Forgeries contest
Findings & Analysis Report

2023-02-28

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 Forgeries smart contract system written in Solidity. The audit contest took place between December 13—December 16 2022.

Wardens

83 Wardens contributed reports to the Forgeries contest:

1. 0x1f8b
2. 0xAgro
3. 0xdeadbeef0x
4. 9svR6w
5. Apocalypto (cRat1st0s, reassor, and M0ndoHEHE)
6. Aymen0909
7. BAHOZ
8. Bnke0x0
9. Bobface

10. Ch_301
11. Deivitto
12. HE1M
13. lllllll
14. Koolex
15. Madalad
16. Matin
17. PaludoX0
18. Rahoz
19. RaymondFam
20. ReyAdmirado
21. Rolezn
22. Ruhum
23. Sathish9098
24. SmartSek (0xDjango and hake)
25. Soosh
26. Titi
27. Trust
28. Zarf
29. adriro
30. aga7hokakological
31. ayeslick
32. bin2chen
33. btk
34. c3phas
35. carrotsmuggler
36. caventa
37. cccz
38. chaduke

39. codeislight
40. csanuragjain
41. ctrlc03
42. deliriusz
43. dic0de
44. dipp
45. evan
46. gasperpre
47. gz627
48. hansfriese
49. hihen
50. imare
51. immeas
52. indijanc
53. izhelyazkov
54. jadezti
55. kaliberpoziomka8552
56. kuldeep
57. ladboy233
58. maks
59. mookimgo
60. nadin
61. neko_nyaa
62. neumo
63. nicobevi
64. obront
65. orion
66. oyc_109
67. petersspetrov

68. poirots (DavideSilva, resende, naps62 and eighty)
69. rvierdiiev
70. sces60107
71. shark
72. sk8erboy
73. subtle77
74. trustindistrust
75. wagmi
76. yixxas
77. zzykxx

This contest was judged by **gzeon**.

Final report assembled by **itsmetechjay**.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 28 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 24 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 Forgeries contest repository**, and is composed of 4 smart contracts, 1 abstract, and 3 interfaces written in the Solidity programming language and includes 423 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] Admin does not have to wait to call `lastResortTimelockOwnerClaimNFT()`

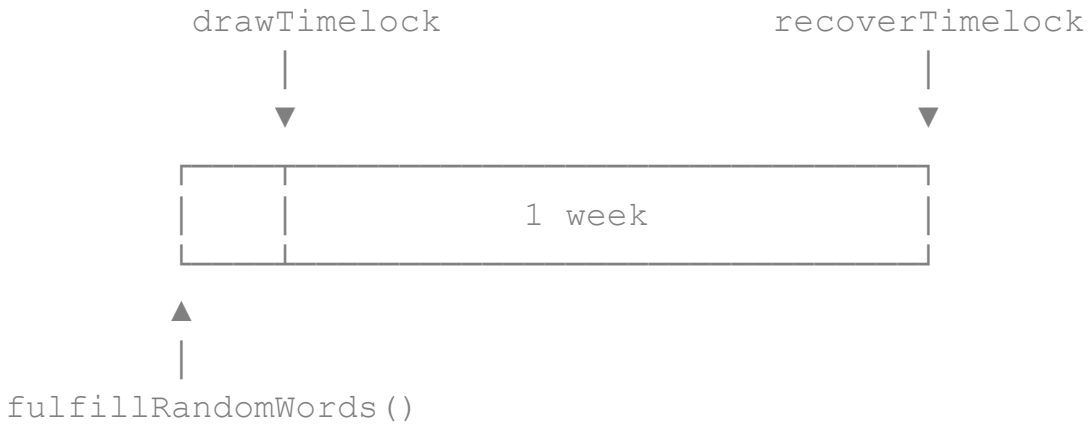
Submitted by Soosh, *also found by* dipp, indijanc, maks, jadezti, gz627, sces60107, Zarf, neumo, Ch_301, imare, Trust, btk, kuldeep, bin2chen, immeas, obront, hansfriese, Koolex, Apocalypto, carrotsmuggler, hihen, HE1M, rvierdiiev, SmartSek, 9svR6w, sk8erboy, ladboy233, Titi, dic0de, *and* csanuragjain

On contest page: `"If no users ultimately claim the NFT, the admin specifies a timelock period after which they can retrieve the raffled NFT."`

Let's assume a recoverTimelock of 1 week.

The specification suggests that 1 week from the winner not having claimed the NFT. Meaning that the admin should only be able to call `lastResortTimelockOwnerClaimNFT()` only after `<block.timestamp at fulfillRandomWords()> + request.drawTimelock + 1 weeks`.

Specification:



- The winner should have up to `drawTimelock` to claim before an admin can call `redraw()` and pick a new winner.
- The winner should have up to `recoverTimelock` to claim before an admin can call `lastResortTimelockOwnerClaimNFT()` to cancel the raffle.

But this is not the case.

recoverTimelock is set in the `initialize(...)` function and nowhere else. That means 1 week from initialization, the admin can call `lastResortTimelockOwnerClaimNFT()`. `redraw()` also does not update `recoverTimelock`.

In fact, `startDraw()` does not have to be called at the same time as `initialize(...)`. That means that if the draw was started after having been initialized for 1 week, the admin can withdraw at any time after that.

Impact

Protocol does not work as intended.

Recommended Mitigation Steps

Just like for `drawTimelock`, `recoverTimelock` should also be updated for each dice roll. `<block.timestamp at fulfillRandomWords()> + request.drawTimelock + <recoverBufferTime>`. Where `<recoverBufferTime>` is essentially the `drawBufferTime` currently used, but for `recoverTimelock`.

Note: currently, `drawTimelock` is updated in the `_requestRoll()` function. This is "technically less correct" as chainlink will take some time before `fulfillRandomWords(...)` callback. So the timelock is actually set before the winner has been chosen. This should be insignificant under normal network conditions (Chainlink VRF shouldn't take > 1min) but both timelocks should be updated in the same function - either `_requestRoll()` or `fulfillRandomWords(...)`.

iainnash (Forgeries) confirmed and commented:

> This seems to be a dupe of a previous issue where the timelock is not passed.

> Give this timelock is validated from the end of the auction the risk here seems Low.

gzeon (judge) increased severity to High and commented:

> #359 (comment)

[H-02] Draw organizer can rig the draw to favor certain participants such as their own account.

