StoryProtocol

Security Assessment

November 2024

FUZZING LABS

Contents

About FuzzingLabs

Founded in 2021 and headquartered in Paris, FuzzingLabs is a cybersecurity startup specializing in vulnerability research, fuzzing, and blockchain security. We combine cutting-edge research with hands-on expertise to secure some of the most critical components in the blockchain ecosystem.

At FuzzingLabs, we aim to uncover and mitigate vulnerabilities before they can be exploited. Over the past year, our tools and methodologies have identified hundreds of vulnerabilities in essential blockchain components, such as RPC libraries, cryptographic systems, compilers, and smart contracts. We collaborate with leading protocols and foundations to deliver open-source security tools, continuous audits, and comprehensive fuzzing services that help secure the future of blockchain technology.

If you're interested, we have a blog available at fuzzinglabs.com and an X account @FuzzingLabs. You can also contact us at contact@fuzzinglabs.com.

Executive Summary

Goals

The primary goal of this audit is to assess the security and resilience of the Story Protocol project through a detailed analysis of its codebase and associated components. The objectives are structured into the following areas:

1. **Code Audit and Security Review**
   - Perform a rapid analysis of the code to detect straightforward patterns and vulnerabilities. This phase will identify low-hanging fruits, such as common security flaws, poor coding practices, or weak implementation patterns.
   - Familiarize ourselves with the architecture and identify areas needing further attention in the next phase.
2. **In-Depth Analysis (Deep Dive):**
   - Conduct a more thorough examination of the codebase to identify more complex patterns, logical bugs, or security vulnerabilities that may not be immediately apparent. This includes looking for subtle issues related to control flow, data validation, and other complex logic errors.
   - Writing fuzzing harnesses will be a key part of this stage to automate the detection of edge cases and unexpected behaviors and expose vulnerabilities that may not be easily found through manual analysis.
3. **Execution Layer Invariant Testing**
   - Fork Medusa to allow maximum compatibility with Story execution layer specifics features.
   - Develop fuzzing harnesses for smart contracts to test invariants under various scenarios. These invariants will include but are not limited to, ensuring the correctness of fund transfers, adherence to protocol rules, and consistency in state transitions.
4. **Consensus Layer Invariant Testing**
   - Stress test the consensus layer against invariants via fuzzing.

Limitations

While the scope of the audit is extensive, certain limitations were established in agreement with the client:

- **Exclusion of legacy code:** Although the audit scope included comprehensive assessments, the client explicitly requested the exclusion of battle-tested legacy code.
- **Exclusion of Cryptographic Analysis**: Another area that was deemed non-essential by the client for this particular audit was cryptographic analysis. As such, we did not conduct any evaluations or validations of cryptographic primitives, protocols, or implementations in the system.

These limitations were client-directed and aligned with the specific risk tolerance and project requirements. They may be revisited in future assessments if required.

Project Summary

About StoryProtocol

Story Protocol is a decentralized network designed to tokenize, program, and manage intellectual property (IP) on the blockchain. It enables creators to register, distribute, and monetize their IP assets seamlessly while ensuring transparency and security. Story Protocol addresses the limitations of traditional IP management systems, which often suffer from inefficiency, lack of transparency, and limited programmability. By leveraging blockchain technology, it transforms intellectual property into programmable assets, empowering creators and developers to innovate collaboratively.
The platform introduces a core innovation: **IP Assets** and **IP Accounts**. IP Assets are tokenized representations of creative works, while IP Accounts serve as smart contracts managing permissions, royalties, and on-chain interactions. Story Protocol provides a **programmable IP layer**, allowing creators to define rights and permissions dynamically, ensuring precise control over how their assets are used.
Story Protocol operates on its own **Story Network**, a purpose-built Layer 1 blockchain that is fully EVM-compatible and optimized for managing complex data structures like IP. This architecture supports high scalability and composability, enabling efficient interactions across applications. It also integrates customizable modules to handle operations like licensing, royalty distribution, and dispute resolution, making it a versatile tool for developers and creators alike.
The platform supports two key functionalities:

1. **IP Tokenization and Management:**
    - Story Protocol allows creators to transform their works into tokenized IP assets with programmable attributes, enabling novel use cases such as collaborative content creation and AI-powered remixing.

2. **Licensing and Royalty Distribution:**
    - With a universal **Programmable IP License (PIL)**, creators can establish transparent and automated licensing frameworks, ensuring fair compensation and compliance with predefined rules.

Story Protocol is particularly valuable for developers and creators seeking to manage intellectual property in a decentralized and efficient manner. Its use cases include collaborative content platforms, decentralized storytelling, AI-driven content generation, and any application requiring programmable and secure IP rights management.

