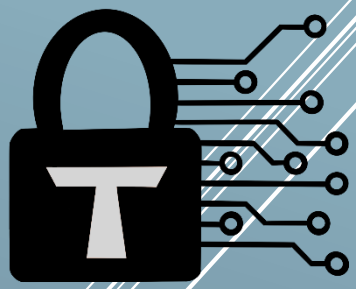
Trust Security

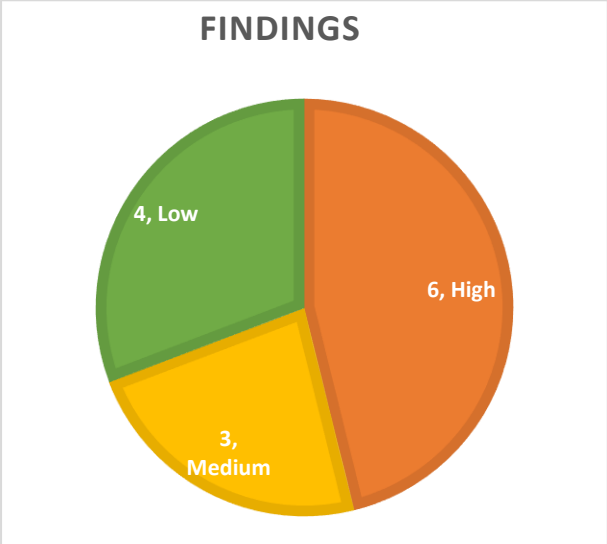


Smart Contract Audit

Story Protocol

09/07/2024

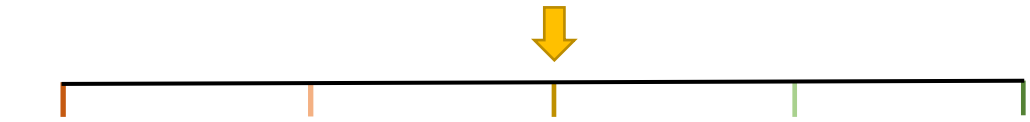
Executive summary



Category	IP management
Audited file count	36
Lines of Code	3719
Auditor	Bernd Artmüller, rvierdiiev
Time period	29/04-20/05

Severity	Total	Fixed	Acknowledged
High	6	6	-
Medium	3	3	-
Low	4	3	1

Centralization score



Signature

Digitally signed by Trust
Date: 2024.07.09 14:06:28 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	20.05.2024	Client report
0.2	09.07.2024	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

- contracts/access/AccessControlled.sol
- contracts/access/AccessController.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/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/ArbitrationPolicySP.sol
- contracts/modules/external/TokenWithdrawalModule.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/metadata/CoreMetadataModule.sol
- contracts/modules/metadata/CoreMetadataViewModule.sol
- contracts/modules/royalty/policies/IpRoyaltyVault.sol
- contracts/modules/royalty/policies/RoyaltyPolicyLAP.sol
- contracts/modules/royalty/RoyaltyModule.sol
- contracts/modules/BaseModule.sol
- contracts/pause/ProtocolPausableUpgradeable.sol
- contracts/pause/ProtocolPauseAdmin.sol
- contracts/registries/IPAccountRegistry.sol
- contracts/registries/IPAssetRegistry.sol
- contracts/registries/LicenseRegistry.sol
- contracts/registries/ModuleRegistry.sol
- contracts/IPAccountImpl.sol
- contracts/IPAccountStorage.sol
- contracts/LicenseToken.sol

Repository details

- **Repository URL:** https://github.com/storyprotocol/protocol-core-v1
- **Commit hash:** 888bb8b3dc0eb706715594591fbb8a5b25013c1f
- **Mitigation review commit hash:** efd2008e259fac75d7bff5f7c495538c87eebf17

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 Inability to override existing term's license token minting fee configuration

