
# Protocol Audit Report

Prepared by: AuditByte

# Table of Contents

# Protocol Summary

The Story protocol is making the legal system for creative Intellectual Property (IP) more efficient by turning IP "programmable" on the blockchain. To do this, Story Network is created: a purpose-built layer 1 blockchain where people or programs alike can license, remix, and monetize IP according to transparent terms set by creators themselves.

# Disclaimer

The AUDITBYTE team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

# Risk Classification

| | | **Impact** | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

# Audit Details

## Scope

Commit: 17eaf993cfc6dea113f2f25639115a5e3eed50ae

```
/contracts/src/protocol
```

### Roles

# Executive Summary

## Issues found

| Severity | Number of issues found |
|---|---|
| High | 5 |
| Medium | 8 |
| Low | 10 |
| Info | 3 |
| Total | 26 |

# Findings

## High

### [H-1] Missing validation for `commissionRate` larger than `maxCommissionRate` in `IPTokenStaking::updateValidatorCommission` can cause operation fail

**Description:** The function does not check if the new `commissionRate` exceeds the `maxCommissionRate`. According to the documentation, If the updated commission rate is larger than max commission rate or the commission rate change delta is larger than max commission rate change, the operation will fail.

```
    function updateValidatorCommission(
        bytes calldata validatorUncmpPubkey,
        uint32 commissionRate
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender)
chargesFee {
        require(commissionRate >= minCommissionRate, "IPTokenStaking:
Commission rate under min");
        emit UpdateValidatorCommssion(validatorUncmpPubkey,
commissionRate);
    }
```

**Impact:** As a result of that, validators can set an unreasonably high commision rate and this could lead to exploitative scenarios where validators overcharge delegators. Delegators may suffer financial losses due to

excessive fees.

**Proof of Code:**

In the proof of code below, `CreateValidator` function was called with a `maxCommissionRate` of 5000, then `UpdateValidatorCommssion` function was called with `commissionRate` of 100000000 which is greater than maximum commission rate previously set and it was executed successfully which against documentation.

▶ Code

```
    function testIPTokenStaking_updateValidatorMaxCommission() public {
        // uint256 stakeAmount = 0.5 ether;
        bytes memory validatorUncmpPubkey = delegatorUncmpPubkey;
        // Network shall allow anyone to create a new validator by staking
validator's own tokens (self-delegation)
        uint256 stakeAmount = ipTokenStaking.minStakeAmount();
        vm.deal(delegatorAddr, stakeAmount);
        vm.prank(delegatorAddr);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.CreateValidator(
            validatorUncmpPubkey,
            "delegator's validator",
            stakeAmount,
            1000,
            5000,
            100,
            1, // supportsUnlocked
            delegatorAddr,
            abi.encode("data")
        );
        ipTokenStaking.createValidator{ value: stakeAmount }({
            validatorUncmpPubkey: delegatorUncmpPubkey,
            moniker: "delegator's validator",
            commissionRate: 1000,
            maxCommissionRate: 5000,
            maxCommissionChangeRate: 100,
            supportsUnlocked: true,
            data: abi.encode("data")
        });

        uint32 commissionRate = 100000000;
        uint256 feeAmount = ipTokenStaking.fee();
        vm.deal(delegatorAddr, feeAmount * 10);
        vm.prank(delegatorAddr);
        vm.expectEmit(address(ipTokenStaking));
        emit
IIPTokenStaking.UpdateValidatorCommssion(delegatorUncmpPubkey,
commissionRate);
        ipTokenStaking.updateValidatorCommission{ value: feeAmount }
(delegatorUncmpPubkey, commissionRate);
    }
```

**Recommended Mitigation:** There should be check to ensure the new `commissionRate` does not exceed `maxCommissionRate`.

```
    function updateValidatorCommission(
        bytes calldata validatorUncmpPubkey,
        uint32 commissionRate
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender)
chargesFee {
        require(commissionRate >= minCommissionRate, "IPTokenStaking:
Commission rate under min");
+       require(commissionRate <= maxCommissionRate, "IPTokenStaking:
Commission rate exceeds max");
        emit UpdateValidatorCommssion(validatorUncmpPubkey,
commissionRate);
    }
```

### [H-2] Missing check for validator existence in `IPTokenStaking::_stake` function can cause delegator to lose tokens.

**Description:** There should be a check in `IPTokenStaking::_stake` for validator existence because according to the natspec the function should creates a validator if it does not exist, then delegates the stake to the validator. According to the documentation, if a delegator delegates to a non-existent validator, the tokens will NOT be refunded.

**Impact:** If the validator does not exist and a delegator delegates to it, the delegator will lose the tokens and will not be refunded. This will lead to financial losses for delegators or users.

