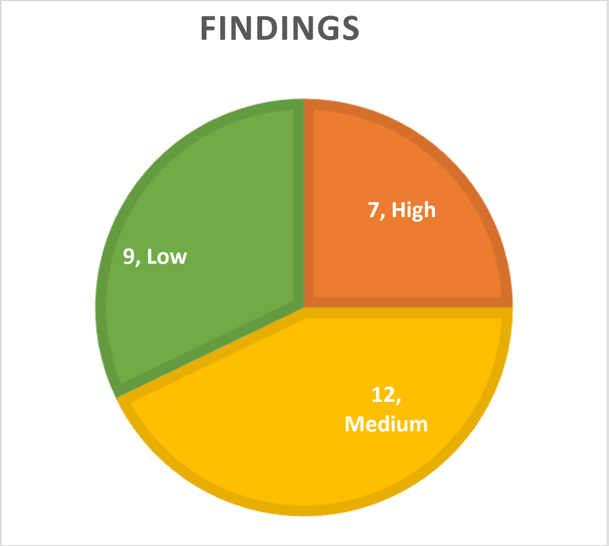
Trust Security

Smart Contract Audit

Story Protocol – Smart Contract Assets

13/12/2024

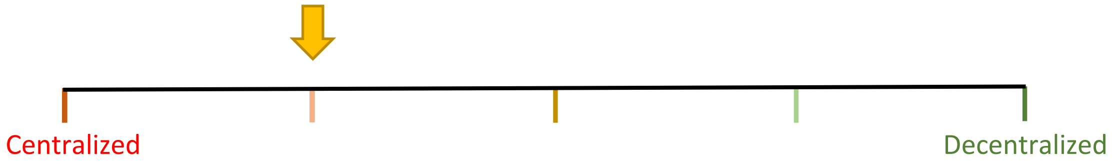
Executive summary



Category	IP management
Audited file count	53
Lines of Code	6243
Auditor	Bernd Artmüller, rvierdiiev
Time period	17/10-18/11

Severity	Total	Pending Resolution	Fixed	Acknowledged
High	7	1	6	-
Medium	12	3	9	-
Low	9	-	9	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.12.13 12:41:38 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	18.11.2024	Client report
0.2	13.12.2024	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

Repo: https://github.com/storyprotocol/protocol-core-v1

File list:

- contracts/access/AccessControlled.sol
- contracts/access/AccessController.sol
- contracts/access/IPGraphACL.sol
- contracts/lib/modules/Module.sol
- contracts/lib/registries/IPAccountChecker.sol
- contracts/lib/AccessPermission.sol
- contracts/lib/ArrayUtils.sol
- contracts/lib/Errors.sol
- contracts/lib/ExpiringOps.sol
- contracts/lib/IPAccountStorageOps.sol
- contracts/lib/Licensing.sol
- contracts/lib/MetaTx.sol
- contracts/lib/PILFlavors.sol
- contracts/lib/PILicenseTemplateErrors.sol
- contracts/lib/ProtocolAdmin.sol
- contracts/lib/ShortStringOps.sol
- contracts/modules/dispute/DisputeModule.sol
- contracts/modules/dispute/policies/UMA/ArbitrationPolicyUMA.sol
- contracts/modules/grouping/EvenSplitGroupPool.sol
- contracts/modules/grouping/GroupingModule.sol
- contracts/modules/licensing/BaseLicenseTemplateUpgradeable.sol
- contracts/modules/licensing/LicensingModule.sol
- contracts/modules/licensing/parameter-helpers/LicensorApprovalChecker.sol
- contracts/modules/licensing/PILicenseTemplate.sol
- contracts/modules/licensing/PILTermsRenderer.sol
- contracts/modules/metadata/CoreMetadataModule.sol
- contracts/modules/metadata/CoreMetadataViewModule.sol
- contracts/modules/royalty/policies/IpRoyaltyVault.sol
- contracts/modules/royalty/policies/VaultController.sol
- contracts/modules/royalty/policies/LAP/RoyaltyPolicyLAP.sol
- contracts/modules/royalty/policies/LRP/RoyaltyPolicyLRP.sol
- contracts/modules/royalty/RoyaltyModule.sol

- contracts/modules/BaseModule.sol
- contracts/pause/ProtocolPausableUpgradeable.sol
- contracts/pause/ProtocolPauseAdmin.sol
- contracts/registries/GroupIPAssetRegistry.sol
- contracts/registries/IPAccountRegistry.sol
- contracts/registries/IPAssetRegistry.sol
- contracts/registries/LicenseRegistry.sol
- contracts/registries/ModuleRegistry.sol
- contracts/GroupNFT.sol
- contracts/IPAccountImpl.sol
- contracts/IPAccountStorage.sol
- contracts/LicenseToken.sol

PR list:

- 291

Repo: https://github.com/piplabs/story

File list:

- contracts/src/deploy/Create3.sol
- contracts/src/libraries/Predeploys.sol
- contracts/src/libraries/Secp256k1.sol
- contracts/src/protocol/IpTokenStaking.sol
- contracts/src/protocol/PubKeyVerifier.sol
- contracts/src/protocol/UBIPool.sol
- contracts/src/protocol/UpgradeEntrypoint.sol
- contracts/src/token/WIP.sol

Repo: https://github.com/piplabs/story-geth

File list:

- core/vm/ipgraph.go

Repository details

- **Repository URL:** https://github.com/storyprotocol/protocol-core-v1
- **Commit hash:** 3ef2a99a99192587d24b4805d5848e3a799dcf5e

- **Repository URL:** https://github.com/piplabs/story
- **Commit hash:** 20fed5ed45d39c9ac59ab17c03ff3b1efac0f7b2

- **Repository URL:** https://github.com/piplabs/story-geth
- **Commit hash:** ab8925de20576333b7d8f638d46cc9eab288b4f5

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

Bernd is a blockchain and smart contract security researcher that has made the transition from a successful full-stack web developer career. His ability to quickly grasp new concepts and technologies and his attention to detail have helped him become a top auditor in the blockchain space. Having conducted 50+ audits, Bernd has identified numerous vulnerabilities across a wide range of DeFi protocols, wallets, bridges, and VMs. He currently splits his time between audit competitions, private audits and bug bounty hunting.

rvierdiiev is a Web3 security researcher who participated in a large number of public audit contests on multiple platforms and has proven track record of experience.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	The code is well modularized to reduce complexity.
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Project introduces several points of centralization vectors (e.g., upgradeability, maintaining allow-lists, etc.)

Findings

High severity findings

TRST-H-1 Group member IPs can repeatedly claim rewards

- **Category:** Logical flaws
- **Source:** EvenSplitGroupPool.sol
- **Status:** Fixed

Description

Group member IPs can claim collected royalty rewards from their group with GroupingModule.claimReward(). The member IPs to distribute rewards to are provided through the **address[] calldata ipIds** function parameter.

Internally, *EvenSplitGroupPool.distributeRewards()* calculates each IP's reward share by enumerating the specified **ipIds**. To avoid multiple claims for the same rewards, the already distributed amount is tracked as debt, which is factored into the calculation of eligible reward distributions using **rewardPerIP - $.ipRewardDebt[groupId][token][ipIds[i]]**.

However, the reward calculation for all **ipIds** in *_getAvailableReward()* occurs before **$.ipRewardDebt** is updated. As a result, if the **ipIds** array contains duplicate IPs, it leads to repeated claims for the same IP, distributing more royalties than it would be eligible for and thus stealing funds from other group member IPs.