The goal of Story Protocol is to revolutionize the management and monetization of intellectual property, fostering a decentralized ecosystem where IP is accessible, programmable, and beneficial to all stakeholders. By integrating IP into the blockchain, it empowers creators and developers to unlock new economic opportunities and drive innovation in the creative economy.

Scope

- **piplabs/story**: This repository contains the official code for the Story Layer 1 consensus client, contracts, and associated tooling. It includes the core components of the Story Protocol blockchain, such as the consensus mechanism and smart contract implementations.
    - **Tag**: 0.12.0 and 0.12.1 once it was released
- **piplabs/story-geth**: This repository is a fork of the Go Ethereum (Geth) client, customized for the Story Protocol. It serves as the execution client, handling transaction processing and state management.
    - **Tag**: 0.10.0
- **storyprotocol/protocol-core-v1**: This repository contains the core protocol code for Story Protocol. It includes smart contracts and scripts essential for the protocol's operation.
    - **Commit:** 3ef2a99
- **piplabs/cosmos-sdk**: This repository is a fork of the Cosmos SDK, tailored for Story Protocol's needs. It provides the framework for building the blockchain's consensus and networking layers.

Audit Timeline

The security audit for Story Protocol was conducted over a structured timeline of **50 man-days** by **4 FuzzingLabs auditors** to comprehensively address its components and ensure the system's security and robustness. The audit was carried out as follows:

1. **Attack Surface and Threat Modeling**
    - Identified key risks and critical components (Execution Layer, Consensus Layer, Cosmos SDK).
    - Developed a threat model for potential vulnerabilities.
2. **L1 Consensus Audit**
    - Performed static analysis and used Go audit tools.
    - Verified staking mechanisms and state synchronization security.
3. **Core Protocol and Solidity Smart Contracts Audit**
    - Reviewed core smart contracts for blockchain-specific vulnerabilities.
    - Developed invariants and conducted automated fuzzing for Solidity contracts.
4. **Cosmos SDK Usage Audit**
    - Assessed SDK customizations for compliance with security standards.
    - Focused on state transitions and module interaction risks.
5. **Documentation, Reporting, and Tool Packaging**
    - Compiled detailed audit reports and recommendations for stakeholders.
    - Produced comprehensive documentation to support future development and maintenance.
    - Packaged custom tools and scripts developed during the audit for ongoing security monitoring.

This structured approach allowed us to cover both the high-level architecture and the critical components of StoryProtocol, ensuring a comprehensive security review.

Threat Model

The **Story Protocol** is a decentralized network designed to tokenize, program, and manage intellectual property (IP) on the blockchain. This threat model aims to identify, categorize, and assess potential security threats to ensure the robustness and integrity of the Story Protocol ecosystem.

1. System Architecture Overview

Understanding the architecture is essential for identifying potential threats. Below are the core components of the Story Protocol:

- **Execution Layer (EL):** Handles transaction processing and state management.
- **Consensus Layer (CL):** Manages the consensus and stacking mechanism, ensuring network agreement.
- **Story Protocol Contracts:** Written in Solidity, they define the protocol's functionality, including IP tokenization and royalty mechanisms.

2. Potential Threat Actors

Understanding who might attack the system helps in tailoring defense strategies. Potential threat actors include:

- **External Hackers:** Seeking financial gain or to disrupt the network.
- **Insider Threats:** Malicious or negligent actions by developers or administrators.
- **Competitors:** Attempting to undermine Story Protocol's market position.
- **Malicious Users:** Exploiting vulnerabilities for personal gain.

3. Potential Vulnerabilities

Identifying specific weaknesses that could be exploited like:

- **Story Protocol Contracts:** Improper access control, miscalculations, griefing, DoS, frontrun, theft of funds, loss of funds.
- **Execution Layer (EL):** Improper access control for IpGraph, miscalculations, synchronization issue withCL.
- **Consensus Layer (CL):** DoS, synchronization issue with EL, theft of funds, loss of funds, logs reordering.
- **Dependency Risks:** Vulnerabilities in third-party libraries or dependencies.

4. Attack Vectors

Potential pathways through which threats can materialize:

- **Network Layer:** Exploiting weaknesses in the communication protocols.
- **Application Layer:** Attacks on client applications or APIs.
- **Smart Contracts:** Direct exploitation of vulnerabilities within Solidity contracts.
- **Consensus Mechanism:** Attacks targeting the consensus layer to disrupt network agreement.
- **Dependency Exploits:** Leveraging vulnerabilities in third-party libraries used by Story Protocol.

