Joyn contest
Findings & Analysis Report

2022-07-25

Table of contents

- Disclosures

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 Joyn smart contract system written in Solidity. The audit contest took place between March 30—April 1 2022.

Wardens

45 Wardens contributed reports to the Joyn contest:

1. leastwood
2. kirk-baird
3. peritoflores
4. hickuphh3
5. ych18
6. wuwe1
7. hyh
8. rayn
9. WatchPug (jtp and ming)
10. Dravee
11. Ruhum
12. hubble (ksk2345 and shri4net)
13. TomFrenchBlockchain
14. robee

15. defsec
16. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
17. saian
18. lllllll
19. 0x (Czar102 and pmerkleplant)
20. 0xDjango
21. pedroais
22. minhquanym
23. m9800
24. rfa
25. 0xkatana
26. kenta
27. BouSalman
28. z3s
29. securerodd
30. cccz
31. 0v3rf10w
32. 0x1f8b
33. hake
34. Hawkeye (0xwags and 0xmint)
35. Tomio
36. Funen
37. blackswordsman
38. 0xNazgul

This contest was judged by Michael De Luca.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities. Of these vulnerabilities, 9 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 25 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 21 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 Joyn contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 1,294 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.

High Risk Findings (9)

[H-01] ERC20 transferFrom return values not checked

Submitted by hickuphh3, also found by 0xDjango, kirk-baird, leastwood, m9800, minhquanym, and pedroais

The `transferFrom()` function returns a boolean value indicating success. This parameter needs to be checked to see if the transfer has been successful. Oddly, `transfer()` function calls were checked.

Some tokens like **EURS** and **BAT** will **not** revert if the transfer failed but return `false` instead. Tokens that don't actually perform the transfer and return `false` are still counted as a correct transfer.

Impact

Users would be able to mint NFTs for free regardless of mint fee if tokens that don't revert on failed transfers were used.

Recommended Mitigation Steps

Check the `success` boolean of all `transferFrom()` calls. Alternatively, use OZ's `SafeERC20`'s `safeTransferFrom()` function.

sofianeOuafir (Joyn) confirmed, disagreed with severity and commented:

> In my opinion, the severity level should be 3 (High Risk) instead of 2 (Med Risk)

> This is clearly an issue that needs to be fixed and represents a high risk. Currently, the current state of the code would allow users to mint tokens even if the payment isn't successful.

deluca-mike (judge) increased severity to High and commented:

> `payableToken` seems to be defined by whomever defines the `Collection` in `createProject`, so it would be possible for that person to define a payable token that, unbeknownst to them, behaves unexpectedly. I agree with high risk (unless there is some person/committee that is curates and validates the `paybaleToken`s ahead of time). Need to handle return from `transfer` and `transferFrom`, as well as erc20s that do not return anything from from `transfer` and `transferFrom`.

[H-02] Splitter: Anyone can call incrementWindow to steal the tokens in the contract

Submitted by cccz, also found by hickuphh3, kirk-baird, leastwood, pedroais, rayn, Ruhum, saian, WatchPug, and wuwe1

In general, the Splitter contract's incrementWindow function is only called when tokens are transfer to the contract, ensuring that the number of tokens stored in balanceForWindow is equal to the contract balance. However, anyone can use a fake RoyaltyVault contract to call the incrementWindow function of the Splitter contract, so that the amount of tokens stored in balanceForWindow is greater than the contract balance, after which the verified user can call the claim or claimForAllWindows functions to steal the tokens in the contract.

```
function incrementWindow(uint256 royaltyAmount) public retur
    uint256 wethBalance;

    require(
        IRoyaltyVault(msg.sender).supportsInterface(IID_IROY
        "Royalty Vault not supported"
    );
    require(
        IRoyaltyVault(msg.sender).getSplitter() == address(t
        "Unauthorised to increment window"
    );

    wethBalance = IERC20(splitAsset).balanceOf(address(this)
    require(wethBalance >= royaltyAmount, "Insufficient fund

    require(royaltyAmount > 0, "No additional funds for wind
    balanceForWindow.push(royaltyAmount);
    currentWindow += 1;
    emit WindowIncremented(currentWindow, royaltyAmount);
    return true;
}
```

Proof of Concept

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/Splitter.sol#L149-L169

Recommended Mitigation Steps

Add the onlyRoyaltyVault modifier to the incrementWindow function of the Splitter contract to ensure that only RoyaltyVault contracts with a specific address can call this function.

sofianeOuafir (Joyn) confirmed and commented:

> This is a high-risk issue and we intend to solve it. The mitigation provided looks good too and will be considered when fixing this issue 👍

deluca-mike (judge) commented:

> See a detailed exploit and recommended solution at #21

[H-03] DoS: `claimForAllWindows()` May Be Made Unusable By An Attacker

Submitted by kirk-baird, also found by hyh and Ruhum

When the value of `currentWindow` is raised sufficiently high `Splitter.claimForAllWindows()` will not be able to be called due to the block gas limit.

`currentWindow` can only ever be incremented and thus will always increase. This value will naturally increase as royalties are paid into the contract.

Furthermore, an attacker can continually increment `currentWindow` by calling `incrementWindow()`. An attacker can impersonate a `IRoyaltyVault` and send 1 WEI worth of WETH to pass the required checks.

Proof of Concept

Excerpt from `Splitter.claimForAllWindows()` demonstrating the for loop over `currentWindow` that will grow indefinitely.

```
for (uint256 i = 0; i < currentWindow; i++) {
```

```
            if (!isClaimed(msg.sender, i)) {
                setClaimed(msg.sender, i);

                amount += scaleAmountByPercentage(
                    balanceForWindow[i],
                    percentageAllocation
                );
            }
        }
```

`Splitter.incrementWindow()` may be called by an attacker increasing `currentWindow`.

```
    function incrementWindow(uint256 royaltyAmount) public retur
        uint256 wethBalance;

        require(
            IRoyaltyVault(msg.sender).supportsInterface(IID_IROY
            "Royalty Vault not supported"
        );
        require(
            IRoyaltyVault(msg.sender).getSplitter() == address(t
            "Unauthorised to increment window"
        );

        wethBalance = IERC20(splitAsset).balanceOf(address(this)
        require(wethBalance >= royaltyAmount, "Insufficient fund

        require(royaltyAmount > 0, "No additional funds for wind
        balanceForWindow.push(royaltyAmount);
        currentWindow += 1;
        emit WindowIncremented(currentWindow, royaltyAmount);
        return true;
    }
```

Recommended Mitigation Steps

Consider modifying the function `claimForAllWindows()` to instead claim for range of windows. Pass the function a `startWindow` and `endWindow` and only iterate through windows in that range. Ensure that `endWindow < currentWindow`.

sofianeOuafir (Joyn) confirmed, disagreed with severity and commented:

> In my opinion, the severity level should be 3 (High Risk) instead of 2 (Med Risk)
> duplicate of #3

deluca-mike (judge) increased severity to High and commented:

> While similar, I believe these issues are separate.

> Issue 3 indicates that the check that `msg.sender` is an authorized `RoyaltyVault` is faulty, since any contract can implement the interface and return the `Splitter` from `getSplitter`. While this should be fixed, as the warden suggested in the Recommended Mitigation Steps in #3, the issue raised in this issue can still occur when enough authorized `RoyaltyVault` contracts call `incrementWindow`.

> `claimForAllWindows` can remain, but as this warden suggests, a `claimForWindows(uint256 startWindow, uint256 endWindow, uint256 percentageAllocation, bytes32[] calldata merkleProof)` should exist, in case `claimForAllWindows` becomes prohibitively expensive, even organically (i.e. `currentWindow` is made very high due to sufficient authorized `incrementWindow` calls).