**Proof of Concepts:**

It is confirmed in the proof of code below that user stake with unexisted validator without checking for the existence of the validator before.

```
    function testIPTokenStaking_stakeWithoutValidator() public {
        bytes memory delegatorUncmpPubkeyx =
hex"04e38d15ae6cc5d41cce27a2307903cb12a406cbf463fe5fef215bdf8aa988ced195e9
327ac89cd362eaa0397f8d7f007c02b2a75642f174e455d339e4a1efe47b"; // pragma:
allowlist-secret
        // Address matching delegatorCmpPubkey
        address delegatorAddrx =
address(0xf398C12A45Bc409b6C652E25bb0a3e702492A4ab);

        // Flexible should produce 0 delegationId
        bytes memory validatorPubkey = delegatorUncmpPubkeyx;
        IIPTokenStaking.StakingPeriod stkPeriod =
IIPTokenStaking.StakingPeriod.FLEXIBLE;
        vm.deal(delegatorAddrx, 10000 ether);
        vm.prank(delegatorAddrx);
```

```
        uint256 delegationId = ipTokenStaking.stake{ value: 1024 ether }(
            delegatorUncmpPubkeyx,
            validatorPubkey,
            stkPeriod,
            ""
        );
        assertEq(delegationId, 0);
    }
```

**Recommended Mitigation:** There should be a check for the validator existence before delegating to it.

- Add validation to check if the validator exists before proceeding with the delegation.
- If the validator does not exist, revert the transaction and ensure that no tokens are transferred.
- Use a mapping of existing validators (e.g., isValidator), which can be checked before proceeding with the staking operation.

```
+   mapping(bytes => bool) private isValidator;
.
.
.
    function _stake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        IIPTokenStaking.StakingPeriod stakingPeriod,
        bytes calldata data
    ) internal returns (uint256) {
        // This can't be tested from Foundry (Solidity), but can be
triggered from js/rpc
        require(stakingPeriod <= IIPTokenStaking.StakingPeriod.LONG,
"IPTokenStaking: Invalid staking period");
        (uint256 stakeAmount, uint256 remainder) =
roundedStakeAmount(msg.value);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake
amount under min");

+       require(isValidator[validatorUncmpPubkey], "IPTokenStaking:
Validator does not exist");

        uint256 delegationId = 0;
        if (stakingPeriod != IIPTokenStaking.StakingPeriod.FLEXIBLE) {
            delegationId = ++_delegationIdCounter;
        }
        emit Deposit(
            delegatorUncmpPubkey,
            validatorUncmpPubkey,
            stakeAmount,
            uint8(stakingPeriod),
            delegationId,
            msg.sender,
            data
        );
```

```
        // We burn staked tokens
        payable(address(0)).transfer(stakeAmount);

        if (remainder > 0) {
            _refundRemainder(remainder);
        }

        return delegationId;
    }
```

Add the code below in the `IPTokenStaking::_createValidator` function before making external call

```
isValidator[validatorUncmpPubkey] = true;
```

### [H-3] Missing zero address checks in `IPTokenStaking::setOperator` can cause loss of operator's access.

**Description:** In `IPTokenStaking::setOperator` function, there is missing zero address check for `operator` which can cause the intended operator not to have access to redelegate or unstake on delegator's behalf contrary to the documentation.

```
    function setOperator(
        bytes calldata uncmpPubkey,
        address operator
    ) external payable verifyUncmpPubkeyWithExpectedAddress(uncmpPubkey,
msg.sender) chargesFee {
        emit SetOperator(uncmpPubkey, operator);
    }
```

**Impact:** If the operator address is wrongly set, this can cause the intended operator not to have access or right to unstake or redelegate on delegator's behalf.

**Proof of Concepts:**

In the proof of code below, the `operator` address was set to zero address but due to lack of zero address check for the operator in the `IPTokenStaking::setOperator` function, the test successfully passed which can cause loss of access by the intended operator.

Insert the code in `IPTokenStaking.t.sol`

```
    function testIPTokenStaking_setOperatorWithZeroAddress() public {
        address operator =
address(0x0000000000000000000000000000000000000000);
        // Network should allow delegators to add operators for themselves
        uint256 feeAmount = 1 ether;
```

```
        vm.deal(delegatorAddr, feeAmount);
        vm.prank(delegatorAddr);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.SetOperator(delegatorUncmpPubkey, operator);
        ipTokenStaking.setOperator{ value: feeAmount }
(delegatorUncmpPubkey, operator);
    }
```