Execution Layer

The **Story execution layer** is a forked version of Geth with the following major features:

- **Stateful IpGraph precompile** to manage the royalty tree.
- **Module to filter sanctioned addresses.**

Potential identified threats specific to Story EL are:

ID	Threat	Impact
1	Call to IpGraph could DoS the EL	**CRITICAL**
2	IpGraph access control bypass	**HIGH**
3	Bypass of sanctions filter	**HIGH**
4	Miscalculation in IpGraph	**HIGH**
5	Call to IpGraph could significantly slow the EL	**MEDIUM**
6	Sanction filter could significantly slow the EL	**MEDIUM**

Consensus Layer

The **Story consensus layer** is built using the Cosmos SDK and CometBFT. It includes several custom modules designed to enable the following features:

- **A bridge between the execution and consensus layers** using EVM events.
- **Native staking** management.
- **Universal Basic Income (UBI) rewards** handling.

Potential identified threats specific to these features are:

ID	Threat	Impact
1	Increase validator staking without a deposit	CRITICAL
2	DoS of the CL	CRITICAL
3	Attack that leads to steal of fund	CRITICAL
4	Balance syncing issue	CRITICAL
5	Attack that leads to loss of fund	HIGH
6	Log ordering issues	MEDIUM

Findings

I. Network can be halted by spamming the mempool

Rating	**Critical**
ID	**FL-SP-01**
Target	**Consensus chain**

Description

Since Story checks in PrepareProposal that there are no transactions in the proposal, the network can be halted by sending transactions to the Tendermint RPC endpoint that will be added to the proposal or shared with the network.

story-consensus/client/x/evmengine/keeper/abci.go

```
package keeper
...
func (k *Keeper) PrepareProposal(ctx sdk.Context, req *abci.RequestPrepareProposal) (
	*abci.ResponsePrepareProposal, error,
) {
	...
	if len(req.Txs) > 0 {
		return nil, errors.New("unexpected transactions in proposal")
	}
}
```

And by default Story CometBFT is configured with the following parameters:

```
...
[rpc]

# TCP or UNIX socket address for the RPC server to listen on
laddr = "tcp://0.0.0.0:26657"
...
[p2p]
```

```
# Address to listen for incoming connections
laddr = "tcp://0.0.0.0:26656"
...
# Comma separated list of seed nodes to connect to
seeds =
"75ac7b193e93e928d6c83c273397517cb60603c0@b1.odyssey-testnet.storyrpc.io:26656,6adbd1e97
4d6bb1c353aabbc7abef72c81e536f5@b2.odyssey-testnet.storyrpc.io:26656"
...
[mempool]

# The type of mempool for this node to use.
#
#  Possible types:
#  - "flood" : concurrent linked list mempool with flooding gossip protocol
#  (default)
#  - "nop"   : nop-mempool (short for no operation; the ABCI app is responsible
#  for storing, disseminating and proposing txs). "create_empty_blocks=false" is
#  not supported.
type = "flood"
...
# Broadcast (default: true) defines whether the mempool should relay
# transactions to other peers. Setting this to false will stop the mempool
# from relaying transactions to other peers until they are included in a
# block. In other words, if Broadcast is disabled, only the peer you send
# the tx to will see it until it is included in a block.
broadcast = true
...
```

The network can be halted by sending transactions to the Tendermint RPC endpoint that will be added to the proposal or broadcast to the network.

Recommendations

- A possible solution would be to disable the mempool broadcast feature by setting broadcast = false in the config.toml file and setting the mempool type to nop in the config.toml file.
- Another possible solution would be to add a check in the CheckTx function to ensure that the transactions are not added to the proposal or shared with the network.

II. IPAccount can steal all group rewards

Rating	**High**
ID	**FL-SP-02**
Target	**EvenSplitGroupPool**

Description

The GroupingModule enables the creation and management of group IPAssets, supporting a royalty pool for the group. The default pool type implements a reward mechanism where all royalties collected are shared evenly among the group's IPAssets.

However, there is a vulnerability in the function responsible for handling the reward distribution that allows an IP to get all the rewards from the group pool.

