an allowed token-gated-drop is reasonable.

In scenario 2, we could also check against allowing a token to register a second token as an allowed-token-gated-drop if that token's `currentSupply < maxSupply` and has the first token registered as its own token-gated drop. This has the caveat that a token could implement itself to have a changeable maxSupply, which would bypass these checks... open to other implementation ideas.

I think both cases should be documented in the comments

Spearbit: Agree with OpenSea regarding a check for a self-allowed token gated drop in scenario 1.

For scenario 2 or a more complex variant of it like (can be even more complex than below):

```
// N1, N2: IERC721SeaDrop tokens with token gated drop promotions
// Each arrow in the diagram below represents an allowed mint mechanism
// A -> B : a person with a t_a token from A can mint a token of B (B can potentially mark t_a as
↪   redeemed on mint)

 M0 -> N1 -> M1 -> M2 -> ... -> Mk ->
       N2 -> O1 -> O2 -> ... -> Oj -> N1
```

It would be hard to have an implementation that would check for these kind of behaviors. But we agree that documenting these scenarios in the comments would be great.

OpenSea: Added `error TokenGatedDropAllowedNftTokenCannotBeDropToken()` and added comments for scenario no 2. See commit 0a91de9.

5.2.5 `ERC721A` has mint caps that are not checked by `ERC721SeaDrop`

Severity: *Medium Risk*

Context: ERC721SeaDrop.sol#L137-L145

Description: `ERC721SeaDrop` inherits from `ERC721A` which packs `balance`, `numberMinted`, `numberBurned`, and an extra data chunk in 1 storage slot (64 bits per substorage) for every address. This would add an inherent cap of $2^{64} - 1$ to all these different fields. Currently, there is no check in `ERC721A`'s `_mint` for `quantity` nor in `ERC721SeaDrop`'s `mintSeaDrop` function.

Also, if we almost reach the max cap for a `balance` by an owner and someone else transfers a token to this owner, there would be an overflow for the `balance` and possibly the number of mints in the `_packedAddressData`. The overflow could possibly reduce the `balance` and the `numberMinted` to a way lower numer and `numberBurned` to a way higher number

Recommendation: We should have an additional check if `quantity` would exceed the mint cap in `mintSeaDrop`.

OpenSea: We will add checks around ERC721A limits. We have added a restraint that `maxSupply` cannot be set to greater than $2^{64} - 1$ so balance nor number minted can exceed this. See the commit 5a98d29.

5.2.6 `ERC721SeaDrop owner` can choose an address they control as the `admin` when the constructor is called.

Severity: *Medium Risk*

Context: ERC721SeaDrop.sol#L83

Description: The `owner/creator` can call the contract directly (skip using the UI) and set the `administrator` as themselves or another address that they can control. Then after they create a `PublicDrop` or `TokenGatedDrop`, they can call either `updatePublicDropFee` or `updateTokenGatedDropFee` and set the `feeBps` to

- zero
- or another number and also call the `updateAllowedFeeRecipient` to add the same or another address they control as a `feeRecipient`.

This way they can circumvent the protocol fee.

Recommendation: Consider implementing the following suggestions

- Not list NFT contracts on the marketplace site that have an administrator who is not in an internal allowed list.
- Or let each allowed SeaDrop implementation's admin/operator to set the admins for the `ERC721SeaDrop` contract. Although, this can still be possibly rigged by a custom hand-craftedd contract that pretends to be an `ERC721SeaDrop` contract.
- `SeaDrop` can have its own set of `admins` independent of the `IERC721SeaDrop` tokens. These `admins` should be able to set the `feeRecipients` and `feeBps` on `SeaDrop` without interacting with the original token.

OpenSea: In practice, this particular implementation will be deployed by OpenSea or a trusted Partner.

In general, an `Administrator` is not required of `ERC721SeaDrop` contracts; OpenSea will ingest events and data, and then selectively decide which mints to surface and fulfill, depending on mint parameters.

In other words, more generally, it's up to an individual marketplace to decide which mints they are willing to list and fulfill, and that decision making happens off-chain

Spearbit: I guess the listing and fulfillment on the OpenSea side is just about the OpenSea marketplace UI. But for example, other aggregators that listen to events from OpenSea deployed `SeaDrops` can/could list these

ERC721SeaDrop on their marketplace. And obviously, users can still interact with the OpenSea deployed SeaDrops directly.

5.2.7 ERC721SeaDrop's admin would need to set feeBps manually after/before creation of each drop by the owner

Severity: *Medium Risk*

Context: ERC721SeaDrop.sol#L180, ERC721SeaDrop.sol#L256

Description: When an owner of a ERC721SeaDrop token creates either a public or a token gated drop by calling updatePublicDrop or updateTokenGatedDrop, the PublicDrop.feeBps/TokenGatedDropStage.feeBps is initially set to 0. So the admin would need to set the feeBps parameter at some point (before or after). Forgetting to set this parameter results in not receiving the protocol fees.

Recommendation: There are mutiple ways to mitigate this:

1. The admin monitors the activities on-chain and if it sees a newly created drop, calls either updatePublicDropFee or updateTokenGatedDropFee (depending on the type of the drop) to set the feeBps.
2. Enforcing that both updatePublicDrop and updatePublicDropFee (or updateTokenGatedDrop and updateTokenGatedDropFee) be called by the owner and the admin before a drop can start. The enforcement can be either on the ERC721SeaDrop side or on the SeaDrop side. Also, there could be a flag set by the admin to waive the protocol fee.