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

Description

When a user mints a license token for a specific IP, the minting fee that needs to be paid is determined by the *LicenseModule::_getTotalMintingFee()* function.

```
function _getTotalMintingFee(
    Licensing.MintingLicenseConfig memory mintingLicenseConfig,
    address licensorIpId,
    address licenseTemplate,
    uint256 licenseTermsId,
    uint256 mintingFeeSetByLicenseTerms,
    uint256 amount
) private view returns (uint256) {
    if (!mintingLicenseConfig.isSet) return mintingFeeSetByLicenseTerms * amount;
    if (mintingLicenseConfig.mintingFeeModule == address(0)) return mintingLicenseConfig.mintingFee *
amount;
    return
        IMintingFeeModule(mintingLicenseConfig.mintingFeeModule).getMintingFee(
            licensorIpId,
            licenseTemplate,
            licenseTermsId,
            amount
        );
}
```

There are three sources from which the minting price is determined, listed by priority:

1. *MintingFeeModule*,
2. *MintingLicenseConfig*, and
3. License terms

If an IP intends to override the fee configuration of a specific license, an individual minting configuration containing the minting fee can be set as the **mintingLicenseConfig**. Alternatively, the configured license terms minting fee is used.

This can be done with the *setMintingLicenseConfigForLicense()* and *setMintingLicenseConfigForIp()* functions in the *LicenseRegistry* contract.

However, as both functions use the **onlyLicensingModule** modifier and *LicensingModule* does not have any means to call them, the IP is not able to set those minting configurations.

Moreover, the **receiverCheckModule**, which is part of **Licensing.MintingLicenseConfig** cannot be used as well, and the license token receiver cannot be verified.

Recommended mitigation

We recommend adding the ability for IPs to set the license token minting configuration. For example, by adding appropriate function to the *LicensingModule* contract that will interact with *LicenseRegistry*. Moreover, ensure that the **verifyPermission** modifier is used to properly check authorization.

Team response

Fixed.

Mitigation review

The issue has been fixed by adding a new function *setLicensingConfig()* to set the licensing configuration for specific license terms of an IP.

TRST-H-2 Missing derivativesReciprocal check when registering derivative

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

Description

License terms has the **derivativesReciprocal** parameter. In case when it is set to **false**, only derivatives on the first inheritance level can be created, and derivatives on the second level are not allowed.

The *mintLicenseTokens()* function internally calls *verifyMintLicenseToken()*, which prevents minting license tokens for registering derivatives on the next level, i.e., derivatives of derivatives.

However, when a derivative is created via *registerDerivative()*, the **derivativesReciprocal** check is missing. As a result, derivatives can be created at more levels and consequently break license terms.

Recommended mitigation

We recommend also checking the **derivativesReciprocal** configuration parameter in the *registerDerivative()* execution flow.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking if the **derivativesReciprocal** configuration parameter is set to **true** when registering a derivative of a derivative with *registerDerivative()*.

TRST-H-3 Incorrectly determining the license token expiration time

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

Description

Each license term has an expiration time. It is possible for a term to have an expiration of **0**, which means that the license does not have an expiration, i.e., it's infinitely valid.

When a derivative is registered, then its expiration is calculated and set based on the parents' terms. It is supposed to be set to the shortest expiration from those terms.

However, *getEarlierExpireTime()* incorrectly calculates the expiry. Specifically, if a parent's expiration is **0**, indicating no expiration, the derivative will also not have an expiration even though the other license terms have a concrete non-zero expiration configured.

```
uint expireTime = _getExpireTime(licenseTermsIds[0], start);
for (uint i = 1; i < licenseTermsIds.length; i++) {
    uint newExpireTime = _getExpireTime(licenseTermsIds[i], start);
    if (newExpireTime < expireTime || expireTime == 0) {
        expireTime = newExpireTime;
    }
}
```

Recommended mitigation

We recommend explicitly checking if **newExpireTime** equals **0** and not assigning it to **expireTime** in this case.

Team response

Fixed.

Mitigation review

The issue has been fixed by only comparing expiration times if the expiration time being compared is larger than 0.

TRST-H-4 Child IP can exceed the parents' expiration

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

Description

Each license term defines an expiration time, and when a derivative is registered, its expiration is calculated and set based on the parents' terms. The expiration time is supposed to be set to the shortest expiration time from its parent terms by calling *getEarlierExpireTime()*.

