// Smart Contract Security Assessment 10.28.2024 - 12.04.2024

Proof of Creativity Protocol

Story

HALBORN

Proof of Creativity Protocol - Story

Prepared by: HALBORN

Last Updated 12/13/2024

Date of Engagement by: October 28th, 2024 - December 4th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
15	2	2	3	3	5

TABLE OF CONTENTS

1. Introduction

Story engaged Halborn to conduct a security assessment on their smart contracts beginning on October 28th and ending on December 4th, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to assess the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Story team. The main ones are the following:

- Pass the address of the user who intends to register the new Group IP account and transfer fees from this address.
- Enforce uniqueness within the ipIds array provided to the claimReward function.
- Implement strict access control in the mintLicenseTokens function to ensure that only the IP owner or authorized operators can mint license tokens for their IPs
- Introduce a short time delay during which the Group IP owner can still add new IPs without the risk of being front-run by an attacker registering a derivative.
- Add checks in the addIp function to ensure that the license terms attached to both the group and the IP Assets are not expired.
- Modify the tagDerivativeIfParentInfringed function by adding a call to _updateActiveArbitrationPolicy before fetching the arbitration policy.
- Enforce the maximum Royalty Tokens limit specified by the user during derivative registration.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local or public testnet deployment (`Foundry`, `Remix IDE`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: protocol-core-v1

(b) Assessed Commit ID: 5bcc4ae

(c) Items in scope:

- contracts/pause/ProtocolPausableUpgradeable.sol
- contracts/pause/ProtocolPauseAdmin.sol
- contracts/IPAccountImpl.sol
- contracts/registries/IPAccountRegistry.sol
- contracts/registries/LicenseRegistry.sol
- contracts/registries/IPAssetRegistry.sol
- contracts/registries/ModuleRegistry.sol
- contracts/registries/GroupIPAssetRegistry.sol
- contracts/access/AccessControlled.sol
- contracts/access/AccessController.sol
- contracts/access/IPGraphACL.sol
- contracts/IPAccountStorage.sol
- contracts/GroupNFT.sol
- contracts/lib/PILFlavors.sol
- contracts/lib/PILicenseTemplateErrors.sol
- contracts/lib/registries/IPAccountChecker.sol
- contracts/lib/ArrayUtils.sol
- contracts/lib/Errors.sol
- contracts/lib/ProtocolAdmin.sol
- contracts/lib/AccessPermission.sol
- contracts/lib/ExpiringOps.sol
- contracts/lib/ShortStringOps.sol
- contracts/lib/Licensing.sol
- contracts/lib/modules/Module.sol
- contracts/lib/MetaTx.sol
- contracts/lib/IPAccountStorageOps.sol
- contracts/modules/royalty/policies/LAP/RoyaltyPolicyLAP.sol
- contracts/modules/royalty/policies/VaultController.sol
- contracts/modules/royalty/policies/LRP/RoyaltyPolicyLRP.sol
- contracts/modules/royalty/policies/IpRoyaltyVault.sol
- contracts/modules/royalty/RoyaltyModule.sol
- contracts/modules/BaseModule.sol
- contracts/modules/grouping/EvenSplitGroupPool.sol
- contracts/modules/grouping/GroupingModule.sol

- contracts/modules/dispute/DisputeModule.sol
- contracts/modules/dispute/policies/UMA/ArbitrationPolicyUMA.sol
- contracts/modules/licensing/PILicenseTemplate.sol
- contracts/modules/licensing/BaseLicenseTemplateUpgradeable.sol
- contracts/modules/licensing/LicensingModule.sol
- contracts/modules/licensing/parameter-helpers/LicensorApprovalChecker.sol
- contracts/modules/licensing/PILTermsRenderer.sol
- contracts/modules/metadata/CoreMetadataModule.sol
- contracts/modules/metadata/CoreMetadataViewModule.sol
- contracts/LicenseToken.sol
- contracts/interfaces/pause/IProtocolPauseAdmin.sol
- contracts/interfaces/IIPAccountStorage.sol
- contracts/interfaces/registries/ILicenseRegistry.sol
- contracts/interfaces/registries/IIPAssetRegistry.sol
- contracts/interfaces/registries/IGroupIPAssetRegistry.sol
- contracts/interfaces/registries/IModuleRegistry.sol
- contracts/interfaces/registries/IIPAccountRegistry.sol
- contracts/interfaces/access/IAccessController.sol
- contracts/interfaces/IGroupNFT.sol
- contracts/interfaces/ILicenseToken.sol
- contracts/interfaces/modules/royalty/policies/IRoyaltyPolicy.sol
- contracts/interfaces/modules/royalty/policies/IExternalRoyaltyPolicy.sol
- contracts/interfaces/modules/royalty/policies/IGraphAwareRoyaltyPolicy.sol
- contracts/interfaces/modules/royalty/policies/IVaultController.sol
- contracts/interfaces/modules/royalty/policies/IIpRoyaltyVault.sol
- contracts/interfaces/modules/royalty/IRoyaltyModule.sol
- contracts/interfaces/modules/grouping/IGroupRewardPool.sol
- contracts/interfaces/modules/grouping/IGroupingModule.sol
- contracts/interfaces/modules/dispute/IDisputeModule.sol
- contracts/interfaces/modules/dispute/policies/IArbitrationPolicy.sol
- contracts/interfaces/modules/dispute/policies/UMA/IOOV3.sol
- contracts/interfaces/modules/dispute/policies/UMA/IOOV3Callbacks.sol
- contracts/interfaces/modules/dispute/policies/UMA/IArbitrationPolicyUMA.sol
- contracts/interfaces/modules/licensing/ILicensingModule.sol
- contracts/interfaces/modules/licensing/IPILicenseTemplate.sol
- contracts/interfaces/modules/licensing/ILicenseTemplate.sol
- contracts/interfaces/modules/licensing/ILicensingHook.sol
- contracts/interfaces/modules/metadata/ICoreMetadataModule.sol
- contracts/interfaces/modules/metadata/ICoreMetadataViewModule.sol
- contracts/interfaces/modules/base/IViewModule.sol
- contracts/interfaces/modules/base/IHookModule.sol
- contracts/interfaces/modules/base/IModule.sol
- contracts/interfaces/IIPAccount.sol
- storyprotocol/protocol-core-v1/pull/291

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: story-geth

(b) Assessed Commit ID: ab8925d

(c) Items in scope:

- core/vm/ipgraph.go
- crypto/secp256r1/publickey.go
- crypto/secp256r1/verifier.go

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- fd96a09
- 110175a
- ceda981
- 31a27ab
- https://github.com/storyprotocol/protocol-core-v1/pull/315
- https://github.com/storyprotocol/protocol-core-v1/pull/309
- https://github.com/storyprotocol/protocol-core-v1/pull/337
- https://github.com/storyprotocol/protocol-core-v1/pull/339
- https://github.com/storyprotocol/protocol-core-v1/pull/329
- https://github.com/storyprotocol/protocol-core-v1/pull/326
- https://github.com/storyprotocol/protocol-core-v1/pull/327
- https://github.com/storyprotocol/protocol-core-v1/pull/333

https://github.com/storyprotocol/protocol-core-v1/pull/338

- https://github.com/storyprotocol/protocol-core-v1/pull/330

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	2	3	3	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DENIAL OF SERVICE CAUSED BY IMPROPER REGISTRATION FEE HANDLING IN GROUP IP ACCOUNT	CRITICAL	SOLVED - 11/15/2024
MALICIOUS IP OWNER CAN DRAIN REWARDS VIA DUPLICATE ENTRIES IN CLAIMREWARD FUNCTION	CRITICAL	SOLVED - 11/12/2024
GRIEFING ATTACK IN GROUP IP MANAGEMENT VIA LICENSE TOKEN MINTING	HIGH	SOLVED - 11/16/2024
FRONT-RUNNING EXPLOIT DISRUPTS GROUP IP MANAGEMENT VIA DERIVATIVE REGISTRATION	HIGH	SOLVED - 11/16/2024
NO EXPIRATION CHECK ENABLES UNAUTHORIZED ACCESS TO REWARD POOLS	MEDIUM	SOLVED - 11/14/2024
OUTDATED ARBITRATION POLICY USED WHEN TAGGING DERIVATIVE IP ASSETS	MEDIUM	SOLVED - 11/26/2024
MALICIOUS IP OWNER CAN FRONT-RUN DERIVATIVE REGISTRATION TO DRAIN USER'S ROYALTY TOKENS	MEDIUM	SOLVED - 11/20/2024
UNINITIALIZED DYNAMIC ARRAY IN GETGROUPMEMBERS FUNCTION LEADS TO RUNTIME ERROR	LOW	SOLVED - 12/11/2024
USAGE OF DIRECT APPROVE CALLS LEADS TO SEVERAL ISSUES	LOW	SOLVED - 12/11/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF PROPER NORMALIZATION IN SECP256R1 CURVE VERIFIER LEADS TO SIGNATURE MALLEABILITY	LOW	RISK ACCEPTED - 12/12/2024
USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS	INFORMATIONAL	SOLVED - 12/03/2024
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	PARTIALLY SOLVED - 12/03/2024
INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS	INFORMATIONAL	SOLVED - 12/03/2024
POTENTIAL LICENSE INCOMPATIBILITIES	INFORMATIONAL	SOLVED - 12/11/2024
UNUSED CUSTOM ERRORS	INFORMATIONAL	SOLVED - 12/04/2024