Recommended mitigation

It is recommended to prevent duplicate items in **ipIds** from being passed to *distributeRewards()*.

Team response

Fixed.

Mitigation review

While the reported issue is fixed now, a new issue has been introduced.

Specifically, the token spending allowance can be inflated by adding duplicate IPs to the **ipIds** array, or IPs that are not added to the group, while the **totalRewards** allowance is not spent by *payRoyaltyOnBehalf()*. This results in issues with tokens such as USDT, which revert when approving while the current allowance is non-zero.

Team response

Fixed in PR 339 and PR 355.

Mitigation review

The issue has been fixed by force-approving the token spending allowance and additionally resetting the allowance to zero at the end of *distributeRewards()*.

TRST-H-2 Front-running group registrations prevents registering groups

- **Category:** Front-running attacks
- **Source:** GroupingModule.sol
- **Status:** Fixed

Description

A new group IP is created using *GroupingModule.registerGroup()*. This process mints a group NFT token and registers the group through *GroupIPAssetRegistry.registerGroup()*, which calls *_register()* to create a *6551Registry* account, akin to the regular IP asset registration.

However, *_register()* does not verify whether the NFT token actually exists and proceeds to register the account regardless. Consequently, it is possible to front-run group registrations via a regular IP asset registration with the same *GroupNFT* contract address and **tokenId** corresponding to the next available group token ID, effectively blocking group registrations.

Because groups utilize sequential token IDs for their NFT tokens, maliciously using such a group NFT token ID would render it impossible to register a new group with that same ID, thereby completely preventing group registrations.

Recommended mitigation

To avoid registering an IP asset with a non-existent NFT token contract and token ID, which means it has not been minted yet, it is recommended to check within *_register()* whether the NFT token exists. Note that such a check in the current implementation will also impact regular IP asset registration, which should be assessed to determine if such a check is needed for those assets as well.

Team response

Fixed.

Mitigation review

The issue has been fixed by making the IP registration idempotent.

TRST-H-3 Fees are not charged for unstake(), redelegate(), and removeOperator() functions in IPTokenStaking allowing to spam the L1 chain

- **Category:** Logical flaws
- **Source:** IPTokenStaking.sol
- **Status:** Fixed

Description

The *unstake()*, *redelegate()*, and *removeOperator()* functions in the *IPTokenStaking* contract do not charge fees for the operations due to the lack of applying the **chargesFee** modifier, even though processing the events on the L1 chain consumes computational resources. As a result, this can be exploited to spam the L1 chain with unnecessary events, potentially leading to a DoS attack.

Notably, *stake()* does also not charge a fee. However, by enforcing a minimum staking amount (**minStakeAmount**), the potential for spamming the L1 chain is limited.

Recommended mitigation

It is recommended to always charge a fee for operations that consume computational resources on the L1 chain.

Team response

Fixed.

Mitigation review

The issue has been fixed by adding the *chargesFee* modifier to all relevant functions that would allow spamming the Story L1.

TRST-H-4 Derivative IPs can be registered with arbitrary license terms that are not attached to the parent IP

- **Category:** Logical flaws
- **Source:** PILicenseTemplate.sol
- **Status:** Pending Resolution

Description

Derivative child IPs must be registered with license terms attached to the parent IP(s). However, the owner of an IP can mint license tokens using *LicensingModule.mintLicenseTokens()* for an IP not yet registered as a derivative, allowing for any license terms to be applied. In contrast, other users can only mint license tokens for explicitly attached license terms. Importantly, this process does not attach the license term to the IP. Instead, it is only associated with the license token. Consequently, using the license token with the arbitrary license term to register the IP as a derivative through *registerDerivativeWithLicenseTokens()*, the **derivativesReciprocal = false** restriction can be circumvented, which is designed to prevent derivatives of derivatives when enabled by the license terms.

For example, given IP assets IP1, IP2, and IP3, along with the license terms T1 and T2, where T1 forbids derivatives of derivatives and is the only attached license term to IP1, the following steps can facilitate the creation of a derivative chain IP1 -> IP2 -> IP3, despite it being prohibited:

1. Before IP2 is registered as a derivative of IP1, the owner of IP2 mints a license token of IP2 with the license term T2 (e.g., using **derivativesReciprocal = true**) via *mintLicenseTokens()*.
2. Register IP2 as a derivative of IP1 via *registerDerivative()*.
3. Use *registerDerivativeWithLicenseTokens()* with the license token minted in step 1 to register IP3 as a derivative of IP2. As the license token is associated with the license term T2 instead of T1, and it fails to check whether T2 is attached to IP1, the derivative is successfully registered as a derivative of IP2. As a result, the derivative chain IP1 -> IP2 -> IP3 is created, even though IP1's license term is supposed to prevent the derivative registration of IP3.

Recommended mitigation

It is recommended to verify in *PILicenseTemplate._verifyRegisterDerivative()* whether the **licenseTermsId** is attached to the parent IP.

Team response

TBD

Mitigation review

TBD

TRST-H-5 Malicious ancestor IP can steal vault tokens from derivative

- **Category:** Logical flaws
- **Source:** LicenseRegistry.sol
- **Status:** Fixed

Description

When a derivative is registered, the **accumulatedRoyaltyPolicies** variable is updated for both the derivative and its parents. Each parent is updated with a new policy, and the derivative is updated with the policies of each parent.

When a new policy is linked to a child, the policy's *getPolicyRtsRequiredToLink()* hook is called, and the returned **rtsRequiredToLink** specifies the amount of royalty vault tokens to be sent to the policy.

Since all policies from the parent are copied to the child, hooks from external policies that the current child does not even directly use are called. It is worth noting that an external royalty policy can be added by anyone. This introduces the risk that a malicious parent can steal royalty tokens from its child without the user's consent.

Steps required for an attacker:

1. Register IP1.
2. Register an external policy P1.
3. Register a new license term T1 with the policy P1.
4. Attach T1 and another license term T2 to IP1.
5. Register a derivative IP2 with T1 (this will copy the external policy P1 to the **accumulatedRoyaltyPolicies** of parent IP1).

When an IP3 registers as a derivative of the malicious IP1 with the license T2, the external policy P1 is copied into IP3's **accumulatedRoyaltyPolicies**. Subsequently, *getPolicyRtsRequiredToLink()* of the malicious policy P1 is called and returns a higher-than-expected **rtsRequiredToLink**, enabling the policy to siphon royalty vault tokens from IP3.

Recommended mitigation

It is recommended to only add policies to the child's list. Avoid storing the policy in the parent's lists, instead, add the policy to the **accParentRoyaltyPolicies** variable after retrieving the parent policies.

Team response

Fixed.

Mitigation review

The issue has been fixed by only copying royalty policies to the children IPs.

TRST-H-6 Registering a derivative using a license token uses potentially updated royalty percentage, allowing the parent IP owner to front-run the registration

- **Category:** Logical flaws
- **Source:** LicensingModule.sol
- **Status:** Fixed

Description

There are two sources from which the revenue share percentage is fetched:

1. *LicensingConfig*
2. License terms

LicensingModule.setLicensingConfig() allows an IP owner to modify the revenue share configuration of a particular license at any time. If there are any adjustments to the revenue share after the license token has been minted but before the derivative is registered with the license token, the registration will reflect the updated revenue share amount rather than the value set at the token minting. This can lead to a higher-than-anticipated revenue share for the registered derivative, ultimately causing a financial loss because of increased royalty payments to the parent IP.