Submitted by Trust

In RandomDraw, the host initiates a draw using `startDraw()` or `redraw()` if the redraw draw expiry has passed. Actual use of Chainlink oracle is done in `\_requestRoll`:

```
    request.currentChainlinkRequestId = coordinator.requestRandomWor
        keyHash: settings.keyHash,
        subId: settings.subscriptionId,
        minimumRequestConfirmations: minimumRequestConfirmations,
        callbackGasLimit: callbackGasLimit,
        numWords: wordsRequested
    });
```

Use of subscription API is explained well here. Chainlink VRFCoordinatorV2 is called with `requestRandomWords()` and emits a random request. After `minimumRequestConfirmations` blocks, an oracle VRF node replies to the coordinator with a provable random, which supplies the random to the requesting contract via `fulfillRandomWords()` call. It is important to note the role of subscription ID. This ID maps to the subscription charged for the request, in LINK tokens. In our contract, the raffle host supplies their subscription ID as a parameter. Sufficient balance check of the request ID is not checked at request-time, but rather checked in Chainlink node code as well as on-chain by VRFCoordinator when the request is satisfied. In the scenario where the subscriptionID lacks funds, there will be a period of 24 hours when user can top up the account and random response will be sent:

"Each subscription must maintain a minimum balance to fund requests from consuming contracts. If your balance is below that minimum, your requests remain pending for up to 24 hours before they expire. After you add sufficient LINK to a subscription, pending requests automatically process as long as they have not expired."

The reason this is extremely interesting is because as soon as redraws are possible, the random response can no longer be treated as fair. Indeed, Draw host can wait until redraw cooldown passed (e.g. 1 hour), and only then fund the subscriptionID. At this point, Chainlink node will send a TX with the random response. If host likes the response (i.e. the draw winner), they will not interfere. If they don't like the response, they can simply frontrun the Chainlink TX with a `redraw()` call. A redraw will create a new random request and discard the old requestId so the previous request will never be accepted.

```
function fulfillRandomWords(
    uint256 _requestId,
    uint256[] memory _randomWords
) internal override {
    // Validate request ID
          // <---------------- swap currentChainlinkRequestId --
    if (_requestId != request.currentChainlinkRequestId) {
        revert REQUEST_DOES_NOT_MATCH_CURRENT_ID();
    }
        ...
}
```

```
    //<------ redraw swaps currentChainlinkRequestId --->
    request.currentChainlinkRequestId = coordinator.requestRandomWor
        keyHash: settings.keyHash,
        subId: settings.subscriptionId,
        minimumRequestConfirmations: minimumRequestConfirmations,
        callbackGasLimit: callbackGasLimit,
        numWords: wordsRequested
    });
```

Chainlink docs **warn** against this usage pattern of the VRF -"Don't accept bids/bets/inputs after you have made a randomness request". In this instance, a low subscription balance allows the host to invalidate the assumption that 1 hour redraw cooldown is enough to guarantee Chainlink answer has been received.

Impact

Draw organizer can rig the draw to favor certain participants such as their own account.

Proof of Concept

Owner offers a BAYC NFT for holders of their NFT collection X. Out of 10,000 tokenIDs, owner has 5,000 Xs. Rest belongs to retail users.

1. Owner subscriptionID is left with 0 LINK balance in coordinator
2. Redraw time is set to 2 hours
3. Owner calls `startDraw()` which will initiate a Chainlink request
4. Owner waits for 2 hours and then tops up their subscriptionID with sufficient LINK
5. Owner scans the mempool for `fulfillRandomWords()`
6. If the raffle winner is tokenID < 5000, it is owner's token
    1. Let fulfill execute and pick up the reward
7. If tokenID >= 5000
    1. Call `redraw()`

    2. fulfill will revert because of requestId mismatch
8. Owner has 75% of claiming the NFT instead of 50%

Note that Forgeries draws are presumably intended as incentives for speculators to buy NFTs from specific collections. Without having a fair shot at receiving rewards from raffles, these NFTs user buys could be worthless. Another way to look at it is that the impact is theft of yield, as host can freely decrease the probability that a token will be chosen for rewards with this method.

Also, I did not categorize it as centralization risk as the counterparty is not Forgeries but rather some unknown third-party host which offers an NFT incentive program. It is a similar situation to the distinction made between 1st party and 3rd party projects here.

Tools Used

Chainlink docs

Chainlink co-ordinator code

Recommended Mitigation Steps

The root cause is that Chainlink response can arrive up to 24 hours from the most request is dispatched, while redraw cooldown can be 1 hour+. The best fix would be to enforce minimum cooldown of 24 hours.

iainnash (Forgeries) confirmed

gzeon (judge) decreased severity to Medium and commented:

> This issue weaponized 133 and 194 to violate the fairness requirement of the protocol. Downgrading this to Medium because the

1. Difficulty of attack is high; you need to a) front-run the fulfillRandomWords call and b) own a meaningful % of the collection
2. Require to use an underfunded subscription This will flag the raffle is fishy, since the owner might as well never fund the subscription.
3. 3rd party can mitigate this by funding the subscription.

> There is another case where the chainlink node waits almost 24 hours before fulfilling the request, but I don't think that is the normal behavior and is out of the attacker's control.

Trust (warden) commented:

> Would like to respectfully state my case and why this finding is clearly HIGH impact. Manipulation of RNG is an extremely serious impact as it undermines assumption of fairness which is the main selling point of raffles, lotteries etc. As proof one can view Chainlink's BBP which lists "Predictable or manipulable RNG that results in abuse of downstream services" as a critical impact, payable up to $3M.

> I would like to relate to the conditions stated by the judge:

1. Difficulty of attack is high; you need to a) front-run the fulfillRandomWords call and b) own a meaningful % of the collection

> frontrunning is done in practically every block by MEV bots proving it's practical and easy to do on mainnet, where the protocol is deployed. Owning a meaningful % of the collection is not necessary, as:

1. Even with 1 / 10,000 NFTs, owner is still multiplying their chances which is a breach of fair random.
2. The exploit can be repeated in every single raffle, exponentially multiplying their edge across time. This also highlights that the frontrunning does not have to be work every time (even though it's high %) in order for the exploitation to work.
3. The draw is chosen by ownership of **_settings.drawingToken**, which is a project-provided token which is already likely they have a large amount of. It is unrelated to the BAYC collection / high value NFT being given out.
4. It is easy to see attacker can easily half the chances of any unwanted recipient to win the raffle - they would have to have the winning ticket in both rounds. Putting the subscriber's boosted win chances aside, it's a clear theft of user's potential high value prize.