7. FINDINGS & TECH DETAILS

7.1 DENIAL OF SERVICE CAUSED BY IMPROPER REGISTRATION FEE HANDLING IN GROUP IP ACCOUNT

// CRITICAL

Description

The `registerGroup` function defined in the **GroupingModule** is an entry point for all the flows related to the Group IP. It allows users to register a new Group IP account. It calls the `registerGroup` function on the **GroupIPAssetRegistry** contract for the actual accounting process:

```
116 function registerGroup(address groupPool) external nonReentrant whenNotPa
117     // mint Group NFT
118     uint256 groupNftId = GROUP_NFT.mintGroupNft(msg.sender, msg.sender);
119     // register Group NFT
120     groupId = GROUP_IP_ASSET_REGISTRY.registerGroup(address(GROUP_NFT), g
121     emit IPGroupRegistered(groupId, groupPool);
122 }
```

This function internally calls the `_register` function, which is defined in the **IPAssetRegistry**, from which the **GroupIPAssetRegistry** abstract contract inherits:

```
53  function registerGroup(
54      address groupNft,
55      uint256 groupNftId,
56      address rewardPool
57  ) external onlyGroupingModule whenNotPaused returns (address groupId) {
58      groupId = _register({ chainid: block.chainid, tokenContract: groupNft
59
60      IIPAccount(payable(groupId)).setBool("GROUP_IPA", true);
61      GroupIPAssetRegistryStorage storage $ = _getGroupIPAssetRegistryStora
62      if (!$.whitelistedGroupRewardPools[rewardPool]) {
63          revert Errors.GroupIPAssetRegistry__GroupRewardPoolNotRegistered(
64      }
65      _getGroupIPAssetRegistryStorage().rewardPools[groupId] = rewardPool;
66  }
```

The `_register` function checks if the registration fee was previously set by the owner and transfers the fixed amount from the `msg.sender`, which in this case is the **GroupingModule** contract:

```
94  function _register(uint256 chainid, address tokenContract, uint256 tokenI
95      IPAssetRegistryStorage storage $ = _getIPAssetRegistryStorage();
96
97      // Pay registration fee
98      uint96 feeAmount = $.feeAmount;
99      if (feeAmount > 0) {
100         address feeToken = $.feeToken;
101         address treasury = $.treasury;
102         IERC20(feeToken).safeTransferFrom(msg.sender, treasury, uint256(f
103         emit IPRegistrationFeePaid(msg.sender, treasury, feeToken, feeAmo
104     }
105
106     id = _registerIpAccount(chainid, tokenContract, tokenId);
107     IIPAccount ipAccount = IIPAccount(payable(id));
108
109     if (bytes(ipAccount.getString("NAME")).length != 0) {
110         revert Errors.IPAssetRegistry__AlreadyRegistered();
111     }
112
113     (string memory name, string memory uri) = _getNameAndUri(chainid, tok
114     uint256 registrationDate = block.timestamp;
115     ipAccount.setString("NAME", name);
116     ipAccount.setString("URI", uri);
117     ipAccount.setUint256("REGISTRATION_DATE", registrationDate);
118
119     $.totalSupply++;
120
121     emit IPRegistered(id, chainid, tokenContract, tokenId, name, uri, reg
122 }
```

The intended behavior is to transfer the fee from the user who is registering the Group IP account. However, as it does not grant the required approval, the entire process will revert, which breaks the core feature and causes a denial of service.

Proof of Concept

The core functionalities are broken due to improper registration fee handling:

```
function test_groupingFee() public {
    //Admin set the registration fee
    vm.prank(u.admin);
    address treasury = address(0x123);
    ipAssetRegistry.setRegistrationFee(treasury, address(erc20), 1000);
```

```
    //User grants approval required for fee transfer
    vm.startPrank(alice);
        erc20.approve(address(ipAssetRegistry), type(uint256).max);

        //Transaction reverts due to improper approval handling
        vm.expectRevert();
        address groupId = groupingModule.registerGroup(address(rewardPool));
}
```

The result is the following:

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_groupingFee() (gas: 217012)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 16.53ms (907.42µs CPU time)

Ran 1 test suite in 151.18ms (16.53ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:C (10.0)

Recommendation

It is recommended to pass the address of the user who intends to register the new Group IP account and transfer fees from this address.

Remediation

SOLVED: The **Story team** solved the issue in the specified commit id. The address of the account that pays the registration fee is passed across the entire flow.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/commit/fd96a09d0ac94b490f66ab011c96f4f048578945

References

storyprotocol/protocol-core-v1/contracts/modules/grouping/GroupingModule.sol#L117-L123
storyprotocol/protocol-core-v1/contracts/registries/GroupIPAssetRegistry.sol#L54-L73
storyprotocol/protocol-core-v1/contracts/registries/IPAssetRegistry.sol#L98-L132

7.2 MALICIOUS IP OWNER CAN DRAIN REWARDS VIA DUPLICATE ENTRIES IN CLAIMREWARD FUNCTION

// CRITICAL

Description

The **RoyaltyModule** is designed to manage the flow of revenue between parent and child IP Assets. Revenue can flow through the system when licenses are minted or when tips are sent directly to an IPA. In such cases, the revenue is intended to be distributed fairly among the IPAs within a group, ensuring that each member receives their rightful share.

However, a vulnerability exists in the reward distribution logic that allows a malicious IP owner to manipulate the system and redirect all rewards to a single IP account within a group.

When a user calls the `claimReward` function defined in the **GroupingModule** contract, they provide an array of `ipIds` representing the IP Assets that should receive rewards.

```
194 function claimReward(address groupId, address token, address[] calldata i
195     IGroupRewardPool pool = IGroupRewardPool(GROUP_IP_ASSET_REGISTRY.getG
196     // Trigger group pool to distribute rewards to group members' vaults
197     uint256[] memory rewards = pool.distributeRewards(groupId, token, ipI
198     emit ClaimedReward(groupId, token, ipIds, rewards);
199 }
```

The `distributeRewards` function, from **EvenSplitGroupPool** contract, then calculates the available rewards for each IP Asset by calling the internal `_getAvailableReward`.

```
123 function distributeRewards(
124     address groupId,
125     address token,
126     address[] calldata ipIds
127 ) external whenNotPaused onlyGroupingModule returns (uint256[] memory rew
128     rewards = _getAvailableReward(groupId, token, ipIds);
129     uint256 totalRewards = 0;
130     for (uint256 i = 0; i < ipIds.length; i++) {
131         totalRewards += rewards[i];
132     }
133     if (totalRewards == 0) return rewards;
134     IERC20(token).approve(address(ROYALTY_MODULE), totalRewards);
135     EvenSplitGroupPoolStorage storage $ = _getEvenSplitGroupPoolStorage()
136     for (uint256 i = 0; i < ipIds.length; i++) {
```

```
            if (rewards[i] == 0) continue;
            // Calculate pending reward for each IP
            $.ipRewardDebt[groupId][token][ipIds[i]] += rewards[i];
            // Call royalty module to transfer reward to IP's vault as royalt
            ROYALTY_MODULE.payRoyaltyOnBehalf(ipIds[i], groupId, token, rewar
        }
    }
```

In the `_getAvailableReward` function, the total accumulated pool balance is divided equally among all IP Assets in the group (`totalAccumulatedPoolBalance / totalIps`) to determine the rewardPerIP. The function then calculates the individual reward for each IP Asset by subtracting any previously recorded reward debt. However, the function does not enforce uniqueness within the ipIds array, allowing the same IP Asset to appear multiple times, before the reward debt is updated.

```
function _getAvailableReward(
    address groupId,
    address token,
    address[] memory ipIds
) internal view returns (uint256[] memory) {
    EvenSplitGroupPoolStorage storage $ = _getEvenSplitGroupPoolStorage()
    uint256 totalIps = $.totalMemberIps[groupId];
    if (totalIps == 0) return new uint256[](ipIds.length);

    uint256 totalAccumulatedPoolBalance = $.poolBalance[groupId][token];
    uint256[] memory rewards = new uint256[](ipIds.length);
    for (uint256 i = 0; i < ipIds.length; i++) {
        // Ignore if IP is not added to pool
        if (!_isIpAdded(groupId, ipIds[i])) {
            rewards[i] = 0;
            continue;
        }
        uint256 rewardPerIP = totalAccumulatedPoolBalance / totalIps;
        rewards[i] = rewardPerIP - $.ipRewardDebt[groupId][token][ipIds[i
    }
    return rewards;
}
```

Because of this oversight, a malicious user can pass the same `ipId` multiple times in the `ipIds` array when calling `claimReward`. Each occurrence of the `ipId` in the array results in the `rewardPerIP` being added to the IP Asset's reward debt multiple times. Consequently, the IP Asset accumulates a disproportionately large share of the total rewards, effectively taking the entire reward pool designated to be shared among other users.