[H-04] CoreCollection can be reinitialized

Submitted by hyh, also found by 0xkatana, hubble, kirk-baird, leastwood, pedroais, rayn, rfa, Ruhum, saian, securerodd, and WatchPug

Reinitialization is possible for CoreCollection as `initialize` function sets `initialized` flag, but doesn't control for it, so the function can be rerun multiple times.

Such types of issues tend to be critical as all core variables can be reset this way, for example `payableToken`, which provides a way to retrieve all the contract funds.

However, setting priority to be medium as `initialize` is `onlyOwner`. A run by an external attacker this way is prohibited, but the possibility of owner initiated reset either by mistake or with a malicious intent remains with the same range of system breaking consequences.

Proof of Concept

`initialize` doesn't control for repetitive runs:

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol#L87

Recommended Mitigation Steps

Add `onlyUnInitialized` modifier to the `initialize` function:

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol#L46-L49

sofianeOuafir (Joyn) confirmed, disagreed with severity and commented:

> This is a high severity issue and we intend to fix it. The mitigation step looks great and will be considered to fix the issue.

> In my opinion, the severity level should be 3 (High Risk) instead of 2 (Med Risk)

deluca-mike (judge) increased severity to High and commented:

> ~~Not convinced this is a high severity issue, since erroneously changing `payableToken` via a re-initialization can simply be corrected by a re-re-initialization to set it back correctly. Further, as the warden mentioned, the initialize function is behind `onlyOwner`.~~

> ~~However, if it can be shown that users other than the owner can end up losing value due to the owner abusing or erroneously using `initialize`, then it can be promoted to High Severity.~~

> And just as I say that, #17 points that out clearly. So, yes, agreed, this is a High Severity issue.

[H-05] Centralisation RIsk: Owner Of `RoyaltyVault` Can Take All Funds

Submitted by kirk-baird, also found by 0xDjango, defsec, Dravee, hubble, hyh, leastwood, minhquanym, Ruhum, TomFrenchBlockchain, and WatchPug

The owner of `RoyaltyVault` can set `_platformFee` to any arbitrary value (e.g. 100% = 10000) and that share of the contracts balance and future balances will be set to the `platformFeeRecipient` (which is in the owners control) rather than the splitter contract.

As a result the owner can steal the entire contract balance and any future balances avoiding the splitter.

Proof of Concept

```
function setPlatformFee(uint256 _platformFee) external overr
    platformFee = _platformFee;
    emit NewRoyaltyVaultPlatformFee(_platformFee);
}
```

Recommended Mitigation Steps

This issue may be mitigated by add a maximum value for the `_platformFee` say 5% (or some reasonable value based on the needs of the platform).

Also consider calling `sendToSplitter()` before adjusting the `platformFee`. This will only allow the owner to change the fee for future value excluding the current contract balance.

Consider the following code.

```
function setPlatformFee(uint256 _platformFee) external overr
    require(_platformFee < MAX_FEE);
    sendToSplitter(); // @audit this will need to be public
    platformFee = _platformFee;
    emit NewRoyaltyVaultPlatformFee(_platformFee);
}
```

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue and we intend to fix it. The recommended mitigation looks good and will be considered.

> We also agree that this is a med risk as this can currently only be done by the contract owner which is us at Joyn

deluca-mike (judge) commented:

> Instead of having to come up with a "reasonable" `MAX_FEE`, consider instead just preventing the fee from ever being raised, and only allowing it to be lowered.

deluca-mike (judge) increased severity to High and commented:

> While I was originally leaning Medium Risk, after taking the arguments made by the duplicate issues into account, I am now leaning High Risk. The rationale is that, a DOS of `sendToSplitter` via a high `platformFee` not only harms stakeholders of the `RoyaltyVault` that would get the remainder of the balance, split, but may also prevent all NFT transfers if `sendToSplitter` is hooked into as part of all token transfer, via royalty payments. A malicious or disgruntled `RoyaltyVault` owner can hold all the NFTs hostage that call `sendToSplitter` atomically on transfers.

> So there are 2 issues that need to be solved here:

- protect NFT holders by ensuring `platformFee` (or any other values) cannot be set to a value that would cause `sendToSplitter` to fail (`splitterShare = 0` or `platformShare > balanceOfVault`), or don't have `sendToSplitter` be called on NFT transfers
- protect royalty split recipients by putting an arbitrary max to the fee, or only allowing the fee to be reduced

[H-06] STORAGE COLLISION BETWEEN PROXY AND IMPLEMENTATION (LACK EIP 1967)

Submitted by peritoflores

Storage collision because of lack of EIP1967 could cause conflicts and override sensible variables

Proof of Concept

```
    contract CoreProxy is Ownable {
            address private immutable _implement;
```

When you implement proxies, logic and implementation share the same storage layout. In order to avoid storage conflicts EIP1967 was proposed. (https://eips.ethereum.org/EIPS/eip-1967) The idea is to set proxy variables at fixed positions (like `impl` and `admin`).

For example, according to the standard, the slot for for logic address should be

`0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc`
(obtained as `bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)`).

In this case, for example, as you inherits from `Ownable` the variable _owner is at the first slot and can be overwritten in the implementation. There is a table at OZ site that explains this scenario more in detail

https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies

section “Unstructured Storaged Proxies”

Recommended Mitigation Steps

Consider using EIP1967

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue we want to investigate and fix if our investigation suggests we indeed need to make improvement on that end.

> At the same time, I have little idea of what is the impact of this issue. I’m not sure if it’s a high risk item

deluca-mike (judge) commented:

> Impact would be that an upgrade could brick a contract by simply rearranging inheritance order, or adding variables to an inherited contract, since the implantation slot will not be where it is expected. As the warden suggests, its critical that the implementation slot be fixed at an explicit location, and not an implicit location derived purely from inheritance and declaration order.

[H-07] Duplicate NFTs Can Be Minted if `payableToken` Has a Callback Attached to it

Submitted by leastwood

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol#L139-L167

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/ERC721Payable.sol#L50-L56

Impact

The `mintToken()` function is called to mint unique tokens from an `ERC721` collection. This function will either require users to provide a merkle proof to claim an airdropped token or pay a fee in the form of a `payableToken`. However, because the `payableToken` is paid before a token is minted, it may be possible to reenter the `mintToken()` function if there is a callback attached before or after the token transfer. Because `totalSupply()` has not been updated for the new token, a user is able to bypass the `totalSupply() + amount <= maxSupply` check. As a result, if the user mints the last token, they can reenter and mint duplicate NFTs as the way `tokenId` is generated will wrap around to the start again.

Proof of Concept

For the sake of this example, let's say `startingIndex = 0` and `maxSupply = 100`. `tokenId` is minted according to `((startingIndex + totalSupply()) % maxSupply) + 1`. If we see that a user mints a token where `totalSupply() = maxSupply - 1 = 99` and they reenter the function, then the next token to mint will actually be of index `1` as `totalSupply() % maxSupply = 0`. Calculating the first

`tokenId`, we get `((0 + 0) % maxSupply) + 1 = 1` which is a duplicate of our example.

Recommended Mitigation Steps

Consider adding reentrancy protections to prevent users from abusing this behaviour. It may also be useful to follow the checks-effects pattern such that all external/state changing calls are made at the end.