5.2.8 owner can reset feeBps set by admin for token gated drops

Severity: *Medium Risk*

Context: ERC721SeaDrop.sol#L233-L245, SeaDrop.sol#L860, SeaDrop.sol#L889-L890

Description: Only the admin can call updateTokenGatedDropFee to update feeBps. However, the owner can call updateTokenGatedDrop(address seaDropImpl, address allowedNftToken, TokenGatedDropStage calldata dropStage) twice after that to reset the feeBps to 0 for a drop.

1. Once with dropStage.maxTotalMintableByWallet equal to 0 to wipe out the storage on the SeaDrop side.
2. Then with the same allowedNftToken address and the other desired parameters, which would retrieve the previously wiped out drop stage data (with feeBps equal to 0).

NOTE: This type of attack does not apply to updatePublicDrop and updatePublicDropFee pair. Since updatePublicDrop cannot remove or update the feeBps. Once updatePublicDropFee is called with a specific feeBps that value remains for this ERC721SeaDrop contract-related storage on SeaDrop (_publicDrops[msg.sender] = publicDrop). And any number of consecutive calls to updatePublicDrop with any parameters cannot change the already set feeBps.

Recommendation: The admins could monitor all the activities for updateTokenGatedDrop calls even when the same old allowedNftToken is used and make sure to set the fees after each call if it is not a removal kind.

OpenSea: We can re-work it so that updateTokenGatedDropFee "initializes" a tokenGatedDrop stage (all params 0 besides feeBps and restrictFeeRecipients), and a partner is free to then edit other params and delete the stage, but not create a new one. I believe that would be a workaround for current issues.

Proposed workaround:

Administrator/OpenSea is the only authorized user that can "initialize" a TokenGatedDrop. Initializing a token-gated drop sets all params to zero except maxTotalMintableByWallet = 1 (struct will not be stored if == 0), feeBps, and restrictFeeRecipients = true. The parameter startTime = 0 means the stage will not be active, and cannot be made active by OpenSea.

The Owner/Partner can then update the initialized TokenGatedDrop stage (potentially including delete, if so desired, but it would need to be re-initialized with a fee by OpenSea).

5.3 Low Risk

5.3.1 Update the start token id for `ERC721SeaDrop` to `1`

Severity: *Low Risk*

Context: ERC721SeaDrop.sol#L144

Description: `ERC721SeaDrop`'s `mintSeaDrop` uses `_mint` from `ERC721A` library which starts the token ids for minting from `0`.

```
/// contracts/ERC721A.sol#L154-L156

/**
    * @dev Returns the starting token ID.
    * To change the starting token ID, please override this function.
    */
function _startTokenId() internal view virtual returns (uint256) {
    return 0;
}
```

Recommendation: Usually `0` is used to signal values that have not been set or have been removed, ... To avoid possible future problems consider using a different starting token id by overriding the `_startTokenId` in `ERC721SeaDrop`.

OpenSea: We can configure it to start at 1 as a QOL improvement.

Fixed in commit e14fa17.

Spearbit: Acknowledged.

5.3.2 Update the `ERC721A` library due to an unpadded `toString()` function

Severity: *Low Risk*

Context: ERC721SeaDrop.sol#L14, chiru-labs/ERC721A/contracts/ERC721A.sol#L1049

Description: The audit repo uses `ERC721A` at `dca00fffdc8978ef517fa2bb6a5a776b544c002a` which does not add a trailing zero padding to the returned string. Some projects have had issues reusing the `toString()` where the off-chain call returned some dirty-bits at the end (similar to Seaport 1.0's `name()`).

Recommendation: Consider upgrading to a version of `ERC721A1` with that fix, even then testing it would be great.

Ref: PR: Add trailing zeros padding to _toString

OpenSea: Fixed in commit 8441e94.

Spearbit: Acknowledged.

5.3.3 Warn contracts implementing `IERC721SeaDrop` to revert on `quantity == 0` case

Severity: *Low Risk*

Context: SeaDrop.sol#L620

Description: There are no checks in Seadrop that prevents minting for the case when `quantity == 0`. This would call the function `mintSeadrop(minter, quantity)` for a contract implementing `IERC721SeaDrop` with `quantity == 0`. It is up to the implementing contract to revert in such cases. The ERC721A library reverts when `quantity == 0`--the correct behaviour.

However, there has been instances in the past where ignoring `quantity == 0` checks have led to security issues.

Recommendation: There are two ways to fix this:

1. Seadrop reverts early when `quantity == 0`. This is never a valid input. As a reference, Seaport avoids any transfers of 0 amount. See TokenTransferrerErrors.sol#L18.

2. Warn contracts implementing `IERC721SeaDrop` to revert on `quantity == 0` case.

OpenSea: We have added error `MintQuantityCannotBeZero` to `_checkMintQuantity` in the commit 69f2854.

Spearbit: Acknowledged.

5.3.4 Missing parameter in `_SIGNED_MINT_TYPEHASH`

Severity: *Low Risk*

Context: SeaDrop.sol#L78, lib/SeaDropStructs.sol#L92

Description: A parameter is missing (`uint256 maxTokenSupplyForStage`) and got caught after reformatting.