Recommended mitigation

It is recommended to store the revenue share amount alongside the license NFT token and use it on registration.

Team response

Fixed.

Mitigation review

The issue has been fixed by storing the revenue share percentage inside the license token metadata and using it when registering a derivative.

TRST-H-7 Registering a derivative with distinct license terms for the same parent IP leads to an inaccurate royalty stack and results in underpayment of royalties

- **Category:** Logical flaws
- **Source:** LicenseRegistry.sol
- **Status:** Fixed

Description

Registering an IP as a derivative with *LicensingModule.registerDerivative()* or *registerDerivativeWithLicenseTokens()* calls *LicenseRegistry.registerDerivativeIp()* internally, which checks if the child IP can be registered as a derivative of the specified parent IPs. For instance, it will revert if duplicate parent IPs with identical license terms are provided.

However, it is possible to register a derivative using different license terms for the same parent IP by calling *registerDerivativeIp()* with **parentIpIds = [IP_A, IP_A]** and **licenseTermsIds = [T1, T2]**. As a result, when linking the child IP to the parent IP through *RoyaltyModule.onLinkToParents()*, the royalty revenue share set by the first license term will be overridden by the royalty configuration of the second license term while still attaching both license terms to the derivative.

Specifically, in the LAP policy, *onLinkToParents()* will set **royaltyStackLAP** to an incorrect royalty value, leading to either insufficient or excess royalty payments. Insufficient payments result in losses for the ancestor IPs, while excess payments cannot be fully claimed by these ancestors. Notably, this is only an issue if the same royalty policy is used for a child IP and parent IP permutation.

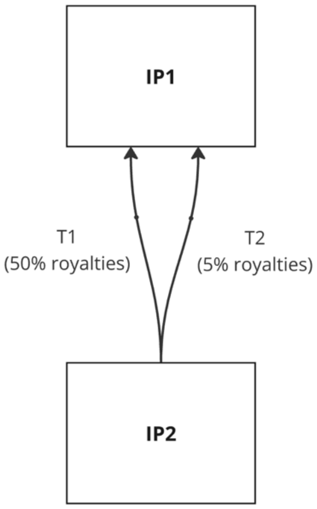


Figure 1. *IP2 registers as a derivative of IP2 using both the T1 and T2 license terms. IP2's royalty stack is overwritten by T2, resulting in a royalty stack of 5% + 5% = 10%*

Similarly, the **$.parentLicenseTerms** storage mapping, which records the license term used to link the child to the parent, is also overwritten. However, this mapping is not currently utilized by the contracts and is primarily accessible for off-chain querying.

Recommended mitigation

It is recommended to prevent registering a derivative using distinct license terms for the same parent IP.

Team response

Fixed.

Mitigation review

The issue has been fixed by preventing using duplicate parent IPs when registering a derivative.

Medium severity findings

TRST-M-1 tagDerivativeIfParentInfringed() may use outdated arbitration policy for the IP

- **Category:** Logical flaws
- **Source:** DisputeModule.sol
- **Status:** Fixed

Description

DisputeModule enables an IP to assign any whitelisted arbitration policy through *setArbitrationPolicy()*. This policy will take effect after a cool-down period and will be activated on the subsequent call to *_updateActiveArbitrationPolicy()*.

However, when a derivative is disputed using *tagDerivativeIfParentInfringed()*, its policy is directly retrieved from **$.arbitrationPolicies**. In this scenario, *_updateActiveArbitrationPolicy()* is not called prior to updating the arbitration policy for the IP. Consequently, if the IP has queued a new arbitration policy and the cooldown period has already elapsed following the *tagDerivativeIfParentInfringed()* call, the function uses an outdated arbitration policy instead.

Recommended mitigation

To ensure that the arbitration policy is always up-to-date, it is recommended to call *_updateActiveArbitrationPolicy()* at the beginning of *tagDerivativeIfParentInfringed().*

Team response

Fixed.

After consideration, instead of calling *_updateActiveArbitrationPolicy()* as in the recommendation - the parent dispute was used instead.

This is because - in our roadmap - we are considering replacing the need for *tagDerivativeIfParentInfringed()* with a precompile that checks if any ancestor is tagged. The closest behavior we can get to - without the precompile - would be propagating the dispute to the derivative with the same arbitration policy it had in the parent.

Mitigation review

The issue has been fixed by using the same arbitration policy as the original (parent) dispute.

TRST-M-2 A participant may register a derivative to restrict group management

- **Category:** Front-running attacks
- **Source:** GroupingModule.sol
- **Status:** Fixed

Description

When a new group is created, the owner should add its IP members. This is accomplished through *addIP()* and *removeIP()*. If the group already has derivatives as child IPs registered or issued license tokens, the call is reverted. The reasoning is that once the first payment is processed, the members should remain unchanged to ensure fair distribution of rewards.

As a consequence, if someone front-runs the group owner who attempts to add group members by registering a derivative or minting a license token, group member management will be restricted indefinitely. This is possible because each IP has a default license term that can be used to register derivatives or to mint license tokens at any time.

Recommended mitigation

It is recommended to implement a mechanism that allows the group owner to block anyone from registering a derivative of the group or from minting license tokens from the group IP until the owner has finished adding or removing group members.

Team response

Fixed.

Mitigation review

The issue has been fixed by allowing the group owner to disable the license.

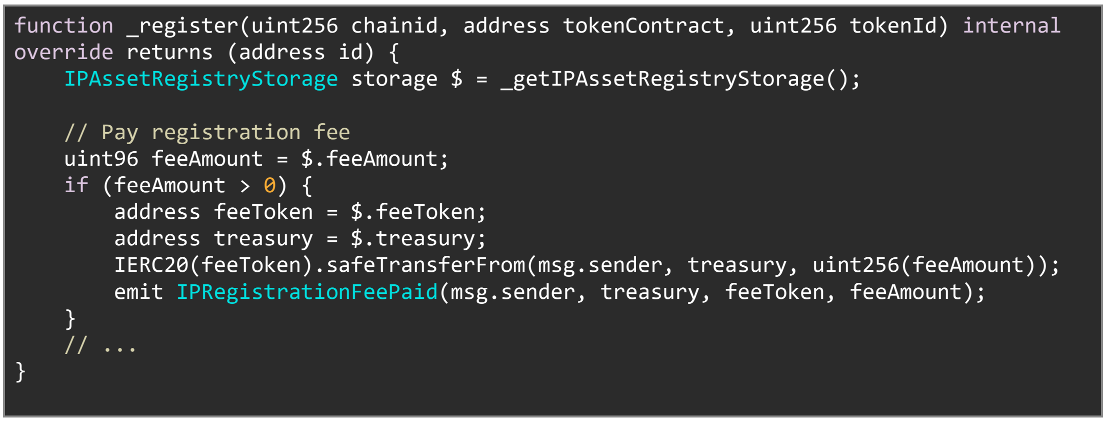
TRST-M-3 Enabled IP asset registration fee breaks group registration

- **Category:** Logical flaws
- **Source:** IPAssetRegistry.sol
- **Status:** Fixed

Description

Registering an IP asset can be subject to a fee, configured as **$.feeAmount** in *IPAssetRegistry* and enabled by Story governance via *setRegistrationFee()*.