Proof of Concept

The following test demonstrates how a malicious user can exploit this vulnerability:

```
function test_GroupingModule_duplicatedIPs() public {
    vm.warp(100);

    //Alice registers a new group IP
    vm.prank(alice);
    address groupId = groupingModule.registerGroup(address(rewardPool));

    uint256 termsId = pilTemplate.registerLicenseTerms(
        PILFlavors.commercialRemix({
            mintingFee: 0,
            commercialRevShare: 10_000_000,
            currencyToken: address(erc20),
            royaltyPolicy: address(royaltyPolicyLAP)
        })
    );

    //IP Owner 1 attaches license terms and mints license tokens
    vm.prank(ipOwner1);
    licensingModule.attachLicenseTerms(ipId1, address(pilTemplate), termsId);
    licensingModule.mintLicenseTokens(ipId1, address(pilTemplate), termsId, 1,

    //IP Owner 2 attaches license terms and mints license tokens
    vm.prank(ipOwner2);
    licensingModule.attachLicenseTerms(ipId2, address(pilTemplate), termsId);
    licensingModule.mintLicenseTokens(ipId2, address(pilTemplate), termsId, 1,

    //IP Owner 3 attaches license terms and mints license tokens
    vm.prank(ipOwner3);
    licensingModule.attachLicenseTerms(ipId3, address(pilTemplate), termsId);
    licensingModule.mintLicenseTokens(ipId3, address(pilTemplate), termsId, 1,

    //Alice attaches license terms to the group IP
    vm.startPrank(alice);
    licensingModule.attachLicenseTerms(groupId, address(pilTemplate), termsId)
    vm.stopPrank();

    //Alice adds IPs to the group
    vm.startPrank(alice);
        address[] memory ipIds = new address[](3);
        ipIds[0] = ipId1;
```

```
    ipIds[1] = ipId2;
    ipIds[2] = ipId3;
    groupingModule.addIp(groupId, ipIds);
    assertEq(ipAssetRegistry.totalMembers(groupId), 3);
    assertEq(rewardPool.getTotalIps(groupId), 3);
vm.stopPrank();

//Register a derivative IP asset
address[] memory parentIpIds = new address[](1);
parentIpIds[0] = groupId;
uint256[] memory licenseTermsIds = new uint256[](1);
licenseTermsIds[0] = termsId;

vm.prank(ipOwner4);
licensingModule.registerDerivative(ipId4, parentIpIds, licenseTermsIds, ac

//IP Owner 4 pays royalties that should be distributed amongs IP owners
erc20.mint(ipOwner4, 150 ether);
vm.startPrank(ipOwner4);
    erc20.approve(address(royaltyModule), 150 ether);
    royaltyModule.payRoyaltyOnBehalf(ipId4, ipOwner4, address(erc20), 150
vm.stopPrank();

//Transfer a portion of the royalties to the group's vault
royaltyPolicyLAP.transferToVault(ipId4, groupId, address(erc20), 15 ether)
vm.warp(vm.getBlockTimestamp() + 7 days);

//Take a snapshot of the group's royalty vault
uint256 snapshotId = IIpRoyaltyVault(royaltyModule.ipRoyaltyVaults(groupId
uint256[] memory snapshotIds = new uint256[](1);
snapshotIds[0] = snapshotId;

//Collect royalties for the group
groupingModule.collectRoyalties(groupId, address(erc20), snapshotIds);

//The same ipId1 is included three times
address[] memory claimIpIds = new address[](3);
claimIpIds[0] = ipId1;
claimIpIds[1] = ipId1;
claimIpIds[2] = ipId1;

//Owner of IP1 claims rewards using the manipulated ipIds array
vm.prank(ipOwner1);
    groupingModule.claimReward(groupId, address(erc20), claimIpIds);
```

```
    //Assert that ipId1 received the entire reward pool
    assertApproxEqAbs(erc20.balanceOf(royaltyModule.ipRoyaltyVaults(ipId1)), 1
```

The result of the test is the following:

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_GroupingModule_duplicatedIPs() (gas: 6390229)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.51ms (5.19ms CPU time)

Ran 1 test suite in 142.10ms (18.51ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:H (9.4)

Recommendation

It is recommended to enforce uniqueness within the `ipIds` array provided to the `claimReward` function. This can be achieved by implementing a check that ensures each IP Asset appears only once in the array before proceeding with the reward distribution.

Remediation

SOLVED: The **Story team** solved the issue in the specified commit id. The vulnerability was fixed by updating each IP Asset's reward debt immediately after calculating its reward within the loop, ensuring that if the same `ipId` appears multiple times in the `ipIds` array, subsequent calculations account for the updated debt and prevent extra rewards from being allocated to duplicates.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/commit/1101175ae7e7f078023d9644bcd96f1f8b1498794

References

storyprotocol/protocol-core-v1/contracts/modules/grouping/EvenSplitGroupPool.sol#L123-L143
storyprotocol/protocol-core-v1/contracts/modules/grouping/EvenSplitGroupPool.sol#L165-L186

7.3 GRIEFING ATTACK IN GROUP IP MANAGEMENT VIA LICENSE TOKEN MINTING

// HIGH

Description

The `registerGroup` function in the `GroupingModule` allows users to register a new Group IP account. It internally calls the `registerGroup` function on the `GroupIPAssetRegistry` contract for the actual accounting process:

```
116 function registerGroup(address groupPool) external nonReentrant whenNotPa
117         // mint Group NFT
118         uint256 groupNftId = GROUP_NFT.mintGroupNft(msg.sender, msg.sende
119         // register Group NFT
120         groupId = GROUP_IP_ASSET_REGISTRY.registerGroup(address(GROUP_NFT
121         emit IPGroupRegistered(groupId, groupPool);
122 }
```

After registering the Group IP, the group owner is intended to attache license terms to it using the `attachLicenseTerms` function in the `LicensingModule`:

```
130 function attachLicenseTerms(
131     address ipId,
132     address licenseTemplate,
133     uint256 licenseTermsId
134 ) external verifyPermission(ipId) {
135     _verifyIpNotDisputed(ipId);
136     LICENSE_REGISTRY.attachLicenseTermsToIp(ipId, licenseTemplate, licens
137     emit LicenseTermsAttached(msg.sender, ipId, licenseTemplate, licenseT
138 }
```

The owner then can attempt to add individual IPs to their group using the `addIp` function in the `GroupingModule`. Before adding IPs, the `_checkIfGroupMembersLocked` function checks if the group is locked due to previous actions, such as having derivative IPs or minted license tokens. The `getTotalTokensByLicensor` function in the `LICENSE_TOKEN` contract returns the total number of license tokens minted under the specified IP:

```
238 function _checkIfGroupMembersLocked(address groupIpId) internal view {
239     if (LICENSE_REGISTRY.hasDerivativeIps(groupIpId)) {
240         revert Errors.GroupingModule__GroupFrozenDueToHasDerivativeIps(gr
241
```

```
        }
        if (LICENSE_TOKEN.getTotalTokensByLicensor(groupIpId) > 0) {
            revert Errors.GroupingModule__GroupFrozenDueToAlreadyMintLicenseT
        }
    }
```

Because there is no permission check to verify that the caller is authorized to mint license tokens for `licensorIpId`, any user can front-run the owner transaction, invoking this function. An attacker can exploit the lack of access control in the `mintLicenseTokens` function to mint license tokens for Alice's Group IP without authorization, as presented in the Proof of Concept.

An attacker can prevent the legitimate owner from managing their Group IP by unauthorized minting of license tokens, causing a denial-of-service condition. This disrupts the core functionality of the Group IP management system and can lead to operational and financial losses for the rightful owner.

Note that the similar scenario applies also to removing IP from the Group IP by using the `removeIp` function, which also uses the internal `_checkIfGroupMembersLocked`.