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue we intend to investigate and fix if indeed it is an issue

deluca-mike (judge) commented:

> This is a valid high risk issue. Also, for reference, the checks-effects-interactions (CEI) pattern suggests you, in this order:
>
> - perform checks that something can be done
> - perform the effects (update storage and emit events)
> - interact with other functions/contracts (since you may not be sure they will call out and re-enter)

[H-08] Funds cannot be withdrawn in `CoreCollection.withdraw`

Submitted by ych18, also found by hickuphh3 and WatchPug

The `CoreCollection.withdraw` function uses `payableToken.transferFrom(address(this), msg.sender, amount)` to transfer tokens from the `CoreCollection` contract to the `msg.sender` (who is the owner of the contract). The usage of `transferFrom` can result in serious issues. In fact, many ERC20 always require that in `transferFrom allowance[from][msg.sender] >= amount`, so in this case the call to the `withdraw` function will revert as the `allowance[CoreCollection][CoreCollection] == 0` and therefore the funds cannot ben withdrawn and will be locked forever in the contract.

Recommendation

Replace `transferFrom` with `transfer`

sofianeOuafir (Joyn) confirmed and commented:

> duplicate of #52

deluca-mike (judge) commented:

> This is not a duplicate, as it pertains to the wrong use of transfer vs transferFrom, which can have implications regarding required allowances.

[H-09] ERC20 tokens with no return value will fail to transfer

Submitted by ych18, also found by wuwe1

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L43-L46

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L51-L57

Vulnerability details

Although the ERC20 standard suggests that a transfer should return true on success, many tokens are non-compliant in this regard (including high profile, like USDT) . In that case, the .transfer() call here will revert even if the transfer is successful, because solidity will check that the RETURNDATASIZE matches the ERC20 interface.

Recommendation

Consider using OpenZeppelin's SafeERC20

sofianeOuafir (Joyn) confirmed and commented:

> duplicate of #52

deluca-mike (judge) commented:

> Actually not a duplicate of #52, since it pertains to return data size handling causing an issue, rather than failure to handle a true/false return at all. Still, same solution (use SafeERC20).

Medium Risk Findings (12)

[M-01] DoS: Attacker May Front-Run `createSplit()` With A `merkleRoot` Causing Future Transactions With The Same `merkleRoot` to Revert

Submitted by kirk-baird

A `merkleRoot` may only be used once in `createSplit()` since it is used as `salt` to the deployment of a `SplitProxy`.

The result is an attacker may front-run any `createSplit()` transaction in the mem pool and create another `createSplit()` transaction with a higher gas price that uses the same `merkleRoot` but changes the other fields such as the `_collectionContract` or `_splitAsset()`. The original transaction will revert and the user will not be able to send any more transaction with this `merkleRoot`.

The user would therefore have to generate a new merkle tree with different address, different allocations or a different order of leaves in the tree to create a new merkle root. However, the attack is repeateable and there is no guarantee this new merkle root will be successfully added to a split without the attacker front-running the transaction again.

Proof of Concept

The excerpt from `createSplitProxy()` shows the `merkleRoot()` being used as a `salt`.

```
splitProxy = address(
  new SplitProxy{salt: keccak256(abi.encode(merkleRoot))}()
);
```

Recommended Mitigation Steps

As seems to be the case here if the transaction address does NOT need to be known ahead of time consider removing the `salt` parameter from the contract deployment.

Otherwise, if the transaction address does need to be known ahead of time then consider concatenating `msg.sender` to the `merkleRoot`. e.g.

```
    splitProxy = address(
        new SplitProxy{salt: keccak256(abi.encode(msg.sender, merkle
      )
```

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue and intend to fix it

[M-02] Fixed Amount of Gas Sent in Call May Be Insufficient

Submitted by kirk-baird

Impact

The function `attemptETHTransfer()` makes a call with a fixed amount of gas, 30,000. If the receiver is a contract this may be insufficient to process the `receive()` function. As a result the user would be unable to receive funds from this function.

Proof of Concept

```
    function attemptETHTransfer(address to, uint256 value)
        private
        returns (bool)
    {
        // Here increase the gas limit a reasonable amount above
        // to send ETH to the recipient.
        // NOTE: This might allow the recipient to attempt a lim
        (bool success, ) = to.call{value: value, gas: 30000}("")
        return success;
```

```
        }
```

Recommended Mitigation Steps

Consider removing the `gas` field to use the default amount and protect from reentrancy by using reentrancy guards and the check-effects-interaction pattern. Note this pattern is already applied correctly.

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue we want to investigate and fix if needed

deluca-mike (judge) commented:

> Consider that changes to the network could result in re-pricing of opcodes which can make:
>
> - opcodes less expensive, which would re-introduce the opportunity for re-entrancy that the old costs prevented
> - opcodes more expensive, which would break this function altogether
>
> If `attemptETHTransfer` is the only way to extract ETH stored in this contract, then an Ethereum upgrade could result in lost funds. Luckily, as long as access lists remain in the protocol, a route would exist to recover and/or restore functionality.

[M-03] `RoyaltyVault.sol` is Not Equipped to Handle On-Chain Royalties From Secondary Sales

Submitted by leastwood

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol

Impact

The Joyn documentation mentions that Joyn royalty vaults should be equipped to handle revenue generated on a collection's primary and secondary sales. Currently, `CoreCollection.sol` allows the collection owner to receive a fee on each token mint, however, there is no existing implementation which allows the owner of a collection to receive fees on secondary sales.

After discussion with the Joyn team, it appears that this will be gathered from Opensea which does not have an on-chain royalty mechanism. As such, each collection will need to be added manually on Opensea, introducing further centralisation risk. It is also possible for users to avoid paying the secondary fee by using other marketplaces such as Foundation.

Recommended Mitigation Steps

Consider implementing the necessary functionality to allow for the collection of fees through an on-chain mechanism. `ERC2981` outlines the appropriate behaviour for this.

sofianeOuafir (Joyn) confirmed and commented:

> This is a great observation. Something we are aware of and intend to fix as well. 👍

[M-04] createProject can be frontrun

Submitted by wuwe1, also found by defsec, and kirk-baird

This is dangerous in scam senario because the malicious user can frontrun and become the owner of the collection. As owner, one can withdraw `paymentToken`. (note that _collections.isForSale can be change by frontrunner)

Proof of Concept

1. Anyone can call `createProject`.

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreFactory.sol#L70-L77

```
    function createProject(
```

```
    string memory _projectId,
    Collection[] memory _collections
  ) external onlyAvailableProject(_projectId) {
    require(
      _collections.length > 0,
      'CoreFactory: should have more at least one collection'
    );
```

Recommended Mitigation Steps

Two ways to mitigate.

1. Consider use white list on project creation.
2. Ask user to sign their address and check the signature against `msg.sender`. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol#L102

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue and we intend to fix it!

deluca-mike (judge) commented:

> The solutions listed in #34 and #35 are better.

[M-05] Gas costs will likely result in any fees sent to the Splitter being economically unviable to recover.

Submitted by TomFrenchBlockchain

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/CoreCollection.sol#L161-L163

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/CoreCollection.sol#L307

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/royalty-vault/contracts/RoyaltyVault.sol#L43-L50

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/splits/contracts/Splitter.sol#L149-L169

Impact

Collection owners will likely lose money by claiming fees unless the fees from a single NFT sale outweighs the cost of claiming it (not guaranteed).

Proof of Concept

Consider a new `Collection` with a `RoyaltyVault` and `Splitter` set and a nonzero mint fee.

When calling `mintToken`, the `_handlePayment` function is called https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/CoreCollection.sol#L161-L163

This will transfer the minting fee to the `RoyaltyVault` contract.

On each transfer of an NFT within the collection (for instance in the `_mint` call which occurs directly after calling `_handlePayment`), the `Collection` contract will call `sendToSplitter` on the `RoyaltyVault`:

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/CoreCollection.sol#L307

This function will forward the collection owners' portion of the minting on to the `Splitter` contract but another important thing to note is that we call `Splitter.incrementWindow`.