The fee is charged in *_register()* and transferred from **msg.sender** to the treasury.



```
function _register(uint256 chainid, address tokenContract, uint256 tokenId) internal
override returns (address id) {
    IPAssetRegistryStorage storage $ = _getIPAssetRegistryStorage();

    // Pay registration fee
    uint96 feeAmount = $.feeAmount;
    if (feeAmount > 0) {
        address feeToken = $.feeToken;
        address treasury = $.treasury;
        IERC20(feeToken).safeTransferFrom(msg.sender, treasury, uint256(feeAmount));
        emit IPRegistrationFeePaid(msg.sender, treasury, feeToken, feeAmount);
    }
    // ...
}
```

This flow works as expected when registering regular IP assets. However, a group is registered via *GroupingModule.registerGroup()*, which means that when calling *GROUP_IP_ASSET_REGISTRY.registerGroup()*, **msg.sender** is the grouping module's address and not the user who registers the group. Consequently, the fee transfer uses the grouping module as the sender, which will fail due to insufficient funds and a lack of approval.

As a result, registering group IPs will not work as soon as the registration fee is enabled.

We classify this issue as medium-severity due to the issue only becoming relevant when the fee is enabled, which can also be disabled again.

Recommended mitigation

It is recommended to forward the **msg.sender** from *registerGroup()* to *_register()* to ensure that the correct sender is used for the fee transfer.

Team response

Fixed.

Mitigation review

The issue has been fixed by explicitly providing the address of the fee payer (which is set to the **msg.sender**).

TRST-M-4 Redelegation is not possible if the balance is less than 1024 IP tokens

- **Category:** Logical flaws
- **Source:** IPTokenStaking.sol
- **Status:** Fixed

Description

According to the documentation, users should be able to redelegate tokens even if their redelegation amount is less than 1024:

"The minimum redelegation amount is 1024 IP. If a delegator's initial stake is 1024 IP but later gets slashed, it can still redelegate its tokens to another validator even if the token amount is less than 1024 IP."

Story's Cosmos SDK modifications do not enforce the minimum amount for redelegations. However, the requested amount is enforced to be less than the currently delegated shares.

Conversely, the smart contracts enforce the minimum stake amount for all redelegate operations. Therefore, requesting a redelegation for a smaller amount than 1024 IP will not be possible if the amount falls below 1024 IP after a slashing operation on L1.

Recommended mitigation

It is recommended to remove the minimum stake amount check for redelegation and implement the check on the L1 side, which allows redelegation of an amount that is smaller than 1024 in case if delegation balance of the user is less than 1024 IP.

Team response

Fixed.

Mitigation review

The issue has now been resolved.

If a delegator has been slashed, it is now possible for them to redelegate an amount larger than their staked amount (after the slashing). This would lead to unbonding the full amount on L1.

Previously, during our review, this was not possible due to the constraint on L1.

TRST-M-5 Group members with an expired license can be added as group members, which dilutes the royalty revenue share for all group members

- **Category:** Logical flaws
- **Source:** GroupingModule.sol
- **Status:** Fixed

Description

The *addIP()* function enables a group owner to add additional member IPs. However, it neglects to verify whether the IP's license has expired and continues to add the IP as a group member. As a result, the IP is eligible to receive equal royalty rewards alongside all members. According to the documentation, expired IPs should not be able to receive royalty revenue. Thus, it is inferred that adding such an IP to a group should not be possible.

Recommended mitigation

It is recommended to add a check to *addIP()* to verify if the IP's license has already expired, permitting only those IPs that have no expiration time defined.

Team response

Fixed.

Mitigation review

The issue has been resolved by verifying if the IP has a non-zero expiration. If an expiration is set, the IP will not be added as a group member.

TRST-M-6 IPs with expired licenses can still earn royalty revenue

- **Category:** Logical flaws
- **Source:** RoyaltyModule.sol
- **Status:** Fixed

Description

In the derivative registration process, a derivative may have a specific expiration based on the license terms or the expiration of the parent IP. As stated in the documentation, expired IPs should not be able to generate royalty revenue. However, *RoyaltyModule.payRoyaltyOnBehalf()* does not check the expiration of **receiverIpId**, which violates the specification and permits new royalty revenue to be distributed to an expired IP.

Recommended mitigation

It is recommended to check if **receiverIpId** is not expired in *RoyaltyModule.payRoyaltyOnBehalf()*.

Team response

Fixed.

Mitigation review

The issue has been fixed by preventing royalty payments to expired IPs.

TRST-M-7 Group IP members may add additional license terms, violating the requirement of a single common license term among all members

- **Category:** Logical flaws
- **Source:** GroupingModule.sol
- **Status:** Pending Resolution

Description

A group IP is a structure designed to organize individual IPs with a common license term. Additionally, the license token minting fee is expected to be zero for both the group IP and its member IPs. Those requirements are verified when adding a new member IP to a group through *GroupingModule.addIp()*.

However, the current implementation allows a group IP to attach additional license terms using *LicensingModule.attachLicenseTerms()* or to set a non-zero minting fee with *LicensingModule.setLicensingConfig()*.

Furthermore, a group IP can be registered as a derivative of another IP, which also allows attaching additional license terms inherited by the parent IP.

Recommended mitigation

To maintain the group invariants, the following changes are recommended:

1. Implement a check in *LicensingModule.attachLicenseTerms()* to prevent a group IP from attaching additional license terms.
2. Add a check in *LicensingModule.setLicensingConfig()* to prohibit a group IP from setting a non-zero minting fee or altering any other licensing configuration that should remain unchanged.
3. Establish restrictions on registering a group IP as a derivative, ensuring that the group IP shares the same parent(s) and license terms as its member IPs.

Team response

TBD

Mitigation review

TBD

TRST-M-8 Royalty policy may charge more royalty tokens than expected

- **Category:** Logical flaws
- **Source:** RoyaltyModule.sol

- **Status:** Fixed

Description

RoyaltyModule._distributeRoyaltyTokensToPolicies() iterates all the parent IP's royalty policies and calls *getPolicyRtsRequiredToLink()*, which returns the amount of royalty tokens that should be sent to the external policy.

As anyone can register a custom external policy, a user must carefully choose any involvement of such an external royalty policy. However, if a user chooses a malicious external policy, a higher-than-expected **rtsRequiredToLink** amount may be requested that was not approved by the user. This results in royalty tokens being transferred to the external policy, which can claim more collected royalty rewards than the user anticipated.

Recommended mitigation

It is recommended to introduce an additional parameter to *LicensingModule.registerDerivative()* and *LicensingModule.registerDerivativeWithLicenseTokens()*, allowing users to specify the maximum amount of royalty tokens that will be transferred to each policy.

Team response

Fixed.

Mitigation review

The issue has been fixed by introducing a new slippage control, **maxRts**, that can be used to restrict the number of royalty tokens distributed to royalty policies.

TRST-M-9 IP account directly deployed via the ERC6551Registry registry can interact with the protocol without the ability to get disputed

- **Category:** Logical flaws
- **Source:** IPAccountChecker.sol
- **Status:** Fixed

Description

Among many checks, *IPAccountChecker.isIpAccount()* checks if the IP account is an account that was registered via the ERC6551 registry. However, it does not check if the IP account was registered directly via the Story *IPAssetRegistry* contract, e.g., whether it has **getString("NAME").length != 0**. As a result, it is not possible to raise a dispute for an IP account that was not properly registered via the IPAssetRegistry contract.