However, this function does not consider the parents' IP expiration time, which makes it possible for a derivative to have an expiration that exceeds the parents.

For example, consider a parent IP that expires in two months, allowing a derivative to extend its term with a 1-year expiration. As a result, the parent IP will expire after two months, but the derivative will still be valid for ten more months.

Recommended mitigation

We recommend checking that the expiration calculated by the *getEarlierExpireTime()* function does not exceed the expiry of any parent IP.

Team response

Fixed.

Mitigation review

The issue has been fixed by incorporating the parent IP's expiration time in the calculation of the earliest expiration time.

TRST-H-5 Attacker can burn arbitrary license tokens from other users

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

Description

registerDerivativeWithLicenseTokens() determines the **childIpOwner** in line 289 by calling *IIPAccount(payable(childIpId)).owner()*. This *owner()* function internally retrieves the owner of the IP asset via *ownerOf(..)*:

```
function owner() public view returns (address) {
    (uint256 chainId, address contractAddress, uint256 tokenId) = token();
    if (chainId != block.chainid) return address(0);
    return IERC721(contractAddress).ownerOf(tokenId);
}
```

However, **contractAddress** might be a malicious (NFT) contract that returns whatever address is desired from the *ownerOf()* function and, thus, also from *owner()*.

Consequently, in *registerDerivativeWithLicenseTokens()*, the **childIpOwner** can be set to any arbitrary address and therefore bypassing the owner check in *LicenseToken.validateLicenseTokensForDerivative()* in lines 142-144. After that, the license tokens are burned.

This effectively allows an attacker to burn license tokens from any other user.

Recommended mitigation

We recommend updating *registerDerivativeWithLicenseTokens()* to only allow the owner of the specified license token(s) to call the function, i.e., verify **msg.sender** to be the license token owner.

Team response

Fixed.

Mitigation review

The issue has been fixed by restricting the caller of *registerDerivativeWithLicenseTokens()*.

TRST-H-6 Circular chaining of Ips allows bypassing of core restrictions

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

Description

Checking if the child already has license terms attached in *LicenseRegistry.registerDerivativeIp()* does not detect whether the parent IP is already a parent node, as the parent IP might not have license terms explicitly added to **attachedLicenseTerms**. For example, if only the default license with the default term ID is used.

As a result, it is possible to create a circular chain of IPs, where a node is simultaneously a parent and a child node, as shown in Figure 1.

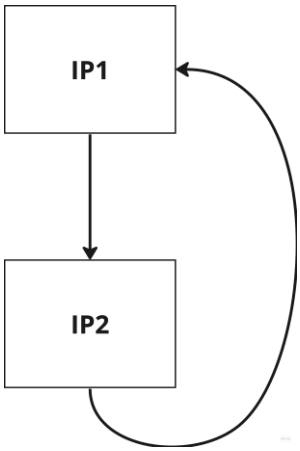


Figure 1. *Circular IP chain. IP1 is the parent of IP2 and the child of IP2.*

In addition to creating circular IP chains, this oversight allows retroactively turning a parent node into a derivative (child node) even though this parent node already has derivatives assigned. For example, this can be exploited to create reciprocal derivatives (derivatives of derivatives) for license terms that explicitly disabled such a configuration via the **derivativesReciprocal** flag.

Please note that this is only possible if there is no royalty policy involved, as in this case, the parent IPs are always flagged with **isUnlinkableToParents**, preventing them from being used as derivatives.

Recommended mitigation

We recommend adding a check to ensure the parent IP cannot become a derivative once it has derivatives assigned.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking whether the child IP already has a child IP before registering it as a derivative.

Medium severity findings

TRST-M-1 Owner-minted license tokens with arbitrary licenses cannot be used to register derivatives

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

Description

License tokens can be minted via the *LicensingModule.mintLicenseTokens* function, either by the IP owner or anyone else. The IP owner has special permissions and is able to use any arbitrary but valid license template and terms, without explicitly attaching it to the IP.

