Foundation Drop contest Findings & Analysis Report

2022-09-29

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Foundation Drop smart contract system written in Solidity. The audit contest took place between August 11—August 15 2022.

Wardens

114 Wardens contributed reports to the Foundation Drop contest:

1. Lambda

2. joestakey
3. KIntern_NA (TrungOre and duc)
4. byndooa
5. shenwilly
6. bin2chen
7. berndartmueller
8. 0x52
9. 0xHarry
10. peritoflores
11. Dravee
12. Saw-mon_and_Natalie
13. 0x1f8b
14. IllIllI
15. c3phas
16. zkhorse (karmacoma and horsefacts)
17. Chom
18. rbserver
19. Deivitto
20. 0xDjango
21. ladboy233
22. wagmi
23. Treasure-Seeker
24. cccz
25. csanuragjain
26. ReyAdmirado
27. auditor0517
28. PwnedNoMore (izhuer, ItsNio, and papr1ka2)
29. thank_you
30. oyc_109

31. Bnke0x0
32. erictee
33. Rolezn
34. durianSausage
35. LeoS
36. Rohan16
37. Sm4rty
38. zeesaw
39. brgltd
40. carlitox477
41. 0xSmartContract
42. simon135
43. MiloTruck
44. gogo
45. JC
46. 0xNazgul
47. d3e4
48. TomJ
49. _141345_
50. robee
51. rvierdiiev
52. DevABDee
53. Aymen0909
54. Waze
55. fatherOfBlocks
56. mics
57. bobirichman
58. ElKu
59. bulej93

60. apostle0x01
61. sikorico
62. Yiko
63. 0xsolstars (Varun_Verma and masterchief)
64. Ruhum
65. Ch_301
66. nine9
67. yixxas
68. itsmeSTYJ
69. yash90
70. 0xSolus
71. danb
72. delfin454000
73. Kumpa
74. ret2basic
75. rokinot
76. jonatascm
77. Vexjon
78. cryptphi
79. 0xackermann
80. 0xmatt
81. iamwhitelights
82. 0xkatana
83. Noah3o6
84. CodingNameKiki
85. Diraco
86. ignacio
87. ajtra
88. jag

89. saian
90. Tomio
91. Trabajo_de_mates (Saintcode_ and tay054)
92. Amithuddar
93. pfapostol
94. 0x040
95. 0xbepresent
96. cRat1st0s
97. Fitraldys
98. Funen
99. gerdusx
100. Metatron
101. samruna
102. SpaceCake
103. zuhaibmohd
104. hakerbaya
105. medikko
106. newfork01
107. sach1r0
108. 0xc0ffEE

This contest was judged by hickuphh3.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 73 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 73 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Foundation Drop contest repository, and is composed of 20 smart contracts written in the Solidity programming language and includes 1,218 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (8)

[M-01] Creator fees may be burned

Submitted by Lambda

`royaltyInfo`, `getRoyalties`, or `getFeeRecipients` may return `address(0)` as the recipient address. While the value 0 is correctly handled for the royalties itself, it is not for the address. In such a case, the ETH amount will be sent to `address(0)`, i.e. it is burned and lost.

Recommended Mitigation Steps

In your logic for determining the recipients, treat `address(0)` as if no recipient was returned such that the other priorities / methods take over.

HardlyDifficult (Foundation) confirmed, but disagreed with severity and commented:

> We are looking into options here to improve.

> We believe this is Medium risk, burning a percent of the sale revenue is a form of leaking value. Otherwise the sale works as expected and the collector does get the NFT they purchased.

> The royalty APIs we use are meant to specific which addresses should receive payments and how much they each should receive. As the warden noted, we try to ignore entries which specify a 0 amount... but did not filter out address(0) recipients with >0 requested. Originally we were thinking this was a way of requesting that a portion of the sale be burned since that seems to be what the data is proposing.

> However we agree that this is more likely a configuration error. Since our market uses ETH and not ERC20 tokens, it's unlikely that creators would actually want a portion of the proceeds burned. We are exploring a change to send the additional revenue to the seller instead of burning the funds in this scenario.

HickupHH3 (judge) decreased severity to Medium and commented:

> Case of protocol leaked value: Medium severity is appropriate.

[M-02] NFT creator sales revenue recipients can steal gas

Submitted by berndartmueller, also found by 0xHarry, peritoflores, and shenwilly

Selling a NFT with `NFTDropMarketFixedPriceSale.mintFromFixedPriceSale` distributes the revenue from the sale to various recipients with the `MarketFees._distributeFunds` function.

Recipients:

- NFT creator(s)
- NFT seller
- Protocol
- Buy referrer (optional)

It is possible to have multiple NFT creators. Sale revenue will be distributed to each NFT creator address. Revenue distribution is done by calling `SendValueWithFallbackWithdraw._sendValueWithFallbackWithdraw` and providing an appropriate gas limit to prevent consuming too much gas. For the revenue distribution to the seller, protocol and the buy referrer, a gas limit of `SEND_VALUE_GAS_LIMIT_SINGLE_RECIPIENT = 20_000` is used. However, for the creators, a limit of `SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS = 210_000` is used. This higher amount of gas is used if `PercentSplitETH` is used as a recipient.

A maximum of `MAX_ROYALTY_RECIPIENTS = 5` NFT creator recipients are allowed.

For example, a once honest NFT collection and its 5 royalty creator recipients could turn “malicious” and could “steal” gas from NFT buyers on each NFT sale and therefore grief NFT sales. On each NFT sell, the 5 creator recipients (smart contracts) could consume the full amount of `SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS = 210_000` forwarded gas. Totalling `5 * 210_000 = 1_050_000` gas. With a gas price of e.g. `20 gwei`, this equals to additional gas costs of `21_000_000 gwei = 0.028156 eth`, with a `ETH` price of `2000`, this would total to ~ `56.31 $` additional costs.

Proof of Concept

mixins/shared/MarketFees.sol#L130

```
  /**
   * @notice Distributes funds to foundation, creator recipients,
   */
  function _distributeFunds(
   address nftContract,
   uint256 tokenId,
   address payable seller,
   uint256 price,
   address payable buyReferrer
  )
   internal
   returns (
     uint256 totalFees,
     uint256 creatorRev,
     uint256 sellerRev
   )
  {
   address payable[] memory creatorRecipients;
   uint256[] memory creatorShares;

   uint256 buyReferrerFee;
   (totalFees, creatorRecipients, creatorShares, sellerRev, buyRe
     nftContract,
     tokenId,
     seller,
     price,
     buyReferrer
   );

   // Pay the creator(s)
   unchecked {
     for (uint256 i = 0; i < creatorRecipients.length; ++i) {
       _sendValueWithFallbackWithdraw(
         creatorRecipients[i],
         creatorShares[i],
         SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS // @audit-info
       );
       // Sum the total creator rev from shares
       // creatorShares is in ETH so creatorRev will not overflow
       creatorRev += creatorShares[i];
     }
   }

   // Pay the seller
   _sendValueWithFallbackWithdraw(seller, sellerRev, SEND_VALUE_(

   // Pay the protocol fee
   _sendValueWithFallbackWithdraw(getFoundationTreasury(), totalF
   // Pay the buy referrer fee
   if (buyReferrerFee != 0) {
     _sendValueWithFallbackWithdraw(buyReferrer, buyReferrerFee,
     emit BuyReferralPaid(nftContract, tokenId, buyReferrer, buyR
     unchecked {
       // Add the referrer fee back into the total fees so that a
       totalFees += buyReferrerFee;
     }
   }
 }
```

Recommended Mitigation Steps

Consider only providing a higher amount of gas (`SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS`) for the first creator recipient. For all following creator recipients, only forward the reduced amount of gas `SEND_VALUE_GAS_LIMIT_SINGLE_RECIPIENT`.

[HardlyDifficult (Foundation) confirmed and commented](#):

> We will be making changes here.

> This seems like a Low risk issue since only gas is at risk, but protecting our collectors is an important goal so we are comfortable with Medium here.