- Require to use an underfunded subscription This will flag the raffle is fishy, since the owner might as well never fund the subscription.
- 3rd party can mitigate this by funding the subscription

> It is unrealistic to expect users of the protocol to be savvy on-chain detectives and also anticipate this specific attack vector. Even so, the topping-up of the subscription is done directly subscriber -> ChainlinkVRFCoordinator, so it's not visible by looking at the raffle contract.

> To summarize, the characteristics of this finding are much more aligned to those of High severity, than those of Medium severity.

gzeon (judge) commented:

> The difficulty arises when only the raffle creator can perform the front running, not any interested MEV searcher. For sure, this is only 1 of the reason I think the risk of this issue is not High.

> As the project seems to be fine with a raffle being created, but never actually started; I think when the attack require a chainlink subscription to be underfunded to begin with also kinda fall in to the "creator decided not to start raffle" category.

> The argument of judging this apart from that is the raffle would looks like it completed but might not be fair, which I think is a very valid issue. However, I don't see this as High risk given the relative difficulty as said and we seems to agree that it is fine if the raffle creator decided not to start the raffle. The end state would basically be the same.

Trust (warden) commented:

> The end states are in my opinion very different. In order to understand the full impact of the vulnerability we need to understand the context in which those raffles take place. The drawing tokens are shilled to give users a chance to win a high valued item. Their worth is correlated to the fair chance users think they have in winning the raffle. The "fake raffle" on display allows the attacker to keep profiting from ticket sales while not giving away high value. I think this is why @iainnash agreed this to be a high risk find.

> I've also listed several other justifications including theft of user's chances of winning which is high impact. I'd be happy to provide additional proof of why frontrunning is easily high enough % if that is the source of difficulty observed.

gzeon (judge) commented:

> The drawing tokens are shilled to give users a chance to win a high valued item. Their worth is correlated to the fair chance users think they have in winning the raffle.

> That's my original thought, but you and the sponsor tried to convince me the raffle is permissioned by design considering startDraw.

> If we think we need to guarantee the raffle token can get something fairly, we will also need to guarantee the raffle will, well, start. So I would say these are very similar since the ticket would be already sold anyway.

> I think I might either keep everything as-is, or I am going to reinstate those other issues that I invalidated due to assuming the permissioned design, and upgrading this to High. Would love to hear more from the sponsor before making the final call.

Trust (warden) commented:

> Regarding your smart observation @gzeon , I think the idea is clearly to make the draw methods decentralized in the future, but owner controlled as a first step. However they were not aware of this exploit, which from day 1 allows to put on a show and drive draw token prices up.

gzeon (judge) increased severity to High and commented:

> #359 (comment)

Medium Risk Findings (3)

[M-01] Raffle creator might not start raffle draw

Submitted by gasperpre, *also found by* evan, hansfriese, SmartSek, *and* orion

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L173

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L304

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L127

Impact

The raffle creator is not required to actually give the NFT away. The NFT that is used for the raffle is transferred to the contract when `startDraw` is executed. Before that, the NFT is in the hands of the creator. This means that he might create a raffle to make users buy NFTs required to participate and then refuse to draw a winner and keep the NFT to himself. Furthermore, he might not even be the owner of the NFT in the first place, which he can achieve by flash loaning the NFT in order to pass the `ownerOf` check in `initialize` function.

Proof of Concept

Example 1

1. User U creates an NFT collection C
2. He buys a BAYC NFT
3. He creates a raffle with it, and requires `drawingToken` to be from collection C
4. Users buy tokens from his collection C
5. He then refuses to execute `startDraw` function and rather sells the BAYC NFT

Example 2

1. User U creates an NFT collection C
2. User U uses an NFT flash loan to borrow a very expensive NFT
3. In the same transaction he creates a raffle with this NFT, and requires `drawingToken` to be from collection C
4. The check that he is the owner will pass, because for the duration of the transaction he in fact is

5. Users see that there is a raffle for a very expensive NFT, so they buy tokens C
6. The winner is never drawn, because the creator does not even own the NFT

Example 3

1. User U has an NFT X
2. He puts X on a sale on some NFT marketplace (which does not require him to lock it in contract)
3. He forgets about it and creates a raffle with it
4. Users buy the tokens necessary for the raffle
5. User U wants to execute the `startDraw` function, but just before it the NFT X is bought from him through the marketplace
6. The winner cannot be drawn

Recommended Mitigation Steps

Transfer the NFT to the contract at the time of creation of the raffle. You can do that by approving the factory contract to transfer the token and do the transfer in `makeNewDraw` function between cloning and `initialization`.

```
address newDrawing = ClonesUpgradeable.clone(implementation);
IERC721(settings.token).transferFrom(msg.sender, newDrawing, set
        // Setup the new drawing
IVRFNFTRandomDraw(newDrawing).initialize(admin, settings);
```

Remember to remove token transfer from `startDraw` function.

Notice that the creator can still claim NFT after a week, without drawing, by executing `lastResortTimelockOwnerClaimNFT`. To prevent that, I would recommend adding a check in `lastResortTimelockOwnerClaimNFT`, if a winner was drawn.

```
 if (!request.hasChosenRandomNumber) {
            revert NEEDS_TO_HAVE_CHOSEN_A_NUMBER();
}
```

So now a user can trust that the NFT is locked in the contract, and it will be claimable only by a winner (or creator if the winner does not claim it). However, there is still no guarantee that the winner will actually be drawn, because the creator has to manually execute `startDraw` function. To fix this, I would recommend allowing anyone to execute `startDraw` function, so there is no need to rely on the creator. But we would need to limit the time window of when `startDraw` can be executed, so users have the time to get tokens before the drawing. That can be done by introducing a new state variable `firstDrawTime`, that acts as a timestamp after which drawing can happen.

```
if(block.timestamp < firstDrawTime) revert CANNOT_DRAW_YET();
```

Notice that now the NFT can only be claimed after the winner has been drawn. This means that we are depending on ChainLink VRF to be successful. For that reason I would recommend adding a role that has the power to change the VRF subscription or restore the NFT in cases where the winner is not picked in reasonable time. This role would be given to protocol owner (owner of the factory) / DAO / someone who would be considered as most reliable.

gzeon (judge) decreased severity to Medium and commented:

> **#359 (comment)**

iainnash (Forgeries) confirmed

[M-02] VRFNFTRandomDraw admin can prevent created or started raffle from taking place

Submitted by **9svR6w**, *also found by* **deliriusz**, **BAHOZ**, **0xdeadbeef0x**, **trustindistrust**, **gasperpre**, *and* **codeislight**