Not attaching the license template and terms is problematic and prevents the license token from being used when registering a derivative via *LicensingModule.registerDerivativeWithLicenseTokens()*, as this function validates the license and terms in line 312 by calling *registerDerivativeIp()*. This function reverts in lines 430-438 if a license is used that has not been attached, which is not the case if the owner used an arbitrary license when minting the license tokens.

As a result, it is not possible to use such license tokens.

Recommended mitigation

We recommend amending the logic to ensure that license tokens with arbitrary licenses can be used when registering a derivative.

Team response

Fixed.

Mitigation review

The issue has been fixed by adding a new parameter **isUsingLicenseToken** that allows the registration of a derivative with license terms that are minted by the IP owner but not attached to the IP Asset.

TRST-M-2 IP account directly deployed via the ERC6551Registry registry can interact with the protocol without the ability to get disputed

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

Description

IP accounts are supposed to be deployed via *IPAssetRegistry::register()*, which internally interacts with the *ERC6551Registry* contract. This *register()* function stores additional data on the IP account, such as **NAME**, **URI**, and **REGISTRATION_DATE**.

In many instances, the protocol only checks if the interacting IP account has been registered with the *ERC6551Registry* contract without also checking whether the IP was registered via the protocol's *IPAssetRegistry.register* function. The only exception is *DisputeModule::raiseDispute()*, which makes use of the *IP_ASSET_REGISTRY::isRegistered()* function, checking if **NAME** is set for the IP to ensure the IP account was correctly registered.

However, if an IP was directly deployed via the *ERC6551Registry* contract, it cannot be disputed due to **NAME** not being initialized.

Recommended mitigation

We recommend always checking if the IP account was correctly registered via *IPAssetRegistry::register()*.

Team response

Fixed.

Mitigation review

The issue has been fixed by making IPAccountRegistry::*registerIpAccount() internal*, ensuring that a IP Account registration can only be done through the *IPAssetRegistry*, providing a single, consistent path for IP registration.

TRST-M-3 Impartial license terms compatibility check for reciprocal derivatives

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

Description

When registering a new derivative of license terms that belong to the *PILicenseTemplate* license template, *PILicenseTemplate::_verifyCompatibleLicenseTerms()* is called to verify whether the license terms are compatible. Specifically, if the parents' terms have **derivativesReciprocal** set to **true**, meaning that unlimited levels of derivatives can be created as long as the license terms are the same, all license terms must have **derivativesReciprocal** set to **true** as well. However, the function fails to check whether the license terms are equal. As a result, it is possible to use license terms with slightly different terms, e.g., different values for **commercialAttribution**.

Recommended mitigation

We recommend ensuring that all **licenseTermsIds** are equal if **derivativesReciprocal** is **true**.

Team response

Fixed.

Mitigation review

The issue has been fixed by adding an additional check to ensure the license term IDs in **licenseTermsIds** are equal.

Low severity findings

TRST-L-1 IP owner can front-run with changing minting configs to make license buyer overpay

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

Description

There are three sources from which the minting price is determined, listed by priority:

1. *MintingFeeModule*,
2. *MintingLicenseConfig*, and
3. License terms

If an IP intends to override the fee configuration of a specific license, an individual minting configuration containing the minting fee can be set as the **mintingLicenseConfig**. Alternatively, the configured license terms minting fee is used.

Currently, the functions that would allow changing this configuration are not callable, as reported in TRST-H-1 Inability to override existing term's license token minting fee configuration.

However, if they were callable, the IP owner would be able to front-run license-buying users by changing the minting license configuration to the buyers' disadvantage, such as increasing the minting fee.

Recommended mitigation

We recommend adding an additional **maxPayment** parameter to *LicensingModule::mintLicenseTokens()*, which would allow a license token minter to specify a maximum payment amount.

Team response

Acknowledged. Since it would have a significant downstream impact due to interface change (SDK, websites, etc.), we will track the issue internally and tackle it in subsequent development iterations.