Recommendation: Reformat these lines into:

```
bytes32 internal immutable _SIGNED_MINT_TYPEHASH =
    keccak256(
        "SignedMint("
            "address nftContract,"
            "address minter,"
            "address feeRecipient,"
            "MintParams mintParams"
        ")"
        "MintParams("
            "uint256 mintPrice,"
            "uint256 maxTotalMintableByWallet,"
            "uint256 startTime,"
            "uint256 endTime,"
            "uint256 dropStageIndex,"
            "uint256 maxTokenSupplyForStage," // <--- missing in the audit repo
            "uint256 feeBps,"
            "bool restrictFeeRecipients"
        ")"
    );
bytes32 internal immutable _EIP_712_DOMAIN_TYPEHASH =
    keccak256(
        "EIP712Domain("
            "string name,"
            "string version,"
            "uint256 chainId,"
            "address verifyingContract"
        ")"
    );
```

5.3.5 Missing `address(0)` check

Severity: *Low Risk*

Context: SeaDrop.sol#L856, SeaDrop.sol#L907-L909, SeaDrop.sol#L927-L929, SeaDrop.sol#L966-L968, ERC721SeaDrop.sol#L245

Description: All `update` functions having an address as an argument check them against `address(0)`. This is missing in updateTokenGatedDrop. This is also not protected in ERC721SeaDrop.sol#updateTokenGatedDrop(), so `address(0)` could pass as a valid value.

Recommendation: Consider adding `address(0)` checks for `allowedNftToken`

OpenSea: Fixed in commit 13deff0.

Spearbit: Acknowledged.

5.3.6 Missing boundary checks on `feeBps`

Severity: *Low Risk*

Context:

- ERC721SeaDrop.sol#L167,
- ERC721SeaDrop.sol#L192,
- ERC721SeaDrop.sol#L241,
- ERC721SeaDrop.sol#L272,
- SeaDrop.sol#L554-L557

Description: There's a missing check when setting `feeBps` from `ERC721SeaDrop.sol` while one exists when the value is used at a later stage in Seadrop.sol, which could cause a `InvalidFeeBps` error.

Recommendation: Consider adding the following checks before setting `feeBps` at the mentioned places in `ERC721SeaDrop.sol`:

```
// Revert if the fee basis points is greater than 10_000.
if (feeBps > 10_000) {
    revert InvalidFeeBps(feeBps);
}
```

OpenSea: This have added this to SeaDrop itself on `updatePublicDrop` and `updateTokenGatedDrop`. See commit 246e1d4.

Spearbit: Acknowledged.

5.3.7 Upgrade `openzeppelin/contracts`'s version

Severity: *Low Risk*

Context: SeaDrop.sol#L318

Description: There are known vulnerabilities in the current `@openzeppelin/contracts` version used. This affects `SeaDrop.sol` with a potential Improper Verification of Cryptographic Signature vulnerability as `ECDSA.recover` is used.

Recommendation: Consider upgrading to `@openzeppelin/contracts@4.7.3`

OpenSea: Fixed in commit d279548.

Spearbit: Acknowledged.

5.3.8 `struct TokenGatedDropStage` is expected to fit into 1 storage slot

Severity: *Low Risk*

Context: SeaDropStructs.sol#L32-L61, SeaDrop.sol#L871-L876

Description: `struct TokenGatedDropStage` is expected to be tightly packed into 1 storage slot, as per announced in its @notice tag. However, the struct actually takes 2 slots. This is unexpected, as only one slot is loaded in the dropStageExists assembly check.

Recommendation: Consider changing `maxTokenSupplyForStage` to `uint32` to fit into 1 slot:

```
struct TokenGatedDropStage {
     uint80 mintPrice; // 80/256 bits
     uint16 maxTotalMintableByWallet;
     uint48 startTime;
     uint48 endTime;
     uint8 dropStageIndex; // non-zero
-    uint40 maxTokenSupplyForStage;
+    uint32 maxTokenSupplyForStage;
     uint16 feeBps;
     bool restrictFeeRecipients;
}
```

5.4 Gas Optimization

5.4.1 Avoid expensive iterations on removal of list elements by providing the index of element to be removed

Severity: *Gas Optimization*

Context: SeaDrop.sol#L1004

Description: Iterating through an array (`address[] storage enumeration`) to find the desired element (`address toRemove`) can be an expensive operation. Instead, it would be best to also provide the index to be removed along with the other parameters to avoid looping over all elements.

Also note in the case of `_removeFromEnumeration(signer, enumeratedStorage)`, hopefully, there wouldn't be too many signers corresponding to a contract. So practically, this wouldn't be an issue. But something to note. Although the `owner` or `admin` can stuff the `signer` list with a lot of signers as the other person would not be able to remove from the list (DoS attack). For example, if the `owner` has stuffed the `signer` list with malicious signers, the `admin` would not be able to remove them.

Recommendation: One way to simplify the removal process would be by providing the index. As an example:

```
function _removeFromEnumeration(
    address toRemove,
    address[] storage enumeration,
    uint index
) internal {
    require(enumeration[index] == toRemove);
    // Do the actual removing--no loops needed.
```

The index needs to be computed off-chain before sending the transaction.