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/royalty-

vault/contracts/RoyaltyVault.sol#L43-L50

This results in the fees newly deposited into the `Splitter` contract being held in a separate "window" to the fees from previous or later mints and need to be claimed separately. Remember that this process happens on every NFT sale so the only funds which will be held in this window will be the minting fees for this particular mint.

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/splits/contracts/Splitter.sol#L149-L169

From this we can see that the `claim` function will only claim the fraction of the fees which are owed to the caller from a single NFT mint.

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/splits/contracts/Splitter.sol#L112-L142

Note that we can attempt to claim from multiple windows in a single transaction using `claimForAllWindow` but as the name suggests it performs an unbounded loop trying to claim all previous windows (even ones which have already been claimed!) and it is likely that with a new window for every NFT sold this function will exceed the gas limit (consider an 10k token collection resulting in trying to do 10k SSTOREs at 20k gas each.), leaving us to claim each window individually with `claim`.

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/splits/contracts/Splitter.sol#L35-L62

We're then forced to claim the royalties from each NFT sold one by one, having to send huge numbers of calls to `claim` incurring the base transaction cost many times over and performing many ERC20 transfers when we could have just performed one.

Compound on this that this needs to be repeated by everyone included in the split, multiplying the costs of claiming.

Medium risk as it's gas inefficiency to the point of significant value leakage where collection owners will lose a large fraction of their royalties.

Recommended Mitigation Steps

It doesn't seem like the "window" mechanism does anything except raise gas costs to the extent that it will be very difficult to withdraw fees so it should be removed.

sofianeOuafir (Joyn) confirmed and commented:

> This is a very fair point and we'll consider fixing this issue.

deluca-mike (judge) commented:

> Aside from the very valid points made by the warden, it seems that the heavy functions called from the `_beforeTokenTransfer` also create a lot of friction for the NFT owners. Might make more sense to have royalty splitting happen asynchronously from NFT transfers (i.e. let the cost of splitting be the burden of royalty stakeholders, not NFT holders).

> If NFT transferring becomes too costly, someone could make a "de-joyn" contract which can "re-tokenize" NFTs sent to it, so that they can be transferred without having to worry about `_beforeTokenTransfer` (or royalties, for that matter).

> See Recommended Mitigation Steps in #37 for more info.

[M-06] CoreCollection's token transfer can be disabled

Submitted by hyh, also found by robee

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L51-L57

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/Splitter.sol#L164

Impact

When royaltyAsset is an ERC20 that doesn't allow zero amount transfers, the following griefing attack is possible, entirely disabling CoreCollection token transfer by precision degradation as both reward distribution and vault balance can be manipulated.

Suppose splitterProxy is set, all addresses and fees are configured correctly, system is in normal operating state.

POC:

Bob the attacker setup a bot which every time it observes positive royaltyVault balance:

1. runs `sendToSplitter()`, distributing the whole current royaltyAsset balance of the vault to splitter and platform, so vault balance becomes zero
2. sends `1 wei` of royaltyAsset to the royaltyVault balance
3. each next CoreCollection token transfer will calculate `platformShare = (balanceOfVault * platformFee) / 10000`, which will be 0 as platformFee is supposed to be less than 100%, and then there will be an attempt to transfer it to `platformFeeRecipient`

If royaltyAsset reverts on zero amount transfers, the whole operation will fail as the success of `IERC20(royaltyAsset).transfer(platformFeeRecipient, platformShare)` is required for each CoreCollection token transfer, which invokes `sendToSplitter()` in `_beforeTokenTransfer()` as vault balance is positive in (3).

Notice, that Bob needn't to front run the transfer, it is enough to empty the balance in a lazy way, so cumulative gas cost of the attack can be kept moderate.

Setting severity to medium as on one hand, the attack is easy to setup and completely blocks token transfers, making the system inoperable, and it looks like system has to be redeployed on such type of attack with some manual management of user funds, which means additional operational costs and reputational damage. On the another, it is limited to the zero amount reverting royaltyAsset case or the case when platformFee is set to 100%.

That is, as an another corner case, if platformFee is set to 100%, `platformShare` will be `1 wei` and `splitterShare` be zero in (3), so this attack be valid for any royaltyAsset as it is required in Splitter's `incrementWindow` that `splitterShare` be positive.

Proof of Concept

As royaltyAsset can be an arbitrary ERC20 it can be reverting on zero value transfers:

https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers

`_beforeTokenTransfer` runs `IRoyaltyVault(royaltyVault).sendToSplitter()` whenever royaltyVault is set and have positive balance:

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol#L307

`sendToSplitter()` leaves vault balance as exactly zero as `splitterShare = balanceOfVault - platformShare`, i.e. no dust is left behind:

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L41

This way the balance opens up for the tiny amount manipulation.

One require that can fail the whole operation is `platformShare` transfer:

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L51-L57

Another is positive `royaltyAmount = splitterShare` requirement:

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/Splitter.sol#L164

Recommended Mitigation Steps

The issue is that token transfer, which is core system operation, require fee splitting to be done on the spot. More failsafe design is to try to send the fees and record the amounts not yet distributed, not requiring immediate success. The logic here is that transfer itself is more important than fee distribution, which is simple enough and can be performed in a variety of ways later.

Another issue is a combination of direct balance usage and the lack of access controls of the sendToSplitter function, but it only affects fee splitting and is somewhat harder to address.

As one approach consider trying, but not requiring `IRoyaltyVault(royaltyVault).sendToSplitter()` to run successfully as it can be executed later with the same result.

Another, a simpler one (the same is in `Griefing attack is possible making Splitter's claimForAllWindows inaccessible` issue), is to introduce action threshold, `MIN_ROYALTY_AMOUNT`, to `sendToSplitter()`, for example:

Now:

```
    /**
     * @dev Send accumulated royalty to splitter.
     */
    function sendToSplitter() external override {
        uint256 balanceOfVault = getVaultBalance();

        require(
            balanceOfVault > 0,
            "Vault does not have enough royalty Asset to send"
        );
            ...

        emit RoyaltySentToSplitter(...);
        emit FeeSentToPlatform(...);
    }
```

To be:

```
    /**
     * @dev Send accumulated royalty to splitter if it's above MIN_R
     */
    function sendToSplitter() external override {
        uint256 balanceOfVault = getVaultBalance();

        if (balanceOfVault > MIN_ROYALTY_AMOUNT) {
                    ...

                emit RoyaltySentToSplitter(...);
                emit FeeSentToPlatform(...);
        }
```

```
    }
```

sofianeOuafir (Joyn) confirmed

[M-07] Ineffective Handling of FoT or Rebasing Tokens

Submitted by kirk-baird, also found by defsec, hickuphh3, and leastwood

Certain ERC20 tokens may change user's balances over time (positively or negatively) or charge a fee when a transfer is called (FoT tokens). The accounting of these tokens is not handled by `RoyaltyVault.sol` or `Splitter.sol` and may result in tokens being stuck in `Splitter` or overstating the balance of a user

Thus, for FoT tokens if all users tried to claim from the Splitter there would be insufficient funds and the last user could not withdraw their tokens.

Proof of Concept

The function `RoyaltyVault.sendToSplitter()` will transfer `splitterShare` tokens to the `Splitter` and then call `incrementWindow(splitterShare)` which tells the contract to split `splitterShare` between each of the users.

```
        require(
            IERC20(royaltyAsset).transfer(splitterProxy, splitte
            "Failed to transfer royalty Asset to splitter"
        );
        require(
            ISplitter(splitterProxy).incrementWindow(splitterSha
            "Failed to increment splitter window"
        );
```

Since the `Splitter` may receive less than `splitterShare` tokens if there is a fee on transfer the `Splitter` will overstate the amount split and each user can claim more than their value (except the last user who claims nothing as the contract will have insufficient funds to transfer them the full amount).