**Recommended Mitigation:** It is advised to add the missing zero address checks for `operator` as this can cause loss of access to unstake or redelegate operation.

```
    function setOperator(
        bytes calldata uncmpPubkey,
        address operator
    ) external payable verifyUncmpPubkeyWithExpectedAddress(uncmpPubkey,
msg.sender) chargesFee {
+       require(operator != address(0), "IPTokenStaking: Zero address not
allowed");
        emit SetOperator(uncmpPubkey, operator);
    }
```

### [H-4] Missing zero address checks in `IPTokenStaking::setWithdrawalAddress` can cause loss of funds.

**Description:** In `IPTokenStaking::setWithdrawalAddress` function, there is missing zero address check for `newWithdrawalAddress`. This can cause loss of fund if the address is wrongly set, for example assuming the `newWithdrawalAddress` is set to zero address, any fund send to it will be lost.

```
    function setWithdrawalAddress(
        bytes calldata delegatorUncmpPubkey,
        address newWithdrawalAddress
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey, msg.sender)
chargesFee {
        emit SetWithdrawalAddress({
            delegatorUncmpPubkey: delegatorUncmpPubkey,
            executionAddress:
bytes32(uint256(uint160(newWithdrawalAddress))) // left-padded bytes32 of
the address
        });
    }
```

**Impact:** If the `newWithdrawalAddress` is set to zero address, any fund or amount send to it will be lost.

**Proof of Concepts:**

Insert the code below in `IPTokenStaking.t.sol`

```
    function testIPTokenStaking_SetZeroWithdrawalAddress() public {
        uint256 feeAmount = ipTokenStaking.fee();
        address zeroWithdrawalAddress = address(0x000);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.SetWithdrawalAddress(
            delegatorUncmpPubkey,

0x0000000000000000000000000000000000000000000000000000000000000000
        );
        vm.prank(delegatorAddr);
        ipTokenStaking.setWithdrawalAddress{ value: feeAmount }
(delegatorUncmpPubkey, zeroWithdrawalAddress);

    }
```

**Recommended Mitigation:** It is advised to add the missing zero address checks for `newWithdrawalAddress` as this can cause loss of funds or asset sent to it.

```
    function setRewardsAddress(
        bytes calldata delegatorUncmpPubkey,
        address newRewardsAddress
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey, msg.sender)
chargesFee {
+       require(newWithdrawalAddress != address(0), "IPTokenStaking: Zero
address not allowed");
        emit SetRewardAddress({
            delegatorUncmpPubkey: delegatorUncmpPubkey,
            executionAddress: bytes32(uint256(uint160(newRewardsAddress)))
// left-padded bytes32 of the address
        });
    }
```

## [H-5] Missing zero address checks in `IPTokenStaking::setRewardsAddress` can cause loss of funds.

**Description:** In `IPTokenStaking::setRewardsAddress` function, there is missing zero address check for `newRewardsAddress`. This can cause loss of fund if the address is wrongly set, for example assuming the `newRewardsAddress` is set to zero address, any reward send to it will be lost.

```
    function setRewardsAddress(
        bytes calldata delegatorUncmpPubkey,
        address newRewardsAddress
    ) external payable
```

```
verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey, msg.sender)
chargesFee {
        emit SetRewardAddress({
            delegatorUncmpPubkey: delegatorUncmpPubkey,
            executionAddress: bytes32(uint256(uint160(newRewardsAddress)))
// left-padded bytes32 of the address
        });
    }
```

**Impact:** If the `newRewardsAddress` is set to zero address, any fund or reward send to it will be lost.

**Proof of Concepts:**

The proof of code below show that network allow the delegators to set zero withdraw address which can result in loss of funds or rewards if withdrawal is processed.

Insert the code below in `IPTokenStaking.t.sol`

```
    function testIPTokenStaking_SetZeroRewardAddress() public {
        uint256 feeAmount = ipTokenStaking.fee();
        address zeroRewardAddress = address(0x000);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.SetRewardAddress(
            delegatorUncmpPubkey,

0x0000000000000000000000000000000000000000000000000000000000000000
        );
        vm.prank(delegatorAddr);
        ipTokenStaking.setRewardsAddress{ value: feeAmount }
(delegatorUncmpPubkey, zeroRewardAddress);
    }
```

**Recommended Mitigation:** Add a validation zero address check for `newRewardsAddress` variables as this can cause loss of funds or reward sent to it.

```
    function setRewardsAddress(
        bytes calldata delegatorUncmpPubkey,
        address newRewardsAddress
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey, msg.sender)
chargesFee {
+       require(newRewardsAddress != address(0), "IPTokenStaking: Zero
address not allowed");
        emit SetRewardAddress({
            delegatorUncmpPubkey: delegatorUncmpPubkey,
            executionAddress: bytes32(uint256(uint160(newRewardsAddress)))
// left-padded bytes32 of the address
        });
```

```
    }
```

## Medium

### [M-1] Missing Check for duplicate validator creation

**Description:** The `IPTokenStaking::createValidator` function, according to documentation is expected to ignore calls made by a validator attempting to create validator instance second time. However, the current implementation does not use this rule. There is no check to prevent a validator from calling the function multiple times.