Proof of Concept

When Alice calls `addIp` function, the `_checkIfGroupMembersLocked` function checks if any license tokens have been minted under `groupId`. Since the attacker has front-run Alice's transaction and minted a license token, `getTotalTokensByLicensor(groupId)` returns a value greater than zero, causing the function to revert and preventing Alice from adding IPs to her group:

```
    function test_GroupingModule_revertOnFrontRunning() public {
        vm.warp(100);

        //Alice registers a new group IP
        vm.prank(alice);
        address groupId = groupingModule.registerGroup(address(rewardPool));

        uint256 termsId = pilTemplate.registerLicenseTerms(
            PILFlavors.commercialRemix({
                mintingFee: 0,
                commercialRevShare: 10_000_000,
                currencyToken: address(erc20),
                royaltyPolicy: address(royaltyPolicyLAP)
            })
        );

        vm.prank(ipOwner1);
            licensingModule.attachLicenseTerms(ipId1, address(pilTemplate), te
```

```
            licensingModule.mintLicenseTokens(ipId1, address(pilTemplate), ter
        vm.prank(ipOwner2);
            licensingModule.attachLicenseTerms(ipId2, address(pilTemplate), te
            licensingModule.mintLicenseTokens(ipId2, address(pilTemplate), ter

        //Alice attaches the license term
        vm.prank(alice);
            licensingModule.attachLicenseTerms(groupId, address(pilTemplate),

        // Attacker front-runs Alice's transaction, minting a new license toke
        vm.prank(attacker);
            licensingModule.mintLicenseTokens(groupId, address(pilTemplate), t

        // Alice tries to add the intended IPs to the group IP
        vm.startPrank(alice);
            address[] memory ipIds = new address[](2);
            ipIds[0] = ipId1;
            ipIds[1] = ipId2;

            //Transaction reverts
            vm.expectRevert();
            groupingModule.addIp(groupId, ipIds);
        vm.stopPrank();
    }
```

The result of the test is the following:

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_GroupingModule_revertOnFrontRunning() (gas: 3608860)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 15.28ms (3.11ms CPU time)

Ran 1 test suite in 146.22ms (15.28ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation

It is recommended to implement strict access control in the `mintLicenseTokens` function to ensure that only the IP owner or authorized operators can mint license tokens for their IPs. This can be achieved by adding a `verifyPermission` modifier to the function.

Remediation

SOLVED: The **Story team** solved the issue in the specified commit id. The `disabled` field has been added to the `LicensingConfig` struct. When set to `true`, both the `mintLicenseTokens` function and

`_executeLicensingHookAndPayMintingFee` (executed during the derivative registration process) will revert. This effectively provides the Group IP owner with the ability to manage their own group and revoke the `disabled` status when it is ready to operate.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/commit/ceda981b48e9cbf4ce45aa7fa747cb6360396a5c

References

storyprotocol/protocol-core-v1/contracts/modules/licensing/LicensingModule.sol#L160-L236
storyprotocol/protocol-core-v1/contracts/modules/grouping/GroupingModule.sol#L250-L257

7.4 FRONT-RUNNING EXPLOIT DISRUPTS GROUP IP MANAGEMENT VIA DERIVATIVE REGISTRATION

// HIGH

Description

The `registerGroup` function in the **GroupingModule** allows users to register a new Group IP account. After registering the Group IP, the group owner is intended to attach license terms to it using the `attachLicenseTerms` function in the `LicensingModule`. The owner then can attempt to add individual IPs to their group using the `addIp` function in the `GroupingModule`. Before adding IPs, the `_checkIfGroupMembersLocked` function checks if the group is locked due to previous actions, such as having derivative IPs or minted license tokens:

```
238 function _checkIfGroupMembersLocked(address groupIpId) internal view {
239     if (LICENSE_REGISTRY.hasDerivativeIps(groupIpId)) {
240         revert Errors.GroupingModule__GroupFrozenDueToHasDerivativeIps(gr
241     }
242     if (LICENSE_TOKEN.getTotalTokensByLicensor(groupIpId) > 0) {
243         revert Errors.GroupingModule__GroupFrozenDueToAlreadyMintLicenseT
244     }
245 }
```

The `hasDerivativeIps` function in the `LICENSE_REGISTRY` checks if there are any derivative IPs registered under the specified IP.

```
315 function hasDerivativeIps(address parentIpId) external view returns (bool
316     return _getLicenseRegistryStorage().childIps[parentIpId].length() > 0
317 }
```

However, any user can front-run the owner's transaction intended to add the specified IP assets to the group IP, by invoking the `registerDerivative` function. By registering an unauthorized derivative IP, the attacker effectively locks the Group IP, preventing the legitimate owner from managing it. This creates a denial-of-service condition, disrupting the core functionality of the Group IP management system.

Note that the similar scenario applies also to removing IP from the Group IP by using the `removeIp` function, which also uses the internal `_checkIfGroupMembersLocked`.

Proof of Concept

When Alice calls the `addIp` function, the `_checkIfGroupMembersLocked` function checks if any derivative IPs have been registered under `groupId`. Since the attacker has front-run Alice's transaction and registered

a derivative IP, `hasDerivativeIps(groupId)` returns `true`, causing the function to revert and preventing Alice from adding IPs to her group:

```
function test_GroupingModule_revertOnFrontRunning_registerDerivative() public
    //Alice registers a new group IP
    vm.prank(alice);
        address groupId = groupingModule.registerGroup(address(rewardPool));

    uint256 termsId = pilTemplate.registerLicenseTerms(
        PILFlavors.commercialRemix({
            mintingFee: 0,
            commercialRevShare: 10_000_000,
            currencyToken: address(erc20),
            royaltyPolicy: address(royaltyPolicyLAP)
        })
    );

    vm.prank(ipOwner1);
        licensingModule.attachLicenseTerms(ipId1, address(pilTemplate), termsI
        licensingModule.mintLicenseTokens(ipId1, address(pilTemplate), termsId
    vm.prank(ipOwner2);
        licensingModule.attachLicenseTerms(ipId2, address(pilTemplate), termsI
        licensingModule.mintLicenseTokens(ipId2, address(pilTemplate), termsId

    //Alice attaches the license term
    vm.prank(alice);
        licensingModule.attachLicenseTerms(groupId, address(pilTemplate), term

    //Attacker front-runs Alice's transaction, registering a new derivative fo
    vm.startPrank(ipOwner3);
        address[] memory parentIpIds = new address[](1);
        uint256[] memory licenseTermsIds = new uint256[](1);
        parentIpIds[0] = groupId;
        licenseTermsIds[0] = termsId;
        licensingModule.registerDerivative(ipId3, parentIpIds, licenseTermsIds
    vm.stopPrank();

    // Alice tries to add the intended IPs to the group IP
    vm.startPrank(alice);
        address[] memory ipIds = new address[](2);
        ipIds[0] = ipId1;
        ipIds[1] = ipId2;

        //Transaction reverts
```

```
        vm.expectRevert();
        groupingModule.addIp(groupId, ipIds);
    vm.stopPrank();
}
```

The result of the test is the following:

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_GroupingModule_revertOnFrontRunning_registerDerivative() (gas: 4401803)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 15.05ms (3.51ms CPU time)

Ran 1 test suite in 140.97ms (15.05ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (7.5)

Recommendation

It is recommended to consider introducing a short time delay during which the Group IP owner can still add new IPs without the risk of being front-run by an attacker registering a derivative. This grace period would help mitigate the front-running vector, allowing the owner to add the intended IP assets even if the derivative was registered beforehand.

Remediation

SOLVED: The **Story team** solved the issue in the specified commit id. The `disabled` field has been added to the `LicensingConfig` struct. When set to `true`, both the `mintLicenseTokens` function and `_executeLicensingHookAndPayMintingFee` (executed during the derivative registration process) will revert. This effectively provides the Group IP owner with the ability to manage their own group and revoke the `disabled` status when it is ready to operate.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/commit/ceda981b48e9cbf4ce45aa7fa747cb6360396a5c

References

storyprotocol/protocol-core-v1/contracts/modules/grouping/GroupingModule.sol#L250-L257

7.5 NO EXPIRATION CHECK ENABLES UNAUTHORIZED ACCESS TO REWARD POOLS

// MEDIUM

Description

The `addIp` function, defined in the **GroupingModule** contract, is responsible for adding IP Assets to a group. It performs several checks to ensure that both the group and the IP Assets meet specific criteria before the addition is allowed:

```
139 function addIp(address groupIpId, address[] calldata ipIds) external when
140     _checkIfGroupMembersLocked(groupIpId);
141     // the group IP must has license terms and minting fee is 0 to be abl
142     if (LICENSE_REGISTRY.getAttachedLicenseTermsCount(groupIpId) == 0) {
143     revert Errors.GroupingModule__GroupIPHasNoLicenseTerms(groupIpId);
144     }
145     (address groupLicenseTemplate, uint256 groupLicenseTermsId) = LICENSE
146         groupIpId,
147         0
148     );
149     PILTerms memory groupLicenseTerms = IPILicenseTemplate(groupLicenseTe
150         groupLicenseTermsId
151     );
152     if (groupLicenseTerms.defaultMintingFee != 0) {
153         revert Errors.GroupingModule__GroupIPHasMintingFee(groupIpId, gro
154     }
155
156     GROUP_IP_ASSET_REGISTRY.addGroupMember(groupIpId, ipIds);
157     IGroupRewardPool pool = IGroupRewardPool(GROUP_IP_ASSET_REGISTRY.getG
158     for (uint256 i = 0; i < ipIds.length; i++) {
159         if (GROUP_IP_ASSET_REGISTRY.isRegisteredGroup(ipIds[i])) {
160             revert Errors.GroupingModule__CannotAddGroupToGroup(groupIpId
161         }
162         // check if the IP has the same license terms as the group
163         if (!LICENSE_REGISTRY.hasIpAttachedLicenseTerms(ipIds[i], groupLi
164             revert Errors.GroupingModule__IpHasNoGroupLicenseTerms(
165                 ipIds[i],
166                 groupLicenseTemplate,
167                 groupLicenseTermsId
168             );
169         }
170         pool.addIp(groupIpId, ipIds[i]);
171
```

```
        }

        emit AddedIpToGroup(groupIpId, ipIds);
    }
```

However, it does not check whether the license terms attached to the IP Assets or the group have expired. As a result, IP Assets with expired license terms can be added to a group, potentially bypassing licensing rules and unfairly participate in the reward distribution.