TRST-L-2 Collecting royalty tokens can possibly be halted by paused token transfers

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

Description

Ancestors collect royalty vault tokens via *IpRoyaltyVault.collectRoyaltyTokens()* and receive their share of the accrued royalty funds, distributed by the internally called *_collectAccruedTokens()* function.

However, if any one of the whitelisted tokens is temporarily paused (or any other cause that halts token transfers), the token transfer errors and the royalty vault tokens cannot be collected until the token transfers are functional again.

Recommended mitigation

We recommend adding an additional boolean flag that lets the ancestor specify if royalties should also be collected.

Team response

Fixed.

Mitigation review

The issue has been fixed by having users collect accrued revenue tokens separately from claiming royalty vault tokens.

TRST-L-3 Terms revenue ceiling is not validated when registering new license terms

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

Description

Both _verifyCommercialUse() and _verifyDerivatives() check and verify the terms configuration, but neglect to check the **commercialRevCelling** and **derivativeRevCelling** values. Those two values are seemingly not used to limit the royalty revenue, instead they are part of the license terms and supposedly used to legally enforce license terms off-chain.

Recommended mitigation

We recommend checking the **commercialRevCelling** value and reverting for non-zero values if the term is non-commercial. Additionally, if derivatives are not allowed, consider enforcing **derivativeRevCelling** to be zero.

Team response

Fixed.

Mitigation review

The issue has been fixed by adding additional checks.

TRST-L-4 License tokens can be minted for a non-existent IP, bypassing the potentially configured receiveCheckModule hook

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

Description

License tokens for an IP with a specific license and terms are minted via *LicensingModule::mintLicenseTokens()*. However, this function does not check if the IP is a legitimately registered account. This can be exploited by front-running an IP registration transaction and minting license tokens before the IP has configured the **MintingLicenseConfig.receiverCheckModule** hook which might act as an allow list to only permit certain users to mint license tokens.

Recommended mitigation

We recommend checking if the licensor IP is registered.

Team response

Fixed.

Mitigation review

The issue has been fixed by checking if the licensor IP is already registered.

Additional recommendations

TRST-R-1 IPAccountImpl.isValidSigner() does not allow to specify to parameter

IPAccountImpl.isValidSigner() checks if the **signer** can execute a particular action. However, it does not include the supplied **to** address when calling *_isValidSigner()*, instead, the zero address is provided instead.

```
if (_isValidSigner(signer, address(0), data)) {
    return IERC6551Account.isValidSigner.selector;
}
```

As permissions might be granted to specific contract addresses, the *isValidSigner()* misleadingly determines if the signer is valid.

We recommend allowing users to specify the **to** address.

Team response

Fixed.

Mitigation review

The **to** address can now be provided to *isValidSigner()* via the **data** parameter.

TRST-R-2 Derivative IPs can be repeatedly tagged

A derivative IP can be repeatedly tagged via the dispute module as the *tagDerivativeIfParentInfringed* function does not check if there is already a dispute for **derivativeIpId**. As a result, the **successfulDisputesPerIp** counter for a given derivative IP can be indefinitely increased. To resolve the dispute(s) for this derivative IP, multiple calls to *resolveDispute()* are required, causing increased gas costs.

Similarly, this is also possible via the regular *raiseDispute()* function. However, raising such a dispute requires paying a bond. Thus, the incentive to spam with a large number of disputes is small.

Team response

Acknowledged. We think it is legitimate to be tagged for different infringements. For example, you may clear one tag by paying royalties back, but your IP might still be flagged for plagiarism, which is a different tag.

TRST-R-3 Default license terms are not validated

The *LicenseRegistry.setDefaultLicenseTerms()* function sets the default license template and terms, **defaultLicenseTemplate** and **defaultLicenseTermsId**. However, those values are not validated. For example, it is not checked whether the license template contains the terms id.

This could be checked by calling the license template's *ILicenseTemplate.exists()* function. Additionally, consider also checking if the default template is already registered in **registeredLicenseTemplates**.

Team response

Acknowledged. We will tackle it in a future release to improve license term validation.