**Impact:** This cause discrepancy between documented behaviour and actual implementation. Validators could misuse this functionality to create multiple validator entries, potentially leading to unforeseen vulnerabilities.

**Proof of Concepts:** The proof of code attempt to demonstrate creating a validator multiple times which is confirmed to execute second time and it opposed the documentation.

Insert the code below in `IPTokenStaking.t.sol`

▶ Code

```
    function testIPTokenStaking_CreateMultipleValidator() public {
        uint256 stakeAmount = 0.5 ether;
        bytes memory validatorUncmpPubkey = delegatorUncmpPubkey;
        stakeAmount = ipTokenStaking.minStakeAmount();
        vm.deal(delegatorAddr, stakeAmount);
        vm.startPrank(delegatorAddr);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.CreateValidator(
            validatorUncmpPubkey,
            "delegator's validator",
            stakeAmount,
            1000,
            5000,
            100,
            1, // supportsUnlocked
            delegatorAddr,
            abi.encode("data")
        );
        ipTokenStaking.createValidator{ value: stakeAmount }({
            validatorUncmpPubkey: delegatorUncmpPubkey,
            moniker: "delegator's validator",
            commissionRate: 1000,
            maxCommissionRate: 5000,
            maxCommissionChangeRate: 100,
            supportsUnlocked: true,
            data: abi.encode("data")
        });
        vm.stopPrank();
```

```
        vm.deal(delegatorAddr, stakeAmount);
        vm.startPrank(delegatorAddr);
        vm.expectEmit(address(ipTokenStaking));
        emit IIPTokenStaking.CreateValidator(
            validatorUncmpPubkey,
            "delegator's validator",
            stakeAmount,
            1000,
            5000,
            100,
            1, // supportsUnlocked
            delegatorAddr,
            abi.encode("data")
        );
        ipTokenStaking.createValidator{ value: stakeAmount }({
            validatorUncmpPubkey: delegatorUncmpPubkey,
            moniker: "delegator's validator",
            commissionRate: 1000,
            maxCommissionRate: 5000,
            maxCommissionChangeRate: 100,
            supportsUnlocked: true,
            data: abi.encode("data")
        });
    }
```

**Recommended Mitigation:** Use a mapping variable to check whether the caller has already created a validator.

```
+   mapping(address => bool) private isValidator;
.
.
.
    function createValidator(
        bytes calldata validatorUncmpPubkey,
        string calldata moniker,
        uint32 commissionRate,
        uint32 maxCommissionRate,
        uint32 maxCommissionChangeRate,
        bool supportsUnlocked,
        bytes calldata data
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender)
nonReentrant {

+       require(!isValidator[msg.sender], "Validator already exists");

+       isValidator[msg.sender] = true;

        _createValidator(
            validatorUncmpPubkey,
```

```
            moniker,
            commissionRate,
            maxCommissionRate,
            maxCommissionChangeRate,
            supportsUnlocked,
            data
        );
    }
```

### [M-2] Missing check for the remaining balance during redelegation in `IPTokenStaking::_redelegate` allows token below threshold to be omitted.

**Description:** According to the documentation, if the remaining balance after redelegation is less than 1024 IP, all remaining tokens will be redelegated together but in `IPTokenStaking::_redelegation` funtion there is no check for remaining token less than minimum (1024IP) to be redelegated together.

**Impact:** This can cause remaining balance less than minimum amount after redelegation not to be redelegated together which is wrong according to the documentation.

**Proof of Concepts:**

**Recommended Mitigation:** Adding a check to know if the remaining balance after redelegation is less than minimum amount.

```
    function _redelegate(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpSrcPubkey,
        bytes calldata validatorUncmpDstPubkey,
        uint256 delegationId,
        uint256 amount
    ) private {
        require(
            keccak256(validatorUncmpSrcPubkey) !=
keccak256(validatorUncmpDstPubkey),
            "IPTokenStaking: Redelegating to same validator"
        );
-       (uint256 stakeAmount, ) = roundedStakeAmount(amount);
+       (uint256 stakeAmount, remainder) = roundedStakeAmount(amount);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake
amount under min");
        require(delegationId <= _delegationIdCounter, "IPTokenStaking:
Invalid delegation id");
+       if (remainder < minStakeAmount) {
            stakeAmount += remainder;
        }

        emit Redelegate(
            delegatorUncmpPubkey,
            validatorUncmpSrcPubkey,
            validatorUncmpDstPubkey,
            delegationId,
```

```
            msg.sender,
            stakeAmount
        );
    }
```

### [M-3] Missing reentrancy guard in `IPTokenStaking::unstake` function

**Description:** `IPTokenStaking::_unstake` function emit `Withdraw` event which is responsible for withdrawing the unstaking amount. It is neccessary for the withdraw action to make an external call, inorder to perform a safe operation in making an external call that is free from reentrancy attack, it is always advisable to use `nonReentrant` guard from openzeppelin.