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L173

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L162-L168

Impact

The admin/owner of `VRFNFTRandomDraw` can `startDraw()` a raffle, including emitting the `SetupDraw` event, but in a way that ensures `fulfillRandomWords()` is never called. For example:

- `keyHash` is not validated within `coordinator.requestRandomWords()`. Providing an invalid `keyHash` will allow the raffle to start but prevent the oracle from actually supplying a random value to determine the raffle result.
    - https://github.com/smartcontractkit/chainlink/blob/00f9c6e41f843f96108cdaa118a6ca740b11df35/contracts/src/v0.8/VRFCoordinatorV2.sol#L407-L409
    - https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L163
- The admin/owner could alternatively ensure that the owner-provided chain.link VRF subscription does not have sufficient funds to pay at the time the oracle attempts to supply random values in `fulfillRandomWords()`.
    - https://github.com/smartcontractkit/chainlink/blob/00f9c6e41f843f96108cdaa118a6ca740b11df35/contracts/src/v0.8/VRFCoordinatorV2.sol#L594-L596

In addition, the owner/admin could simply avoid ever calling `startDraw()` in the first place.

Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

Recommended Mitigation Steps

Depending on the desired functionality with respect to the raffle owner, a successful callback to `fulfillRandomWords()` could be a precondition of the admin/owner reclaiming the reward NFT. This would help ensure the owner does not create raffles that they intend will never pay out a reward.

iainnash (Forgeries) confirmed

[M-03] Protocol safeguards for time durations are skewed by a factor of 7. Protocol may potentially lock NFT for period of 7 years.

Submitted by Trust*, also found by* subtle77*,* wagmi*,* Madalad*,* Matin*,* mookimgo*,* evan*,* Apocalypto*,* kaliberpoziomka8552*,* poirots*,* aga7hokakological*, and* yixxas

In `VRFNFtRandomDraw\.sol initialize()` , the MONTH/WSECONDS variable is used to validate two values:

- configured time between redraws is under 1 month
- recoverTimelock (when NFT can be returned to owner) is less than 1 year in the future

```
if (_settings.drawBufferTime > MONTH_IN_SECONDS) {
    revert REDRAW_TIMELOCK_NEEDS_TO_BE_LESS_THAN_A_MONTH();
}
...
if (
    _settings.recoverTimelock >
    block.timestamp + (MONTH_IN_SECONDS * 12)
) {
    revert RECOVER_TIMELOCK_NEEDS_TO_BE_LESS_THAN_A_YEAR();
}
```

The issue is that MONTH/WSECONDS is calculated incorrectly:

```
/// @dev 60 seconds in a min, 60 mins in an hour
uint256 immutable HOUR_IN_SECONDS = 60 * 60;
/// @dev 24 hours in a day 7 days in a week
```

```
    uint256 immutable WEEK_IN_SECONDS = (3600 * 24 * 7);
    // @dev about 30 days in a month
    uint256 immutable MONTH_IN_SECONDS = (3600 * 24 * 7) * 30;
```

MONTH/*N*SECONDS multiplies by 7 incorrectly, as it was copied from WEEK/*N*SECONDS. Therefore, actual seconds calculated is equivalent of 7 months. Therefore, recoverTimelock can be up to a non-sensible value of 7 years, and re-draws every up to 7 months.

Impact

Protocol safeguards for time durations are skewed by a factor of 7. Protocol may potentially lock NFT for period of 7 years.

Recommended Mitigation Steps

Fix MONTH/*N*SECONDS calculation: `uint256 immutable MONTH_IN_SECONDS = (3600 * 24) * 30;`

iainnash (Forgeries) confirmed

Low Risk and Non-Critical Issues

For this contest, 27 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **poirots** received the top score from the judge.

The following wardens also submitted reports: **deliriusz**, **Aymen0909**, **adriro**, **zzykxx**, **IIIIIII**, **Zarf**, **ayeslick**, **Madalad**, **0xAgro**, **Deivitto**, **caventa**, **immeas**, **shark**, **obront**, **0xdeadbeef0x**, **petersspetrov**, **Bobface**, **rvierdiiev**, **0x1f8b**, **Ruhum**, **RaymondFam**, **9svR6w**, **cccz**, **BnkeOx0**, **oyc_109**, *and* **Rolezn** .

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	`redraw()` should be called by anyone	1
[L-02]	An `owner` can resign and lead to locked NFT	1

Number	Issues Details	Context
[L-03]	NFTs are not guaranteed to have sequential IDs	1

Total: 3 issues

[L-01] `redraw()` should be called by anyone

VRFNFTRandomDraw.sol#L203-L225

Redrawing a raffle already protects the winner through the timelocking mechanism. Dependency on the owner should be avoidable in this instance by removing the modifier `onlyOwner`, allowing anyone to redraw the raffle.

[L-02] An owner can resign and lead to locked NFTs

Since there's a possibility of unclaimed drafts, the owner is the only one able to rescue the prize NFT from the raffle contract. Thus, having the ability to resign ownership (including non-intentional) could lead to stuck NFTs.

Consider altering or removing resignOwnership method:

```
/// @notice Resign ownership of contract
/// @dev only callably by the owner, dangerous call.
function resignOwnership() public onlyOwner {
    _transferOwnership(address(0));
}
```

[L-03] NFTs are not guaranteed to have sequential IDs

https://github.com/code-423n4/2022-12-forgeries/blob/main/src/interfaces/IVRFNFTRandomDraw.sol#L71-L90

https://github.com/code-423n4/2022-12-forgeries/blob/main/src/VRFNFTRandomDraw.sol#L249-L256

https://github.com/code-423n4/2022-12-forgeries/blob/main/src/VRFNFTRandomDraw.sol#L271

Impact

Accordingly to EIP-721:

> **NFT Identifiers**
>
> Every NFT is identified by a unique uint256 ID inside the ERC-721 smart contract. This identifying number SHALL NOT change for the life of the contract. The pair (contract address, uint256 tokenId) will then be a globally unique and fully-qualified identifier for a specific asset on an Ethereum chain. While some ERC-721 smart contracts may find it convenient to start with ID 0 and simply increment by one for each new NFT, **callers SHALL NOT assume that ID numbers have any specific pattern to them, and MUST treat the ID as a "black box".** Also note that NFTs MAY become invalid (be destroyed). Please see the enumeration functions for a supported enumeration interface.

> The choice of uint256 allows a wide variety of applications because UUIDs and sha3 hashes are directly convertible to uint256.