The distributeRewards() function does not check against duplicates in the ipIds list. As a result, a malicious IP owner can trigger reward distribution using a list with multiple instances of the same ipId.
The duplicated IP will receive multiple reward shares, effectively stealing rewards from other IPs.

contracts/modules/grouping/EvenSplitGroupPool.sol

```
/// @notice Distributes rewards to the given IP accounts in pool
/// @param groupId The group ID
/// @param token The reward tokens
/// @param ipIds The IP IDs
function distributeRewards(
    address groupId,
    address token,
    address[] calldata ipIds // @audit Duplicates in ipIds will receive the reward
several times
) external whenNotPaused onlyGroupingModule returns (uint256[] memory rewards) {
    rewards = _getAvailableReward(groupId, token, ipIds);
    uint256 totalRewards = 0;
    for (uint256 i = 0; i < ipIds.length; i++) {
        totalRewards += rewards[i];
    }
    if (totalRewards == 0) return rewards;
```

```
    IERC20(token).approve(address(ROYALTY_MODULE), totalRewards);
    EvenSplitGroupPoolStorage storage $ = _getEvenSplitGroupPoolStorage();
    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 royalty
        ROYALTY_MODULE.payRoyaltyOnBehalf(ipIds[i], groupId, token, rewards[i]);
    }
}
```

Recommendations

- Verify that the ipIds array parameter in the distributeRewards() function does not contain duplicates.

III. createValidator can be frontrun leading to grief and loss of funds

Rating	High
ID	**FL-SP-03**
Target	**IPTokenStaking**

Description

This issue arises from the non-atomic nature of the createValidator process. The function execution is divided into two distinct steps:

1. **Execution Layer**: The createValidator function initiates the validator creation process.
2. **Consensus Layer**: The ProcessCreateValidator function is triggered to complete the operation.

Since these steps are executed sequentially and not atomically, the intermediate state between the two can lead to inconsistencies or potential vulnerabilities.

story-consensus-0.12.1/contracts/src/protocol/IPTokenStaking.sol

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity ^0.8.23;
...
contract IPTokenStaking is IIPTokenStaking, Ownable2StepUpgradeable,
ReentrancyGuardUpgradeable, PubKeyVerifier {
    ...
    function createValidator(
        ...
    ) external payable verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey,
msg.sender) nonReentrant {
        _createValidator(
            validatorUncmpPubkey,
            moniker,
            commissionRate,
            maxCommissionRate,
            maxCommissionChangeRate,
            supportsUnlocked,
            data
        );
    }
```

```
    function createValidatorOnBehalf(
        ...
    ) external payable verifyUncmpPubkey(validatorUncmpPubkey) nonReentrant {
        _createValidator(
            validatorUncmpPubkey,
            moniker,
            commissionRate,
            maxCommissionRate,
            maxCommissionChangeRate,
            supportsUnlocked,
            data
        );
    }

    function _createValidator(
        ...
    ) internal {
        ...
        payable(address(0)).transfer(stakeAmount);
        emit CreateValidator(
            validatorUncmpPubkey,
            moniker,
            stakeAmount,
            commissionRate,
            maxCommissionRate,
            maxCommissionChangeRate,
            supportsUnlocked ? 1 : 0,
            msg.sender,
            data
        );
        if (remainder > 0) {
            _refundRemainder(remainder);
        }
    }
}
```

So when a user calls the createValidator function, it can be front-run by another user calling the createValidatorOnBehalf with a higher gas price. This will result in the second user's transaction being executed first.

Since the createValidator function is not atomic, the first user's transaction will still be executed. When the log is processed it will throw an error because the validator already exists, leading to a loss of funds.

story-consensus-0.12.1/client/x/evmstaking/keeper/validator.go

```
package keeper

func (k Keeper) ProcessCreateValidator(ctx context.Context, ev
*bindings.IPTokenStakingCreateValidator) (err error) {
	...
	if _, err = k.stakingKeeper.GetValidator(ctx, validatorAddr); err == nil {
		return errors.WrapErrWithCode(errors.ValidatorAlreadyExists,
errors.New("validator already exists"))
	} else if !errors.Is(err, stypes.ErrNoValidatorFound) {
		// Either the validator does not exist, or unknown error.
		return errors.Wrap(err, "get validator")
	}
	...
}
```

Recommendations

- Implement error handling in the ProcessCreateValidator function to prevent the loss of funds on error.

IV. onERC721Received is never called when new license tokens are minted

Rating	**Medium**
ID	**FL-SP-04**
Target	**LicenseToken**

Description

In the mintLicenseTokens function, using _mint instead of _safeMint does not check for onERC721Received callback.
According to the ERC721 standard, the onERC721Received callback must be invoked during mint or transfer operations to notify the recipient contract. However, in this implementation, smart contracts interacting as recipients must be properly notified via the onERC721Received callback, as required by the standard.
This deviation could lead to unexpected behavior or incompatibility with other smart contracts that rely on this callback for proper functioning.

protocol-core-v1/contracts/LicenseToken.sol

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.26;
...
contract LicenseToken is ILicenseToken, ERC721EnumerableUpgradeable,
AccessManagedUpgradeable, UUPSUpgradeable {
    ...
    function mintLicenseTokens(
        ...
    ) external onlyLicensingModule returns (uint256 startLicenseTokenId) {
        ...
        for (uint256 i = 0; i < amount; i++) {
            ...
            _mint(receiver, tokenId);
            ...
        }
    }
    ...
}
```

Recommendations

- Use _safeMint instead of _mint to trigger the onERC721Received.