**Impact:** If the function is prone to rentrancy attack, malicious user or attacker can drain all the funds in the contract by repeatedly calling the `unstake` function multiple time in a single transaction.

**Proof of Concepts:**

Below is the `unstake` function in `IPTokenStaking`

```
    function _unstake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        uint256 delegationId,
        uint256 amount,
        bytes calldata data
    ) private {
        require(delegationId <= _delegationIdCounter, "IPTokenStaking:
Invalid delegation id");
        require(amount >= minUnstakeAmount, "IPTokenStaking: Unstake
amount under min");
        require(amount % STAKE_ROUNDING == 0, "IPTokenStaking: Amount must
be rounded to STAKE_ROUNDING");
        emit Withdraw(delegatorUncmpPubkey, validatorUncmpPubkey, amount,
delegationId, msg.sender, data);
    }
```

The fields in the `withdraw` event is copied below

```
    event Withdraw(
        bytes delegatorUncmpPubkey,
        bytes validatorUncmpPubkey,
        uint256 stakeAmount,
        uint256 delegationId,
        address operatorAddress,
        bytes data
    );
```

Note in the `withdraw` event above there is `operatorAddress` field which can be an attacker contract address, if not correctly handle on chain it can expose the contract to reentrancy attack.

**Recommended Mitigation:** It is advised to add `nonReentrant` guard (modifier) to the `IPTokenStaking::unstake` function.

```
    function unstake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        uint256 delegationId,
        uint256 amount,
        bytes calldata data
    )
        external
        payable
+       nonReentrant
        verifyUncmpPubkeyWithExpectedAddress(delegatorUncmpPubkey,
msg.sender)
        verifyUncmpPubkey(validatorUncmpPubkey)
        chargesFee
    {
        _unstake(delegatorUncmpPubkey, validatorUncmpPubkey, delegationId,
amount, data);
    }
```

### [M-4] Initializers could be front-run in `UpgradeEntrypoint` contract

**Description:** Assuming `UpgradeEntryPoint.sol` contract was deployed but it was not initialized immediately or there was delay in initialization through `UpgradeEntryPoint::initialize` function, then somebody else initialize the contract with his own address. This result in front-running situation where an attacker takes advantage of information about a transaction that has not yet been mined and submits their own transaction to execute it first, allowing attacker to set ownership of the contract.

```
    function initialize(address owner) public initializer {
        require(owner != address(0), "UpgradeEntrypoint: owner cannot be
zero address");
        __Ownable_init(owner);
    }
```

**Impact:** An attacker could front-run the contract, allowing it to set the owner of the contract with intended or any address or stealing the ownership of the contract.

**Recommended Mitigation:**

- Do not forget to initialize the contract immediately it is deployed and always make sure the initialization is very fast to prevent front-run attack.

- Implement access control for the `initialize` function to ensure that only authorized accounts can call it.

### [M-5] Missing check for `maxUBIPercentage` in `UBIPool` constructor allow setting UBI percentage over the maximum.

**Description:** According to the documentation, maximum UBI percentage that can be set is 20% but in the code implementation the `constructor` does not have the check for `maxUBIPercentage` greater than 20%. This can cause setting `maxUBIPercentage` above the intended maximum amount which is against the documentation plan.

**Impact:** Due to this, if the `maxUBIPercentage` is given portion of newly minted token more than maximum UBI percentage, this can cause validators to be over incentivized and affected the amount of available minted token.

```
    constructor(uint32 maxUBIPercentage) {
        MAX_UBI_PERCENTAGE = maxUBIPercentage;
        _disableInitializers();
    }
```

**Proof of Concepts:**

The maximum UBI percentage that can be set is 20% according to the documentation but from the code below 25% is set and accepted.