Furthermore, if the token rebase their value of the tokens down while they are sitting in the `Splitter` the same issue will occur. If the tokens rebase their value up then this will not be accounted for in the protocol.

Recommended Mitigation Steps

It is recommend documenting clearly that rebasing token should not be used in the protocol.

Alternatively, if it is a requirement to handle rebasing tokens balance checks should be done before and after the transfer to ensure accurate accounting. Note: this makes the contract vulnerable to reentrancy and so a reentrancy guard must be placed over the function `sendToSplitter()`.

```
uint256 balanceBefore = IERC20(royaltyAsset).balanceOf(s
require(
    IERC20(royaltyAsset).transfer(splitterProxy, splitte
    "Failed to transfer royalty Asset to splitter"
);
uint256 balanceAfter = IERC20(royaltyAsset).balanceOf(sp
require(
    ISplitter(splitterProxy).incrementWindow(balanceAfte
    "Failed to increment splitter window"
);
```

sofianeOuafir (Joyn) confirmed

[M-08] CoreCollection: Starting index is pseudo-randomly generated, allowing for gameable NFT launches

Submitted by hickuphh3

In Paradigm's article **"A Guide to Designing Effective NFT Launches"**, one of the desirable properties of an NFT launch is **unexploitable fairness:** Launches *must* have true randomness to ensure that predatory users cannot snipe the rarest items at the expense of less sophisticated users.

It is therefore highly recommended to find a good source of entropy for the generation of the starting index. The `block.number` isn't random at all; it only incrementally increases, allowing anyone to easily compute the starting indexes of the next 10,000 blocks for instance.

```
contract FortuneTeller {
  function predictStartingIndexes(uint256 maxSupply, uint256 numl
    external
    view
    returns
    (uint256[] memory startingIndexes) {
    startingIndexes = new uint[](numBlocks);
    for (uint256 i = 0; i < numBlocks; ++i) {
        startingIndexes[i] = (uint256(
            keccak256(abi.encodePacked("CoreCollection", block.n
        ) % maxSupply) +
        1;
    }
  }
}
```

Coupled with the fact that the `_baseUri` is set upon initialization, the metadata could be scrapped beforehand to determine the rare NFTs.

Thus, NFT mints can be gamed / exploited.

Recommended Mitigation Steps

Consider exploring the use of commit-reveal schemes (eg. blockhash of a future block, less gameable but not foolproof) or VRFs.

sofianeOuafir (Joyn) acknowledged and commented:

> This is a known issue and for now, we're not going to solve it

deluca-mike (judge) commented:

> I'd reconsider not completely it, but rather doing `keccak256(abi.encodePacked("CoreCollection", blockhash(block.number - 1), block.coinbase, msg.sender, i))` so at least there is a much smaller set of

> users that will know what the previous blockhash and the current miner will be by the time the tx mines, and it will be salted with the sender.

> It's not perfect, but it's significantly better.

[M-09] Differing percentage denominators causes confusion and potentially brick claims

Submitted by hickuphh3, also found by 0x

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/Splitter.sol#L14

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/Splitter.sol#L103

Details & Impact

There is a `PERCENTAGE_SCALE = 10e5` defined, but the actual denominator used is `10000`. This is aggravated by the following factors:

1. Split contracts are created by collection owners, not the factory owner. Hence, there is a likelihood for someone to mistakenly use `PERCENTAGE_SCALE` instead of `10000`.
2. The merkle root for split distribution can only be set once, and a collection's split and royalty vault can't be changed once created.

Thus, if an incorrect denominator is used, the calculated claimable amount could exceed the actual available funds in the contract, causing claims to fail and funds to be permanently locked.

Recommended Mitigation Steps

Remove `PERCENTAGE_SCALE` because it is unused, or replace its value with `10_000` and use that instead.

P.S: there is an issue with the example scaled percentage given for platform fees `(5% = 200)`. Should be `500` instead of `200`.

sofianeOuafir (Joyn) confirmed and commented:

> This is an issue and we intend to fix it

[M-10] Add a timelock to `setPlatformFee()`

Submitted by Dravee

https://github.com/code-423n4/2022-03-joyn/blob/main/splits/contracts/SplitFactory.sol#L120

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L67

Impact

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate.

Here, no timelock capabilities seem to be used

I believe this impacts multiple users enough to make them want to react / be notified ahead of time.

Recommended Mitigation Steps

Consider adding a timelock to `setPlatformFee()`

sofianeOuafir (Joyn) acknowledged and commented:

> This is a good idea. We will consider mitigating this but at the same time it might not be something we will solve

[M-11] Not handling return value of transferFrom command can create inconsistency

Submitted by hubble

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/CoreCollection.sol#L175-L176

https://github.com/code-423n4/2022-03-joyn/blob/c9297ccd925ebb2c44dbc6eaa3effd8db5d2368a/core-contracts/contracts/ERC721Payable.sol#L54-L55

Vulnerability details

The below transferFrom command is called at two places in the core contracts, followed by an emit event

```
payableToken.transferFrom(msg.sender,recipient,_amount)
emit ...(...);
```

The return value is not checked during the payableToken.transferFrom

Impact

In the event of failure of payableToken.transferFrom(...), the emit event is still generated causing the downstream applications to capture wrong transaction / state of the protocol.

Proof of Concept

1. Contract CoreCollection.sol
   function withdraw()
2. Contract ERC721Payable.sol function _handlePayment

Recommended Mitigation Steps

Add a require statement as being used in the RoyaltyVault.sol

```
require( payableToken.transferFrom(msg.sender,recipient,_amount)
            "Failed to transfer amount to recipient" );
```

sofianeOuafir (Joyn) confirmed, disagreed with severity and commented:

> In my opinion, the severity level should be 3 (High Risk) instead of 2 (Med Risk)
> duplicate of #52

deluca-mike (judge) commented:

> No longer a duplicate because this issue pertains specifically to the false emission of events when an underlying call would have failed.

[M-12] `CoreCollection.setRoyaltyVault` doesn't check `royaltyVault.royaltyAsset` against `payableToken`, resulting in potential permanent lock of `payableTokens` in royaltyVault

Submitted by rayn

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/CoreCollection.sol#L185

https://github.com/code-423n4/2022-03-joyn/blob/main/core-contracts/contracts/ERC721Payable.sol#L50

https://github.com/code-423n4/2022-03-joyn/blob/main/royalty-vault/contracts/RoyaltyVault.sol#L31

Impact

Each CoreProxy is allowed to be associated with a RoyaltyVault, the latter which would be responsible for collecting minting fees and distributing to beneficiaries. Potential mismatch between token used in CoreProxy and RoyaltyVault might result in minting tokens being permanently stuck in RoyaltyVault.