TRST-R-4 Remove unused LicenseRegistry.setExpireTime() function

The *LicenseRegistry.setExpireTime()* function, only callable by the *LicensingModule* contract, is currently unused as *LicensingModule* does not provide a publicly accessible function to call it. Moreover, if *setExpireTime()* were to be made callable, it would allow retroactively modifying an IP's expiration time. This could lead to conflicts with the expiration times of any existing derivatives if the IP already has derivatives associated with it.

Team response

Fixed.

Mitigation review

LicenseRegistry.setExpireTime() has been removed.

TRST-R-5 Dispute tag IN_DISPUTE should not be allowed to add to the tag allow list

Adding a dispute tag to the **whitelistDisputeTag** allow list should not allow adding a tag with the same bytes as the special **IN_DISPUTE** tag. Otherwise, it would allow raising a dispute with the target tag set to **IN_DISPUTE**, subsequently setting the dispute's current tag to **IN_DISPUTE**. As a result, the *setDisputeJudgement()* function might get repeatedly called for this dispute, draining the arbitration policy contract funds (if coordinated with the arbitration relayer).

Team response

Fixed.

Mitigation review

The **IN_DISPUTE** tag is now prevented from being allowlisted by governance.

TRST-R-6 Consider adding a license token activation grace period

License terms might define an **expiration**. But this expiration applies to the derivative and starts when the derivative IP is registered.

But if the minted license token is not used for an extended period, the license token owner can always use it later to register a derivative. Consider adding an optional terms configuration value (e.g., **licenseTokenExpiration**) that expires the license token.

Team response

Acknowledged. It was a design decision to have the LNFT not expire for the moment. The recommendation makes sense, but we chose to simplify the legal concepts for the users to reduce friction, plus, ERC721 expiring is also a somewhat new concept. If there is demand, we may introduce this later on.

TRST-R-7 isAllProtocolPaused returns true even when there are no pausables configured

Currently, the *isAllProtocolPaused()* function in the *ProtocolPauseAdmin* contract returns **true** if no contracts have been added to **pausables**, misleadingly indicating that the protocol is paused.

We recommend returning **false** from *isAllProtocolPaused()* if there are no **pausables** (i.e., **pausables.length() == 0**) configured.

Team response

Fixed.

Mitigation review

isAllProtocolPaused() now returns **false** when no pausable contracts have been added.

TRST-R-8 Collected royalties of disputed IPs remain locked in the vault

If an IP is tagged, i.e., disputed, the already collected and unclaimed royalties are unrecoverable from the vault. We recommend adding governance functionality to recover said funds from the *IpRoyaltyVault* vault.

Team response

Acknowledged. Since we are still designing an appeal process (for example, if a dispute decision might be wrong, or the infringement might be fixable by the accused, maybe with lower penalty), we don't feel it's right at this moment in time for the protocol to withdraw those royalties. We consider this a Feature Request for next versions.

TRST-R-9 Add reentrancy protection as a safety precaution

Most functions already employ reentrancy protection by using OpenZeppelin's nonReentrant modifier. However, to mitigate the risk of potential reentrancy attacks we recommend adding reentrancy protection to other potentially vulnerable functions as well, such as:

- DisputeModule::resolveDispute ()
- IpRoyaltyVault::claimRevenueBySnapshotBatch ()

Team response

Fixed.

Mitigation review

nonReentrant modifier has been added to *resolveDispute()* and *claimRevenueBySnapshotBatch().*

Centralization risks

TRST-CR-1 Withdrawing funds from the ArbitrationPolicySP contract prevents dispute judgment

Governance can withdraw funds from the dispute arbitration *ArbitrationPolicySP* policy contract via the *ArbitrationPolicySP.governanceWithdraw()* function. However, if there are currently disputes pending judgment, judging a dispute with **decision = true** would error as there are insufficient funds to refund the dispute initiator.

Consider only withdrawing funds from settled disputes.

Team response

Fixed.

Mitigation review

governanceWithdraw() has been removed and replaced with transferring the funds directly to a treasury address whenever a dispute judgment tags an IP as infringing.