Insert code below in the `UBIPool.t.sol` file

```
    function test_recommendedMaxUbiPercntage20_00() public {
        UBIPool ubiPoolContract;
        uint32 RECOMMENDED_MAX_UBI_PERCENTAGE = 20_00;
        ubiPoolContract = new UBIPool(25_00);
        assert(ubiPoolContract.MAX_UBI_PERCENTAGE() !=
RECOMMENDED_MAX_UBI_PERCENTAGE);
    }
```

**Recommended Mitigation:** It is a good pattern to add a check for `maxUBIPercentage` not to be more than the intended maximum percentage specified by the documentation.

```
    constructor(uint32 maxUBIPercentage) {
+        require(maxUBIPercentage <= 20, "UBI: Percentage exceeds maximum
limit 20");
        MAX_UBI_PERCENTAGE = maxUBIPercentage;
        _disableInitializers();
    }
```

### [M-6] Centralization risk for trusted owners in `UpgradeEntrypoint::planUpgrade` function.

**Description:** In `UpgradeEntrypoint::planUpgrade` function, owners have privilege rights to perform admin tasks and these owners need to be trusted not to perform malicious updates or drain funds.

```
    function planUpgrade(string calldata name, int64 height, string
calldata info) external onlyOwner {
        emit SoftwareUpgrade({ name: name, height: height, info: info });
    }
```

**Impact:** Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (1):

```
function planUpgrade(string calldata name, int64 height, string calldata
info) external onlyOwner {
```

**Recommended Mitigation:** Ensure only trusted owner is permitted to run `UpgradeEntrypoint::planUpgrade` function to protect the contract from malicious attack.

### [M-7] Missing reentrancy guard in `IPTokenStaking::unstakeOnBehalf` function

**Description:** `IPTokenStaking::_unstake` function emit `Withdraw` event which is responsible for withdrawing the unstaking amount. It is neccessary for the withdraw action to make an external call, inorder to perform a safe operation in making an external call that is free from reentrancy attack, it is always advisable to use `nonReentrant` guard from openzeppelin.

**Impact:** If the `IPTokenStaking::_unstake` function is prone to rentrancy attack, malicious user or attacker can drain all the funds in the contract by repeatedly calling the `unstake` function multiple time in a single transaction.

**Proof of Concepts:**

Below is the `unstake` function in `IPTokenStaking` contract.

```
    function _unstake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        uint256 delegationId,
        uint256 amount,
        bytes calldata data
    ) private {
        require(delegationId <= _delegationIdCounter, "IPTokenStaking:
Invalid delegation id");
        require(amount >= minUnstakeAmount, "IPTokenStaking: Unstake
```

```
amount under min");
        require(amount % STAKE_ROUNDING == 0, "IPTokenStaking: Amount must
be rounded to STAKE_ROUNDING");
        emit Withdraw(delegatorUncmpPubkey, validatorUncmpPubkey, amount,
delegationId, msg.sender, data);
    }
```

The fields in the `withdraw` event is copied below

```
    event Withdraw(
        bytes delegatorUncmpPubkey,
        bytes validatorUncmpPubkey,
        uint256 stakeAmount,
        uint256 delegationId,
        address operatorAddress,
        bytes data
    );
```

Note in the `withdraw` event above there is `operatorAddress` field which can be an attacker contract address, if not correctly handle on chain it can expose the contract to reentrancy attack.

**Recommended Mitigation:** It is advised to add `nonReentrant` guard (modifier) to the `IPTokenStaking::unstakeOnBehalf` function.

### [M-8] Centralization risk for trusted owners in `UpgradeEntrypoint::cancelUpgrade` function

**Description:** In `UpgradeEntrypoint::cancelUpgrade`, owners have privilege rights to perform admin tasks and these owners need to be trusted not to perform malicious updates by cancelling upgrade that is not supposed to cancel.

**Impact:** Contract has owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates of cancelling upgrade that is needed in `cancelUpgrade` function.

Instances (1):

```
function cancelUpgrade() external onlyOwner {
```

**Recommended Mitigation:** Ensure only trusted owner is permitted to run `UpgradeEntrypoint::cancelUpgrade` function to protect the contract from malicious attack of cancelling the upgrade.

## Low

### [L-1] `IPTokenStaking` can be initialized by anyone due to front-running

**Description:** The `initialize` function within the `IPTokenStaking.sol` is vulnerable to front-running. This function allows an attacker to make external call to it, set their own values and take ownership of the contract.

```
    function initialize(IIPTokenStaking.InitializerArgs calldata args)
public initializer {
        __ReentrancyGuard_init();
        __Ownable_init(args.owner);
        _setMinStakeAmount(args.minStakeAmount);
        _setMinUnstakeAmount(args.minUnstakeAmount);
        _setMinCommissionRate(args.minCommissionRate);
        _setFee(args.fee);
    }
```

**Impact:** An attacker could front-run the intended owner and initialize the contract with their own parameters. This could lead to changing the ownership of the contract, changing the contract's state variable like `minStakeAmount`, `minUnstakeAmount`, `minCommissionRate`, `fee`.