Proof of Concept

Each RoyaltyVault can only handle the `royaltyVault.royaltyAsset` token assigned upon creation, if any other kind of tokens are sent to the vault, it would get stuck inside the vault forever.

```
function sendToSplitter() external override {
    ...
    require(
        IERC20(royaltyAsset).transfer(splitterProxy, splitte
        "Failed to transfer royalty Asset to splitter"
    );
    ...
    require(
        IERC20(royaltyAsset).transfer(
            platformFeeRecipient,
            platformShare
        ) == true,
        "Failed to transfer royalty Asset to platform fee re
    );
    ...
}
```

Considering that pairing of CoreProxy and RoyaltyVault is not necessarily handled automatically, and can sometimes be manually assigned, and further combined with the fact that once assigned, CoreProxy does not allow modifications of the pairing RoyaltyVault. We can easily conclude that if a CoreProxy is paired with an incompatible RoyaltyVault, the `payableToken` minting fees automatically transferred to RoyaltyVault by `_handlePayment` will get permanently stuck.

```
function setRoyaltyVault(address _royaltyVault)
    external
    onlyVaultUninitialized
{
    ...
    royaltyVault = _royaltyVault;
    ...
}

function _handlePayment(uint256 _amount) internal {
    address recipient = royaltyVaultInitialized()
        ? royaltyVault
        : address(this);
    payableToken.transferFrom(msg.sender, recipient, _amoun
    ...
}
```

Tools Used

vim, ganache-cli

Recommended Mitigation Steps

While assigning vaults to CoreProxy, check if `payableToken` is the same as `royaltyVault.royaltyAsset`

```
        function setRoyaltyVault(address _royaltyVault)
            external
            onlyVaultUninitialized
        {
            require(
                payableToken == _royaltyVault.royaltyAsset(),
                "CoreCollection : payableToken must be same as roya
            );
            ...
            royaltyVault = _royaltyVault;
            ...
        }
```

sofianeOuafir (Joyn) confirmed

deluca-mike (judge) decreased severity to Medium and commented:

> Downgraded to medium because, while a more automated and validated way of assigning a compatible royalty vault would prevent this issue, in the current framework you'd need to make a user error (albeit one that is not easy to spot), to lose funds.

Low Risk and Non-Critical Issues

For this contest, 25 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **CertoraInc** received the top score from the judge.

The following wardens also submitted reports: **robee**, **rayn**, **defsec**, **hyh**, **BouSalman**, **lllllll**, **0xDjango**, **kenta**, **hubble**, **Dravee**, **z3s**, **saian**, **Hawkeye**, **kirk-baird**, **0v3rf10w**, **wuwe1**, **hake**, **0x**, **Ruhum**, **0x1f8b**, **leastwood**, **ych18**, **WatchPug**, *and* **0xkatana**.

1. Different pragma versions - the core-contracts use `pragma solidity ^0.8.0` and the rest of the contracts use `pragma solidity ^0.8.4`
2. Use a specific solidity version instead of using `^`, to prevent future solidity versions impacting your code and creating issues.
3. In the comments and variable names you wrote ETH instead of wETH, which is un-correct (that's an ERC20 so it must be wETH) `sol function transferSplitAsset(address to, uint256 value) private returns (bool didSucceed) { // Try to transfer ETH to the given recipient. didSucceed = IERC20(splitAsset).transfer(to, value); require(didSucceed, "Failed to transfer ETH"); emit TransferETH(to, value, didSucceed); }`
4. In the comment before the function, you wrote returns instead of the known `@return` tag `sol /** * @notice Mint token * @dev A starting index is calculated at the time of first mint * returns a tokenId * @param _to Token recipient */ function mint(address _to) private returns (uint256 tokenId) { if (startingIndex == 0) { setStartingIndex(); } tokenId = ((startingIndex + totalSupply()) % maxSupply) + 1; _mint(_to, tokenId); }`
5. Low level calls (call, delegate call and static call) return success if the called contract doesn't exist (not deployed or destructed). This can be seen here https://github.com/Uniswap/v3-core/blob/main/audits/tob/audit.pdf (report #9) and here https://docs.soliditylang.org/en/develop/control-structures.html#error-handling-assert-require-revert-and-exceptions.

````
That means that in `attemptETHTransfer`, if `to` doesn't exist the call
```sol
function attemptETHTransfer(address to, uint256 value)
    private
    returns (bool)
    {
        // Here increase the gas limit a reasonable amount above the de
        // to send ETH to the recipient.
        // NOTE: This might allow the recipient to attempt a limited re
        (bool success, ) = to.call{value: value, gas: 30000}("");
        return success;
    }
```
````

6. Add `onlyUnInitialized` modifier to the `initialize` function, otherwise the owner can initialize the contract more than one time
7. `HASHED_PROOF` - upper case variable name that is not constant
8. If `startingIndex + totalSupply()` will reach `type(uint256).max` the system will be in a stuck state, that's because the calculation in the _mint function will overflow
9. Contracts not declaring that they implement their interfaces - for example `CoreCollection` and `CoreFactory` don't declare that they implement `ICoreCollection` and `ICoreFactory`
10. `ICoreFactory` is imported but not used in `CoreProxy`
11. Didn't check that the address of the given vault is not zero in the `setPlatformFee` function
12. Wrong comment in `RoyaltyVaultFactory` and `SplitFactory` ` sol /** * @dev Set Platform fee for collection contract. * @param *platformFee Platform fee in scaled percentage. (5% = 200) * @param _vault vault address. */ function setPlatformFee(address _vault, uint256 _platformFee) external { IRoyaltyVault(*vault).setPlatformFee(_platformFee); }

```
/**
* @dev Set Platform fee recipient for collection contract.
* @param _vault vault address.
* @param _platformFeeRecipient Platform fee recipient.
*/
function setPlatformFeeRecipient(
    address _vault,
    address _platformFeeRecipient
) external {
    require(_vault != address(0), "Invalid vault");
    require(
        _platformFeeRecipient != address(0),
        "Invalid platform fee recipient"
    );
    IRoyaltyVault(_vault).setPlatformFeeRecipient(_platformFeeRecipient
}
```

[sofianeOuafir (Joyn) commented](#):

> high quality report

deluca-mike (judge) commented:

> add onlyUnInitialized modifier to the initialize function, otherwise the owner can initialize the contract more than one time

Gas Optimizations

For this contest, 21 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: saian, rfa, Dravee, robee, 0xkatana, kenta, WatchPug, defsec, Tomio, z3s, Funen, 0x1f8b, 0v3rf10w, minhquanym, hake, ych18, securerodd, rayn, blackswordsman, *and* 0xNazgul.

[G-01] `require()` / `revert()` strings longer than 32 bytes cost extra gas

1. File: core-contracts/contracts/ERC721Payable.sol (lines 21-24)

```
require(
  !royaltyVaultInitialized(),
  'CoreCollection: Royalty Vault already initialized'
);
```

2. File: core-contracts/contracts/ERC721Payable.sol (lines 29-32)

```
require(
  royaltyVaultInitialized(),
  'CoreCollection: Royalty Vault not initialized'
);
```

3. File: core-contracts/contracts/ERC721Claimable.sol (line 23)

```
require(!claimableSet(), 'ERC721Claimable: Claimable is alre
```

4. File: core-contracts/contracts/CoreCollection.sol (line 47)

```
require(!initialized, "CoreCollection: Already initializ
```

5. File: core-contracts/contracts/CoreCollection.sol (lines 52-55)

```
require(
    _maxSupply > 0,
    "CoreCollection: Max supply should be greater than 0
);
```

6. File: core-contracts/contracts/CoreCollection.sol (line 146)

```
require(amount > 0, "CoreCollection: Amount should be gr
```

7. File: core-contracts/contracts/CoreCollection.sol (lines 189-192)

```
require(
    msg.sender == splitFactory || msg.sender == owner(),
    "CoreCollection: Only Split Factory or owner can ini
);
```

8. File: core-contracts/contracts/CoreCollection.sol (lines 204-207)

```
require(
    bytes(HASHED_PROOF).length == 0,
    "CoreCollection: Hashed Proof is set"
);
```

9. File: core-contracts/contracts/CoreCollection.sol (lines 220-223)

```
require(
    startingIndex == 0,
    "CoreCollection: Starting index is already set"
```

```
        );
```

10. File: core-contracts/contracts/CoreFactory.sol (lines 35-38)

```
        require(
          projects[_projectId].creator == address(0),
          'CoreFactory: Unavailable project id'
        );
```

11. File: core-contracts/contracts/CoreFactory.sol (lines 43-46)

```
        require(
          projects[_projectId].creator == msg.sender,
          'CoreFactory: Not an owner of the project'
        );
```