V. onERC721Received is never called when new group nft are minted

Rating	**Medium**
ID	**FL-SP-05**
Target	**GroupNFT**

Description

In the mintGroupNft function, using _mint instead of _safeMint does not check for onERC721Received callback.
According to the ERC721 standard, the onERC721Received callback must be invoked during mint or transfer operations to notify the recipient contract. However, in this implementation, smart contracts interacting as recipients must be properly notified via the onERC721Received callback, as required by the standard.
This deviation could lead to unexpected behavior or incompatibility with other smart contracts that rely on this callback for proper functioning.

protocol-core-v1/contracts/GroupNFT.sol

```
// SPDX-License-Identifier: BUSL-1.1
...
contract GroupNFT is IGroupNFT, ERC721Upgradeable, AccessManagedUpgradeable,
UUPSUpgradeable {
    ...
    function mintGroupNft(address minter, address receiver) external onlyGroupingModule
returns (uint256 groupNftId) {
        ...
        _mint(receiver, groupNftId);
        ...
    }
    ...
}
```

Recommendations

- Use _safeMint instead of _mint to trigger the onERC721Received.

VI. Static, delegate, and callcode calls to the IpGraph precompile update the state

Rating	Low
ID	**FL-SP-06**
Target	**IpGraph**

Description

The issue is that performing a static, delegate, or callcode call to the IpGraph precompile updates the state of the contract.

These are unexpected behaviors that are non-compliant with EVM specifications.

Recommendations

- Disable all call variants except CALL for stateful functions of the IpGraph precompile.

VII. Casting and cropping issues in royalty calculations could lead to unexpected behavior

Rating	Low
ID	FL-SP-07
Target	IpGraph

Description

No checks on the royalty value are done when calling setRoyalty, for the solidity side a royalty is a uint32 but when calling setRoyalty a privileged user can set the royalty to a uint256 since the precompile does not check the size of the input.

story-geth/core/vm/ipgraph.go

```
...
func (c *ipGraph) setRoyalty(input []byte, evm *EVM, ipGraphAddress common.Address)
([]byte, error) {
	...
	royalty := new(big.Int).SetBytes(getData(input, 96, 32))
	slot := crypto.Keccak256Hash(ipId.Bytes(), parentIpId.Bytes(),
royaltyPolicyKind.Bytes()).Big()
	...
}
...
```

Every royalty calculation is done with big.Int which is a type that can hold any integer value.

This can lead to value cropping when the result is cast back to a uint256 using common.BigToHash before returning it to the caller.

story-geth/core/vm/ipgraph.go

```
...
func (c *ipGraph) getRoyalty(input []byte, evm *EVM, ipGraphAddress common.Address)
([]byte, error) {
```

```
        ...
        return common.BigToHash(totalRoyalty).Bytes(), nil
}
...
```

The caller is expected to cast the result back to a uint32 but this is not checked by the precompile and the solidity cast does not check for overfitting values.

protocol-core-v1/contracts/modules/royalty/policies/LAP/RoyaltyPolicyLAP.sol

```
...
  function _getRoyaltyLAP(address ipId, address ancestorIpId) internal returns (uint32)
{
        (bool success, bytes memory returnData) = IP_GRAPH.call(
            abi.encodeWithSignature("getRoyalty(address,address,uint256)", ipId,
ancestorIpId, uint256(0))
        );
        require(success, "Call failed");
        return uint32(abi.decode(returnData, (uint256)));
    }
...
```

Recommendations

- Check the royalty value in setRoyalty to make sure, at least, to fit in a uint32.
- Check for overflows in the royalty calculation to avoid cropping or having a result that does not fit in a uint32.
- Check on the caller side that the result of getRoyalty* fits in a uint32 before casting it back.

VIII. Calls to IpGraph should be non-payable

Rating	**Low**
ID	**FL-SP-08**
Target	**IpGraph**

Description

When calling the IpGraph a user could send funds by error and lock them forever.