This project, aims to create a raffle specifying that the potential winner will be between a range, where the lower limit is set by the candidate with the lowest TokenId and the candidate with the highest TokenId (`+1` to be included in the draw) sets the upper limit. As stated in the previous quote, this could generate gigantic ranges with numerous empty tokens given how it is calculated (see ENS as an example of empty slots and how the ids are generated).

After generating the random number via VRF, the winner is selected by *moduluing* by the range plus the initial token id. The result is then used to determine the winner.

Considering the costs to query VRF and the waiting time to claim the prize, this issue may turn the contract unusable.

Note that any used method when casting the collection to IERC721EnumerableUpgradeable has the same effect as casting to IERC721, and giving how a raffle is setup, it seems the original intent was to use indexes instead of ids.

External requirements:

- non-sequential NFTs.

Proof of Concept

Consider a scenario where:

```
Drawing NFT Collection: ABC
TokenIDs: [1,10,1000]
TotalSupply: 3
```

Setting a raffle to include each possible token results in [1..1001[alternatives. Since there are only three possible winners, there's only `0.3%` of a successful draw.

Recommended Mitigation Steps

Depending on the direction the project takes:

1. Change the way the setup is performed;
2. Give more guarantees that only collections with sequential ids are used (note that the same problem might happen in nfts with high number of burns);
3. Only use indexed collections.

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	`getRequestDetails()` should include the tokenid	1
[N-02]	Avoid setting time variables manually	1
[N-03]	Use constants instead of immutable variables	1
[N-04]	Uppercase immutable variables	6
[N-05]	Empty blocks should be avoided	1
[N-06]	Missing *NatSpec*	1
[N-07]	Contracts that extend interfaces should override its methods	3
[N-08]	`_requestRoll()` after confirming that the raffle is viable	1
[N-09]	`IERC721EnumerableUpgradeable` may lead to false assumptions	6

Number	Issues Details	Context
[N-10]	`drawingTokenEndId` should be inclusive or altered to a range	1
[N-11]	`fulfillRandomWords` must not revert	1

Total: 11 issues

[N-01] `getRequestDetails()` should include the tokenid

In VRFNFTRandomDraw.sol#getRequestDetails() should include `currentChosenTokenId` (at) and ease integrations with other tools.

[N-02] Avoid setting time variables manually

Use solidity Time Units to avoid mistakes in defining time variables. In VRFNFTRandomDraw.sol#L28-L33 (the `MONTH_IN_SECONDS` leads to a medium issue):

```
/// @dev 60 seconds in a min, 60 mins in an hour
uint256 immutable HOUR_IN_SECONDS = 60 * 60;
/// @dev 24 hours in a day 7 days in a week
uint256 immutable WEEK_IN_SECONDS = (3600 * 24 * 7);
// @dev about 30 days in a month
uint256 immutable MONTH_IN_SECONDS = (3600 * 24 * 7) * 30;
```

Consider changing to:

```
/// @dev 60 seconds in a min, 60 mins in an hour
uint256 immutable HOUR_IN_SECONDS = 1 hours;
/// @dev 24 hours in a day 7 days in a week
uint256 immutable WEEK_IN_SECONDS = 1 weeks;
// @dev about 30 days in a month
uint256 immutable MONTH_IN_SECONDS = 30 days;
```

[N-03] Use constants instead of immutable variables

Variables defined in VRFNFTRandomDraw.sol#L21-L33 should be constants, since they aren't defined at contract creation:

```
uint32 immutable callbackGasLimit = 200_000;
/// @notice Chainlink request confirmations, left at the def
uint16 immutable minimumRequestConfirmations = 3;
/// @notice Number of words requested in a drawing
uint16 immutable wordsRequested = 1;

/// @dev 60 seconds in a min, 60 mins in an hour
uint256 immutable HOUR_IN_SECONDS = 60 * 60;
/// @dev 24 hours in a day 7 days in a week
uint256 immutable WEEK_IN_SECONDS = (3600 * 24 * 7);
// @dev about 30 days in a month
uint256 immutable MONTH_IN_SECONDS = (3600 * 24 * 7) * 30;
```

If these are rules, consider changing them to `IVRFNFTRandomDraw` interface:

```
uint32 constant callbackGasLimit = 200_000;
/// @notice Chainlink request confirmations, left at the def
uint16 constant minimumRequestConfirmations = 3;
/// @notice Number of words requested in a drawing
uint16 constant wordsRequested = 1;

/// @dev 60 seconds in a min, 60 mins in an hour
uint256 constant HOUR_IN_SECONDS = 60 * 60;
/// @dev 24 hours in a day 7 days in a week
uint256 constant WEEK_IN_SECONDS = (3600 * 24 * 7);
// @dev about 30 days in a month
uint256 constant MONTH_IN_SECONDS = (3600 * 24 * 7) * 30;
```

[N-04] Uppercase immutable variables

In [VRFNFTRandomDraw.sol#L22-L26]:

```
uint32 immutable callbackGasLimit = 200_000;
/// @notice Chainlink request confirmations, left at the def
uint16 immutable minimumRequestConfirmations = 3;
/// @notice Number of words requested in a drawing
uint16 immutable wordsRequested = 1;
```

In [VRFNFTRandomDraw.sol#L37]:

```
VRFCoordinatorV2Interface immutable coordinator;
```

In Version.sol#L5:

```
uint32 private immutable __version;
```

In VRFNFTRandomDrawFactory.sol#L21:

```
address public immutable implementation;
```

[N-05] Empty blocks should be avoided

Avoid using code blocks, such as:

In VRFNFTRandomDrawFactory.sol#L53-L59:

```
/// @notice Allows only the owner to upgrade the contract
/// @param newImplementation proposed new upgrade implementa
function _authorizeUpgrade(address newImplementation)
    internal
    override
    onlyOwner
{}
```

Consider emitting an event.

[N-06] Missing *NatSpec*

Consider adding specification to the following code blocks:

In IVRFNFTRandomDraw.sol#L28:

```
error REDRAW_TIMELOCK_NEEDS_TO_BE_LESS_THAN_A_MONTH();
```

[N-07] Contracts that extend interfaces should override its methods

Consider using the `override` keyword to indicate which methods are implementing the interface.

For `VRFNFTRandomDraw` regarding `IVRFNFTRandomDraw`: `initialize`, `startDraw`, `redraw`, `hasUserWon`, `winnerClaimNFT`, `lastResortTimelockOwnerClaimNFT`, `getRequestDetails`. For `VRFNFTRandomDrawFactory` regarding `IVRFNFTRandomDrawFactory`: `initialize`, `startDraw`.