12. File: core-contracts/contracts/CoreFactory.sol (lines 51-54)

```
        require(
          collections[_collectionId] == address(0),
          'CoreFactory: Unavailable collection id'
        );
```

13. File: core-contracts/contracts/CoreFactory.sol (lines 74-77)

```
        require(
          _collections.length > 0,
          'CoreFactory: should have more at least one collection'
        );
```

14. File: royalty-vault/contracts/RoyaltyVault.sol (lines 34-37)

```
            require(
                balanceOfVault > 0,
                "Vault does not have enough royalty Asset to send"
```

```
        );
```

15. File: royalty-vault/contracts/RoyaltyVault.sol (lines 43-46)

```
        require(
            IERC20(royaltyAsset).transfer(splitterProxy, splitte
            "Failed to transfer royalty Asset to splitter"
        );
```

16. File: royalty-vault/contracts/RoyaltyVault.sol (lines 47-50)

```
        require(
            ISplitter(splitterProxy).incrementWindow(splitterSha
            "Failed to increment splitter window"
        );
```

17. File: royalty-vault/contracts/RoyaltyVault.sol (lines 51-57)

```
        require(
            IERC20(royaltyAsset).transfer(
                platformFeeRecipient,
                platformShare
            ) == true,
            "Failed to transfer royalty Asset to platform fee re
        );
```

18. File: splits/contracts/SplitFactory.sol (lines 48-51)

```
      require(
        splits[_splitId] == address(0),
        'SplitFactory : Split ID already in use'
      );
```

19. File: splits/contracts/SplitFactory.sol (lines 81-84)

```
      require(
```

```
        ICoreCollection(_collectionContract).owner() == msg.sender
        'Transaction sender is not collection owner'
      );
```

20. File: splits/contracts/Splitter.sol (lines 118-121)

```
        require(
            !isClaimed(msg.sender, window),
            "NFT has already claimed the given window"
        );
```

[G-02] Use a more recent version of solidity

Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

1. File: royalty-vault/contracts/RoyaltyVault.sol (line 2)

```
pragma solidity ^0.8.4;
```

2. File: royalty-vault/contracts/ProxyVault.sol (line 2)

```
pragma solidity ^0.8.4;
```

3. File: splits/contracts/SplitFactory.sol (line 2)

```
pragma solidity ^0.8.4;
```

4. File: splits/contracts/SplitProxy.sol (line 2)

```
pragma solidity ^0.8.4;
```

5. File: splits/contracts/Splitter.sol (line 2)

```
pragma solidity ^0.8.4;
```

[G-03] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

1. File: core-contracts/contracts/ERC721Payable.sol (line 2)

```
pragma solidity ^0.8.0;
```

2. File: core-contracts/contracts/CoreProxy.sol (line 2)

```
pragma solidity ^0.8.0;
```

3. File: core-contracts/contracts/ERC721Claimable.sol (line 2)

```
pragma solidity ^0.8.0;
```

4. File: core-contracts/contracts/CoreCollection.sol (line 2)

```
pragma solidity ^0.8.0;
```

5. File: core-contracts/contracts/CoreFactory.sol (line 2)

```
pragma solidity ^0.8.0;
```

[G-04] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

1. File: core-contracts/contracts/ERC721Payable.sol (line 8)

```
bool public isForSale;
```

2. File: core-contracts/contracts/CoreCollection.sol (line 20)

```
bool public initialized;
```

[G-05] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

1. File: core-contracts/contracts/CoreCollection.sol (lines 52-55)

```
require(
    _maxSupply > 0,
    "CoreCollection: Max supply should be greater than 0
);
```

2. File: core-contracts/contracts/CoreCollection.sol (line 146)

```
require(amount > 0, "CoreCollection: Amount should be gr
```

3. File: royalty-vault/contracts/RoyaltyVault.sol (lines 34-37)

```
        require(
            balanceOfVault > 0,
            "Vault does not have enough royalty Asset to send"
        );
```

4. File: splits/contracts/Splitter.sol (line 164)

```
        require(royaltyAmount > 0, "No additional funds for wind
```

[G-06] `<array>.length` should not be looked up in every loop of a `for` -loop

Even memory arrays incur the overhead of bit tests and bit shifts to calculate the array length

1. File: core-contracts/contracts/CoreFactory.sol (line 79)

```
        for (uint256 i; i < _collections.length; i++) {
```

2. File: splits/contracts/Splitter.sol (line 274)

```
        for (uint256 i = 0; i < proof.length; i++) {
```

[G-07] It costs more gas to initialize variables to zero than to let the default of zero be applied

1. File: core-contracts/contracts/CoreCollection.sol (line 279)

```
        for (uint256 i = 0; i < _amount; i++) {
```

2. File: splits/contracts/Splitter.sol (line 49)

```
uint256 amount = 0;
```

3. File: splits/contracts/Splitter.sol (line 50)

```
for (uint256 i = 0; i < currentWindow; i++) {
```

4. File: splits/contracts/Splitter.sol (line 274)

```
for (uint256 i = 0; i < proof.length; i++) {
```

[G-08] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second access of a state variable within a function. Less obvious optimizations include having local storage variables of mappings within state variable mappings or mappings within state variable structs, having local storage variables of structs within mappings, or having local caches of state variable contracts/addresses.

1. File: core-contracts/contracts/CoreCollection.sol (line 231)

```
emit StartingIndexSet(startingIndex);
```

2. File: core-contracts/contracts/CoreCollection.sol (line 264)

```
tokenId = ((startingIndex + totalSupply()) % maxSupply)
```

3. File: splits/contracts/SplitFactory.sol (line 161)

```
delete merkleRoot;
```

4. File: splits/contracts/SplitFactory.sol (line 171)

```
delete splitterProxy;
```

[G-09] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

1. File: core-contracts/contracts/CoreCollection.sol (line 79)

```
string memory _collectionName,
```

2. File: core-contracts/contracts/CoreCollection.sol (line 80)

```
string memory _collectionSymbol,
```

3. File: core-contracts/contracts/CoreCollection.sol (line 81)

```
string memory _collectionURI,
```

4. File: core-contracts/contracts/CoreCollection.sol (line 122)

```
string memory _collectionName,
```

5. File: core-contracts/contracts/CoreCollection.sol (line 123)

```
string memory _collectionSymbol
```

6. File: core-contracts/contracts/CoreFactory.sol (line 71)

```
string memory _projectId,
```

7. File: core-contracts/contracts/CoreFactory.sol (line 72)

```
Collection[] memory _collections
```

8. File: core-contracts/contracts/CoreFactory.sol (line 109)

```
string memory _projectId,
```

9. File: core-contracts/contracts/CoreFactory.sol (line 110)

```
Collection memory _collection
```

10. File: core-contracts/contracts/CoreFactory.sol (line 128)

```
function getProject(string memory _projectId)
```

11. File: splits/contracts/SplitFactory.sol (line 79)

```
string memory _splitId
```

12. File: splits/contracts/SplitFactory.sol (line 105)

```
string memory _splitId
```

[G-10] `++i` / `i++` should be `unchecked{++i}` / `unchecked{++i}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

1. File: core-contracts/contracts/CoreCollection.sol (line 279)

```
for (uint256 i = 0; i < _amount; i++) {
```

2. File: core-contracts/contracts/CoreFactory.sol (line 79)

```
        for (uint256 i; i < _collections.length; i++) {
```

3. File: splits/contracts/Splitter.sol (line 50)

```
            for (uint256 i = 0; i < currentWindow; i++) {
```

4. File: splits/contracts/Splitter.sol (line 274)

```
            for (uint256 i = 0; i < proof.length; i++) {
```

[G-11] `++i` costs less gas than `++i`, especially when it's used in `for`-loops (`--i` / `i--` too)