Recommendations

- Make all calls to the IpGraph non-payable.

IX. Adding parents' ip erases the previous ones

Rating	Informational
ID	FL-SP-09
Target	IpGraph

Description

The addParentIp function overwrites the previous parent slots of the IP without checking existing parent relationships. This behavior deviates from the expected functionality, as it does not preserve the previous parents in a set-like structure, unlike the Solidity implementation. This could lead to future issues where parent relationships are unintentionally lost, potentially impacting the system's integrity.

story-geth/core/vm/ipgraph.go

```
...
func (c *ipGraph) addParentIp(input []byte, evm *EVM, ipGraphAddress common.Address)
([]byte, error) {
	...
	for i := 0; i < int(parentCount.Uint64()); i++ {
		parentIpId := common.BytesToAddress(input[96+i*32 : 96+(i+1)*32])
		index := uint64(i)
		slot := crypto.Keccak256Hash(ipId.Bytes()).Big()
		slot.Add(slot, new(big.Int).SetUint64(index))
		log.Info("addParentIp", "ipId", ipId, "parentIpId", parentIpId, "slot",
slot)
		evm.StateDB.SetState(ipGraphAddress, common.BigToHash(slot),
common.BytesToHash(parentIpId.Bytes()))
	}
    ...
}
```

Recommendations

- Implement a way to prevent the previous parents from being erased when adding a new parent by using the slot has a set of parents like in the solidity implementation.

X. Royalty policies are not properly validated during Derivative registration

Rating	Informational
ID	**FL-SP-10**
Target	**LicensingModule**

Description

The registerDerivativeWithLicenseTokens function of the LicensingModule is supposed to check that all royalty policies of the parent IPs are identical, but fails to do so. Here is the vulnerable code snippet:

contracts/modules/licensing/LicensingModule.sol

```
// Confirm that the royalty policies defined in all license terms of the parent IPs are
identical.
address[] memory rPolicies = new address[](parentIpIds.length);
uint32[] memory rPercents = new uint32[](parentIpIds.length);
for (uint256 i = 0; i < parentIpIds.length; i++) {
    (address royaltyPolicy, uint32 royaltyPercent, , ) =
lct.getRoyaltyPolicy(licenseTermsIds[i]);
    Licensing.LicensingConfig memory lsc = LICENSE_REGISTRY.getLicensingConfig(
        parentIpIds[i],
        licenseTemplate,
        licenseTermsIds[i]
    );
    if (lsc.isSet && lsc.commercialRevShare != 0) {
        royaltyPercent = lsc.commercialRevShare;
    }
    rPercents[i] = royaltyPercent;
    rPolicies[i] = royaltyPolicy;
}
```

The problem arises in the following check that assumes all the addresses of the rPolicies array are identical, and so only the first address is checked:

```
if (rPolicies.length != 0 && rPolicies[0] != address(0)) {
    // Notify the royalty module
    ROYALTY_MODULE.onLinkToParents(childIpId, parentIpIds, rPolicies, rPercents,
```

```
royaltyContext);
}
```

There could be a case where the rPolicies array starts with a zero address followed by other non-zero policies. This would allow to bypass the royalty registration for that derivative.

The same bug pattern is also present in the registerDerivative function.

Note that this issue is only informational because the current LicenseTemplate available prevents the registration of Derivatives with such incompatible LicenseTerms. Currently, the PILicenseTemplate::_verifyCompatibleLicenseTerms() does not allow to have different values of commercialUse in the Parents' License Terms and we cannot register Terms with both commercialUse set to true and royaltyPolicy at address(0).

However, this attack vector might become exploitable in the future with the registration of new LicenseTemplates.

Recommendations

- Implement a verification that all the addresses of rPolicies are identical, directly in the Licensing Module to not rely on the LicenseTemplate.