Proof of Concept

The following test demonstrates how an IP Asset with expired license terms can be added to a group:

```
function test_registerExpired() public {
    vm.warp(block.timestamp + 10 days);

    //Alice registers a new group IP
    vm.prank(alice);
        address groupId = groupingModule.registerGroup(address(rewardPool));

    //Create outdated term
    PILTerms memory terms = PILFlavors.commercialRemix({
        mintingFee: 0,
        commercialRevShare: 10,
        currencyToken: address(erc20),
        royaltyPolicy: address(royaltyPolicyLAP)
    });
    //Set expiration to 5 days ago
    terms.expiration = block.timestamp - 5 days;

    //Register the expired license terms
    uint256 termsId = pilTemplate.registerLicenseTerms(terms);

    // IP Owner 1 has attached the expired license terms to their IP Asset
    vm.prank(ipOwner1);
        licensingModule.attachLicenseTerms(ipId1, address(pilTemplate), termsI
        licensingModule.mintLicenseTokens(ipId1, address(pilTemplate), termsId

    //Alice attaches the same expired license terms to the group IP
    vm.startPrank(alice);
        licensingModule.attachLicenseTerms(groupId, address(pilTemplate), term
        vm.stopPrank();
```

```
    //Alice adds the IP Asset with expired license terms to the group
    vm.startPrank(alice);
        address[] memory ipIds = new address[](1);
        ipIds[0] = ipId1;
        groupingModule.addIp(groupId, ipIds);
    vm.stopPrank();
}
```

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_registerExpired() (gas: 2233963)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 14.33ms (2.39ms CPU time)

Ran 1 test suite in 140.20ms (14.33ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:U (5.0)

Recommendation

It is recommended to add checks in the `addIp` function to ensure that the license terms attached to both the group and the IP Assets are not expired.

Remediation

SOLVED: The **Story team** solved the issue in the specified commit id. The check to ensure whether the license term has expired was added.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/commit/31a27ab9583e0bbcf8e9e3ead9160680cd34fcf8

References

storyprotocol/protocol-core-v1/contracts/modules/grouping/GroupingModule.sol#L139-L174

7.6 OUTDATED ARBITRATION POLICY USED WHEN TAGGING DERIVATIVE IP ASSETS

// MEDIUM

Description

The purpose of the **DisputeModule** contract is to allow users to raise and resolve disputes related to IP assets. Disputes are handled through arbitration policies, which define the rules and procedures for dispute resolution. The arbitration policies can be set by IP owners and are subject to a cooldown period before becoming active. The IP owners can set a new arbitration policy for their IP assets by calling the `setArbitrationPolicy` function.

```
177 function setArbitrationPolicy(address ipId, address nextArbitrationPolicy
178     DisputeModuleStorage storage $ = _getDisputeModuleStorage();
179     if (!$.isWhitelistedArbitrationPolicy[nextArbitrationPolicy])
180         revert Errors.DisputeModule__NotWhitelistedArbitrationPolicy();
181
182     $.nextArbitrationPolicies[ipId] = nextArbitrationPolicy;
183
184     uint256 nextArbitrationUpdateTimestamp = block.timestamp + $.arbitrat
185     $.nextArbitrationUpdateTimestamps[ipId] = nextArbitrationUpdateTimest
186
187     emit ArbitrationPolicySet(ipId, nextArbitrationPolicy, nextArbitratio
188 }
```

Users can raise disputes against an IP asset by calling `raiseDispute` function. This function updates the active arbitration policy for the IP asset before proceeding. It checks if the cooldown period has passed and updates the active arbitration policy accordingly. Then the dispute is processed according to the active arbitration policy.

```
196 function raiseDispute(
197     address targetIpId,
198     bytes32 disputeEvidenceHash,
199     bytes32 targetTag,
200     bytes calldata data
201 ) external nonReentrant whenNotPaused returns (uint256) {
202     if (!IP_ASSET_REGISTRY.isRegistered(targetIpId)) revert Errors.Disput
203     DisputeModuleStorage storage $ = _getDisputeModuleStorage();
204     if (!$.isWhitelistedDisputeTag[targetTag]) revert Errors.DisputeModul
205     if (disputeEvidenceHash == bytes32(0)) revert Errors.DisputeModule__Z
206
207
```

```
    address arbitrationPolicy = _updateActiveArbitrationPolicy(targetIpId
    uint256 disputeId = ++$.disputeCounter;
    uint256 disputeTimestamp = block.timestamp;

    $.disputes[disputeId] = Dispute({
        targetIpId: targetIpId,
        disputeInitiator: msg.sender,
        disputeTimestamp: disputeTimestamp,
        arbitrationPolicy: arbitrationPolicy,
        disputeEvidenceHash: disputeEvidenceHash,
        targetTag: targetTag,
        currentTag: IN_DISPUTE,
        parentDisputeId: 0
    });

    IArbitrationPolicy(arbitrationPolicy).onRaiseDispute(msg.sender, data

    emit DisputeRaised(
        disputeId,
        targetIpId,
        msg.sender,
        disputeTimestamp,
        arbitrationPolicy,
        disputeEvidenceHash,
        targetTag,
        data
    );

    return disputeId;
}
```

Also, if a parent IP asset has been tagged with an infringement (e.g., plagiarism), derivative IP assets can be automatically tagged by calling the `tagDerivativeIfParentInfringed` function. It tags the derivative IP with the same infringement tag as the parent.

```
function tagDerivativeIfParentInfringed(
    address parentIpId,
    address derivativeIpId,
    uint256 parentDisputeId
) external whenNotPaused {
    DisputeModuleStorage storage $ = _getDisputeModuleStorage();

    Dispute memory parentDispute = $.disputes[parentDisputeId];
```

```
297     if (parentDispute.targetIpId != parentIpId) revert Errors.DisputeModu
298
299     // a dispute current tag prior to being resolved can be in 3 states -
300     // by restricting IN_DISPUTE and 0 - it is ensire the parent has been
301     if (parentDispute.currentTag == IN_DISPUTE || parentDispute.currentTa
302         revert Errors.DisputeModule__ParentNotTagged();
303
304     if (!LICENSE_REGISTRY.isParentIp(parentIpId, derivativeIpId)) revert
305
306     address arbitrationPolicy = $.arbitrationPolicies[derivativeIpId];
307     if (!$.isWhitelistedArbitrationPolicy[arbitrationPolicy]) arbitration
308
309     uint256 disputeId = ++$.disputeCounter;
310     uint256 disputeTimestamp = block.timestamp;
311
312     $.disputes[disputeId] = Dispute({
313         targetIpId: derivativeIpId,
314         disputeInitiator: msg.sender,
315         disputeTimestamp: disputeTimestamp,
316         arbitrationPolicy: arbitrationPolicy,
317         disputeEvidenceHash: "",
318         targetTag: parentDispute.currentTag,
319         currentTag: parentDispute.currentTag,
320         parentDisputeId: parentDisputeId
321     });
322
323     $.successfulDisputesPerIp[derivativeIpId]++;
324
325     emit DerivativeTaggedOnParentInfringement(
326         parentIpId,
327         derivativeIpId,
328         parentDisputeId,
329         parentDispute.currentTag,
330         disputeTimestamp
331     );
332 }
```

However, in the current implementation of `tagDerivativeIfParentInfringed`, the function fetches the arbitration policy for the `derivativeIpId` directly from the storage (`$.arbitrationPolicies[derivativeIpId]`) without updating it beforehand. The function does not call `_updateActiveArbitrationPolicy`, so it does not check if the cooldown period has passed and whether the arbitration policy should be updated. As a result, if the arbitration policy was changed and the cooldown period has passed, the `tagDerivativeIfParentInfringed` will use an outdated arbitration policy, which is

not intended by the IP owner. This will lead to disputes being handled incorrectly, violating the expectations of the IP owner.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

It is recommended to modify the `tagDerivativeIfParentInfringed` function by adding a call to `_updateActiveArbitrationPolicy` before fetching the arbitration policy. This ensures that the arbitration policy is current and reflects any changes that may occur after the cooldown period.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR. Instead of calling `_updateActiveArbitrationPolicy`, the parent dispute was used. According to the team:
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 behaviour we can get to - without the precompile - would be to propagate the dispute to the derivative with the same arbitration policy it had in the parent.*

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/315

References

storyprotocol/protocol-core-v1/contracts/modules/dispute/DisputeModule.sol#L289-L333

7.7 MALICIOUS IP OWNER CAN FRONT-RUN DERIVATIVE REGISTRATION TO DRAIN USER'S ROYALTY TOKENS

// MEDIUM

Description

The **RoyaltyModule** is designed to manage the distribution of royalty tokens among IP owners and royalty policies when a derivative IP is linked to one or more parent IPs. External Royalty Policies enable IP owners to define custom royalty distribution rules by implementing the **IExternalRoyaltyPolicy** interface.

```
5  interface IExternalRoyaltyPolicy {
6      /// @notice Returns the amount of royalty tokens required to link a c
7      /// @param ipId The ipId of the IP asset
8      /// @param licensePercent The percentage of the license
9      /// @return The amount of royalty tokens required to link a child to
10     function getPolicyRtsRequiredToLink(address ipId, uint32 licensePerce
11 }
```

IP Owners can deploy and register an External Royalty Policy contract that implements IExternalRoyaltyPolicy and mint license tokens specifying the intended policy. Other users can then create derivative IPs by linking to parent IPs using the specified licenses and royalty policies. During the linking process, the **RoyaltyModule** calculates and distributes royalty tokens, associated with the user's **IPRoyaltyVault**, to royalty policies and IP owners based on the outputs of **getPolicyRtsRequiredToLink**.