5.4.2 `mintParams.allowedNftToken` should be cached

Severity: *Gas Optimization*

Context: SeaDrop.sol#L345-L436

Description: `mintParams.allowedNftToken` is accessed several times in the `mintAllowedTokenHolder` function. It would be cheaper to cache it:

```
// Put the allowedNftToken on the stack for more efficient access.
address allowedNftToken = mintParams.allowedNftToken;
```

Recommendation: Consider the following diff:

```
+        // Put the allowedNftToken on the stack for more efficient access.
+        address allowedNftToken = mintParams.allowedNftToken;
+
         // Set the dropStage to a variable.
         TokenGatedDropStage memory dropStage = _tokenGatedDrops[nftContract][
-            mintParams.allowedNftToken
+            allowedNftToken
         ];
...
             // Check that the sender is the owner of the allowedNftTokenId.
             if (
-                IERC721(mintParams.allowedNftToken).ownerOf(tokenId) != minter
+                IERC721(allowedNftToken).ownerOf(tokenId) != minter
             ) {
                 revert TokenGatedNotTokenOwner(
                     nftContract,
-                    mintParams.allowedNftToken,
+                    allowedNftToken,
                     tokenId
                 );
             }

             // Check that the token id has not already been redeemed.
             if (
-                _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken][
+                _tokenGatedRedeemed[nftContract][allowedNftToken][
                     tokenId
                 ] == true
             ) {
                 revert TokenGatedTokenIdAlreadyRedeemed(
                     nftContract,
-                    mintParams.allowedNftToken,
+                    allowedNftToken,
                     tokenId
                 );
             }

             // Mark the token id as redeemed.
-            _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken][
+            _tokenGatedRedeemed[nftContract][allowedNftToken][
```

OpenSea: Added in commit 48823a3.

Spearbit: Acknowledged.

5.4.3 Immutables which are calculated using `keccak256` of a string literal can be made constant.

Severity: *Informational*

Context: SeaDrop.sol#L76, SeaDrop.sol#L80

Description: Since Solidity 0.6.12, `keccak256` expressions are evaluated at compile-time:

> Code Generator: Evaluate keccak256 of string literals at compile-time.

The suggestion of marking these expressions as `immutable` to save gas isn't true for compiler versions >= `0.6.12`. As a reminder, before that, the occurrences of `constant keccak256` expressions were replaced by the expressions instead of the computed values, which added a computation cost.

Recommendation: In `SeaDrop`, `_SIGNED_MINT_TYPEHASH` and `_EIP_712_DOMAIN_TYPEHASH` are defined as `immutable` but can be safely tuned into `constant`.

OpenSea: Will update to a constant.

5.4.4 Combine a pair of mapping to a list and mapping to a mapping into mapping to a linked-list

Severity: *Gas Optimization*

Context: SeaDrop.sol#L54-L68

Description: `SeaDrop` uses 3 pairs of mapping to a list and mapping to a mapping that can be combined into just one mapping. The pairs:

1. `_allowedFeeRecipients` and `_enumeratedFeeRecipients`
2. `_signers` and `_enumeratedSigners`
3. `_tokenGatedDrops` and `_enumeratedTokenGatedTokens`

Here we have variables that come in pairs. One variable is used for data retrievals (a flag or a custom struct) and the other for iteration/enumeration.

```
mapping(address => mapping(address => CustomStructOrBool)) private variable;
mapping(address => address[]) private _enumeratedVariable;
```

Recommendation: We can combine each pair into just one variable that maps `address` for `nftContracts` into a (cyclic) doubly-linked list. Then retrievals, insertions, and removals would cost $\mathcal{O}(1)$, iteration would remain $\mathcal{O}(n)$. Removals is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ (this would save us gas on any call that would trigger `_removeFromEnumeration`). Also the storage structure would look more simplified.

For example for the case of `bool` inner value (`_allowedFeeRecipients`, `_signers`), we can define the doubly-linked list node/element as the following struct:

```
struct Node {
    bool value;
    address prev;
    address next;
}
```

And our mapped variable would be:

```
mapping(address => mapping(address => Node)) private variable;
```

We can have the address `0x1` as our flagged address (start/end) of our doubly-linked list.

Related: *Combine `_allowedSeaDrop` and `_enumeratedAllowedSeaDrop` in `ERC721SeaDrop` to save storage and gas.*.

5.4.5 The `onlyAllowedSeaDrop` modifier is redundant

Severity: *Gas Optimization*

Context:

- ERC721SeaDrop.sol#L65-L74,
- ERC721SeaDrop.sol#L157,
- ERC721SeaDrop.sol#L184,
- ERC721SeaDrop.sol#L213,
- ERC721SeaDrop.sol#L232,
- ERC721SeaDrop.sol#L265,
- ERC721SeaDrop.sol#L290,
- ERC721SeaDrop.sol#L306,
- ERC721SeaDrop.sol#L324,

- ERC721SeaDrop.sol#L346

Description: The `onlyAllowedSeaDrop` modifier is always used next to another one (`onlyOwner`, `onlyAdministrator` or `onlyOwnerOrAdministrator`). As the `owner`, which is the least privileged role, already has the privilege to update the allowed SeaDrop registry list for this contract (by calling `updateAllowedSeaDrop`), this makes this second modifier redundant.

Recommendation: Remove the `onlyAllowedSeaDrop` modifier. As additional note, keep in mind that the `onlySeaDrop` modifier is indeed useful. It is used on a function where checking against the stored allowed Sea Drop registry list for this contract is relevant. Without the `onlySeaDrop` modifier, anyone could call the `mintSeaDrop` endpoint to mint. It restricts calls to only an allowed `msg.sender`.

5.4.6 Combine `_allowedSeaDrop` and `_enumeratedAllowedSeaDrop` in `ERC721SeaDrop` to save storage and gas.

Severity: *Gas Optimization*

Context: ERC721SeaDrop.sol#L48-L52

Description: Combine `_allowedSeaDrop` and `_enumeratedAllowedSeaDrop` into just one variable using a cyclic linked-list data structure. This would reduce storage space and save gas when storing or retrieving parameters.