XI. Nested IpAccount signature verification in executeWithSig is broken in some cases

Rating	Informational
ID	**FL-SP-11**
Target	**IpAccount**

Description

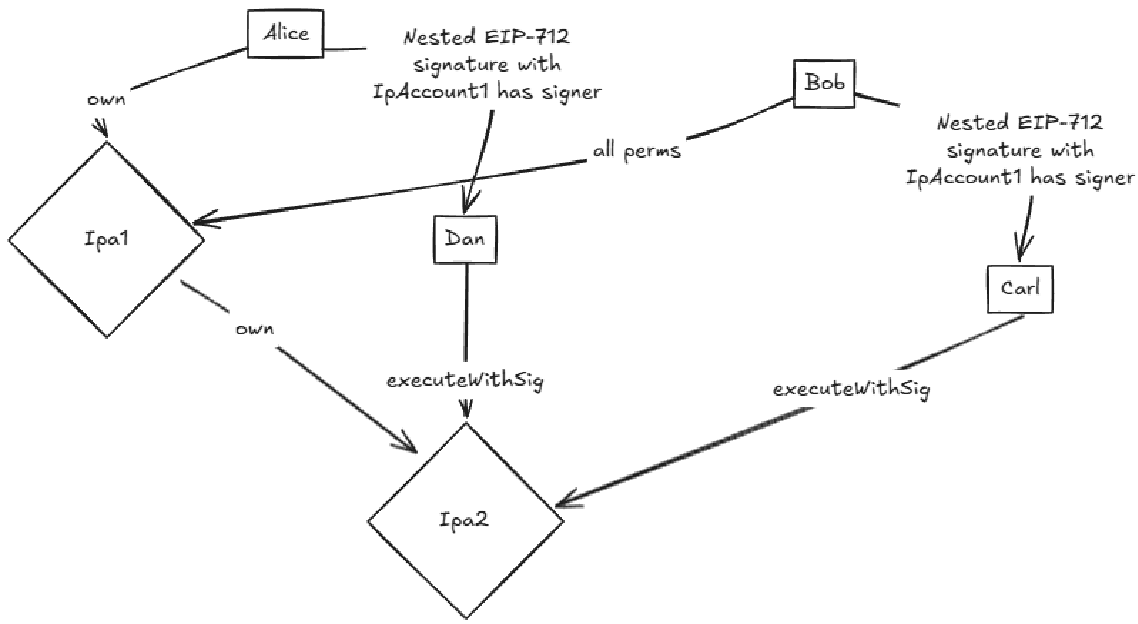
The IPAccount is a tailored implementation of the ERC-6551 (Token Bound Account) standard, incorporating a specific access control mechanism. As a result, certain functions must be overridden to integrate the required custom logic and ensure the implementation aligns with the unique requirements of the system.

- isValidSigner
- execute
- executeBatch

A new function, executeWithSig, has been introduced to enable transaction execution using a signature. However, the isValidSignature function has not been overridden to incorporate the custom access control mechanism. This omission may result in inconsistencies or vulnerabilities in signature validation, potentially undermining the security of the new functionality.

This could lead to several issues:

- isValidSignature returns invalid for signatures that are valid If someone attempts to check his signature using the function, it will return false even if the signature is valid.
- executeWithSig fails on specific signature verification workflow that involves EIP-1271



Alice owns IpAccount1 and IpAccount1 owns IpAccount2, Bob has full permission on IpAccount1.

Scenarios:

- Alice signs a valid nested EIP-712 signature for calling executeWithSig on IpAccount2 with IpAccount1 as the signer.
  Dan calls executeWithSig on IpAccount2 with the signature from Alice.
  It passes the signature verification and the call is successful.
- Bob signs a valid nested EIP-712 signature for calling executeWithSig on IpAccount2 with IpAccount1 as the signer.
  Carl calls executeWithSig on IpAccount2 with the signature from Bob.
  It fails the signature verification and the call is reverted.

Since IpAccount has implemented the EIP-1271 standard and executeWithSig use SignatureChecker.isValidSignatureNow to verify the signature which also checks the EIP-1271 standard. The second scenario should succeed.

Recommendations

- Overwrite the isValidSignature to implement a check against the EIP-1271 standard that involves the access control features.

XII. Nested IpAccount permissions are not updated on parent owner change

Rating	Informational
ID	**FL-SP-12**
Target	**IpAccount**

Description

Since the AccessController contract uses the direct owner of the IpAccount as key to store the permissions, when the owner of the IpAccount is changed, the permissions of his children are not updated leading to a potential access control issue.

This is the intended behavior.

XIII. Noncompliant ABI decoding

Rating	Informational
ID	**FL-SP-13**
Target	**IpGraph**

Description

The addParentIp makes a noncompliant decoding of the parents' array.

```
func (c *ipGraph) addParentIp(input []byte, evm *EVM, ipGraphAddress common.Address)
([]byte, error) {
	...
	ipId := common.BytesToAddress(input[0:32])
	log.Info("addParentIp", "ipId", ipId)
	parentCount := new(big.Int).SetBytes(getData(input, 64, 32))
	...
}
```

The addParentIp function in the ipGraph implementation does not adhere to ABI (Application Binary Interface) decoding standards for handling dynamic data structures, such as arrays. Specifically, it assumes a static offset location (input[0:32]) for decoding the ipId and uses a hardcoded offset to extract parentCount. This approach is problematic because it does not account for the dynamic nature of the ABI format, where the offset of data fields can vary depending on the structure of the input.