[N-08] `_requestRoll()` after confirming that the raffle is viable

In startDraw(), the contract makes a request for a random number before confirming that it has the prize to raffle.

Consider confirming first that the contract has the NFT to raffle before wasting resources calling for a random.

[N-09] `IERC721EnumerableUpgradeable` may lead to false assumptions

Throughout the contract, there's a wrapper of NFT collections to `IERC721EnumerableUpgradeable` instances:

```
src/VRFNFTRandomDraw.sol:127:              IERC721EnumerableUpgrac
src/VRFNFTRandomDraw.sol:187:              IERC721EnumerableUpgrac
src/VRFNFTRandomDraw.sol:216:              IERC721EnumerableUpgrac
src/VRFNFTRandomDraw.sol:271:              IERC721EnumerableUpgrac
src/VRFNFTRandomDraw.sol:295:         IERC721EnumerableUpgradeabl
src/VRFNFTRandomDraw.sol:315:         IERC721EnumerableUpgradeabl
```

This could lead to false assumptions when working with this contract (particularly when considering how settings are defined).

Consider altering to `IERC721` if the goal is to allow any NFT collection compliant with EIP-721.

[N-10] `drawingTokenEndId` should be inclusive or altered to a range

Natspec specifies that the last ID is exclusive in the raffle, but the variable's name could lead to wrong assumptions.

Consider altering the logic to the contract to include the ID or to change the logic to a range definition, since it is only used twice (1,2) and could avoid misinterpretations.

[N-11] `fulfillRandomWords` must not revert

Accordingly to ChainLinks' documentation:

> `fulfillRandomWords` must not revert If your `fulfillRandomWords()` implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert. Consider simply storing the randomness and taking more complex follow-on actions in separate contract calls made by you, your users, or an Automation Node.

This project's current implementation does revert in two instances, although they are not expected to materialize.

Nevertheless, consider altering the logic to drop the random generated whenever the requestId does not match and ignore extra words if the array received is greater than the expected amount.

Gas Optimizations

For this contest, 24 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **c3phas** received the top score from the judge.

The following wardens also submitted reports: indijanc, Aymen0909, adriro, PaludoX0, IIIIIII, izhelyazkov, ctrlc03, kuldeep, neko_nyaa, shark, Sathish9098,

Bobface, rvierdiiev, nadin, 0x1f8b, RaymondFam, chaduke, codeislight, ReyAdmirado, BnkeOx0, Rahoz, nicobevi, *and* Rolezn .

Gas Optimizations Overview

NB: Some functions have been truncated where necessary to just show affected parts of the code.

Throughout the report some places might be denoted with audit tags to show the actual place affected.

I've tried to give the exact amount of gas saved from running the included tests. Whenever the function is within the test coverage, the average gas before and after will be included, and often a diff of the code will also accompany this.

Some functions are not covered by the test cases or are internal/private functions. In this case, the gas can be estimated by looking at the opcodes involved.

[G-01] Pack structs by putting data types in ascending size (We can save up to ~6k gas)

As the solidity EVM works with 32 bytes, variables less than 32 bytes should be packed inside a struct so that they can be stored in the same slot, this saves gas when writing to storage ~20000 gas.

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/interfaces/IVRFNFTRandomDraw.sol#L59-L68

We can use a smaller type for uint256 drawTimelock as it's simply a timestamp. Using uint64 should be safe for 532 years. We save 1 Storage SLOT from 4 SLOTS to 3 SLOTS (~2K gas)

```
File: /src/interfaces/IVRFNFTRandomDraw.sol
59:     struct CurrentRequest {
60:         /// @notice current chainlink request id
61:         uint256 currentChainlinkRequestId;
62:         /// @notice current chosen random number
63:         uint256 currentChosenTokenId;
```

```
64:         /// @notice has chosen a random number (in case randc
65:         bool hasChosenRandomNumber;
66:         /// @notice time lock (block.timestamp) that a re-dra
67:         uint256 drawTimelock;
68:     }
```

```
diff --git a/src/interfaces/IVRFNFTRandomDraw.sol b/src/interfac
index 4775288..af1d928 100644
--- a/src/interfaces/IVRFNFTRandomDraw.sol
+++ b/src/interfaces/IVRFNFTRandomDraw.sol
@@ -64,7 +64,7 @@ interface IVRFNFTRandomDraw {
         /// @notice has chosen a random number (in case random
         bool hasChosenRandomNumber;
         /// @notice time lock (block.timestamp) that a re-draw
-        uint256 drawTimelock;
+        uint64 drawTimelock;
     }
```

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/interfaces/IVRFNFTRandomDraw.sol#L71-L90

We can save 2 SLOTs here by packing address token with uint64 subscriptionId and also changing the type of uint256 recoverTimelock which is a timestamp to uint64 which should be safe for more than 500 years (Saves ~4k gas)

```
File: /src/interfaces/IVRFNFTRandomDraw.sol
71:     struct Settings {
72:         /// @notice Token Contract to put up for raffle
73:         address token;
74:         /// @notice Token ID to put up for raffle
75:         uint256 tokenId;
76:         /// @notice Token that each (sequential) ID has a ent
77:         address drawingToken;
78:         /// @notice Start token ID for the drawing (if totals
79:         uint256 drawingTokenStartId;
80:         /// @notice End token ID for the drawing (exclusive)
81:         uint256 drawingTokenEndId;
82:         /// @notice Draw buffer time - time until a re-drawir
83:         uint256 drawBufferTime;
```

```
84:         /// @notice block.timestamp that the admin can recove
85:         uint256 recoverTimelock;
86:         /// @notice Chainlink gas keyhash
87:         bytes32 keyHash;
88:         /// @notice Chainlink subscription id
89:         uint64 subscriptionId;
90:     }
```

```
diff --git a/src/interfaces/IVRFNFTRandomDraw.sol b/src/interfac
index 4775288..7923c29 100644
--- a/src/interfaces/IVRFNFTRandomDraw.sol
+++ b/src/interfaces/IVRFNFTRandomDraw.sol
@@ -69,24 +69,24 @@ interface IVRFNFTRandomDraw {

     /// @notice Struct to organize user settings
     struct Settings {
+        /// @notice Chainlink subscription id
+        uint64 subscriptionId;
         /// @notice Token Contract to put up for raffle
         address token;
         /// @notice Token ID to put up for raffle
         uint256 tokenId;
         /// @notice Token that each (sequential) ID has a entry
         address drawingToken;
+        /// @notice block.timestamp that the admin can recover
+        uint64 recoverTimelock;
         /// @notice Start token ID for the drawing (if totalSup
         uint256 drawingTokenStartId;
         /// @notice End token ID for the drawing (exclusive) (t
         uint256 drawingTokenEndId;
         /// @notice Draw buffer time - time until a re-drawing
         uint256 drawBufferTime;
-        /// @notice block.timestamp that the admin can recover
-        uint256 recoverTimelock;
         /// @notice Chainlink gas keyhash
         bytes32 keyHash;
-        /// @notice Chainlink subscription id
-        uint64 subscriptionId;
     }
```

[G-02] Emitting storage values instead of the memory one.