Recommended mitigation

It is recommended to call *IPAssetRegistry.isRegistered()* in *isIpAccount()*, which checks if the IP account was registered via *IPAssetRegistry* by validating **getString("NAME").length != 0**.

Team response

Fixed.

Mitigation review

The issue has been fixed by using *IPAssetRegistry.isRegistered()*.

TRST-M-10 IP owner can front-run with changing revenue share percent to make derivative pay more royalty

- **Category:** Front-running attacks
- **Source:** LicensingModule.sol
- **Status:** Pending Resolution

Description

There are two sources from which the revenue share percentage is fetched:

1. *LicensingConfig*
2. License terms

Using *LicensingModule.setLicensingConfig()*, an IP owner may override the revenue share configuration of a specific license.

As a result, the IP owner can front-run derivative registrations and change the minting license configuration to the buyers' disadvantage, such as increasing the revenue share amount.

Recommended mitigation

It is recommended to add an additional **maxRevenueShare** parameter to *LicensingModule.registerDerivative()* and *LicensingModule.registerDerivativeWithLicenseTokens()*, which would allow a derivative IP to specify a maximum revenue share amount.

Team response

TBD

Mitigation review

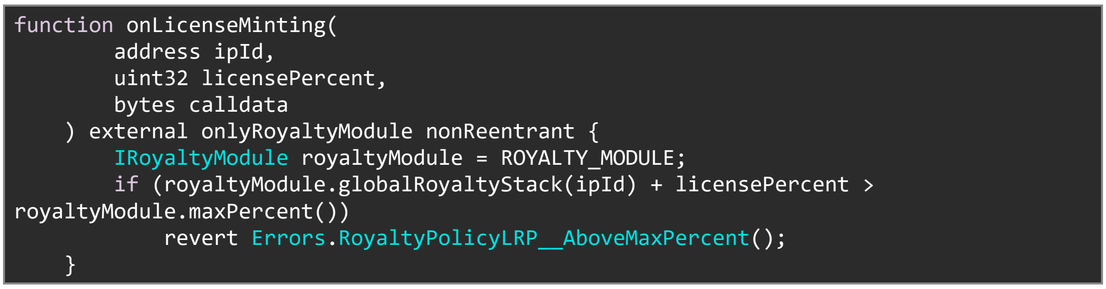
TBD

TRST-M-11 License token minting reverts for the LRP policy when exceeding the limit of 100%, unnecessarily limiting the derivative chain

- **Category:** Logical flaws
- **Source:** RoyaltyPolicyLRP.sol
- **Status:** Fixed

Description

IRoyaltyPolicy(royaltyPolicy).onLicenseMinting() of the whitelisted LAP and LRP policy is called during license minting with the intent to check if linking to the current parent IP is permitted. Specifically, it is checked whether the global royalty stack of the parent IP exceeds 100% after the new license is added.



```
function onLicenseMinting(
        address ipId,
        uint32 licensePercent,
        bytes calldata
    ) external onlyRoyaltyModule nonReentrant {
        IRoyaltyModule royaltyModule = ROYALTY_MODULE;
        if (royaltyModule.globalRoyaltyStack(ipId) + licensePercent >
royaltyModule.maxPercent())
            revert Errors.RoyaltyPolicyLRP__AboveMaxPercent();
    }
```

This check is reasonable for the Liquid Absolute Percentage (LAP) policy because a total royalty stack exceeding 100% implies that the derivative would owe more than 100% to its ancestors, which is not feasible. In contrast, the Liquid Relative Percentage (LRP) policy dictates that a derivative only pays the parent's royalty percentage. For instance, if a parent IP1 has a royalty stack of 50% and the license term's royalty percentage is set at 60%, this would result in an error as the total exceeds the maximum limit of 100%. Although, this is assumed to be a legitimate scenario for an LRP royalty policy.

Recommended mitigation

Given the potential to remix policies, a clear solution to this problem isn't apparent. The global royalty stack for IP combines royalty percentages from all policies, including both LAP and LRP, making it challenging to identify which portion belongs to each. If the royalty stack were to be divided by LAP and LRP policies, it would allow for restricting the LAP royalty stack to 100% while leaving the LRP stack without limits.

Team response

Fixed.

Mitigation review

The issue has been fixed by incorporating the licensor's LAP royalty stack in the check in *LRP.onLicenseMinting()*.

TRST-M-12 Group members without a deployed vault cannot receive royalty rewards from the group

- **Category:** Logical flaws
- **Source:** RoyaltyModule.sol
- **Status:** Pending Resolution

Description

When distributing group royalties to member IPs using *EvenSplitGroupPool.distributeRewards()*, the rewards are transferred via *RoyaltyModule.payRoyaltyOnBehalf()*. However, for *payRoyaltyOnBehalf()* to work, the receiving IP must have an associated royalty vault. Otherwise, an error occurs in *_payToReceiverVault()*, preventing the distribution of rewards.

Currently, when an IP is added as a group member, there is no verification if it has a vault deployed. A royalty vault is only deployed when a license token is minted for an IP or when a derivative is registered. As a result, group royalty rewards cannot be distributed to those member IPs.

Notably, this issue is temporary because once the IP group member has a vault deployed, *distributeRewards()* functions as intended again.

Recommended mitigation

It is recommended to either prevent adding group member IPs without a vault or, alternatively, deploy a royalty vault when adding the IP to the group.

Team response

TBD

Mitigation review

TBD

Low severity findings

TRST-L-1 Lack of UBI token distribution segregation might prevent validators from claiming their entitled rewards

- **Category:** Logical flaws
- **Source:** UBIPool.sol
- **Status:** Fixed

Description

UBIPool.setUBIDistribution() asserts that the allocated amount of UBI tokens for the newly created distribution does not exceed the contract's UBI balance. However, these funds are not segregated from other distributions since it is not accounted for not yet claimed rewards from previous distributions. Some validators might not have claimed their eligible rewards from previous distributions, even while a new distribution is already in progress. This new distribution attempts to allocate UBI tokens intended for past distributions. As a result, UBI token distributions are not fully "backed", meaning that not all validators can claim their entitled rewards.

Recommended mitigation

To ensure UBI token distributions are always able to be fully distributed, it is recommended to strictly account for already distributed funds across all distributions to be able to determine exactly how many funds are available for a new distribution.

Team response

Fixed.

Mitigation review

The issue has been fixed by tracking the pending (not claimed) UBI and using it to ensure the contract's balance is sufficient.

TRST-L-2 DisputeModule.setArbitrationPolicy() may reset previously queued policy

- **Category:** Logical flaws
- **Source:** DisputeModule.sol
- **Status:** Fixed

Description

When a new arbitration policy is set, a cooldown period should pass before the policy becomes active. In case a new arbitration policy is going to be set after the previous one was queued and a cooldown period has passed, then the previously queued arbitration policy will not become active and will be erased.

Recommended mitigation

It is recommended to call *_updateActiveArbitrationPolicy()* at the beginning of *setArbitrationPolicy()* to update the arbitration policy.

Team response

Fixed.

Mitigation review

The issue has been fixed by updating the arbitration policy within *setArbitrationPolicy()* and calling *_updateActiveArbitrationPolicy().*

TRST-L-3 Disputed IP can be added to a group as a member

- **Category:** Logical flaws
- **Source:** GroupingModule.sol
- **Status:** Fixed

Description

When a new member is added to a group, it is required to check if a member is not disputed as disputed IPs violate the protocol specification. However, the current implementation does not have such restrictions, thus allowing a group owner to add disputed IPs as members.