> As the warden has noted, we use gas caps consistently when interacting with external addresses/contracts. This is important to ensure that the cost to collectors does not become unwieldy.. and that the calls cannot revert (e.g. if the receiver gets stuck in a loop).

> The gas limits we set are high enough to allow some custom logic to be performed, and to support smart contract wallets such as Gnosis Safe. For the scenario highlighted here, we have used a very high limit in order to work with contracts such as PercentSplitETH (which will push payments to up to 5 different recipients, and those recipients may be smart contract wallets themselves).

> However we were too flexible here. And in total, the max potential gas costs are higher than they should be. We have changed the logic to only use `SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS` when 1 recipient is defined,

> otherwise use `SEND_VALUE_GAS_LIMIT_SINGLE_RECIPIENT`. This will support our PercentSplitETH scenario and use cases like it, while restricting the worst case scenario to something much more reasonable.

HickupHH3 (judge) commented:

> Keeping the Medium severity because users are potentially paying more than necessary.

[M-03] Forget to check "Some manifolds contracts of ERC-2981 return (address(this), 0) when royalties are not defined" in 3rd priority - MarketFees.sol

Submitted by KIntern_NA, also found by bin2chen and Lambda

Wrong return of `cretorShares` and `creatorRecipients` can make real royalties party can't gain the revenue of sale.

Proof of concept

Function `getFees()` firstly call to function `internalGetImmutableRoyalties` to get the list of `creatorRecipients` and `creatorShares` if the `nftContract` define ERC2981 royalties.

```
    try implementationAddress.internalGetImmutableRoyalties(nftContr
      address payable[] memory _recipients,
      uint256[] memory _splitPerRecipientInBasisPoints
    ) {
      (creatorRecipients, creatorShares) = (_recipients, _splitPerRe
    } catch // solhint-disable-next-line no-empty-blocks
    {
      // Fall through
    }
```

In the 1st priority it check the `nftContract` define the function `royaltyInfo` or not. If yes, it get the return value `receiver` and `royaltyAmount`. In some manifold contracts of erc2981, it `return (address(this), 0)` when royalties are not defined. So we ignore it when the `royaltyAmount = 0`

```
try IRoyaltyInfo(nftContract).royaltyInfo{ gas: READ_ONLY_GAS_
  address receiver,
  uint256 royaltyAmount
) {
  // Manifold contracts return (address(this), 0) when royalti
  // - so ignore results when the amount is 0
  if (royaltyAmount > 0) {
    recipients = new address payable[](1);
    recipients[0] = payable(receiver);
    splitPerRecipientInBasisPoints = new uint256[](1);
    // The split amount is assumed to be 100% when only 1 reci
    return (recipients, splitPerRecipientInBasisPoints);
  }
```

In the same sense, the **3rd priority** (it can reach to 3rd priority when function `internalGetImmutableRoyalies` fail to return some royalties) should check same as the 1st priority with the `royaltyRegistry.getRoyaltyLookupAddress`. But the 3rd priority forget to check the case when `royaltyAmount == 0`.

```
try IRoyaltyInfo(nftContract).royaltyInfo{ gas: READ_ONLY_GAS_
  address receiver,
  uint256 /* royaltyAmount */
) {
  recipients = new address payable[](1);
  recipients[0] = payable(receiver);
  splitPerRecipientInBasisPoints = new uint256[](1);
  // The split amount is assumed to be 100% when only 1 recipi
  return (recipients, splitPerRecipientInBasisPoints);
}
```

It will make **function** `_distributeFunds()` transfer to the wrong `creatorRecipients` (for example erc2981 return `(address(this), 0)`, market will transfer creator revenue to `address(this)` - market contract, and make the fund freeze in contract forever).

This case just happen when

- `nftContract` doesn't have any support for royalties info

- `overrideContract` which was fetched from `royaltyRegistry.getRoyaltyLookupAddress(nftContract)` implements both function `getRoyalties` and `royaltyInfo` but doesn't support `royaltyInfo` by returning `(address(this), 0)`.

Recommended Mitigation Steps

Add check if `royaltyAmount > 0` or not in 3rd priority.

HardlyDifficult (Foundation) confirmed and commented:

> This was a great catch. We will be making the recommended change.

> Medium risk seems correct as this is a form of potentially leaking value.

> We agree that any contract returning `(address(this), 0)` should be treated as no royalties defined instead of paying to `address(this)`.

HickupHH3 (judge) commented:

> Yes, agree that zero royalty amount check is missing for 3rd priority.

[M-04] Possible to bypass saleConfig.limitPerAccount

Submitted by itsmeSTYJ, also found by 0x1f8b, 0x52, 0xDjango, auditor0517, byndooa, cccz, Ch_301, Chom, csanuragjain, KIntern_NA, ladboy233, nine9, PwnedNoMore, shenwilly, thank_you, Treasure-Seeker, wagmi, yixxas, and zkhorse

It is possible to bypass the `saleConfig.limitPerAccount` set by the creator by transferring the NFTs out. For highly sought after NFT drops, a single smart contract can buy out the entire drop simply by calling `mintFromFixedPriceSale` then transferring the NFTs out and repeating the process multiple times.

Proof of Concept

Modify the `FixedPriceDrop.sol` Foundry test with the following changes.

```
diff --git a/FixedPriceDrop.sol.orig b/FixedPriceDrop.sol
index 0a6d698..56808f8 100644
--- a/FixedPriceDrop.sol.orig
+++ b/FixedPriceDrop.sol
@@ -71,14 +71,26 @@ contract TestFixedPriceDrop is Test {

     /** List for sale **/
     uint80 price = 0.5 ether;
-    uint16 limitPerAccount = 10;
+    uint16 limitPerAccount = 3;
     vm.prank(creator);
     nftDropMarket.createFixedPriceSale(address(nftDropCollectic

     /** Mint from sale **/
     uint16 count = 3;
     vm.deal(collector, 999 ether);
-    vm.prank(collector);
+    vm.startPrank(collector);
+    nftDropMarket.mintFromFixedPriceSale{ value: price * count
+
+    // Check that available count for collector is 0
+    uint256 remaining = nftDropMarket.getAvailableCountFromFixe
+    assertEq(remaining, 0);
+
+    // Transfer all bought NFTs out
+    nftDropCollection.transferFrom(collector, address(5), 1);
+    nftDropCollection.transferFrom(collector, address(5), 2);
+    nftDropCollection.transferFrom(collector, address(5), 3);
+
+    // Buy 3 more NFT
     nftDropMarket.mintFromFixedPriceSale{ value: price * count
   }
 }
```

Tools Used

Foundry

Recommended Mitigation Steps

Use a mapping to track how many NFTs an address has bought instead of relying on `balanceOf`.

HardlyDifficult (Foundation) acknowledged and commented:

> This is accurate! We had several meetings about this concern while building the contract, ultimately deciding to move forward with this approach knowing that it has limitations. The trouble is every limit solution suggested and used in the wild today can be gamed, it’s just varying levels of friction for an attacker to work around it. Once someone has coded up a workaround, it could easily be used on any of the collections being sold by our marketplace. So we decided to KISS.

> But if it can be gamed, why include a limit at all? Creators want one. It sets expectations with the community and makes the sale feel more fair. Many users will respect the limit as communicated - we suspect more often than not, this simple limit check will be sufficient.

> What if it’s not sufficient? If someone were to clearly abuse the system it may degrade the value of the collection for all. There are options available to the creator at that point. For example, the creator could create a new collection to replace the original - airdropping NFTs to their legit holders, or allowing them to do an NFT swap (so the original collection can slowly be removed from circulation) — this swap could also have a deny list so that the abused sales cannot be used to redeem from the new collection, and presumably the original collection will quickly lose value so long as the creator’s community is on board with this process. Or the creator and their community could choose to simply accept that the sale went down this way and wait for things to balance out again on the secondary market.

> I selected this instance as the primary submission for having a simple & clear coded POC.

> We agree Medium risk is appropriate for this since it could “leak value with a hypothetical attack path with stated assumptions”.