The vulnerability lies in the `_distributeRoyaltyTokensToPolicies` function within the **RoyaltyModule** contract. This function relies on the `getPolicyRtsRequiredToLink` method from External Royalty Policies to determine the number of RTs to distribute. However, there is no validation on the value returned by this method.

```
486 function _distributeRoyaltyTokensToPolicies(
487     address ipId,
488     address[] calldata parentIpIds,
489     address[] calldata licenseRoyaltyPolicies,
490     uint32[] calldata licensesPercent,
491     address ipRoyaltyVault
492 ) internal {
493     RoyaltyModuleStorage storage $ = _getRoyaltyModuleStorage();
494
495     uint32 totalRtsRequiredToLink;
496     // this loop is limited to maxParents
```

```
    for (uint256 i = 0; i < parentIpIds.length; i++) {
        if (parentIpIds[i] == address(0)) revert Errors.RoyaltyModule__Ze
        if (licenseRoyaltyPolicies[i] == address(0)) revert Errors.Royalt
        _addToAccumulatedRoyaltyPolicies(parentIpIds[i], licenseRoyaltyPo
        address[] memory accParentRoyaltyPolicies = $.accumulatedRoyaltyP

        // this loop is limited to accumulatedRoyaltyPoliciesLimit
        for (uint256 j = 0; j < accParentRoyaltyPolicies.length; j++) {
            // add the parent ancestor royalty policies to the child
            _addToAccumulatedRoyaltyPolicies(ipId, accParentRoyaltyPolici
            // transfer the required royalty tokens to each policy
            uint32 licensePercent = accParentRoyaltyPolicies[j] == licens
                ? licensesPercent[i]
                : 0;
            uint32 rtsRequiredToLink = IRoyaltyPolicy(accParentRoyaltyPol
                parentIpIds[i],
                licensePercent
            );
            totalRtsRequiredToLink += rtsRequiredToLink;
            if (totalRtsRequiredToLink > MAX_PERCENT) revert Errors.Royal
            IERC20(ipRoyaltyVault).safeTransfer(accParentRoyaltyPolicies[
        }
    }

    if ($.accumulatedRoyaltyPolicies[ipId].length() > $.maxAccumulatedRoy
        revert Errors.RoyaltyModule__AboveAccumulatedRoyaltyPoliciesLimit

    // sends remaining royalty tokens to the ipId address or
    // in the case the ipId is a group then send to the group reward pool
    address receiver = IP_ASSET_REGISTRY.isRegisteredGroup(ipId)
        ? IP_ASSET_REGISTRY.getGroupRewardPool(ipId)
        : ipId;
    IERC20(ipRoyaltyVault).safeTransfer(receiver, MAX_PERCENT - totalRtsR
}
```

A malicious IP owner can observe a pending derivative registration transaction and front-run it by increasing the RT requirements in their royalty policy. This forces the user to transfer more Royalty Tokens than anticipated, potentially draining the user's RTs.

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:N/Y:H/R:N/S:U (5.0)

Recommendation

It is recommended to enforce the maximum Royalty Tokens limit specified by the user during derivative registration. If the required RTs exceed the maximum limit, the transaction should revert, preventing the user from overpaying.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR. A new variable, `maxRts`, defined by the user when registering a derivative, was implemented to ensure that the royalty tokens distributed to royalty policies do not exceed the specified limit.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/309

References

storyprotocol/protocol-core-v1/contracts/modules/royalty/RoyaltyModule.sol#L486-L530

7.8 UNINITIALIZED DYNAMIC ARRAY IN GETGROUPMEMBERS FUNCTION LEADS TO RUNTIME ERROR

// LOW

Description

The `getGroupMembers` function, implemented in the **GroupIPAssetRegistry** contract, is intended to retrieve a subset of IP addresses that are members of a specific group:

```
141 function getGroupMembers(
142     address groupId,
143     uint256 startIndex,
144     uint256 size
145 ) external view returns (address[] memory results) {
146     EnumerableSet.AddressSet storage allMemberIpIds = _getGroupIPAssetReg
147     uint256 totalSize = allMemberIpIds.length();
148     if (startIndex >= totalSize) return results;
149 
150     uint256 resultsSize = (startIndex + size) > totalSize ? size - ((star
151     for (uint256 i = 0; i < resultsSize; i++) {
152         results[i] = allMemberIpIds.at(startIndex + i);
153     }
154     return results;
155 }
```

The function declares a dynamic array results to hold the returned addresses but does not initialize it with a specific length before accessing its elements. In Solidity, when working with dynamic arrays in memory, it must be allocated with a defined length before accessing or assigning values to their elements. Attempting to assign values to results[i] without initializing results leads to a runtime error (`Panic: Index out of bounds`), causing the transaction to revert and making the `getGroupMembers` function broken.

Proof of Concept

The test will fail because the contract reverts when trying to assign a value to `results[0]` without initializing results.

```
function test_GroupingModule_getGroupMembers() public {
    vm.warp(100);

    //Alice registers a new group IP
    vm.prank(alice);
    address groupId = groupingModule.registerGroup(address(rewardPool));
```

```
    uint256 termsId = pilTemplate.registerLicenseTerms(
        PILFlavors.commercialRemix({
            mintingFee: 0,
            commercialRevShare: 10_000_000,
            currencyToken: address(erc20),
            royaltyPolicy: address(royaltyPolicyLAP)
        })
    );

    //IP owners attach license terms and mint license tokens
    vm.prank(ipOwner1);
        licensingModule.attachLicenseTerms(ipId1, address(pilTemplate), termsI
        licensingModule.mintLicenseTokens(ipId1, address(pilTemplate), termsId
    vm.prank(ipOwner2);
        licensingModule.attachLicenseTerms(ipId2, address(pilTemplate), termsI
        licensingModule.mintLicenseTokens(ipId2, address(pilTemplate), termsId

    //Alice attaches the license term
    vm.prank(alice);
        licensingModule.attachLicenseTerms(groupId, address(pilTemplate), term

    //Alice adds the intended IPs to the group IP
    vm.startPrank(alice);
        address[] memory ipIds = new address[](2);
        ipIds[0] = ipId1;
        ipIds[1] = ipId2;
        groupingModule.addIp(groupId, ipIds);
    vm.stopPrank();

    //Expect a revert due to uninitialized results array in getGroupMembers fu
    vm.expectRevert();
    ipAssetRegistry.getGroupMembers(groupId, 0, 1);
}
```

```
Ran 1 test for test/foundry/modules/grouping/GroupingModule.t.sol:GroupingModuleTest
[PASS] test_GroupingModule_brokengetGroupMembers() (gas: 3202935)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 20.26ms (3.16ms CPU time)

Ran 1 test suite in 146.44ms (20.26ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to initialize the results array properly with the correct length before accessing or assigning values to it. This can be achieved by allocating memory for results:

```
results = new address[](resultsSize);
```

Remediation

SOLVED: The **Story team** solved the issue in the specified PR. The `getGroupMembers` function was updated to initialize the results array dynamically based on the size of the group members.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/337

References

storyprotocol/protocol-core-v1/contracts/registries/GroupIPAssetRegistry.sol#L141-L155

7.9 USAGE OF DIRECT APPROVE CALLS LEADS TO SEVERAL ISSUES

// LOW

Description

The `distributeRewards` function in the **EvenSplitGroupPool** contract calls the approve method on tokens intended for reward distribution:

```
123 function distributeRewards(
124     address groupId,
125     address token,
126      address[] calldata ipIds
127 ) external whenNotPaused onlyGroupingModule returns (uint256[] memory rew
128     rewards = _getAvailableReward(groupId, token, ipIds);
129     uint256 totalRewards = 0;
130     for (uint256 i = 0; i < ipIds.length; i++) {
131         totalRewards += rewards[i];
132     }
133     if (totalRewards == 0) return rewards;
134     IERC20(token).approve(address(ROYALTY_MODULE), totalRewards);
135     EvenSplitGroupPoolStorage storage $ = _getEvenSplitGroupPoolStorage()
136     for (uint256 i = 0; i < ipIds.length; i++) {
137         if (rewards[i] == 0) continue;
138         // calculate pending reward for each IP
139         $.ipRewardDebt[groupId][token][ipIds[i]] += rewards[i];
140         // call royalty module to transfer reward to IP's vault as royalt
141         ROYALTY_MODULE.payRoyaltyOnBehalf(ipIds[i], groupId, token, rewar
142     }
143 }
```

However, certain tokens like USDT do not strictly adhere to the ERC-20 standard, leading to potential failures in the contract's operation. Specifically:

- **Approval Race Conditions**: Some tokens, to protect against race conditions, do not allow approving an amount **M > 0** when an existing amount **N > 0** is already approved. This restriction can cause the approve call to revert if not handled correctly.
- **Non-Standard Return Values**: The approve call does not return a boolean in some tokens, which can cause compatibility issues with contracts expecting a standard ERC-20 boolean response.
- **Approval Value Limits**: The function reverts if the approval value exceeds **uint96**, which can happen if large token amounts are involved.

These issues can cause the `distributeRewards` function to revert, preventing the distribution of rewards when using affected tokens. This not only disrupts the reward mechanism but also affects user trust and the protocol's functionality, especially if popular tokens like USDT are intended for use.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation

It is recommended to use SafeERC20's `forceApprove` method instead of direct `approve`, to ensure compatibility with all ERC20 tokens, including those with non-standard implementations, This will provide a more robust solution for managing token allowances across various ERC20 token contracts.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR. The `distributeRewards` function was updated to use SafeERC20's `forceApprove()` method.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/339

References

storyprotocol/protocol-core-v1/contracts/modules/grouping/EvenSplitGroupPool.sol#L123-L143

7.10 LACK OF PROPER NORMALIZATION IN SECP256R1 CURVE VERIFIER LEADS TO SIGNATURE MALLEABILITY

// LOW

Description

In ECDSA, a signature is composed of two values, `r` and `s`, derived from the message hash and the signer's private key. The malleability issue stems from the fact that for any valid signature `(r, s)`, the signature `(r, n - s)` is also valid, where `n` is the order of the elliptic curve group.

In Go, the `crypto/ecdsa` package provides functions for ECDSA signature generation and verification. The `ecdsa.Verify` function checks the validity of a signature by recalculating the signature values using the public key and message hash:

```
10 func Verify(hash []byte, r, s, x, y *big.Int) bool {
11     // Create the public key format
12     publicKey := newPublicKey(x, y)
13
14     // Check if they are invalid public key coordinates
15     if publicKey == nil {
16         return false
17     }
18
19     // Verify the signature with the public key,
20     // then return true if it's valid, false otherwise
21     return ecdsa.Verify(publicKey, hash, r, s)
22 }
```

However, the `ecdsa.Verify` function verifies signatures without normalizing the `s` value. Specifically, it accepts signatures where `s` is in the range `(0,n)` without normalizing `s` to ensure `s ≤ n/2`.

As a result, both `s` and `n - s` are considered valid, making the signature verification process susceptible to malleability attacks.

Proof of Concept

The `Verify` function is susceptible to signature malleability, showing that both the original signature `(r, s)` and the altered signature `(r, n - s)` are valid for the same message and public key:

```
package main

import (
```

```
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "crypto/sha256"
    "fmt"
    "math/big"
)

// Generates appropriate public key format from given coordinates
func newPublicKey(x, y *big.Int) *ecdsa.PublicKey {
    //Check if the given coordinates are valid
    if x == nil || y == nil || !elliptic.P256().IsOnCurve(x, y) {
        return nil
    }

    return &ecdsa.PublicKey{
        Curve: elliptic.P256(),
        X:     x,
        Y:     y,
    }
}

// Verify verifies the given signature (r, s) for the given hash and public ke
// It returns true if the signature is valid, false otherwise.
func Verify(hash []byte, r, s, x, y *big.Int) bool {
    //Create the public key format
    publicKey := newPublicKey(x, y)

    //Check if they are invalid public key coordinates
    if publicKey == nil {
        return false
    }

    //Verify the signature with the public key,
    return ecdsa.Verify(publicKey, hash, r, s)
}

func main() {
    //Generate a new private key using P-256 curve
    privateKey, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
    if err != nil {
        fmt.Println("Error generating private key:", err)
        return
    }
```

```
    //Get public key components
    x := privateKey.PublicKey.X
    y := privateKey.PublicKey.Y

    //Message to be signed
    message := []byte("xyz")

    //Hash the message using SHA-256
    hash := sha256.Sum256(message)

    //Sign the hash using the private key
    r, s, err := ecdsa.Sign(rand.Reader, privateKey, hash[:])
    if err != nil {
        fmt.Println("Error signing message:", err)
        return
    }

    //Verify the original signature
    validOriginal := Verify(hash[:], r, s, x, y)
    fmt.Printf("Original signature valid: %v\n", validOriginal)

    //Get the curve order n
    nCurve := elliptic.P256().Params().N

    //Compute s' = n - s
    sPrime := new(big.Int).Sub(nCurve, s)

    //Verify the malleable signature
    validMalleable := Verify(hash[:], r, sPrime, x, y)
    fmt.Printf("Malleable signature valid: %v\n", validMalleable)

    if validMalleable {
        fmt.Println("Malleability demonstrated in Verify function.")
    } else {
        fmt.Println("Malleability not demonstrated.")
    }
}
```

The result of the test is the following:

```
qba@HAL-LT156-MBP gotest % go run verifyMalleability.go
Original signature valid: true
Malleable signature valid: true
Malleability demonstrated in Verify function.
```

BVSS

AO:A/AC:M/AX:H/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (2.2)

Recommendation

To mitigate the vulnerability, modify the signature verification process to enforce the canonical form of `s` by normalizing it to `s ≤ n / 2`.

```
func Verify(hash []byte, r, s, x, y *big.Int) ([]byte, error) {
    // Create the public key format
    publicKey := newPublicKey(x, y)
    if publicKey == nil {
        return nil, errors.New("invalid public key coordinates")
    }

    if checkMalleability(s) {
        return nil, errors.New("malleability issue")
    }

    // Verify the signature with the public key and return 1 if it's valid, 0
    if ok := ecdsa.Verify(publicKey, hash, r, s); ok {
        return common.LeftPadBytes(common.Big1.Bytes(), 32), nil
    }

    return common.LeftPadBytes(common.Big0.Bytes(), 32), nil
}

// Check the malleability issue
func checkMalleability(s *big.Int) bool {
    return s.Cmp(secp256k1halfN) <= 0
}
```

Remediation

RISK ACCEPTED: The **Story team** accepted the risk of this issue and states the following:

> To ensure compatibility with the NIST P-256 specification, the malleability check was not implemented. The necessary protection will be applied at the contract layer whenever the precompile is invoked.

References

piplabs/story-geth/crypto/secp256r1/verifier.go

7.11 USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS

// INFORMATIONAL

Description

In the **IPAccountImpl**, **RoyaltyModule**, **RoyaltyPolicyLAP RoyaltyPolicyLRP** and **LicenseRegistry** contract, `require` statements are used. In Solidity development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hardcoded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts. The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption.

This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommend to replace all revert strings with custom errors.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/329

7.12 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

During the audit, it was identified that **DisputeModule**, **IPAccountStorage, AccessController, LicensorApprovalChecker** and **IPGraphACL** contracts do not use named mappings, unlike the other contracts in the protocol.

The project is using Solidity version greater than 0.8.18, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice will enhance code readability and make the purpose of each mapping more explicit.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider refactoring all the mappings to use named arguments.

Remediation

PARTIALLY SOLVED: The **Story team** partially solved the issue in the specified PR. The mappings in **DisputeModule** contract have been refactored.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/326

7.13 INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS

// INFORMATIONAL

Description

To mitigate the risk of reentrancy attacks, a modifier named `nonReentrant` is commonly used. This modifier acts as a lock, ensuring that a function cannot be called recursively while it is still in execution. A typical implementation of the `nonReentrant` modifier locks the function at the beginning and unlocks it at the end. However, it is vital to place the `nonReentrant` modifier before all other modifiers in a function. Placing it first ensures that all other modifiers cannot bypass the reentrancy protection. In the current implementation, some functions use other modifiers before `nonReentrant`, which may compromise the protection it provides.

During the audit it was identified that the following functions does not use `nonReentrant` as a first modifier:

- **ArbitrationPolicyUMA.sol**
  - `onRaiseDispute`

- **LicensingModule.sol**
  - `mintLicenseTokens`
  - `registerDerivative`

- **RoyaltyPolicyLAP.sol**
  - `onLicenseMinting`
  - `onLinkToParents`

- **RoyaltyPolicyLRP.sol**
  - `onLicenseMinting`
  - `onLinkToParents`

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to follow the best practice of placing the `nonReentrant` modifier before all other modifiers. By doing so, one can reduce the risk of reentrancy-related vulnerabilities. This simple yet effective approach can help enhance the security posture of any Solidity smart contract.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/327

7.14 POTENTIAL LICENSE INCOMPATIBILITIES

// INFORMATIONAL

Description

The inconsistent use of open-source licenses across the files in the protocol was identified. The **VaultController** and **IpRoyaltyVault** contracts specifying the **MIT** license, the **IPAccountStorageOps** is unlicensed and others using the more restrictive **BUSL-1.1** license. Specifically:

- Certain files are marked with `// SPDX-License-Identifier: MIT`, which is a permissive open-source license that allows anyone to use, copy, modify, and distribute the code, provided the original license is included.
- Other files use `// SPDX-License-Identifier: BUSL-1.1`, a more restrictive license that limits commercial use for a period of time, making it incompatible with **MIT** in terms of redistributing or combining the code.

This discrepancy could lead to **license incompatibility issues** when attempting to combine or redistribute the project's code as a whole, potentially violating the terms of one or both licenses. This is particularly problematic in open-source projects, where license compliance is crucial for legal and community standards.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Ensure consistent licensing across the project by reviewing and aligning all file headers to use the same license, depending on the intended usage and distribution model.

Remediation

SOLVED: The **Story team** solved the issue in the specified PRs. All contracts use the BUSL-1.1 license.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/333 https://github.com/storyprotocol/protocol-core-v1/pull/338

7.15 UNUSED CUSTOM ERRORS

// INFORMATIONAL

Description

Throughout the code in scope, there are several custom errors that are declared but never used.

In `Errors.sol`:

- `error GroupingModule__InvalidGroupRewardPool(address rewardPool);`
- `error GroupingModule__CallerIsNotLicensingModule(address caller);`
- `error GroupIPAssetRegistry__GroupFrozen(address groupId);`
- `error GroupIPAssetRegistry__GroupNotFrozen(address groupId);`
- `error LicenseRegistry__NotTransferable();`
- `error LicenseRegistry__NoParentIp();`
- `error LicenseToken__ZeroLicensingModule();`
- `error LicenseToken__ZeroDisputeModule();`
- `error LicensingModule__ReceiverCheckFailed(address receiver);`
- `error LicensingModule__ZeroGroupingModule();`
- `error RoyaltyPolicyLAP__IpTagged();`
- `error RoyaltyPolicyLAP__AlreadyClaimed();`
- `error RoyaltyPolicyLAP__ClaimerNotAnAncestor();`
- `error RoyaltyPolicyLAP__NotAllRevenueTokensHaveBeenClaimed();`
- `error RoyaltyPolicyLAP__InvalidTargetIpId();`
- `error EvenSplitGroupPool__UnregisteredCurrencyToken(address currencyToken);`
- `error EvenSplitGroupPool__UnregisteredGroupIP(address groupId);`

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to remove all unused errors.

Remediation

SOLVED: The **Story team** solved the issue in the specified PR. The mentioned custom errors were removed.

Remediation Hash

https://github.com/storyprotocol/protocol-core-v1/pull/330

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

The security team assessed all findings identified by the Slither software and everything was categorised as false positives.

Results