Recommended mitigation

It is recommended to disallow adding disputed IPs to the group.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking if the IP is tagged within *addIp().*

TRST-L-4 Tagged group member IPs dilute royalty rewards

- **Category:** Validation issues
- **Source:** GroupingModule.sol
- **Status:** Fixed

Description

EvenSplitGroupPool evenly splits and distributes collected royalty rewards to all group member IPs by transferring them to the IPs' royalty vaults via *RoyaltyModule.payRoyaltyOnBehalf()*. However, if a group contains member IPs tagged via a dispute (e.g., due to license terms infringement), those IPs cannot claim their rewards from the vault as the claimable reward is set to 0.

```
function _claimableRevenue(address account, uint256 snapshotId, address token)
internal view returns (uint256) {
    IpRoyaltyVaultStorage storage $ = _getIpRoyaltyVaultStorage();

    // if the ip is tagged, then the unclaimed royalties are unavailable until the
dispute is resolved
    if (DISPUTE_MODULE.isIpTagged($.ipId)) return 0;
```

Consequently, the disputed IPs dilute the rewards for the other non-tagged IPs within a group. Furthermore, member IPs cannot be removed from a group since it becomes locked once a license token is minted for the group IP or a derivative of it is registered.

Recommended mitigation

To mitigate the potential for such royalty reward dilution caused by tagged IPs it is recommended to check whether an IP is tagged when adding it to a group.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking if the IP is tagged within *addIp().*

TRST-L-5 Royalty policy's transferToVault() can be front-ran to grief royalty reward transfers to a vault

- **Category:** Front-running issues
- **Source:** RoyaltyPolicyLRP.sol, RoyaltyPolicyLAP.sol
- **Status:** Fixed

Description

The *transferToVault()* function in both *RoyaltyPolicyLRP* and *RoyaltyPolicyLAP* checks whether the requested transfer **amount** is eligible to be transferred and reverts otherwise:

```
if (transferredAmount + amount > maxAmount) revert
Errors.RoyaltyPolicyLAP__ExceedsClaimableRoyalty();
```

This introduces a griefing vector where a user calling *transferToVault()* with a transfer **amount** equal to the maximum transferrable value, i.e., **maxAmount - transferredAmount**, can be front-ran by another user who calls *transferToVault()* with **amount** being set to 1 wei. Consequently, the second call will error with **Errors.RoyaltyPolicyLRP__ExceedsClaimableRoyalty()** due to surpassing the maximum transferrable amount by 1 wei.

Recommended mitigation

To prevent griefing users calling *transferToVault()*, it is recommended to adjust **amount** to the maximum transfer amount.

```
- if (transferredAmount + amount > maxAmount) revert
Errors.RoyaltyPolicyLAP__ExceedsClaimableRoyalty();
+ if (transferredAmount + amount > maxAmount) {
+   amount = maxAmount - transferredAmount;
+
+   if (amount == 0) revert Errors.RoyaltyPolicyLAP__ZeroAmount();
+ }
```

Team response

Fixed.

Mitigation review

Fixed by removing **amount** variable.

TRST-L-6 License hook may be called when license config is not set

- **Category:** Logical flaws
- **Source:** LicensingModule.sol
- **Status:** Fixed

Description

If an IP wants to override any license term configuration, it can do so using *LicensingModule.setLicensingConfig()*. In this case, **isSet** variable of **LicensingConfig** is **true**, which signals that data from config can be used.

However, there are two places when the data is used even if the licensing config **isSet** is **false**.

1. In the *mintLicenseTokens().*
2. In the *_executeLicensingHookAndPayMintingFee().*

Recommended mitigation

We recommend checking if the license config is set in those places.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking **isSet** in those instances.

TRST-L-7 Empty group can mint license token or register derivative

- **Category:** Logical flaws
- **Source:** LicensingModule.sol
- **Status:** Fixed

Description

A group IP is created to logically group IPs with the same licenses. According to the Story team, if a group is empty, it is not valid, and thus, it should be disallowed for such groups to become derivatives or parent IPs. However, this is currently possible.

Recommended mitigation

It is recommended that an empty group not become a parent or derivative. Additionally, it should be considered to disallow minting license tokens for empty groups.

Team response

Fixed.

Mitigation review

The issue has been fixed. Minting license tokens for the group with no members is restricted. When registering a derivative, the child IP must not be a group itself, and the parent, if a group, must have non-zero members.

TRST-L-8 Group reward pool whitelist is not consistently checked

- **Category:** Logical flaws
- **Source:** GroupingModule.sol, RoyaltyModule.sol#L260, L527
- **Status:** Fixed

Description

The *IGroupRewardPool* (e.g., *EvenSplitGroupPool*) must be whitelisted via *GroupIPAssetRegistry.whitelistGroupRewardPool()* for use in the royalty vault. However, this whitelist check is only enforced in the royalty vault. In other instances where the group's reward pool is retrieved and interacted with, the whitelist status is not checked. If a pool is later un-whitelisted, it generally does not affect its functionality, and interacting with the pool is still possible.

Specifically, *GroupingModule.claimReward()*, *RoyaltyModule.onLicenseMinting()*, and *_distributeRoyaltyTokensToPolicies()* do not check the whitelist status and proceed without an error. Generally, wherever *getGroupRewardPool()* is used to retrieve the group's reward pool , the whitelist status is ignored.

Recommended mitigation

To ensure the whitelist is consistently checked, it is recommended to check whether the pool is whitelisted whenever it is interacted with it.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking the pool's whitelist status.

TRST-L-9 Unsafe cast from int256 to uint256 in IpRoyaltyVault._claimableRevenue()

- **Category:** Logical flaws
- **Source:** IpRoyaltyVault.sol
- **Status:** Fixed

Description

IpRoyaltyVault._claimableRevenue() calculates the claimable revenue share for a claimer of a given token. As part of the calculation, the resulting value is explicitly cast from **int256** to **uint256**:

```
return uint256(int256((accBalance * userAmount) / totalSupply()) - rewardDebt);
```

While we did not find a scenario where this calculation would result in a negative value, if it does happen, the resulting unsigned integer will be very large, resulting in an inflated claimable amount.

Recommended mitigation

Before casting the **int256** value to **uint256**, it is recommended to verify that it is positive. Otherwise, revert if it is negative.

Team response

Fixed.

Mitigation review

The issue has been fixed by verifying and reverting if the value is negative.

Additional recommendations

TRST-R-1 IPTokenStaking.STAKE_ROUNDING should be hardcoded

STAKE_ROUNDING variable of **IPTokenStaking** can be set to any non-zero value during initialization. However, the L1 expects this value to be 1e9.

We recommend hardcoding this value to **1e9** instead of making it configurable to avoid any potential issues.

Team response

Fixed.

Mitigation review

Resolved as suggested by hardcoding **STAKE_ROUNDING** to 1 gwei.

TRST-R-2 Predeploys.NamespaceSize is incorrect and should be changed to 2048

NamespaceSize in the **Predeploys** library is currently set to 1024. However, based on the *isPredeploy()*, this value is inaccurate. Based on the **>> 11** bit shift, the actual namespace size should be changed to 2048.

Team response

Fixed.

Mitigation review

Resolved. Addresses are now consistently identified as proxied addresses (within the address set with a length of 1024) through the *proxied()* function.