**Proof of Concepts:**

**Recommended Mitigation:** Implement access control for the `initialize` function to ensure that only authorized accounts can call it.

### [L-2] Centralization risk for trusted owners

**Description:**

**Impact:** Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (4):

```
    File: /src/protocol/IPTokenStaking.sol

    function setMinStakeAmount(uint256 newMinStakeAmount) external
onlyOwner {
    function setMinUnstakeAmount(uint256 newMinUnstakeAmount) external
onlyOwner {
    function setFee(uint256 newFee) external onlyOwner {
    function setMinCommissionRate(uint256 newValue) external onlyOwner {
```

**Proof of Concepts:**

**Recommended Mitigation:** There should be proper monitoring of all processes and actions carried out in the protocol.

### [L-3] Miscrepancy in `IPTokenStaking::_createValidator` function and the documentation

**Description:** According to the documentation the initial staking amount needs to be larger than a threshold but the `_createValidator` function checks for `stakeAmount` less than or equivalent to `minStakeAmount`

**Impact:** If the staking amount is exactly the same as threshold, this will result to violation of minimum staking amount rule. This discrepancy could reduce the effectiveness and undermines user trust in the protocol.

**Proof of Concepts:**

**Recommended Mitigation:** For safe operation of the `IPTokenStaking::_createValidator` function, either the `minStakeAmount` state should be larger than a threshold according to documentation or `stakeAmount` greater than (>) `minStakeAmount`

```
function _createValidator(
        bytes calldata validatorUncmpPubkey,
        string memory moniker,
        uint32 commissionRate,
        uint32 maxCommissionRate,
        uint32 maxCommissionChangeRate,
        bool supportsUnlocked,
        bytes calldata data
    ) internal {
        (uint256 stakeAmount, uint256 remainder) =
roundedStakeAmount(msg.value);
-       require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake
amount under min");
+       require(stakeAmount > minStakeAmount, "IPTokenStaking: Stake
amount under min");
        require(commissionRate >= minCommissionRate, "IPTokenStaking:
Commission rate under min");
        require(commissionRate <= maxCommissionRate, "IPTokenStaking:
Commission rate over max");
        require(bytes(moniker).length <= MAX_MONIKER_LENGTH,
"IPTokenStaking: Moniker length over max");

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

```
    }
```

### [L-4] Missing check in `IPTokenStaking::_redelegate` function for the source and destination validators supported token type

**Description:** The documentation stated that redelegation can only be triggered when the source and destination validators support the same token type, but there is no check for that in the `_redelegate` function

```
    function _redelegate(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpSrcPubkey,
        bytes calldata validatorUncmpDstPubkey,
        uint256 delegationId,
        uint256 amount
    ) private {
        require(
            keccak256(validatorUncmpSrcPubkey) !=
keccak256(validatorUncmpDstPubkey),
            "IPTokenStaking: Redelegating to same validator"
        );
        (uint256 stakeAmount, ) = roundedStakeAmount(amount);
        require(stakeAmount >= minStakeAmount, "IPTokenStaking: Stake
amount under min");
        require(delegationId <= _delegationIdCounter, "IPTokenStaking:
Invalid delegation id");

        emit Redelegate(
            delegatorUncmpPubkey,
            validatorUncmpSrcPubkey,
            validatorUncmpDstPubkey,
            delegationId,
            msg.sender,
            stakeAmount
        );
    }
```

**Impact:** This will result to incorrect handling of the user request by invoking the `redelegate` function when not supposed to, incase there is source and destination validators token type mismatch. This alter the normal behaviour of the protocol according to the documentation.

**Proof of Concepts:**

**Recommended Mitigation:** Validate token type compatibility by ensuring the source and destination validators support the same token type in the `IPTokenStaking::_redelegate` function

### [L-5] Initializers in `UBIPool` contract could be front-run

**Description:** Assuming `UBIPool` contract was deployed but it was not initialized immediately or there was delay in initialization through `UBIPool::initialize` function, then somebody else initialize the contract with his own address. This result in front-running situation where an attacker takes advantage of information about a transaction that has not yet been mined and submits their own transaction to execute it first, allowing attacker to set ownership of the contract.