[M-05] User may get all of the creator fees by specifying high number for himself

Submitted by Lambda, also found by 0x52, KIntern_NA, and shenwilly

If one creator specified a share that is larger than `BASIS_POINTS`, the first creator gets all of the royalties. Depending on how these are set (which is not in the control of the project), this can be exploited by the first creator.

Proof Of Concept

A collective of artists have implemented a process where everyone can set its own share of the fees to the value that he thinks is fair and these values are returned by `getRoyalties`. Bob, the first entry in the list of `_recipients` sets its value to a value that is slightly larger than `BASIS_POINTS` such that he gets all of the royalties.

Recommended Mitigation Steps

There is no need for this check / logic, as the whole sum (`totalShares`) is used anyway to normalize the values.

HardlyDifficult (Foundation) acknowledged, but disagreed with severity and commented:

> We believe this is Low risk. For the Foundation collections, the royalty rate is hard coded to 10% or via `PercentSplitETH` which is not subject to this issue. For 3rd party collections, there are more direct ways to change the distribution if the creator was attempting to be malicious towards their partners — esp via the Royalty Registry.

> This report is true. And the recommendation seems reasonable. However we will not be making this change. We are currently investigating changing our royalty logic in order to use the values returned by collections directly, instead of normalizing it to 10% like we do now. Most of the royalty APIs used here are not official standards, but are becoming industry standards based on growing adoption — and they are expecting the percent amounts to be defined in Basis Points.

> We do not want to mislead the community too much to ease the pain of the potential upcoming change I mentioned above. If they are returning values > 10,000 we don't want that pattern to be adopted by more collections.

> Another option may be to ignore the results if `totalShares` sums to > 10,000 - that's tempting but we are going to defer making a change like that until a future

> workstream which will be more dedicated to rethinking royalties.

HickupHH3 (judge) commented:

> Am siding with the warden here, because for 3rd party collections, it may be the case that they use a larger denomination than basis points. As mentioned in a different issue, royalty standards are still in its infancy.

> Most of the royalty APIs used here are not official standards, but are becoming industry standards based on growing adoption — and they are expecting the percent amounts to be defined in Basis Points.

> Hopeful for this to be the case so there is less ambiguity, and non-compliance can be ignored as suggested by the sponsor.

[M-06] Malicious Creator can steal from collectors upon minting with a custom NFT contract

Submitted by joestakey, also found by byndooa

In the case of a fixed price sale where `nftContract` is a custom NFT contract that adheres to `INFTDropCollectionMint`, a malicious creator can set a malicious implementation of `INFTDropCollectionMint.mintCountTo()` that would result in collectors calling this function losing funds without receiving the expected amount of NFTs.

Proof Of Concept

Here is a Foundry test that shows a fixed price sale with a malicious NFT contract, where a collector pays for 10 NFTs while only receiving one. It can be described as follow:

- A creator creates a malicious `nftContract` with `mintCountTo` minting only one NFT per call, regardless of the value of `count`
- The creator calls `NFTDropMarketFixedPriceSale.createFixedPriceSale()` to create a sale for `nftContract`, with `limit` set to `15`.

- Bob is monitoring the `CreateFixedPriceSale` event. Upon noticing `CreateFixedPriceSale(customERC721, Alice, price, limit)`, he calls `NFTDropMarketFixedPriceSale.mintFromFixedPriceSale(customERC721, count == 10,)`. He pays the price of `count = 10` NFTs, but because of the logic in `mintCountTo`, only receives one NFT.

Note that `mintCountTo` can be implemented in many malicious ways, this is only one example. Another implementation could simply return `firstTokenId` without performing any minting.

Tools Used

Foundry

Recommended Mitigation Steps

The problem here lies in the implementation of `INFTDropCollectionMint(nftContract).mintCountTo()`. You could add an additional check in `NFTDropMarketFixedPriceSale.mintCountTo()` using `ERC721(nftContract).balanceOf()`.

```
+ uint256 balanceBefore = IERC721(nftContract).balanceOf(msg.ser
207:      firstTokenId = INFTDropCollectionMint(nftContract).mint
+ uint256 balanceAfter = IERC721(nftContract).balanceOf(msg.senc
+ require(balanceAfter == balanceBefore + count, "minting failec
```

itsmeSTYJ (warden) commented:

> This assumes a custom NFT contract with a bad implementation of `mintCountTo` which *may* be a stretch but I agree that your mitigation steps should be added as a sanity check.

HardlyDifficult (Foundation) confirmed and commented:

> We will be making the recommended change.

> There's not really anything we can do to completely stop malicious contracts - this is an inherit risk with NFT marketplaces. Even the recommended solution here is

> something a malicious contract could fake in order to bypass that requirement.

> What sold us on making a change here was not malicious creators / contracts but instead potential errors in the implementation or misunderstanding of the interface requirements our marketplace expects. To prevent these errors, we are introducing the recommended change (and it only added 1,300 gas to the mint costs!)

[M-07] NFT of NFT collection or NFT drop collection can be locked when calling _mint or mintCountTo function to mint it to a contract that does not support ERC721 protocol

Submitted by rbserver, also found by 0xc0ffEE, 0xsolstars, berndartmueller, Bnke0x0, brgltd, cccz, CodingNameKiki, Deivitto, Diraco, Dravee, durianSausage, erictee, ignacio, IllIIll, joestakey, KIntern_NA, Lambda, LeoS, Noah3o6, oyc_109, ReyAdmirado, Rohan16, Rolezn, Sm4rty, Treasure-Seeker, zeesaw, and zkhorse

https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTCollection.sol#L262-L274

https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTDropCollection.sol#L171-L187

Impact

When calling the following `_mint` or `mintCountTo` function for minting an NFT of a NFT collection or NFT drop collection, the OpenZeppelin's `ERC721Upgradeable` contract's `_mint` function is used to mint the NFT to a receiver. If such receiver is a contract that does not support the ERC721 protocol, the NFT will be locked and cannot be retrieved.

https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTCollection.sol#L262-L274

```
  function _mint(string calldata tokenCID) private onlyCreator r
    require(bytes(tokenCID).length != 0, "NFTCollection: tokenCI
    require(!cidToMinted[tokenCID], "NFTCollection: NFT was alre
    unchecked {
      // Number of tokens cannot overflow 256 bits.
      tokenId = ++latestTokenId;
      require(maxTokenId == 0 || tokenId <= maxTokenId, "NFTColl
      cidToMinted[tokenCID] = true;
      _tokenCIDs[tokenId] = tokenCID;
      _mint(msg.sender, tokenId);
      emit Minted(msg.sender, tokenId, tokenCID, tokenCID);
    }
  }
```

https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTDropCollection.sol#L171-L187

```
function mintCountTo(uint16 count, address to) external onlyMi
  require(count != 0, "NFTDropCollection: `count` must be grea

  unchecked {
    // If +1 overflows then +count would also overflow, unless
    firstTokenId = latestTokenId + 1;
  }
  latestTokenId = latestTokenId + count;
  require(latestTokenId <= maxTokenId, "NFTDropCollection: Exc

  for (uint256 i = firstTokenId; i <= latestTokenId; ) {
    _mint(to, i);
    unchecked {
      ++i;
    }
  }
}
```

For reference, OpenZeppelin's documentation for `_mint` states: "Usage of this method is discouraged, use _safeMint whenever possible".

Proof of Concept

The following steps can occur when minting an NFT of a NFT collection or NFT drop collection.

1. The `_mint` or `mintCountTo` function is called with `msg.sender` or the `to` input corresponding to a contract.

2. The OpenZeppelin’s `ERC721Upgradeable` contract’s `_mint` function is called with `msg.sender` or `to` used in Step 1 as the receiver address.
3. Since calling the OpenZeppelin’s `ERC721Upgradeable` contract’s `_mint` function does not execute the same contract’s `_checkOnERC721Received` function, it is unknown if the receiving contract inherits from the `IERC721ReceiverUpgradeable` interface and implements the `onERC721Received` function or not. It is possible that the receiving contract does not support the ERC721 protocol, which causes the minted NFT to be locked.