TRST-R-3 IPTokenStaking stake() and stakeOnBehalf() do not check validator key to be valid

It is recommended to consistently check the validator key provided for *stake()* and *stakeOnBehalf()* to be valid, as it is done in other places of the *IPTokenStaking* contract.

Team response

Fixed.

Mitigation review

Resolved by consistently checking the uncompressed pubkeys.

TRST-R-4 Check zero amounts after the rounding in IPTokenStaking contract

_setMinStakeAmount() and *_setMinUnstakeAmount()* assert that **newMinStakeAmount** and **newMinUnstakeAmount** are non-zero. However, this check should be done after applying the rounding correction. Otherwise, it is possible to set the **minStakeAmount** value to 0.

Team response

Fixed.

Mitigation review

Resolved by checking whether the value is 0 after applying the rounding correction.

TRST-R-5 Apply rounding mechanism to IPTokenStaking._unstake()

To ensure consistency and prevent rounding issues, apply the same rounding method to *IPTokenStaking._unstake()* as used in the staking and redelegation functionalities. This is important because the L1 will internally round down the amount when processing the Withdraw event.

Team response

Fixed.

Mitigation review

Resolved by verifying that the remainder is 0.

TRST-R-6 Remove unused childIpOwner param from LicensingModule._executeLicensingHookAndPayMintingFee()

The **childIpOwner** parameter in *LicensingModule._executeLicensingHookAndPayMintingFee()* is unused. Consider removing the parameter.

Team response

Fixed.

Mitigation review

Resolved by removing the unused **childIpOwner** parameter.

TRST-R-7 Use ERC165 interface to check if contract supports specific functionality

During the registration of an external policy, *getPolicyRtsRequiredToLink()* is executed in order to check if the external policy supports the **IExternalRoyaltyPolicy** interface. However, it is recommended to use the ERC165 interface checks instead.

Team response

Fixed.

Mitigation review

Resolved by using **ERC165** for external royalty policy validation.

TRST-R-8 Apply nonReentrant modifier in all places of GroupingModule contract

The **GroupingModule** contract inherits the **nonReentrant** modifier from **ReentrancyGuardUpgradeable**, but only uses it with the *registerGroup()*. It is recommended to apply the **nonReentrant** modifier to *collectRoyalties()*, *claimReward()*, *addIp()*, *removeIp()* to be extra cautious.

Team response

Fixed.

Mitigation review

Resolved by adding the **nonReentrant** modifier to the listed functions.

TRST-R-9 Emit events in IPGraphACL for better off-chain monitoring

IPGraphACL.whitelistAddress() and *revokeWhitelistedAddress()* whitelist access to the IPGraph precompile for a specified address or revoke access for an address that has already been whitelisted, respectively. To ensure off-chain monitoring of these functions, it is recommended to emit events, containing details of the whitelisted or revoked address.

Team response

Fixed.

Mitigation review

Resolved by emitting events in both functions.

TRST-R-10 Explicitly document that royalty tokens can only be used to claim rewards within the same derivative chain

Royalty vaults might receive royalty tokens from other vaults, for example, by registering a derivative and using an external royalty policy that requests a certain amount of royalty tokens of the child IP's vault to allow linking to the parent IP. Whenever the derivative receives royalty revenues, the revenue is shared pro-rata with the royalty token holders. In the case where a vault is eligible for royalty rewards from another vault, *IpRoyaltyVault.claimByTokenBatchAsSelf()* and *claimBySnapshotBatchAsSelf()* are supposed to be used, to ensure the claiming vault has its balance updated with the claimed rewards accordingly.

However, both functions ensure that only royalty rewards from target IPs within the same derivative chain are claimable.

```
// ensures that the target ipId is from a descendant ip which in turn ensures that
// all accumulated royalty policies from the ancestor ip have been checked when
// a payment was made to said descendant ip
if (!ROYALTY_MODULE.hasAncestorIp(targetIpId, _getIpRoyaltyVaultStorage().ipId))
    revert Errors.IpRoyaltyVault__VaultDoesNotBelongToAnAncestor();
```

Consequently, if a vault owns royalty tokens from an IP that does not belong to the derivative chain, rewards cannot be claimed, resulting in those tokens being worthless within the vault. To discourage users from purchasing royalty tokens on the open market and moving them to their IP's vault under the assumption that they can claim rewards, it is recommended to explicitly outline this behavior both in the documentation and the code.

Team response

The client acknowledges it and will add it to their online documentation.

Mitigation review

Resolved.

TRST-R-11 ArbitrationPolicyUMA arbitration relayer conflicts should be avoided

In *DisputeModule*, the **$.isWhitelistedArbitrationRelayer** mapping holds the whitelisted relayers for a specific arbitration policy, which is set by Story governance through *whitelistArbitrationRelayer()*. For the UMA arbitration policy, it is essential that the *ArbitrationPolicyUMA* contract is whitelisted to be able to call *DisputeModule.setDisputeJudgement()*.

However, there may be an additional arbitration relayer also whitelisted for *ArbitrationPolicyUMA*, allowing it to judge disputes prior to the policy's assertion being resolved. In this scenario, when the UMA oracle calls *ArbitrationPolicyUMA.assertionResolvedCallback()*, the internal call to *DisputeModule.setDisputeJudgement()* will fail due to the dispute.currentTag != IN_DISPUTE check. As a result, the assertion bond cannot be returned. Therefore, it is recommended not to whitelist any other arbitration relayer than the *ArbitrationPolicyUMA* or modify the code so that the designated arbitration policy for a dispute is the sole actor capable of judging a dispute using *setDisputeJudgement()*.

Team response

Fixed.

Mitigation review

Mitigated by only having a single arbitration relayer per arbitration policy.

TRST-R-12 Use the whenNotPaused modifier in ArbitrationPolicyUMA to reduce risks associated with relaying UMA decisions from ETH Mainnet to Story L1

The *ArbitrationPolicyUMA* arbitration policy uses UMA for disputes. Based on the UMA docs, disputes are handled by the DVM (i.e., the UMA token holders vote collectively on the outcome). The DVM contract is deployed on Ethereum Mainnet, so it is required that a dispute outcome must be relayed to Story L1. Currently, there are two approaches recommended by UMA:

1. Centralized approach - UMA acts as a trusted party relaying DVM decisions from ETH Mainnet to Story L1.
2. Decentralized approach – Using a cross-chain messaging protocol (e.g., Layer Zero) to relay DVM decisions from ETH Mainnet to Story blockchain.

Initially, it is reasonable to adopt a centralized approach while later transitioning to a decentralized mechanism. To address relay issues, such as when the DVM decision cannot temporarily be relayed to Story L1, it is recommended to use the **whenNotPaused** modifier *onRaiseDispute()*, *disputeAssertion()*, and *assertionResolvedCallback()*. This would allow pausing *ArbitrationPolicyUMA* in situations where known relay issues or delays occur with relaying the dispute decision to Story L1. Otherwise, a malicious actor may take advantage of such a relay delay by disputing an IP, which may not be possible to refute (depending on the maximum liveness period), and so the IP dispute is resolved and the IP tagged.

Team response

Fixed.