1. File: core-contracts/contracts/CoreCollection.sol (line 279)

```
            for (uint256 i = 0; i < _amount; i++) {
```

2. File: core-contracts/contracts/CoreFactory.sol (line 79)

```
        for (uint256 i; i < _collections.length; i++) {
```

3. File: splits/contracts/Splitter.sol (line 50)

```
            for (uint256 i = 0; i < currentWindow; i++) {
```

4. File: splits/contracts/Splitter.sol (line 274)

```
            for (uint256 i = 0; i < proof.length; i++) {
```

[G-12] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

1. File: splits/contracts/Splitter.sol (line 217)

```
function amountFromPercent(uint256 amount, uint32 percent)
```

[G-13] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code

1. File: splits/contracts/Splitter.sol (line 14)

```
uint256 public constant PERCENTAGE_SCALE = 10e5;
```

2. File: splits/contracts/Splitter.sol (line 15)

```
bytes4 public constant IID_IROYALTY = type(IRoyaltyVault).in
```

[G-14] Don't compare boolean expressions to boolean literals

`if (<x> == true) => if (<x>), if (<x> == false) => if (!<x>)`

1. File: royalty-vault/contracts/RoyaltyVault.sol (line 44)

```
IERC20(royaltyAsset).transfer(splitterProxy, splitte
```

2. File: royalty-vault/contracts/RoyaltyVault.sol (line 48)

```
ISplitter(splitterProxy).incrementWindow(splitterSha
```

3. File: royalty-vault/contracts/RoyaltyVault.sol (lines 52-55)

```
IERC20(royaltyAsset).transfer(
    platformFeeRecipient,
    platformShare
) == true,
```

[G-15] Remove unused variables

1. File: splits/contracts/Splitter.sol (line 14)

```
uint256 public constant PERCENTAGE_SCALE = 10e5;
```

[G-16] State variables only set in the constructor should be declared `immutable`

1. File: royalty-vault/contracts/ProxyVault.sol (line 9)

```
address internal royaltyVault;
```

[G-17] `require()` or `revert()` statements that check input arguments should be at the top of the function

1. File: splits/contracts/Splitter.sol (line 164)

```
require(royaltyAmount > 0, "No additional funds for wind
```

[G-18] `private` functions not called by the contract should be removed to save deployment gas

1. File: splits/contracts/Splitter.sol (lines 217-220)

```
function amountFromPercent(uint256 amount, uint32 percent)
    private
    pure
    returns (uint256)
```

2. File: splits/contracts/Splitter.sol (lines 248-250)

```
function attemptETHTransfer(address to, uint256 value)
    private
    returns (bool)
```

[G-19] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

1. File: royalty-vault/contracts/RoyaltyVault.sol (lines 95-100)

```
function supportsInterface(bytes4 interfaceId)
    public
    view
    virtual
    override(IRoyaltyVault, ERC165)
    returns (bool)
```

2. File: royalty-vault/contracts/RoyaltyVault.sol (line 88)

```
function getSplitter() public view override returns (address
```

[G-20] Use custom errors rather than `revert()` / `require()` strings to save deployment gas

1. File: royalty-vault/contracts/RoyaltyVault.sol (Various lines throughout the file)
2. File: splits/contracts/SplitFactory.sol (Various lines throughout the file)
3. File: splits/contracts/Splitter.sol (Various lines throughout the file)

[G-21] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

1. File: core-contracts/contracts/CoreCollection.sol (lines 78-87)

```
    function initialize(
        string memory _collectionName,
        string memory _collectionSymbol,
        string memory _collectionURI,
        uint256 _maxSupply,
        uint256 _mintFee,
        address _payableToken,
        bool _isForSale,
        address _splitFactory
    ) external onlyOwner onlyValidSupply(_maxSupply) {
```

2. File: core-contracts/contracts/CoreCollection.sol (lines 78-87)

```
    function initialize(
        string memory _collectionName,
        string memory _collectionSymbol,
        string memory _collectionURI,
        uint256 _maxSupply,
        uint256 _mintFee,
        address _payableToken,
        bool _isForSale,
        address _splitFactory
```

```
) external onlyOwner onlyValidSupply(_maxSupply) {
```

3. File: core-contracts/contracts/CoreCollection.sol (lines 105-109)

```
function initializeClaims(bytes32 _root)
    external
    onlyOwner
    onlyNotClaimableSet
    onlyValidRoot(_root)
```

4. File: core-contracts/contracts/CoreCollection.sol (lines 105-109)

```
function initializeClaims(bytes32 _root)
    external
    onlyOwner
    onlyNotClaimableSet
    onlyValidRoot(_root)
```

5. File: core-contracts/contracts/CoreCollection.sol (lines 105-109)

```
function initializeClaims(bytes32 _root)
    external
    onlyOwner
    onlyNotClaimableSet
    onlyValidRoot(_root)
```

6. File: core-contracts/contracts/CoreCollection.sol (lines 121-124)

```
function setCollectionMeta(
    string memory _collectionName,
    string memory _collectionSymbol
) external onlyOwner {
```

7. File: core-contracts/contracts/CoreCollection.sol (lines 139-145)

```
function mintToken(
```

```
        address to,
        bool isClaim,
        uint256 claimableAmount,
        uint256 amount,
        bytes32[] calldata merkleProof
    ) external onlyInitialized {
```

8. File: core-contracts/contracts/CoreCollection.sol (line 173)

```
    function withdraw() external onlyOwner {
```

9. File: core-contracts/contracts/CoreCollection.sol (lines 185-187)

```
    function setRoyaltyVault(address _royaltyVault)
        external
        onlyVaultUninitialized
```

10. File: core-contracts/contracts/CoreCollection.sol (line 203)

```
    function setHashedProof(string calldata _proof) external onl
```

11. File: core-contracts/contracts/CoreFactory.sol (lines 70-73)

```
  function createProject(
    string memory _projectId,
    Collection[] memory _collections
  ) external onlyAvailableProject(_projectId) {
```

12. File: core-contracts/contracts/CoreFactory.sol (lines 108-111)

```
  function addCollection(
    string memory _projectId,
    Collection memory _collection
  ) external onlyProjectOwner(_projectId) returns (address) {
```

13. File: core-contracts/contracts/CoreFactory.sol (lines 142-145)

```
function _createCollection(Collection memory _collection)
  private
  onlyAvailableCollection(_collection.id)
  returns (address)
```

14. File: royalty-vault/contracts/RoyaltyVault.sol (line 67)

```
  function setPlatformFee(uint256 _platformFee) external overr
```

15. File: royalty-vault/contracts/RoyaltyVault.sol (lines 76-79)

```
  function setPlatformFeeRecipient(address _platformFeeRecipie
      external
      override
      onlyOwner
```

16. File: splits/contracts/SplitFactory.sol (lines 75-80)

```
function createSplit(
  bytes32 _merkleRoot,
  address _splitAsset,
  address _collectionContract,
  string memory _splitId
) external onlyAvailableSplit(_splitId) returns (address split
```

17. File: splits/contracts/SplitFactory.sol (lines 102-106)

```
function createSplit(
  bytes32 _merkleRoot,
  address _splitAsset,
  string memory _splitId
) external onlyAvailableSplit(_splitId) returns (address split
```

18. File: splits/contracts/SplitFactory.sol (lines 120-122)

```
function setPlatformFee(address _vault, uint256 _platformFee)
  external
  onlyOwner
```

19. File: splits/contracts/SplitFactory.sol (lines 132-135)

```
function setPlatformFeeRecipient(
  address _vault,
  address _platformFeeRecipient
) external onlyOwner {
```

sofianeOuafir (Joyn) confirmed and commented:

> high quality report

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