Here, the values emitted shouldn't be read from storage. The existing memory values should be used instead:

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L75-L138

Save 499 gas on average

	Min	Average	Median	Max
Before	43790	146546	175523	192923
After	43790	146047	174692	192092

```
File: /src/VRFNFTRandomDraw.sol
75:     function initialize(address admin, Settings memory _setti
76:         public
77:         initializer
78:     {
79:         // Set new settings
80:         settings = _settings;

122:         // Emit initialized event for indexing
123:         emit InitializedDraw(msg.sender, settings);
```

```
diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..7955234 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -120,7 +120,7 @@ contract VRFNFTRandomDraw is
         __Ownable_init(admin);

         // Emit initialized event for indexing
-        emit InitializedDraw(msg.sender, settings);
+        emit InitializedDraw(msg.sender, _settings);

         // Get owner of raffled tokenId and ensure the current
```

[G-03] Cache storage values in memory to minimize SLOADs

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.

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L141-L169

VRFNFTRandomDraw.sol._requestRoll(): We could cache request.drawTimelock instead of calling it twice

```
File: /src/VRFNFTRandomDraw.sol
141:     function _requestRoll() internal {

148:         // If the number has been drawn and
149:         if (
150:             request.hasChosenRandomNumber &&
151:             // Draw timelock not yet used
152:             request.drawTimelock != 0 && //@audit: 1st call
153:             request.drawTimelock > block.timestamp //@audit:
154:         ) {
155:             revert STILL_IN_WAITING_PERIOD_BEFORE_REDRAWING(
156:         }

158:         // Setup re-draw timelock
159:         request.drawTimelock = block.timestamp + settings.dr

169:     }
```

```
diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..e235c0b 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -145,12 +145,13 @@ contract VRFNFTRandomDraw is
             revert REQUEST_IN_FLIGHT();
         }

-        // If the number has been drawn and
```

```
+         // If the number has been drawn
+         uint256 _requestDrawTimelock = request.drawTimelock ;
          if (
              request.hasChosenRandomNumber &&
              // Draw timelock not yet used
-             request.drawTimelock != 0 &&
-             request.drawTimelock > block.timestamp
+             _requestDrawTimelock != 0 &&
+             _requestDrawTimelock > block.timestamp
          ) {
              revert STILL_IN_WAITING_PERIOD_BEFORE_REDRAWING();
          }
```

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L277-L300

VRFNFTRandomDraw.sol.winnerClaimNFT(): settings.token and settings.tokenId should be cached. Also no need to cast settings.token as it's an address already - Saves 62 gas on average

	Min	Average	Median	Max
Before	422	11638	2422	27624
After	422	11576	2422	27469

```
File: /src/VRFNFTRandomDraw.sol
277:     function winnerClaimNFT() external {

287:         emit WinnerSentNFT(
288:             user,
289:             address(settings.token),
290:             settings.tokenId,
291:             settings
292:         );

294:         // Transfer token to the winter.
295:         IERC721EnumerableUpgradeable(settings.token).transfe
296:             address(this),
297:             msg.sender,
298:             settings.tokenId
299:         );
```

```
300:     }


diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..407a5f4 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -283,19 +283,21 @@ contract VRFNFTRandomDraw is
             revert USER_HAS_NOT_WON();
         }

+        address _token = settings.token;
+        uint256 _tokenId = settings.tokenId;
         // Emit a celebratory event
         emit WinnerSentNFT(
             user,
-            address(settings.token),
-            settings.tokenId,
+            _token,
+            _tokenId,
             settings
         );

         // Transfer token to the winter.
-        IERC721EnumerableUpgradeable(settings.token).transferFr
+        IERC721EnumerableUpgradeable(_token).transferFrom(
             address(this),
             msg.sender,
-            settings.tokenId
+            _tokenId
         );
     }
```

[G-04] The result of a function call should be cached rather than re-calling the function

Consider caching the following:

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L304-L320

VRFNFTRandomDraw.sol.lastResortTimelockOwnerClaimNFT(): The results of `owner()` should be cached instead of calling it twice

	Min	Average	Median	Max
Before	381	11061	11061	21741
After	381	10992	10992	21604

```
File: /src/VRFNFTRandomDraw.sol
304:     function lastResortTimelockOwnerClaimNFT() external only
305:         // If recoverTimelock is not setup, or if not yet oc
306:         if (settings.recoverTimelock > block.timestamp) {
307:             // Stop the withdraw
308:             revert RECOVERY_IS_NOT_YET_POSSIBLE();
309:         }

311:         // Send event for indexing that the owner reclaimed
312:         emit OwnerReclaimedNFT(owner()); //@audit: Initial c

314:         // Transfer token to the admin/owner.
315:         IERC721EnumerableUpgradeable(settings.token).transfe
316:             address(this),
317:             owner(),//@audit: Second call
318:             settings.tokenId
319:         );
320:     }
```

```
diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..00f000d 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -307,14 +307,15 @@ contract VRFNFTRandomDraw is
             // Stop the withdraw
             revert RECOVERY_IS_NOT_YET_POSSIBLE();
         }
+        address _ownerAddr = owner();

         // Send event for indexing that the owner reclaimed the
-        emit OwnerReclaimedNFT(owner());
+        emit OwnerReclaimedNFT(_ownerAddr);

         // Transfer token to the admin/owner.
         IERC721EnumerableUpgradeable(settings.token).transferFr
```

```
                address(this),
-               owner(),
+               _ownerAddr,
                settings.tokenId
            );
        }
```

[G-05] Using unchecked blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block.

See resource.