Recommendation: An example of structures that could be used instead:

```
mapping(address => address) private _allowedSeaDrops;
```

When creating a cyclic linked-list use a flagged `address` so that later you will be able to iterate through the list. Let's say you have been given a list of allowed `SeaDrop` addresses `a`, then set `_allowedSeaDrops[0x01] = a[0]`. This would allow you to iterate later on by fetching the data for `0x01` first. Also the last address would need to point to `0x01`. Now if `_allowedSeaDrops[x] != 0` it is an allowed `SeaDrop` address (except `0x01`).

Here are how some of the functions would look like after implementing this type of data structure (rough sketch):

```
// ADDRESS_ZERO = address(uint160(0))
// ADDRESS_ONE  = address(uint160(1))

modifier onlySeaDrop() {
    if (_allowedSeaDrop[msg.sender] == ADDRESS_ZERO || seaDrop == ADDRESS_ONE) {
        revert OnlySeaDrop();
    }
    _;
}

modifier onlyAllowedSeaDrop(address seaDrop) {
    if (_allowedSeaDrop[seaDrop] == ADDRESS_ZERO || seaDrop == ADDRESS_ONE ) {
        revert OnlySeaDrop();
    }
    _;
}

function updateAllowedSeaDrop(address[] calldata allowedSeaDrop)
    external
    override
    onlyOwnerOrAdministrator
{
    // Reset the old mapping.
    address seaDrop = _allowedSeaDrop[ADDRESS_ONE];
    while ( seaDrop ) {
        address nextSeaDrop = _allowedSeaDrop[seaDrop];
        delete _allowedSeaDrop[seaDrop];
        seaDrop = nextSeaDrop;
    }
    seaDrop = ADDRESS_ONE;

    uint i;
    uint256 allowedSeaDropLength = allowedSeaDrop.length;

    // Set the new mapping for allowed SeaDrop contracts.
    for(; i < allowedSeaDropLength;) {
        address nextSeaDrop = allowedSeaDrop[i]
        _allowedSeaDrop[seaDrop] = nextSeaDrop;
        seaDrop = nextSeaDrop;
        unchecked {
            ++i;
        }
    }

    if( allowedSeaDropLength ) {
        _allowedSeaDrop[seaDrop] = ADDRESS_ONE;
    }

    // Emit an event for the update.
    emit AllowedSeaDropUpdated(allowedSeaDrop);
}
```

Also, the `constructor` would need to be updated accordingly. Use the above implementation of `updateAllowedSeaDrop` as a reference.

5.4.7 Use `dropStageDoesNotExist` instead of `dropStageExists`

Severity: *Gas Optimization*

Context: SeaDrop.sol#L871-L886

Recommendation: Instead of using `dropStageExists`, we can change that to `dropStageDoesNotExist` which would save us 2 `NOT`s, 1 `EQ` and 1 `PUSH1 0`.

Modified piece:

```
bool dropStageDoesNotExist;
assembly {
    dropStageDoesNotExist:= iszero(sload(existingDropStageData.slot))
}

if (addOrUpdateDropStage) {
    _tokenGatedDrops[msg.sender][allowedNftToken] = dropStage;
    // Add to enumeration if it does not exist already.
    if (dropStageDoesNotExist) {
        enumeratedTokens.push(allowedNftToken);
    }
} else {
    // Check we are not deleting a drop stage that does not exist.
    if (dropStageDoesNotExist) {
        revert TokenGatedDropStageNotPresent();
    }
    // Clear storage slot and remove from enumeration.
```

Amount of gas saved:

```
src/SeaDrop.sol:SeaDrop contract

Function Name                    min              avg     median  max     # calls

-  updateTokenGatedDrop            7087             72743   93461   97461   25
+  updateTokenGatedDrop            7087             72741   93458   97458   25
```

OpenSea: Updated in commit ac34900.

5.4.8 `<array>.length` should not be looked up in every loop of a `for-loop`

Severity: *Gas Optimization*

Context: ERC721SeaDrop.sol#L87, ERC721SeaDrop.sol#L109, ERC721SeaDrop.sol#L117

Description: Reading an array's length at each iteration of a loop consumes more gas than necessary.

Recommendation: Consider caching the array's length in a variable before the for-loop, and use this new variable instead. This should save around **3 gas** per iteration.

OpenSea: Fixed in the commit 0b90c9e.

Spearbit: Acknowledged.

5.4.9 A `storage` pointer should be cached instead of computed multiple times

Severity: *Gas Optimization*

Context: SeaDrop.sol#L405-L407, SeaDrop.sol#L417-L419

Description: Caching a mapping's value in a local `storage` variable when the value is accessed multiple times saves gas due to not having to perform the same offset calculation every time.

Recommendation: Consider declaring `mapping(uint256 => bool) storage redeemedTokenIds = _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken];`:

```
File: SeaDrop.sol
            // Check that the token id has not already been redeemed.
+           mapping(uint256 => bool) storage redeemedTokenIds =
↪   _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken];
            if (
-                _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken][
-                    tokenId
-                ] == true
+                redeemedTokenIds[tokenId] == true
            ) {
                revert TokenGatedTokenIdAlreadyRedeemed(
                    nftContract,
                    mintParams.allowedNftToken,
                    tokenId
                );
            }

            // Mark the token id as redeemed.
-           _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken][
-               tokenId
-           ] = true;
+           redeemedTokenIds[tokenId] = true;
```

Amount of gas saved:

OpenSea: Fixed in commit 3febba4.