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTCollection.sol#L271 can be changed to the following code.

```
    _safeMint(msg.sender, tokenId);
```

Also, https://github.com/code-423n4/2022-08-foundation/blob/main/contracts/NFTDropCollection.sol#L182 can be changed to the following code.

```
    _safeMint(to, i);
```

HardlyDifficult (Foundation) confirmed and commented:

> Agree will fix.

> Generally we are inclined to skip “safe” by default - it can introduce reentrancy & reverting risk and increase gas costs. When a user is making an action to buy or mint an NFT for themselves, it’s very clear that they are trying to acquire an NFT - so using safe to ensure that they support NFTs seems like a Low risk concern and

> we are inclined to avoid potential reentrancy/reverts and save costs for the common user paths.

> However in this scenario the part that stood out as different is instead of minting for yourself (the msg.sender) we support minting to an arbitrary `to` address, e.g. for an airdrop type use case. Here specifically it does seem that sending to a list of addresses could be error prone, where a contract address without 721 support was incorrectly captured. To guard against that scenario specifically we are moving forward with this change.

> Then for consistency we have decided to use safeMint for both collection types because the difference is nuanced.

[M-08] `mintFromFixedPriceSale` for a custom contract can lead to users losing funds

Submitted by joestakey

`NFTDropMarketFixedPriceSale.createFixedPriceSale` allows creators to create a sale drop. A creator can create a drop sale for their custom NFT Contract that adheres to `INFTDropCollectionMint`.

`INFTDropCollectionMint.mintCountTo` must return the `firstTokenId` being minted, but it is not clear as to what should be returned upon all tokens being minted. A valid implementation could for instance return `0` if called after the last token has been minted.

But the drop market expects the call to mintCountTo to revert upon the last token being minted, meaning a user calling it afterwards would lose the ETH they sent.

Proof Of Concept

- Alice creates a `customERC721` contract adhering to `INFTDropCollectionMint`. She writes `mintCountTo()` so that it returns `0` if called when all the tokens have been minted.
- The sale happens and collectors call `mintFromFixedPriceSale` until all the tokens have been minted.

- Bob now calls `mintFromFixedPriceSale`. Because all the tokens have been minted, the call to mintCountTo does not revert but returns `0`.
- The function call then proceeds to distribute the funds.
- Bob have lost `mintCost` ETH, while not receiving any NFT.

You can find this Foundry test reproducing the issue.

Note that this is not an issue of a malicious creator rugging collectors with a malicious implementation: they have implemented their contract to adhere to `INFTDropCollectionMint`, and the sale went as expected.

It is not unrealistic to imagine collectors monitoring `CreateFixedPriceSale` and calling `mintFromFixedPriceSale` based on it. In this case, all the `mintFromFixedPriceSale` processed after the last token being minted would lead to loss of funds.

Tools Used

Foundry

Recommend Mitigation Steps

You can add an additional check in `NFTDropMarketFixedPriceSale.mintCountTo()` using `ERC721(nftContract).balanceOf()`.

```
+ uint256 balanceBefore = IERC721(nftContract).balanceOf(msg.ser
207:      firstTokenId = INFTDropCollectionMint(nftContract).mint
+ uint256 balanceAfter = IERC721(nftContract).balanceOf(msg.senc
+ require(balanceAfter == balanceBefore + count, "minting failec
```

You can also specify in `INFTDropCollectionMint` that `mintCountTo` must revert if called after all tokens have been minted.

HardlyDifficult (Foundation) marked as duplicate and commented:

> Although this submission uses a different POC, we believe it's the same issue & root cause as #211.

> Dupe of #211.

joestakey (warden) commented:

> I will argue this issue is actually different than #211, although they both come from the same function call:

- In #211, the issue lies in the logic performed in `INFTDropCollectionMint.mintCountTo()` , more precisely the fact that a malicious implementation can perform incorrect state logic, which results in any collector calling `NFTDropMarketFixedPriceSale.mintFromFixedPriceSale` losing funds without receiving the expected amount of NFTs.
- Here, the issue lies in the return value of `INFTDropCollectionMint.mintCountTo()` in an edge case - when all the tokens have been minted. There is no malicious implementation or wrong state logic: users calling `NFTDropMarketFixedPriceSale.mintFromFixedPriceSale` will receive the expected amount of NFTs. The problem is when the minting is done: the `DropMarket` expect subsequent calls to `mintCountTo()` to fail. While you can argue not reverting after the final token has been minted is breaking a semantic requirement, it still complies with the interface `INFTDropCollectionMint` . Not reverting on failure is a behavior that exists in other standards, such as some ERC20 tokens for instance, like ZRX.

> To illustrate the difference between the two issues, take the NFT contract used in the PoC for this issue: users calling `NFTDropMarketFixedPriceSale.mintFromFixedPriceSale` will receive the expected amount of NFTs - i.e. it is not affected by the issue #211. The problem arises only upon the final token being minted.

> In summary, #211 is about malicious implementations that users should be made aware of (docs or UI warnings) while this issue has to do with the fact `INFTDropCollectionMint.mintCountTo()` should define a stricter behavior when the last token has been minted, perhaps by adding a comment such as:

```
    /// if the minting fails it MUST result in the transaction being
```

> @HickupHH3 thoughts?

HickupHH3 (judge) commented:

> The root cause for both issues is about the “potential errors in the implementation or misunderstanding of the interface requirements”. Simply put, the ambiguity regarding the specification of `mintCountTo()` allows for it to be exploited. As you pointed out, #211 is exploited by malicious implementations while this issue happens even if the implementation is seemingly compliant to the interface because of the ambiguity.

> It’s a tough decision because while the methods are different, the root cause and consequence (users losing funds) are the same.

> I’ll side with you on this one, because the attack vectors are quite distinct. It’s similar to how I separated the strategist rug vectors for the Rubicon contest.

Low Risk and Non-Critical Issues

For this contest, 73 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Saw-mon_and_Natalie** received the top score from the judge.

The following wardens also submitted reports: Dravee, Lambda, IllIllI, rbserver, 0x1f8b, berndartmueller, Chom, zkhorse, carlitox477, 0xSmartContract, csanuragjain, joestakey, 0xsolstars, bin2chen, Ruhum, shenwilly, Deivitto, 0xNazgul, d3e4, yash90, 0xDjango, 0xSolus, c3phas, Rolezn, simon135, Bnke0x0, robee, rvierdiiev, 0x52, auditor0517, danb, delfin454000, durianSausage, erictee, ladboy233, oyc_109, PwnedNoMore, TomJ, gogo, Kumpa, mics, ret2basic, rokinot, thank_you, Treasure-Seeker, wagmi, Waze, _141345_, brgltd, bulej93, JC, zeesaw, fatherOfBlocks, sikorico, apostle0x01, jonatascm, Vexjon, cryptphi, 0xackermann, 0xmatt, bobirichman, DevABDee, ElKu, iamwhitelights, LeoS, Rohan16, Sm4rty, MiloTruck, ReyAdmirado, Aymen0909, cccz, *and* Yiko.

[1] contracts/mixins/collections/CollectionRoyalties.sol

On line 80, `supportsInterface` can be rewritten to avoid the `if/esle` branching:

```
function supportsInterface(bytes4 interfaceId) public view virtu
        return (
                interfaceId == type(IRoyaltyInfo).interfaceId ||
                interfaceId == type(ITokenCreator).interfaceId |
                interfaceId == type(IGetRoyalties).interfaceId |
                interfaceId == type(IGetFees).interfaceId ||
                super.supportsInterface(interfaceId)
        );
}
```

[2] contracts/interfaces/ICollectionFactory.sol

In `ICollectionFactory` on line 6, `IProxyCall` is never used and can safely be removed. Unless there is a plan to use it in the future. Maybe a comment explaining why it was imported here would be helpful.