```
INFO:Detectors:
IPAccountImpl._execute(address,address,uint256,bytes) (contracts/IPAccountImpl.sol#207-223) sends eth to arbitrary user
        Dangerous calls:
        - (success,result) = to.call{value: value}(data) (contracts/IPAccountImpl.sol#216)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
         - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
ERC6551.fallback() (node_modules/solady/src/accounts/ERC6551.sol#389-396) calls ERC6551.receiverFallback() (node_modules/solady/src/accounts/ERC6551.sol#326-380) which halt the execution return(uint256,uint256)(0x5
c,0x20) (node_modules/solady/src/accounts/ERC6551.sol#364)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-return-in-assembly
INFO:Detectors:
ERC6551._cachedChainId (node_modules/solady/src/accounts/ERC6551.sol#82) shadows:
        - EIP712._cachedChainId (node_modules/solady/src/utils/EIP712.sol#25)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

```
INFO:Detectors:
LibZip.flzCompress.asm_0.mt()._o_flzCompress_asm_0_mt (node_modules/solady/src/utils/LibZip.sol#49) is written in both
        _o_flzCompress_asm_0_mt = ms8(ms8(ms8(o__flzCompress_asm_0_mt,224 + d__flzCompress_asm_0_mt >> 8),l__flzCompress_asm_0_mt - 7),0xff & d__flzCompress_asm_0_mt) (node_modules/solady/src/utils/LibZip.sol#54)
        _o_flzCompress_asm_0_mt = ms8(ms8(o__flzCompress_asm_0_mt,l__flzCompress_asm_0_mt << 5 + d__flzCompress_asm_0_mt >> 8),0xff & d__flzCompress_asm_0_mt) (node_modules/solady/src/utils/LibZip.sol#57)
SignatureCheckerLib.isValidSignatureNow(address,bytes32,bytes).isValid (node_modules/solady/src/utils/SignatureCheckerLib.sol#35) is written in both
        isValid = 1 (node_modules/solady/src/utils/SignatureCheckerLib.sol#79)
        isValid = mload(uint256)(d_isValidSignatureNow_asm_0) == f_isValidSignatureNow_asm_0 & staticcall(uint256,uint256,uint256,uint256,uint256,uint256)(gas()(),signer,m_isValidSignatureNow_asm_0,returndatasize()
() + 0x44,d_isValidSignatureNow_asm_0,0x20) (node_modules/solady/src/utils/SignatureCheckerLib.sol#97-111)
SignatureCheckerLib.isValidSignatureNowCalldata(address,bytes32,bytes).isValid (node_modules/solady/src/utils/SignatureCheckerLib.sol#123) is written in both
        isValid = 1 (node_modules/solady/src/utils/SignatureCheckerLib.sol#167)
        isValid = mload(uint256)(d_isValidSignatureNowCalldata_asm_0) == f_isValidSignatureNowCalldata_asm_0 & staticcall(uint256,uint256,uint256,uint256,uint256,uint256)(gas()(),signer,m_isValidSignatureNowCalldat
a_asm_0,signature + 0x64,d_isValidSignatureNowCalldata_asm_0,0x20) (node_modules/solady/src/utils/SignatureCheckerLib.sol#185-199)
SignatureCheckerLib.isValidSignatureNow(address,bytes32,bytes32,bytes32).isValid (node_modules/solady/src/utils/SignatureCheckerLib.sol#211) is written in both
        isValid = 1 (node_modules/solady/src/utils/SignatureCheckerLib.sol#233)
        isValid = mload(uint256)(d_isValidSignatureNow_asm_0) == f_isValidSignatureNow_asm_0 & staticcall(uint256,uint256,uint256,uint256,uint256,uint256)(gas()(),signer,m_isValidSignatureNow_asm_0,0xa5,d_isValidSi
gnatureNow_asm_0,0x20) (node_modules/solady/src/utils/SignatureCheckerLib.sol#249-263)
SignatureCheckerLib.isValidSignatureNow(address,bytes32,uint8,bytes32,bytes32).isValid (node_modules/solady/src/utils/SignatureCheckerLib.sol#277) is written in both
        isValid = 1 (node_modules/solady/src/utils/SignatureCheckerLib.sol#299)
        isValid = mload(uint256)(d_isValidSignatureNow_asm_0) == f_isValidSignatureNow_asm_0 & staticcall(uint256,uint256,uint256,uint256,uint256,uint256)(gas()(),signer,m_isValidSignatureNow_asm_0,0xa5,d_isValidSi
gnatureNow_asm_0,0x20) (node_modules/solady/src/utils/SignatureCheckerLib.sol#315-329)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#write-after-write
INFO:Detectors:
The function IPAccountImpl.isValidSigner(address,bytes) (contracts/IPAccountImpl.sol#74-88) reads this.isValidSigner(signer,to,callData) (contracts/IPAccountImpl.sol#85) with `this` which adds an extra STATICCALL.
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-variable-read-in-external-context
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Base64.encode(bytes) (node_modules/@openzeppelin/contracts/utils/Base64.sol#18-98) performs a multiplication on the result of a division:
        - result = new string(4 * ((data.length + 2) / 3)) (node_modules/@openzeppelin/contracts/utils/Base64.sol#34)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

```
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return value by Address.functionDelegateCall(newImplementation,data) (node_modules/@o
penzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores return value by Address.functionDelegateCall(IBeacon(newBeacon).implementatio
n(),data) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#178)
Time.get(Time.Delay) (node_modules/@openzeppelin/contracts/utils/types/Time.sol#90-93) ignores return value by (delay,None,None) = self.getFull() (node_modules/@openzeppelin/contracts/utils/types/Time.sol#91)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Base64.encode(bytes) (node_modules/@openzeppelin/contracts/utils/Base64.sol#18-98) performs a multiplication on the result of a division:
        - result = new string(4 * ((data.length + 2) / 3)) (node_modules/@openzeppelin/contracts/utils/Base64.sol#34)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return value by Address.functionDelegateCall(newImplementation,data) (node_modules/@o
penzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores return value by Address.functionDelegateCall(IBeacon(newBeacon).implementatio
n(),data) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#178)
Time.get(Time.Delay) (node_modules/@openzeppelin/contracts/utils/types/Time.sol#90-93) ignores return value by (delay,None,None) = self.getFull() (node_modules/@openzeppelin/contracts/utils/types/Time.sol#91)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.