Spearbit: Acknowledged.

5.4.10 Comparing a boolean to a constant

Severity: *Gas Optimization*

Context: SeaDrop.sol#L407, SeaDrop.sol#L469-L470

Description: Comparing to a constant (`true` or `false`) is a bit more expensive than directly checking the returned boolean value.

Recommendation: Consider applying the following:

```
             if (
                 _tokenGatedRedeemed[nftContract][mintParams.allowedNftToken][
                     tokenId
-                 ] == true
+                 ]
             ) {
...
-         if (restrictFeeRecipients == true)
+         if (restrictFeeRecipients)
-             if (_allowedFeeRecipients[nftContract][feeRecipient] == false) {
+             if (!_allowedFeeRecipients[nftContract][feeRecipient]) {
```

OpenSea: This was fixed in previous commits in the branch, found one more instance in the commit 16a4837.

5.5 Informational

5.5.1 `mintAllowList`, `mintSigned`, or `mintAllowedTokenHolder` have an inherent cap for minting

Severity: *Informational*

Context: SeaDropStructs.sol#L58

Description: `mintAllowedTokenHolder` is stored in a `uint40` (after this audit `uint32`) which limits the maximum token id that can be minted using `mintAllowList`, `mintSigned`, or `mintAllowedTokenHolder`.

Recommendation: It would be best to warn the the `owner` that `mintAllowList`, `mintSigned`, or `mintAllowedTokenHolder` cannot mint tokens with tokenID more than $2^{40} - 1$. Basically, the early drop mints are limited to this range since `maxTokenSupplyForStage` is not `uint256`.

OpenSea: `mintSigned` and `mintAllowList` have `maxTokenSupplyForStage` as uint256 since they are not stored on chain, but this is true for `mintAllowedTokenHolder` where the struct is stored on chain. Notes added in commit 0c05761.

Spearbit: Acknowledged.

5.5.2 Add warning for NFT contracts to implement authentication properly for `updateAllowList`

Severity: *Informational*

Context: SeaDrop.sol#L827

Recommendation: Need to warn NFT contracts to implement authentication properly for allowed list mints.

OpenSea: Implemented in commit 77ca81f.

Spearbit: Acknowledged.

5.5.3 Consider replacing `minterIfNotPayer` parameter to always correspond to the minter

Severity: *Informational*

Context: SeaDrop.sol#L145

Description: Currently, the variable `minterIfNotPayer` is treated in the following way: if the value is `0`, then `msg.sender` would be considered as the minter. Otherwise, `minterIfNotPayer` would be considered as the minter. The logic can be simplified to always treat this variable as the minter. The `0` can be replaced by setting `msg.sender` as `minterIfNotPayer`. The variable should then be renamed as well--we recommend calling it `minter` afterwards.

Recommendation: If the change is implemented, make sure that the backend code is appropriately changed. Supplying the wrong `calldata` may lead to reverts or loss of funds.

OpenSea: The idea here was to cut down on `calldata` costs in the case where `msg.sender` is the minter, which would be most normal use-cases. Zero-value `calldata` should save ~200 gas in the average case, even with branching, right?

Spearbit: The gas savings sound accurate. However, we should try to simplify the code.

5.5.4 The interface `IERC721ContractMetadata` does not extend `IERC721` interface

Severity: *Informational*

Context: IERC721ContractMetadata.sol#L4

Description: The current interface `IERC721ContractMetadata` does not include the ERC-721 functions. As a comparision, OpenZeppelin's IERC721Metadata.sol extends the `IERC721` interface.

Recommendation: Inherit from the `IERC721` interface.

OpenSea: This should extend `IERC721`.

5.5.5 Add unit tests for `mintSigned` and `mintAllowList` in `SeaDrop`

Severity: *Informational*

Context: SeaDrop.sol#L259

Description: The only test for the `mintSigned` and the `mintAllowList` functions are fuzz tests.

Recommendation: It would be great to include some basic unit tests for these functions.

OpenSea: We are working on full unit-test coverage in Hardhat! Unit tests with complete contract code coverage have been added. SeaDrop-mintSigned.spec.ts, SeaDrop-mintAllowList.spec.ts

Spearbit: Acknowledged.

5.5.6 Rename a variable with a misleading name

Severity: *Informational*

Context: SeaDrop.sol#L1002-L1008

Description: `enumeratedDropsLength` variable name in `SeaDrop._removeFromEnumeration` is a bit misleading since `_removeFromEnumeration` is used also for signer lists, `feeRecipient` lists, etc..

Recommendation: Rename `enumeratedDropsLength` to `enumerationLength`.

OpenSea: Fixed in commit b970ed4.

Spearbit: Acknowledged.

5.5.7 The protocol rounds the fees in the favour of `creatorPaymentAddress`

Severity: *Informational*

Context: SeaDrop.sol#L576

Description: The `feeAmount` calculation rounds down, i.e., rounds in the favour of `creatorPaymentAddress` and against `feeRecipient`. For a minuscule amount of ETH (`price` such that `price * feeBps < 10000`), the fees received by the `feeRecipient` would be `0`. An interesting case here would be if the value `quantity * price * feeBps` is greater than or equal to `10000` and `price * feeBps < 10000`. In this case, the user can split the mint transaction into multiple transactions to skip the fees. However, this is unlikely to be profitable, considering the gas overhead involved as well as the minuscule amount of savings.