[3] contracts/mixins/shared/ContractFactory.sol

On line 31, there is a check for `_contractFactory` to see if it already has a code. I guess this is an extra check that can be removed. Since if `_contractFactory` calls the constructor here in its own constructor by then `_contractFactory.isContract() = _contractFactory.code.length == 0`. Also, it is possible that a wrong contract address is passed here, so the check would not really do anything. This will only check against accidental EOA addresses used for `_contractFactory`. So we could possibly remove the following lines:

```
 5      import "@openzeppelin/contracts-upgradeable/utils/Addres
13      using AddressUpgradeable for address;
31      require(_contractFactory.isContract(), "ContractFactory:
```

If there is a stricter condition for the allowed `contractFactory` addresses, maybe we could use that instead. One possible idea is an array of implementation contract code hashes that we could check. Or maybe contracts that have a function similar to `supportsInterface` that returns a magic number which we could check here.

[4] contracts/NFTCollection.sol

4.1 Shorter inheritance list

The inheritance contracts on line 29-40 can be consolidated into a shorter list:

```
contract NFTCollection is
  INFTCollectionInitializer,
  ContractFactory,
  SequentialMintCollection,
  CollectionRoyalties
{
```

Then you would need to adjust the overrides on lines 255 and 316

4.2 `CID` need to be unique per `tokenID`

Different `tokenID` s can not share the same `CID` by design. Although it is possible to design the contract so that some tokens share the same `CID` to save storage and also server space for off-chain contents.

[5] contracts/NFTCollectionFactory.sol

5.1 `.isContract()`

On lines 182, and 203 instead of checking if `addr.isContract()` to avoid setting the addresses to EOA by mistake it would be best to pass the code hash instead and check the code hash at those addresses. So for example:

Before:

```
constructor(address _rolesContract) {
        require(_rolesContract.isContract(), "NFTCollectionFacto

        rolesContract = IRoles(_rolesContract);
}
```

After:

```
    constructor(address _rolesContract, bytes32 codehash) {
            require(_rolesContract.codehash == codehash, "NFTCollect

            rolesContract = IRoles(_rolesContract);
    }
```

This is a stronger requirement since it would guarantee that the addresses are contracts and also they have the required code hash. For the functions to pass the `require` statements you would need to make 2 mistakes, one for the address and the other for the code hash. The probability of making this mistake should be theoretically lower than just passing a wrong address.

5.2 `versionNFTDropCollection`

Doesn't have an initializer like `versionNFTCollection`.

5.3 a better name can be chosen for `rolesContract`

`rolerManager` might be a better name for this immutable variable and would make it easier to remember what it does (ref. line 104).

[6] contracts/NFTDropCollection.sol

6.1 `supportsInterface` function

We can rewrite `supportsInterface` function (Lines 284-294) like the following block which would make it easier to read and possibly would save some gas.

```
    function supportsInterface(bytes4 interfaceId)
            public
            view
            override(ERC165Upgradeable, AccessControlUpgradeable, ER
            returns (bool)
    {
            return (
                    interfaceId == type(INFTDropCollectionMint).inte
                    super.supportsInterface(interfaceId)
            );
    }
```

6.2 The comment on line 175 needs a bit of correction

So the current comment says:

> If +1 overflows then +count would also overflow, **unless count==0 in which case the loop would exceed gas limits**

But `count` can not be zero if we have reached this line. Since we have already checked for a non-zero `count` on line 172

So we can change the comment to

```
    // If +1 overflows then +count would also overflow, since count
```

6.3 Shorter inheritance list

Like `NFTCollection`, the inheritence list for `NFTDropCollection` contract on lines 28-46 can be consolidated more.

```
    contract NFTDropCollection is
      INFTDropCollectionInitializer,
      INFTDropCollectionMint,
      ContractFactory,
      MinterRole,
      SequentialMintCollection,
      CollectionRoyalties
    {
```

The `overrides` on lines 245 and 287 would also need to be modified accordingly.

[7] contracts/mixins/nftDropMarket/NFTDropMarketFixedPriceSale.sol

In `mintFromFixedPriceSale` we can avoid the nested `if` blocks on lines 182-189. This would improve readability and analyze and it would have the same effect. On the plus side, it will also save gas for a reverting call where `saleConfig.limitPerAccount` is zero by avoiding the outer `if` block in the original code.

```
// Confirm that the collection has a sale in progress.
if (saleConfig.limitPerAccount == 0) {
        revert NFTDropMarketFixedPriceSale_Must_Have_Sale_In_Prc
}
// Confirm that the buyer will not exceed the limit specified af
if (IERC721(nftContract).balanceOf(msg.sender) + count > saleCor
        revert NFTDropMarketFixedPriceSale_Cannot_Buy_More_Than_
}
```

[8] contracts/FETH.sol (Out of Scope)

In `constructor` instead of passing `_lockupDuration` pass `_lockupInterval` to save on the exact division check.

So taking that into consideration the `constructor` would look like this:

```
constructor(
        address payable _foundationMarket,
        address payable _foundationDropMarket,
        uint256 _lockupInterval
) {
        if (!_foundationMarket.isContract()) {
                revert FETH_Market_Must_Be_A_Contract();
        }
        if (!_foundationDropMarket.isContract()) {
                revert FETH_Market_Must_Be_A_Contract();
        }
        if (_lockupInterval == 0) {
                revert FETH_Invalid_Lockup_Duration();
        }

        foundationMarket = _foundationMarket;
        foundationDropMarket = _foundationDropMarket;
        lockupInterval = _lockupInterval;
        lockupDuration = _lockupInterval * 24;
}
```

Also, the `_lockupInterval` check is moved up before the assignments to save gas in case of a revert. If there will be no revert, moving up the `if` block would not introduce any gas changes, since the check will be performed eventually.

[9] Line Width

Keep line width to max 120 characters for better readability.

[10] Hard-coded gas limits

In `contracts/mixins/shared/Constants.sol` we have 3 gas limit constants:

```
uint256 constant READ_ONLY_GAS_LIMIT = 40000;
uint256 constant SEND_VALUE_GAS_LIMIT_MULTIPLE_RECIPIENTS = 2100
uint256 constant SEND_VALUE_GAS_LIMIT_SINGLE_RECIPIENT = 20000;
```

These numbers are not future-proof as some hardforks introduce changes to gas costs. These potential future changes to gas costs might break some of the functionalities of the smart contracts that use these constants. This is something to keep in mind. If some hardfork, would break a smart contract using these numbers you would need to deploy new contracts with adjusted gas limit constants. Or you can also have these gas limits be mutable by admins on-chain. For example, all 3 of these values can be stored on-chain in 1 storage slot.

[11] `address.isContract` check

Lots of the contracts in this project import `import "@openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol"` and use `address.isContract()` to check if an address is a contract and not a EOA. I guess this is only a check if the deployer by mistake provides the wrong address. I think this should be double-checked off-chain. If an on-chain check is needed, there are other checks that can be done that are even more strict than just checking against EOA mistakes. For example, we can provide the contract as the second input to the constructor and check the address's codehash against that. Here is a template as an example:

```
    constructor(address c, bytes32 h) {
            if( c.codehash != h) {
                    revert CustomError();
            }
    }
```

Not only does this check for address with code, but also pinpoints the contract hash to a specific hash. Another type of check that can be used is to check if the provided contract address supports a specific `interfaceSupport` or call an endpoint of the contract expecting it to return a specific magic number.

Here is a list of places `isContract` has been used:

1. FETH.sol - L201
2. FETH.sol - L204
3. NFTCollectionFactory.sol - L182
4. NFTCollectionFactory.sol - L203
5. NFTCollectionFactory.sol - L227
6. PercentSplitETH.sol - L171
7. AddressLibrary.sol - L31
8. ContractFactory.sol - L31
9. FETHNode.sol - L23
10. FoundationTreasuryNode.sol - L48

[12] Simplify `supportsInterface` check

12.1 NFTDropCollection.sol

`NFTDropCollection.supportsInterface` (lines 284-295) can be changed to:

```
    function supportsInterface(bytes4 interfaceId)
            public
            view
            override(ERC165Upgradeable, AccessControlUpgradeable, ER
            returns (bool)
    {
            return (
                    interfaceId == type(INFTDropCollectionMint).inte
                    super.supportsInterface(interfaceId);
            );
    }
```

12.2 CollectionRoyalties.sol

`CollectionRoyalties.supportsInterface` (lines 80-91) can be changed to:

```
    function supportsInterface(bytes4 interfaceId) public view virtu
            return (
                    interfaceId == type(IRoyaltyInfo).interfaceId ||
                    interfaceId == type(ITokenCreator).interfaceId |
                    interfaceId == type(IGetRoyalties).interfaceId |
                    interfaceId == type(IGetFees).interfaceId ||
                    interfaceSupported = super.supportsInterface(int
            )
    }
```

[13] Floating Solidity Pragma Version

It’s best to use the same compiler version across all project files/team members. So having a fixed version pragma is a good practice. Most contracts use a floating pragma which would allow the patch number to be equal or higher than the specified patch number.