Recommendations

- Define an ABI then use the abigen package to generate Go binding that will make compliant decoding.

XIV. IPGraph gas cost can be improved

Rating	**Undetermined**
ID	**FL-SP-14**
Target	**IpGraph**

Description

The RequiredGas function currently returns a fixed gas value for each selector, regardless of the operation's input data or complexity. This approach is inaccurate, as gas costs should be dynamically calculated based on the size and complexity of the operation being executed. This fixed-value implementation may lead to inefficiencies or even failed transactions due to underestimation of required gas.

Recommendations

We recommend reviewing the gas cost calculation in the RequiredGas function to ensure that it accurately reflects the complexity of the operation.

addParentIp

- Currently, the addParentIp function returns a fixed gas cost of ipGraphWriteGas = 100. We recommend reviewing the gas cost of this operation to ensure that it accurately reflects the complexity of the operation by multiplying the gas cost by the number of parent IPs being added ipGraphWriteGas * len(parentIps).

hasParentIp

- Currently, the hasParentIp function returns a fixed gas cost of ipGraphReadGas * averageParentIpCount = 40. A potential fix could be reviewing the gas cost of this to ipGraphReadGas + ipGraphReadGas * len(currentLength).

getParentIps

- Same as hasParentIp.

getParentIpsCount

- Same as hasParentIp.

getAncestorIps

- Currently, the getAncestorIps function returns a fixed gas cost of ipGraphReadGas * averageAncestorIpCount * 2 = 600. A potential fix could be reviewing the gas cost of this to sum(ancestor => ipGraphReadGas + ipGraphReadGas * len(currentAncestorLength) * 2).

getAncestorIpsCount

- Same as getAncestorIps.

hasAncestorIps

- Same as getAncestorIps.

getRoyalty

- Currently, the getRoyalty function returns a fixed gas cost of ipGraphReadGas * (averageAncestorIpCount * 3) = 900 or ipGraphReadGas * (averageAncestorIpCount*2 + 2) = 620. A potential fix could be reviewing the gas cost of this to sum(ancestor => ipGraphReadGas + ipGraphReadGas * len(currentAncestorLength) * 3) and sum(ancestor => ipGraphReadGas + ipGraphReadGas * (len(currentAncestorLength) * 2 + 2)).

getRoyaltyStack

- Currently, the getRoyaltyStack function returns a fixed gas cost of ipGraphReadGas * (averageParentIpCount + 1) = 50 or ipGraphReadGas * (averageAncestorIpCount * 2) = 600. A potential fix could be reviewing the gas cost of this to ipGraphReadGas + ipGraphReadGas * (len(currentParentLength) + 1) and sum(ancestor => ipGraphReadGas + ipGraphReadGas * len(currentAncestorLength) * 2).

Conclusion

The security assessment of Story Protocol provided a detailed evaluation of its execution layer, consensus mechanisms, smart contracts, and associated modules. The review uncovered vulnerabilities ranging from critical issue to minor areas requiring improvement for long-term resilience.

Our audit of Story Protocol highlighted both its innovative potential and the challenges it faces. While several vulnerabilities were identified, the swift and proactive response from the Story Protocol team reflects their dedication to security and transparency. Their commitment to addressing these issues, combined with the protocol's robust framework for decentralized intellectual property management, sets Story Protocol apart as a strong contender in this evolving space.

Specific recommendations were outlined to mitigate these vulnerabilities, focusing on enhanced input validation, and stricter controls. Addressing these recommendations will strengthen the platform’s overall security posture and support its goal of becoming a reliable decentralized layer 1 blockchain designed specifically for intellectual property.

The Story team, by conducting regular audits, having already implemented fuzzing/invariants and unit tests, clearly demonstrates its goal of creating an innovative protocol with a high level of security and good practices in the development process.

Disclaimer

This report is provided under the terms of the agreement between FuzzingLabs and the client. It is intended solely for the client's use and may not be shared or referenced without FuzzingLabs' prior written consent.
The findings in this report are based on a point-in-time assessment and do not guarantee the absence of vulnerabilities or security flaws. FuzzingLabs cannot ensure future security as systems and threats evolve. We recommend continuous monitoring, independent assessments, and a bug bounty program.
This report is not financial, legal, or investment advice, and should not be used for decision-making regarding project involvement or investment. FuzzingLabs aims to help reduce risks, but we do not provide any guarantees regarding the complete security of the technology assessed.