Recommendation: There are no recommended actions needed here, except documenting that fees are rounded in favour of the creator.

OpenSea: Fixed in commit d2a9f29.

Spearbit: Acknowledged.

5.5.8 Consider using `type(uint).max` as the magic value for `maxTokenSupplyForStage` instead of `0`

Severity: *Informational*

Context: SeaDrop.sol#L516

Description: The value `0` is currently used as magic value to mean that `maxTokenSupplyForStage` to mean that the check `quantity + currentTotalSupply > maxTokenSupplyForStage`. However, the value `type(uint).max` is a more appropriate magic value in this case. This also avoids the need for additional branching `if (maxTokenSupplyForStage != MAGIC_VALUE)` as the condition `quantity + currentTotalSupply > type(uint).max` is never true.

Recommendation: Consider implementing the following suggestions

1. Use `type(uint).max` instead of `0`.
2. Remove the redundant `if (maxTokenSupplyForStage != MAGIC_VALUE)`.
3. If the magic value is kept as `0`, consider making it a named constant, and using the name everywhere.

OpenSea: Fixed in commits d4101c2 and cbd5ce5.

Spearbit: Acknowledged.

5.5.9 Missing edge case tests on uninitialized AllowList

Severity: *Informational*

Context: SeaDrop.sol#L226-L231

Description: The default value for `_allowListMerkleRoots[nftContract]` is `0`. A transaction that tries to mint an NFT in this case with an empty proof (or any other proof) should revert. There were no tests for this case.

Recommendation: Add missing tests.

OpenSea: Test added in commit 66ee1176.

Spearbit: Acknowledged.

5.5.10 Consider naming state variables as `public` to replace the user-defined getters

Severity: *Informational*

Context: SeaDrop.sol#L43

Description: Several state variables, for example, `mapping(address => PublicDrop) private _publicDrops;` have private visibility, but have corresponding getters defined (`function getPublicDrop(address nftContract)`). Replacing `private` by `public` and renaming the variable name can decrease the code.

There are several examples of the above pattern in the codebase, however we are only listing one here for brevity.

Recommendation: For all user defined getter functions, consider replacing the function by marking the visibility of the corresponding state variable to `public`. To keep the same name of the getter / ABI, the state variable can be renamed, e.g., from `_publicDrops` to `getPublicDrop`.

OpenSea: "Won't fix" as this doesn't seem to play well with the interfaces we've defined.

Spearbit: Acknowledged.

5.5.11 Use `bytes.concat` instead of `abi.encodePacked` for concatenation

Severity: *Informational*

Context: SeaDrop.sol#L297-L312

Description: While one of the uses of `abi.encodePacked` is to perform concatenation, the Solidity language does contain a reserved function for this: `bytes.concat`.

Recommendation: Consider using `bytes.concat` instead of `abi.encodePacked` for concatenation:

```diff
        bytes32 digest = keccak256(
-           abi.encodePacked(
+           bytes.concat(
                // EIP-191: `0x19` as set prefix, `0x01` as version byte
                bytes2(0x1901),
                _domainSeparator(),
                keccak256(
                    abi.encode(
                        _SIGNED_MINT_TYPEHASH,
                        nftContract,
                        minter,
                        feeRecipient,
                        mintParams
                    )
                )
            )
        );
```

OpenSea: Updated in commit 47c50ed.

Spearbit: Acknowledged.

5.5.12 Misleading comment

Severity: *Informational*

Context: SeaDrop.sol#L392-L395

Description: The comment says `// Check that the sender is the owner of the allowedNftTokenId.`. However, minter isn't necessarily the sender due to how it's set: `address minter = minterIfNotPayer != address(0) ? minterIfNotPayer : msg.sender;`.

Recommendation: Consider editing the comment as such:

```
-            // Check that the sender is the owner of the allowedNftTokenId.
+            // Check that the minter is the owner of the allowedNftTokenId.
            if (
                IERC721(mintParams.allowedNftToken).ownerOf(tokenId) != minter
            ) {
```

OpenSea: Fixed in commit 1bf3225.

Spearbit: Acknowledged.

5.5.13 Use `i` instead of `j` as an index name for a non-nested for-loop

Severity: *Informational*

Context: SeaDrop.sol#L388

Description: Using an index named `j` instead of `i` is confusing, as this naming convention makes developers expect that the for-loop is nested, but this is not the case. Using `i` is more standard and less surprising.

Recommendation: Use `i` instead of `j` as an index name

OpenSea: Fixed in commit 318c851.

Spearbit: Acknowledged.

5.5.14 Avoid duplicating code for consistency

Severity: *Informational*

Context: SeaDrop.sol#L129-L136, SeaDrop.sol#L196, SeaDrop.sol#L268, SeaDrop.sol#L362

Description: The `_checkActive` function is used in every `mint` function besides `mintPublic` where the code is almost the same.

Recommendation: Consider maintaining consistency by using `_checkActive(publicDrop.startTime, type(uint64).max)`:

```
 function mintPublic(
 ...
         // Ensure that the drop has started.
-     if (block.timestamp < publicDrop.startTime) {
-         revert NotActive(
-             block.timestamp,
-             publicDrop.startTime,
-             type(uint64).max
-          );
-      }
+     _checkActive(publicDrop.startTime, type(uint64).max)
```

This would also save some deployment cost and some gas on average:

OpenSea: Fixed in commit 7f1456d.