[14] Avoid Nested `if` Blocks

For better readability and analysis it is better to avoid nested `if` blocks. Here is an example:

14.1 FETH.sol (lines 482-492)

After edit:

```
    if (spenderAllowance == type(uint256).max) {
            return ;
    }

    if (spenderAllowance < amount) {
            revert FETH_Insufficient_Allowance(spenderAllowance);
    }
    // The check above ensures allowance cannot underflow.
    unchecked {
            spenderAllowance -= amount;
    }
    accountInfo.allowance[msg.sender] = spenderAllowance;
    emit Approval(from, msg.sender, spenderAllowance);
```

HardlyDifficult (judge) commented:

> Very detailed and thoughtful feedback — thank you!

> [1] supportsInterface can be rewritten to avoid the if/else branching:

> I think I do like this style more, will consider the change.

> [2] contracts/interfaces/ICollectionFactory.sol

> Agree, fixed.

> [3] contracts/mixins/shared/ContractFactory.sol

> Not sure I'm following this suggestion. There does not appear to be another .code.length type check included at the moment. Considering a stricter check is compelling but since this is an admin function call I think that may be overkill here.

> [4.1] Shorter inheritance list

> True but for top-level contracts I like to expand all inherited contracts to make it clear what all the dependencies are and the lineriazation order they are included in.

> [4.2] CID need to be unique per tokenID

> Agree. This is a primary goal of the NFTDropCollection. As you note there are other more flexible ways we could run with this type of approach and we may

> consider those in the future.
> [5.1] .isContract()

> Fair feedback. Considering a stricter check is compelling but since this is an admin function call I think that may be overkill here.

> [5.2] versionNFTDropCollection

> By design - the default value of 0 is correct there. NFTCollections were previously created by a different factory contract, we wanted the new factory to pick up version where that left off. But drops are new so starting at 0 is correct.

> [5.3] a better name can be chosen for rolesContract

> Agree, I like that name more and will update.

> [6.2] The comment on line 175 needs a bit of correction

> Good catch — this was missed after adding a require against count == 0. Will fix.

> [7] contracts/mixins/nftDropMarket/NFTDropMarketFixedPriceSale.sol

> Although minor, this approach was used to save gas for the happy case scenario since it avoids a second if condition.

> [8] contracts/FETH.sol (Out of Scope)

> Fair feedback, but I think the current approach is easier to reason about. And saving admin-only gas is not a goal for us.

> [9] Line Width

> Our linter is configured to require 120… although maybe you mean we are adding new lines too early in some instances (?)

> [10] Hard-coded gas limits

> Fair feedback. However the use case requires some gas limit to be defined and it’s not clear there is a viable alternative here.

> [11] address.isContract check

> This is good feedback. ATM these checks are there to help avoid simple errors by the admin. I'm not sure that the stricter check is worth the complexity to maintain.

> [12] Use fixed pragma

> Disagree. We intentionally use a floating pragma in order to make integrating with contracts easier. Other contract developers are looking to interact with our contracts and they may be on a different version than we use. The pragma selected for our contracts is the minimum required in order to correctly compile and function. This way integration is easier if they lag a few versions behind, or if they use the latest but we don't bump our packages frequently enough, and when we do upgrade versions unless there was a breaking solidity change — it should just swap in by incrementing our npm package version.

> [14] Avoid Nested if Blocks

> (out of scope) I agree that style is better, will fix.

HickupHH3 (judge) commented:

> Slightly disagree with #3. Agree with sponsor that the suggestion isn't clear.

Gas Optimizations

For this contest, 73 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: c3phas, 0x1f8b, Deivitto, IllIllI, 0xkatana, Saw-mon_and_Natalie, ReyAdmirado, oyc_109, Bnke0x0, MiloTruck, simon135, gogo, JC, ajtra, erictee, 0xSmartContract, jag, saian, TomJ, zkhorse, _141345_, Rolezn, Aymen0909, DevABDee, joestakey, d3e4, fatherOfBlocks, Noah3o6, Tomio, Trabajo_de_mates, Waze, durianSausage, Amithuddar, LeoS, robee, Rohan16, Sm4rty, 0xNazgul, bobirichman, carlitox477, CodingNameKiki, ElKu, pfapostol, 0x040, 0xbepresent, 0xDjango, mics, apostle0x01, cRat1st0s, Diraco, Fitraldys, Funen, gerdusx, ignacio, Metatron, samruna, SpaceCake, zeesaw, zuhaibmohd, 0xHarry, brgltd, bulej93, Chom, hakerbaya, ladboy233, medikko, newfork01, rvierdiiev, sach1r0, sikorico, wagmi, *and* Yiko.

Code Impressions

Overall, the code is pretty optimized:

- Using clones to deploy contracts is an excellent call
- The `unchecked` statements are well used
- Storage variables are tightly packed

Just one particular finding was present across the whole project:

- The revert strings are too long. Please try to make them fit in 32 bytes (use the first letters of the contract as a prefix, as an example, like `NFTCF` instead of `NFTCollectionFactory`), or use Custom Errors consistently

Due to some inconsistencies with the `gas-stories.txt` file, I unfortunately did not attach it.

[G-01] Check for `bytes(_symbol).length > 0` before calling `NFTDropCollection.initialize()`, like it's done for `NFTCollection.initialize()`

This could save a lot of gas if the revert condition is met.

For `NFTCollection`, the check is made in `NFTCollectionFactory.createNFTCollection()`.

- NFTCollectionFactory.sol#L262

```
File: NFTCollectionFactory.sol
257:   function createNFTCollection( //@audit-ok OK function
258:     string calldata name,
259:     string calldata symbol,
260:     uint256 nonce
261:   ) external returns (address collection) {
262:     require(bytes(symbol).length != 0, "NFTCollectionFactor
263:
264:     // This reverts if the NFT was previously created usinc
265:     collection = implementationNFTCollection.cloneDetermini
```

```
266:
267:      INFTCollectionInitializer(collection).initialize(payabl
268:
269:      emit NFTCollectionCreated(collection, msg.sender, versi
270:    }
```

However, for `NFTDropCollection`, the check is made way further, after even the contract's creation (during the initialization):

- [NFTDropCollection.sol#L130]()

```
File: NFTDropCollection.sol
120:    function initialize(
121:      address payable _creator,
122:      string calldata _name,
123:      string calldata _symbol,
124:      string calldata _baseURI,
125:      bytes32 _postRevealBaseURIHash,
126:      uint32 _maxTokenId,
127:      address _approvedMinter,
128:      address payable _paymentAddress
129:    ) external initializer onlyContractFactory validBaseURI(_
130:      require(bytes(_symbol).length > 0, "NFTDropCollection:
131:      require(_maxTokenId > 0, "NFTDropCollection: `_maxToken
```

Consider moving the check in `NFTCollectionFactory._createNFTDropCollection()`:

- [NFTCollectionFactory.sol#L396]()

```
File: NFTCollectionFactory.sol
386:    function _createNFTDropCollection(
387:      string calldata name,
388:      string calldata symbol,
389:      string calldata baseURI,
390:      bytes32 postRevealBaseURIHash,
391:      uint32 maxTokenId,
392:      address approvedMinter,
393:      address payable paymentAddress,
394:      uint256 nonce
```

```
395:   ) private returns (address collection) {
396:     // This reverts if the NFT was previously created usin
+ 396:     require(bytes(symbol).length ! 0, "NFTDropCollection:
397:     collection = implementationNFTDropCollection.cloneDeter
398:
399:     INFTDropCollectionInitializer(collection).initialize(
400:       payable(msg.sender),
401:       name,
402:       symbol,
403:       baseURI,
404:       postRevealBaseURIHash,
405:       maxTokenId,
406:       approvedMinter,
407:       paymentAddress
408:     );
```

This would save the deployment cost of an impossible to initialize contract (which would further need to be destroyed before being redeployed).