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L112-L115

Saves 43 gas on average

	Min	Average	Median	Max	
Before	43790	146546	175523	192923	
After	43790	146503	175451	192851	

```
File: /src/VRFNFTRandomDraw.sol
112:         if (
113:             _settings.drawingTokenEndId < _settings.drawingT
114:             _settings.drawingTokenEndId - _settings.drawingT
115:         ) {
```

The operation `_settings.drawingTokenEndId - _settings.drawingTokenStartId` cannot underflow as it would only be performed if the operation `_settings.drawingTokenEndId < _settings.drawingTokenStartId` is false(Short circuit rules)

```
diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..b33b93e 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -109,12 +109,15 @@ contract VRFNFTRandomDraw is

         // Validate token range: end needs to be greater than s
         // and the size of the range needs to be at least 2 (er
-        if (
+        unchecked {
+            if (
             _settings.drawingTokenEndId < _settings.drawingToke
             _settings.drawingTokenEndId - _settings.drawingToke
         ) {
             revert DRAWING_TOKEN_RANGE_INVALID();
         }
+        }
+
```

[G-06] A modifier used only once and not being inherited should be inlined to save gas

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/ownable/OwnableUpgradeable.sol#L44-L49

```
File: /src/ownable/OwnableUpgradeable.sol
44:     modifier onlyPendingOwner() {
45:         if (msg.sender != _pendingOwner) {
46:             revert ONLY_PENDING_OWNER();
47:         }
48:         _;
49:     }
```

The above modifer is only used in the following:

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/ownable/OwnableUpgradeable.sol#L119-L125

```
File: /src/ownable/OwnableUpgradeable.sol
119:     function acceptOwnership() public onlyPendingOwner {
120:         emit OwnerUpdated(_owner, msg.sender);

122:         _owner = _pendingOwner;

124:         delete _pendingOwner;
125:     }
```

```
diff --git a/src/ownable/OwnableUpgradeable.sol b/src/ownable/Ow
index bfc7eef..d27530c 100644
--- a/src/ownable/OwnableUpgradeable.sol
+++ b/src/ownable/OwnableUpgradeable.sol
@@ -116,7 +116,10 @@ abstract contract OwnableUpgradeable is IOw
     }

     /// @notice Accepts an ownership transfer
-    function acceptOwnership() public onlyPendingOwner {
+    function acceptOwnership() public{
+      if (msg.sender != _pendingOwner) {
+            revert ONLY_PENDING_OWNER();
+        }
         emit OwnerUpdated(_owner, msg.sender);

         _owner = _pendingOwner;

                 delete _pendingOwner;
```

[G-07] Caching global variables is more expensive than using the actual variable (use msg.sender instead of caching it)

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDrawFactory.sol#L38-L51

	Min	Average	Median	Max
Before	46872	183639	213232	239795
After	46860	183631	213224	239783

```
File: /src/VRFNFTRandomDrawFactory.sol
38:     function makeNewDraw(IVRFNFTRandomDraw.Settings memory se
39:         external
40:         returns (address)
41:     {
42:         address admin = msg.sender;
43:         // Clone the contract
44:         address newDrawing = ClonesUpgradeable.clone(implemer
45:         // Setup the new drawing
46:         IVRFNFTRandomDraw(newDrawing).initialize(admin, setti
47:         // Emit event for indexing
48:         emit SetupNewDrawing(admin, newDrawing);
49:         // Return address for integration or testing
50:         return newDrawing;
51:     }
```

```
diff --git a/src/VRFNFTRandomDrawFactory.sol b/src/VRFNFTRandomI
index 84caedb..616cb0a 100644
--- a/src/VRFNFTRandomDrawFactory.sol
+++ b/src/VRFNFTRandomDrawFactory.sol
@@ -39,13 +39,12 @@ contract VRFNFTRandomDrawFactory is
         external
         returns (address)
     {
-        address admin = msg.sender;
         // Clone the contract
         address newDrawing = ClonesUpgradeable.clone(implementa
         // Setup the new drawing
-        IVRFNFTRandomDraw(newDrawing).initialize(admin, setting
+        IVRFNFTRandomDraw(newDrawing).initialize(msg.sender, se
         // Emit event for indexing
-        emit SetupNewDrawing(admin, newDrawing);
+        emit SetupNewDrawing(msg.sender, newDrawing);
         // Return address for integration or testing
         return newDrawing;
     }
```

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L277-L300

```
File: /src/VRFNFTRandomDraw.sol
277:     function winnerClaimNFT() external {
278:         // Assume (potential) winner calls this fn, cache.
279:         address user = msg.sender;

281:         // Check if this user has indeed won.
282:         if (!hasUserWon(user)) {
283:             revert USER_HAS_NOT_WON();
284:         }

286:         // Emit a celebratory event
287:         emit WinnerSentNFT(
288:             user,
289:             address(settings.token),
290:             settings.tokenId,
291:             settings
292:         );

294:         // Transfer token to the winter.
295:         IERC721EnumerableUpgradeable(settings.token).transfe
296:             address(this),
297:             msg.sender,
298:             settings.tokenId
299:         );
300:     }
```

```
diff --git a/src/VRFNFTRandomDraw.sol b/src/VRFNFTRandomDraw.sol
index 668bc56..06ae5b2 100644
--- a/src/VRFNFTRandomDraw.sol
+++ b/src/VRFNFTRandomDraw.sol
@@ -276,16 +276,15 @@ contract VRFNFTRandomDraw is
     /// @notice Function for the winner to call to retrieve the
     function winnerClaimNFT() external {
         // Assume (potential) winner calls this fn, cache.
-        address user = msg.sender;

         // Check if this user has indeed won.
-        if (!hasUserWon(user)) {
+        if (!hasUserWon(msg.sender)) {
             revert USER_HAS_NOT_WON();
         }

         // Emit a celebratory event
         emit WinnerSentNFT(
```

```
-               user,
+               msg.sender,
                address(settings.token),
                settings.tokenId,
                settings
```

[G-08] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that [returns a tuple](https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDrawFactory.sol#L21) of the values of all currently-public constants.

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDrawFactory.sol#L21

```
File: /src/VRFNFTRandomDrawFactory.sol
21:     address public immutable implementation;
```

[G-09] 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.The extra opcodes avoided costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

https://github.com/code-423n4/2022-12-forgeries/blob/fc271cf20c05ce857d967728edfb368c58881d85/src/VRFNFTRandomDraw.sol#L173

```
File: /src/VRFNFTRandomDraw.sol
173:     function startDraw() external onlyOwner returns (uint256

203:     function redraw() external onlyOwner returns (uint256) {
```

```
304:     function lastResortTimelockOwnerClaimNFT() external only
```

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