Spearbit: Acknowledged.

5.5.15 `restrictFeeRecipients` is always true for either `PublicDrop` or `TokenGatedDrop` in `ERC721SeaDrop`

Severity: *Informational*

Context:

- ERC721SeaDrop.sol#L168,
- ERC721SeaDrop.sol#L193,
- ERC721SeaDrop.sol#L242,
- ERC721SeaDrop.sol#L273

Description: `restrictFeeRecipients` is always `true` for either `PublicDrop`s or `TokenGatedDrop`s. When either one of these drops gets created/updated by calling one of the four functions below on a `ERC721SeaDrop` contract, its value is hardcoded as `true`:

- `updatePublicDrop`
- `updatePublicDropFee`
- `updateTokenGatedDrop`
- `updateTokenGatedDropFee`

Recommendation: Unless there is a plan to add some functionality for cases when `restrictFeeRecipients == false`, it would be best to rewrite the contracts to remove this variable and assume it's always true. On the `SeaDrop` end, there are scenarios where minting with a signature or proof for allowed mints can have a `restrictFeeRecipients == false`.

OpenSea: This logic is now updated where the admin can set to false if they would like in the commit cbd5ce5.

5.5.16 Reformat lines for better readability

Severity: *Informational*

Context: SeaDrop.sol#L76-L83

Description: These lines are too long to be readable. A mistake isn't easy to spot.

Recommendation: Reformat these lines into:

```
bytes32 internal immutable _SIGNED_MINT_TYPEHASH =
    keccak256(
        "SignedMint("
            "address nftContract,"
            "address minter,"
            "address feeRecipient,"
            "MintParams mintParams"
        ")"
        "MintParams("
            "uint256 mintPrice,"
            "uint256 maxTotalMintableByWallet,"
            "uint256 startTime,"
            "uint256 endTime,"
            "uint256 dropStageIndex,"
            "uint256 maxTokenSupplyForStage," // <--- this was also missing in the audit repo
            "uint256 feeBps,"
            "bool restrictFeeRecipients"
        ")"
    );
bytes32 internal immutable _EIP_712_DOMAIN_TYPEHASH =
    keccak256(
        "EIP712Domain("
            "string name,"
            "string version,"
            "uint256 chainId,"
            "address verifyingContract"
        ")"
    );
```

OpenSea: Fixed: fd07a8b.

Spearbit: Acknowledged.

5.5.17 Comment is a copy-paste

Severity: *Informational*

Context: SeaDrop.sol#L51, SeaDrop.sol#L54

Description: This comment is exactly the same as this one. This is a copy-paste mistake.

Recommendation: Consider writing another description on L54.

OpenSea: Fixed in commit 8ebff0d.

Spearbit: Acknowledged.

5.5.18 Replace `setBatchTokenURIs()`

Severity: *Informational*

Context: ERC721ContractMetadata.sol#L94-L108

Description: This was discussed with OpenSea:

> This method should be renamed to something like `emitBatchTokenURIUpdated` and have the `string calldata` parameter removed. It was meant as a shortcut for emitting the event when only subsets of token metadata have actually changed, and is mostly informational/instructional.

OpenSea: Fixed in commit b8566fa.

Spearbit: Acknowledged.

5.5.19 Remove unused imports

Severity: *Informational*

Context: ERC721ContractMetadata.sol#L6-L27, SeaDrop.sol#L16

Recommendation: Removing the unused imports improves code-quality.

In `ERC721ContractMetadata.sol`, the following imports are unused: `MaxMintable`, `TwoStepOwnable`, `AllowList`, `Ownable`, `ECDSA`, `ConstructorInitializable` and `IERC721ContractMetadata`.

In `SeaDrop.sol`, the import `ERC20` is unused.

5.5.20 Missing comment about `PublicDrop`s `endTimestamp`

Severity: *Informational*

Context: lib/SeaDropErrorsAndEvents.sol#L7-L14, SeaDrop.sol#L134

Recommendation: Consider adding a comment that for `PublicDrop`s `endTimestamp` value should be `type(uint64).max`.

OpenSea: Fixed in commits 7f1456d and 27e3435.

Spearbit: Acknowledged.

5.5.21 Misaligned lines

Severity: *Informational*

Context: SeaDropErrorsAndEvents.sol#L160-L177

Recommendation: Consider adding indents to align with other lines.

OpenSea: Fixed in commit 24ed58d.

Spearbit: Acknowledged.

5.5.22 Usage of floating `pragma` is not recommended

Severity: *Informational*

Context:

- SeaDrop.sol#L2,
- ERC721SeaDrop.sol#L2,
- ERC721ContractMetadata.sol#L2,
- lib/SeaDropStructs.sol#L2,
- lib/SeaDropErrorsAndEvents.sol#L2,
- interfaces/ISeaDrop.sol#L2,
- interfaces/IERC721SeaDrop.sol#L2,
- interfaces/IERC721ContractMetadata.sol#L2

Description:

- `^0.8.11` is declared in files.
- In `foundry.toml`: `solc_version = '0.8.15'` is used for the default build profile.
- In `hardhat.config.ts` and `hardhat-coverage.config.ts`: `"0.8.14"` is used.

Recommendation: Consider documenting the actual compiler version and flags used to get the compiled bytecode which is going to be deployed on-chain

OpenSea: Fixed in commit 9d87bfc.

Spearbit: Acknowledged.