[G-02] Caching storage values in memory

The code can be optimized by minimizing the number of SLOADs.

SLOADs are expensive (100 gas after the 1st one) compared to MLOADs/MSTOREs (3 gas each). Storage values read multiple times should instead be cached in memory the first time (costing 1 SLOAD) and then read from this cache to avoid multiple SLOADs.

- Saving many SLOADs (including in a for-loop):

```
File: NFTDropCollection.sol
171:   function mintCountTo(uint16 count, address to) external
172:     require(count != 0, "NFTDropCollection: `count` must be
173:
+ 173:      uint32 _latestTokenId = latestTokenId;
174:     unchecked {
175:       // If +1 overflows then +count would also overflow,
- 176:        firstTokenId = latestTokenId + 1; //@audit gas: SL
+ 176:        firstTokenId = _latestTokenId + 1;
177:     }
- 178:      latestTokenId = latestTokenId + count; //@audit gas:
+ 178:      _latestTokenId = _latestTokenId + count;
+ 178:     latestTokenId = _latestTokenId;
- 179:     require(latestTokenId <= maxTokenId, "NFTDropCollecti
+ 179:     require(_latestTokenId <= maxTokenId, "NFTDropCollect
180:
- 181:     for (uint256 i = firstTokenId; i <= latestTokenId; )
+ 181:     for (uint256 i = firstTokenId; i <= _latestTokenId; )
182:       _mint(to, i);
183:       unchecked {
184:         ++i;
185:       }
186:     }
187:   }
```

- [Saving 3 SLOADs (+ a pre-increment is cheaper, but this is counter-balanced with the memory variable)](#):

```
File: NFTCollectionFactory.sol
202:   function adminUpdateNFTCollectionImplementation(address _
203:     require(_implementation.isContract(), "NFTCollectionFac
204:     implementationNFTCollection = _implementation;
+ 204:     uint32 _versionNFTCollection;
205:     unchecked {
206:       // Version cannot overflow 256 bits.
- 207:       versionNFTCollection++;
+ 207:       _versionNFTCollection = ++versionNFTCollection;
208:     }
209:
210:     // The implementation is initialized when assigned so t
211:     INFTCollectionInitializer(_implementation).initialize(
212:       payable(address(rolesContract)),
- 213:       string.concat("NFT Collection Implementation v", ve
+ 213:       string.concat("NFT Collection Implementation v", _v
- 214:       string.concat("NFTv", versionNFTCollection.toString
+ 214:       string.concat("NFTv", _versionNFTCollection.toStrin
215:     );
216:
- 217:     emit ImplementationNFTCollectionUpdated(_implementati
+ 217:     emit ImplementationNFTCollectionUpdated(_implementati
218:   }
```

- [Saving 3 SLOADs (+ a pre-increment is cheaper, but this is counter-balanced with the memory variable)](#)

```
File: NFTCollectionFactory.sol
226:   function adminUpdateNFTDropCollectionImplementation(addre
227:     require(_implementation.isContract(), "NFTCollectionFac
228:     implementationNFTDropCollection = _implementation;
+ 228:        uint32 _versionNFTDropCollection;
229:     unchecked {
230:       // Version cannot overflow 256 bits.
- 231:       versionNFTDropCollection++;
+ 231:       _versionNFTDropCollection = ++versionNFTDropCollect
232:     }
233:
- 234:     emit ImplementationNFTDropCollectionUpdated(_implemen
+ 234:     emit ImplementationNFTDropCollectionUpdated(_implemen
235:
236:     // The implementation is initialized when assigned so t
237:     INFTDropCollectionInitializer(_implementation).initiali
238:       payable(address(this)),
- 239:       string.concat("NFT Drop Collection Implementation v
- 240:       string.concat("NFTDropV", versionNFTDropCollection.
+ 239:       string.concat("NFT Drop Collection Implementation v
+ 240:       string.concat("NFTDropV", _versionNFTDropCollectio
241:       "ipfs://bafybeibvxnuaqtvaxu26gdgly2rm4g2piu7b2tqlx2ds
242:       0x13370000000000000000000000000000000000000000000000
243:       1,
244:       address(0),
245:       payable(0)
246:     );
247:   }
```

- [Saving 1 SLOAD. If we’re optimistic towards the presence of a `baseURI_` string here, this should be cached]:

```
File: NFTCollection.sol
332:   function _baseURI() internal view override returns (strin
- 333:     if (bytes(baseURI_).length != 0) {
+ 333:     string memory memBaseURI = baseURI_;
+ 333:     if (bytes(memBaseURI).length != 0) {
- 334:       return baseURI_;
+ 334:       return memBaseURI;
335:     }
336:     return "ipfs://";
337:   }
```

[G-03] Avoid emitting a storage variable when a memory value is available

When they are the same, consider emitting the memory value instead of the storage value:

- [NFTDropCollection.sol#L242]()

```
File: NFTDropCollection.sol
232:   function updatePreRevealContent(string calldata _baseURI,
233:     external
234:     validBaseURI(_baseURI)
235:     onlyWhileUnrevealed
236:     onlyAdmin
237:   {
238:     require(_postRevealBaseURIHash != bytes32(0), "NFTDropC
239:
240:     postRevealBaseURIHash = _postRevealBaseURIHash;
241:     baseURI = _baseURI;
- 242:     emit URIUpdated(baseURI, postRevealBaseURIHash);
+ 242:     emit URIUpdated(_baseURI, _postRevealBaseURIHash);
243:   }
```

- [NFTDropMarketFixedPriceSale.sol#L156]()

```
File: NFTDropMarketFixedPriceSale.sol
152:     // Save the sale details.
153:     saleConfig.seller = payable(msg.sender);
154:     saleConfig.price = price;
155:     saleConfig.limitPerAccount = limitPerAccount;
- 156:     emit CreateFixedPriceSale(nftContract, saleConfig.sel
+ 156:     emit CreateFixedPriceSale(nftContract, payable(msg.se
```

[G-04] Unchecking arithmetics operations that can't underflow/overflow

While this is inside an `external view` function, consider wrapping this in an `unchecked` statement so that external contracts calling this might save some gas:

- L245 can be unchecked due to L240

```
File: NFTDropMarketFixedPriceSale.sol
240:     if (currentBalance >= limitPerAccount) {
241:       // User has exhausted their limit.
242:       return 0;
243:     }
244:
- 245:     uint256 availableToMint = limitPerAccount - currentBa
+ 245:     uint256 availableToMint;
+ 245:     unchecked { availableToMint = limitPerAccount - curre
```

[G-05] Use `calldata` instead of `memory`

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly bypasses this loop.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gas-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

Affected code (around **60 gas** to be saved):

```
File: NFTCollectionFactory.sol
363:   function createNFTDropCollectionWithPaymentFactory(
364:     string calldata name,
365:     string calldata symbol,
366:     string calldata baseURI,
367:     bytes32 postRevealBaseURIHash,
368:     uint32 maxTokenId,
369:     address approvedMinter,
370:     uint256 nonce,
- 371:     CallWithoutValue memory paymentAddressFactoryCall
+ 371:     CallWithoutValue calldata paymentAddressFactoryCall
```

```
372:    ) external returns (address collection) {
```