Mitigation review

Resolved. **whenNotPaused** modifier has been added to *onRaiseDispute(), disputeAssertion()*, and *assertionResolvedCallback().*

TRST-R-13 Remove unused Predeploys.isPredeploy() and isActivePredeploy()

Predeploys.isPredeploy() checks if the provided address is located in the predeploy namespace, which is supposed to be of size 1024. However, due to the right shift of the address by 11 bits, it will incorrectly use a predeploy namespace size of 2048. As *proxied()* in the *Predeploys* library already implements the correct behavior, it is recommended to remove *isPredeploy()*. Furthermore, *isActivePredeploy()* is unused and should be removed as well.

Team response

Fixed.

Mitigation review

Resolved by removing both *isPredeploy()* and *isActivePredeploy().*

TRST-R-14 Update the documentation around the WIP token deployment and predeploy namespaces

It is recommended to outline the reasoning behind the wrapped IP token's (WIP) distinct deployment method, which employs the chain ID as the address rather than utilizing the predeploy namespace for the token contract. Furthermore, improve the documentation on deployment standards for WIP, Create3, and other predeploys, ensuring that the purpose and function of each deployment method are well-explained.

Team response

Fixed.

Mitigation review

Resolved by adding additional documentation.

TRST-R-15 Update documentation to clarify that group IPs cannot have other groups as members

In the documentation, it is mentioned that a group IP account can be composed of both individual IPs and other group IPs. However, the current implementation errors when attempting to add a group via *addIp()* as a member.

```
if (GROUP_IP_ASSET_REGISTRY.isRegisteredGroup(ipIds[i])) {
    revert Errors.GroupingModule__CannotAddGroupToGroup(groupIpId, ipIds[i]);
}
```

To prevent misunderstandings and clearly outline the requirements for group members, it is recommended to update the documentation to indicate that adding groups as members is not permitted.

Team response

The Client updated the documentation accordingly.

Mitigation review

Resolved by updating the documentation.

TRST-R-16 Restrict setLicensingConfig() to prevent derivative IPs from reducing commercialRevShare when using the LRP royalty policy

The *setLicensingConfig()* function in the Licensing Royalty Policy (LRP) contract allows descendant derivative IPs to reduce **commercialRevShare** to the point where it leads to significant royalty dilution for ancestor IP holders when these derivatives collect royalties. Although a disclaimer in *RoyaltyPolicyLRP* addresses this concern, additional protections are recommended. One option is to require that derivatives can only increase the **commercialRevShare** value passed down from the parent, preventing royalties from being minimized for ancestor IPs. Furthermore, the documentation and code comments should emphasize that by utilizing the *LicensorApprovalChecker*, IPs can explicitly approve or reject derivatives, thus providing greater control over derivative registrations.

Team response

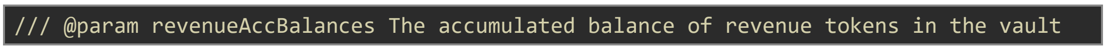
Fixed.

Mitigation review

Resolved by ensuring the new royalty percentage for the license is not less than the default one.

TRST-R-17 Rename outdated NatSpec revenueAccBalances parameter to vaultAccBalances

The NatSpec parameter documentation for the **vaultAccBalances** storage value is outdated as it incorrectly refers to the non-existent **revenueAccBalances** variable.



```
/// @param revenueAccBalances The accumulated balance of revenue tokens in the vault
```

To ensure consistent documentation, update the NatSpec comment.

Team response

Fixed.

Mitigation review

Resolved by updating the NatSpec parameter documentation.

Centralization risks

TRST-CR-1 The trusted UBIPool contract owner manually creates UBI reward distributions

UBIPool.setUBIDistribution() depends on the trusted contract owner (i.e., Story) to set up UBI reward distributions for validators. However, these configured validators may not necessarily be the actual validators, or there could be no (monthly) distribution at all.

Team response

The distribution process will be public and transparent. The calculations will be posted to a public forum where the community can audit them with enough time before being posted. The transaction to post is also time-locked, so there is time to verify correctness. To add context, this is a program that will have an end date, not a long-term feature of Story network.

TRST-CR-2 Most contracts can be upgraded

Most Story protocol contracts are deployed as proxy contracts. This means the protocol can be upgraded at any time, allowing for changes to the logic. It is recommended that the authority to upgrade contracts be transferred to the governance system.

Team response

Initially, these roles are owned by a multisig, responsible for executing the decisions made by governance.

All upgrades are time locked via *AccessManager* (possibly other admin methods) for security reasons and in order to give the community time to react.

The goal is to reduce admin methods and freeze upgradeability when the protocol matures.

TRST-CR-3 Permissioned and trusted contract authority can change important protocol parameters

In many instances, *AccessManagedUpgradeable* is used to restrict calling functions that change important protocol parameters to an authorized address via the **restricted** modifier. Although validations exist in most of those restricted setter functions to prevent setting arbitrary values, the ability to modify parameters at any time presents a centralization risk. Below is a non-exhaustive list illustrating the authorized address's capabilities:

1. *GroupNFT.setLicensingImageUrl()* and *LicensingToken.setLicensingImageUrl()* allows changing the NFT's base image.
2. *IPGraphACL.whitelistAddress()* permits an address to call the ipgraph precompile, which can be used to manipulate the stored derivative IP chains.

3. *RoyaltyModule.setRoyaltyFeePercent()* sets the treasury's royalty fee share, capped at a maximum of 100%.
4. *VaultController.setSnapshotInterval()* sets the minimum interval at which ERC20 balance snapshots can be taken. For example, if this value is set to 0, it allows snapshots to be taken at any moment, potentially resulting in a significantly higher number of snapshots, which increases gas usage when retrieving balances.

Team response

We acknowledge this issue. Execution of these methods will happen after transparent communication with the community, through governance processes/multisig operation, and in some cases behind a delay (through *AccessManager*), so the community will have time to react.

If possible, future iterations of PoC will remove admin methods in the spirit of progressive decentralization.

Systemic risks

TRST-SR-1 UMA arbitration result must be relayed from Ethereum Mainnet to Story L1

IP disputes involving *ArbitrationPolicyUMA* must relay their outcomes from the Ethereum Mainnet to Story L1, whether through centralized or decentralized means. Both approaches carry potential risks of delay or manipulation.

Team response

UMA will be responsible for properly relaying the decision from ETH mainnet to Story L1.

As last resort, Story always has the option of blacklisting any arbitration policy from any future dispute arbitration related to the protocol.

It is the intention to add new arbitration policies and offer a wider range of options to users as the protocol grows.

TRST-SR-2 Group IP royalty rewards are diluted by tagged member IPs

Royalty rewards are equally shared among member IPs, but only non-tagged IPs can receive their eligible portion. Disputed or tagged member IPs thus reduce the royalty amounts available to other IPs.

Team response

Acknowledged, will be added to Story's online documentation.

TRST-SR-3 External royalty policies can be registered by anyone and are potentially malicious

External royalty policies can be registered by anyone. Consequently, users need to exercise caution when choosing an external policy. As the external policy's *getPolicyRtsRequiredToLink()* hook is called when registering a derivative, it might prevent registering a derivative by reverting the call, which may render a previously paid license token unusable. Additionally, as outlined in H-5 and M-8, royalty vault tokens are susceptible to theft.

Team response

External policies can be registered in a permissionless way. The goal is that external developers could "plug-in" their own royalty rules in order to give more freedom to application developers building on top.

The communication between the protocol and external royalty policies is made via a view function that cannot change state to partially mitigate the risk.

However, it is true that users need to be aware that when interacting with external royalty policies there is an additional degree of risk associated and those external royalty policies potentially being malicious.