[G-06] Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
libraries/AddressLibrary.sol:31:    require(contractAddress.isCo
mixins/collections/SequentialMintCollection.sol:58:    require(m
mixins/collections/SequentialMintCollection.sol:63:    require(_
mixins/collections/SequentialMintCollection.sol:74:    require(t
mixins/collections/SequentialMintCollection.sol:87:    require(_
mixins/collections/SequentialMintCollection.sol:88:    require(m
mixins/collections/SequentialMintCollection.sol:89:    require(l
mixins/roles/AdminRole.sol:19:    require(hasRole(DEFAULT_ADMIN_
mixins/roles/MinterRole.sol:22:    require(isMinter(msg.sender)
mixins/shared/ContractFactory.sol:22:    require(msg.sender == c
mixins/shared/ContractFactory.sol:31:    require(_contractFactor
NFTCollection.sol:158:    require(tokenCreatorPaymentAddress !=
NFTCollection.sol:263:    require(bytes(tokenCID).length != 0, "
NFTCollection.sol:264:    require(!cidToMinted[tokenCID], "NFTCo
NFTCollection.sol:268:      require(maxTokenId == 0 || tokenId <
NFTCollection.sol:327:    require(_exists(tokenId), "NFTCollecti
NFTCollectionFactory.sol:173:    require(rolesContract.isAdmin(m
NFTCollectionFactory.sol:182:    require(_rolesContract.isContra
NFTCollectionFactory.sol:203:    require(_implementation.isContr
NFTCollectionFactory.sol:227:    require(_implementation.isContr
NFTCollectionFactory.sol:262:    require(bytes(symbol).length !=
NFTDropCollection.sol:88:    require(bytes(_baseURI).length > 0,
NFTDropCollection.sol:93:    require(postRevealBaseURIHash != by
NFTDropCollection.sol:130:    require(bytes(_symbol).length > 0,
NFTDropCollection.sol:131:    require(_maxTokenId > 0, "NFTDropC
NFTDropCollection.sol:172:    require(count != 0, "NFTDropColle
NFTDropCollection.sol:179:    require(latestTokenId <= maxTokenI
NFTDropCollection.sol:238:    require(_postRevealBaseURIHash !=
```

Consider shortening the revert strings to fit in 32 bytes.

[G-07] Duplicated conditions should be refactored to a modifier or function to save deployment costs

```
NFTCollectionFactory.sol:203:    require(_implementation.isContr
NFTCollectionFactory.sol:227:    require(_implementation.isContr
```

[G-08] Redundant check

The following require statement is redundant:

- SequentialMintCollection.sol#L63

```
File: SequentialMintCollection.sol
62:   function _initializeSequentialMintCollection(address payab
- 63:     require(_creator != address(0), "SequentialMintCollect
64:
65:     owner = _creator;
66:     maxTokenId = _maxTokenId;
67:   }
```

This is due to the fact that the `initialize()` methods have the `onlyContractFactory` modifier already, and that calls to `initialize` from the factory are not using `address(0)` (and hardly ever will in the future of the solution). See these initializations where the first argument is `creator`:

```
contracts/NFTCollectionFactory.sol:
  211:     INFTCollectionInitializer(_implementation).initialize
  212        payable(address(rolesContract)),


  237:     INFTDropCollectionInitializer(_implementation).initia
  238        payable(address(this)),
267:     INFTCollectionInitializer(collection).initialize(paya
```

```
399:     INFTDropCollectionInitializer(collection).initialize(
400        payable(msg.sender),
```

Consider removing this check.

[G-09] Pre-Solidity `0.8.13`: `> 0` is less efficient than `!= 0` for unsigned integers

Up until Solidity `0.8.13`: `!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

As the Solidity version used here is `0.8.12`, consider changing `> 0` with `!= 0` here:

```
NFTDropCollection.sol:88:    require(bytes(_baseURI).length > 0,
NFTDropCollection.sol:130:    require(bytes(_symbol).length > 0,
NFTDropCollection.sol:131:    require(_maxTokenId > 0, "NFTDropC
```

Also, please enable the Optimizer.

[G-10] `<array>.length` should not be looked up in every loop of a `for-loop`

Reading array length at each iteration of the loop consumes more gas than necessary.

In the best case scenario (length read on a memory variable), caching the array length in the stack saves around **3 gas** per iteration. In the worst case scenario (external calls at each iteration), the amount of gas wasted can be massive.

Here, consider storing the array's length in a variable before the for-loop, and use this new variable instead:

```
mixins/shared/MarketFees.sol:126:      for (uint256 i = 0; i < c
mixins/shared/MarketFees.sol:198:    for (uint256 i = 0; i < cre
mixins/shared/MarketFees.sol:484:          for (uint256 i = 0; i
mixins/shared/MarketFees.sol:503:      for (uint256 i = 1; i < c
```

[G-11] `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)

Pre-increments and pre-decrements are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Decrement:

- `i -= 1` is the most expensive form
- `i--` costs 11 gas less than `i -= 1`
- `--i` costs 5 gas less than `i--` (16 gas less than `i -= 1`)

Note that post-increments (or post-decrements) return the old value before incrementing or decrementing, hence the name *post-increment*:

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after
```

However, pre-increments (or pre-decrements) return the new value:

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before
```

In the pre-increment case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`.

Affected code:

```
NFTCollectionFactory.sol:207:      versionNFTCollection++;
NFTCollectionFactory.sol:231:      versionNFTDropCollection++;
```

Consider using pre-increments and pre-decrements where they are relevant (meaning: not where post-increments/decrements logic are relevant).

[G-12] Increments/decrements can be unchecked in for-loops

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Consider wrapping with an `unchecked` block here (around **25 gas saved** per instance):

```
mixins/shared/MarketFees.sol:198:    for (uint256 i = 0; i < cre
mixins/shared/MarketFees.sol:484:          for (uint256 i = 0; i
```

The change would be:

```
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
    // ...
+   unchecked { ++i; }
  }
```

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for `uint256` here.

[G-13] Use Custom Errors instead of Revert Strings to save Gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

Additionally, custom errors can be used inside and outside of contracts (including interfaces and libraries).

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Consider replacing **all revert strings** with custom errors in the solution, and particularly those that have multiple occurrences:

```
NFTCollectionFactory.sol:203:    require(_implementation.isContr
NFTCollectionFactory.sol:227:    require(_implementation.isContr
```

HardlyDifficult (Foundation) commented:

> Great report, the code diffs really help to understand your points. And the statements like `Saving 3 SLOADs` makes the impact clear. Thanks!

> [G-01] Check for bytes(_symbol).length > 0

> Agree, and it's good for consistency. Fixed.

> [G=02] Caching storage values in memory

> Agree, will fix this up. Except for the admin update functions since we are not trying to optimize for the admin and I think the code is a little cleaner as is.

> [G-03] Avoid emitting a storage variable when a memory value is available

> Agree, fixed.

> [G-04] Unchecking arithmetics operations that can't underflow/overflow

> Agree, changed.

> [G-05] calldata

> Valid & will fix. This saves ~60 gas on `createNFTDropCollectionWithPaymentFactory`

> [G-06] Use short error messages

> Agree but won't fix. We use up to 64 bytes, aiming to respect the incremental cost but 32 bytes is a bit too short to provide descriptive error messages for our users.

> [G-07] Duplicated conditions should be refactored to a modifier

> Agree, will consider a change here.

> [G-08] Redundant check

> Good catch! Agree, will fix

> [G-09] Pre-Solidity 0.8.13: > 0 is less efficient than != 0 for unsigned integers

> Ahh that's where it got fixed. I've been calling this invalid after testing — good to know where that had changed. We are compiling with 0.8.16 even though we have a floating 0.8.12.

> [G-10] Cache Array Length Outside of Loop

> May be theoretically valid, but won't fix. I tested this: gas-reporter and our gas-stories suite is reporting a small regression using this technique. It also hurts readability a bit so we wouldn't want to include it unless it was a clear win.

> [G-11] ++i costs less than i++

> Agree and will fix.

> [G-12] unchecked loop in `getFeesAndRecipients`

> `getFeesAndRecipients` is a read only function not intended to be used on-chain, but as a best practice we will add unchecked there as well.

> The other example provided was already unchecked — invalid.

> [G-13] Custom errors

> Agree but won't fix at this time. We use these in the market but not in collections. Unfortunately custom errors are still not as good of an experience for users (e.g. on etherscan). We used them in the market originally because we were nearing the max contract size limit and this was a good way to reduce the bytecode. We'll consider this in the future as tooling continues to improve.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top