```
    function initialize(address owner) public initializer {
        require(owner != address(0), "UBIPool: owner cannot be zero
address");
        __Ownable_init(owner);
    }
```

**Impact:** An attacker could front-run the contract, allowing it to set the owner of the contract with intended or any address or stealing the ownership of the contract.

**Proof of Concepts:**

**Recommended Mitigation:**

- Do not forget to initialize the contract immediately it is deployed and always make sure the initialization is very fast to prevent front-run attack.
- Implement access control for the `initialize` function to ensure that only authorized accounts can call it.

### [L-6] Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

▶ 3 Found Instances

- Found in src/protocol/IPTokenStaking.sol Line: 59

```
        payable(address(0x0)).transfer(msg.value);
```

- Found in src/protocol/IPTokenStaking.sol Line: 263

```
        payable(address(0)).transfer(stakeAmount);
```

- Found in src/protocol/IPTokenStaking.sol Line: 398

```
        payable(address(0)).transfer(stakeAmount);
```

### [L-7] The `nonReentrant modifier` should occur before all other modifiers

This is a best-practice to protect against reentrancy in other modifiers.

▶ 3 Found Instances

- Found in src/protocol/IPTokenStaking.sol Line: 225

```
    ) external payable
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender)
nonReentrant {
```

- Found in src/protocol/IPTokenStaking.sol Line: 329

```
        nonReentrant
```

- Found in src/protocol/IPTokenStaking.sol Line: 355

```
        nonReentrant
```

### [L-8] Loop contains `require`/`revert` statements

Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

▶ 1 Found Instances

- Found in src/protocol/UBIPool.sol Line: 80

```
        for (uint256 i = 0; i < amounts.length; i++) {
```

### [L-9]: Costly operations inside loops.

Invoking `SSTORE`operations in loops may lead to Out-of-gas errors. Use a local variable to hold the loop computation result.

▶ 1 Found Instances

- Found in src/protocol/UBIPool.sol Line: 80

```
        for (uint256 i = 0; i < amounts.length; i++) {
```

### [L-10]: State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

▶ 1 Found Instances

- Found in src/protocol/UBIPool.sol Line: 98

```
    function claimUBI(
```

# Informational

## [I-1] Missing check for the unstake amount larger than total unstakable tokens in `IPTokenStaking::_unstake` function

**Description:** According to the documentation, if the unstake amount passed in is larger than the total unstakable tokens, the current total unstakable amounts will be unstaked but there is no check to confirm unstake amount passed and this result in unexpected error and makes the contract vulnerables to attacker.

```
function _unstake(
        bytes calldata delegatorUncmpPubkey,
        bytes calldata validatorUncmpPubkey,
        uint256 delegationId,
        uint256 amount,
        bytes calldata data
    ) private {
        require(delegationId <= _delegationIdCounter, "IPTokenStaking:
Invalid delegation id");
        require(amount >= minUnstakeAmount, "IPTokenStaking: Unstake
amount under min");
        require(amount % STAKE_ROUNDING == 0, "IPTokenStaking: Amount must
be rounded to STAKE_ROUNDING");
        emit Withdraw(delegatorUncmpPubkey, validatorUncmpPubkey, amount,
delegationId, msg.sender, data);
    }
```

**Impact:** This can result in unexpected error. If not correctly handle on the consensus chain, it can make the contract vulnerables to the attacker and drain the wallet by supplying fake amount. This will have effect on both `unstake` and `unstakeOnBehalf` function, if not properly handle.

**Recommended Mitigation:** It is good to add a check for the unstake amount larger than total unstakable tokens.

## [I-2] Sending fee to zero address in `IPTokenStaking::chargesFee` modifier burns all the fee.

**Description:** fee collected from an account using `msg.value` is sent to zero address which actually burns it without profiting the protocol. From the `constructor` DEFAULT_MIN_FEE is greater than 1 gwei =

0.000000009 Ether, and in `IPTokenStaking::_setFee` function "fee" is set to greater than DEFAULT_MIN_FEE.

```
    modifier chargesFee() {
        require(msg.value == fee, "IPTokenStaking: Invalid fee amount");
        payable(address(0x0)).transfer(msg.value);
        _;
    }
```

**Impact:** Sending any fund to zero address means burning the fund or any token send to it.

**Proof of Concepts:**

**Recommended Mitigation:** `chargesFee` modifier needs to pay proper attention to, because it is been used in many functions and constantly keeps on sending funds or any value to zero address.

### [I-3] Check effects Interactions Pattern not being followed in `UBIPool::claimUBI` function.

**Description:** Checks effects interactions(CEI) is not being followed in the `claimUBI` function of `UBIPool.sol`. There is external call being made before updating the `totalPendingClaims` variable. It is always a good pattern to follow Check Effect Interactions(CEI) by updating all the state first before making an external call.

```
    function claimUBI(
        uint256 distributionId,
        bytes calldata validatorUncmpPubkey
    ) external nonReentrant
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender) {
        uint256 amount = validatorUBIAmounts[distributionId]
[validatorUncmpPubkey];
        require(amount > 0, "UBIPool: no UBI to claim");
        validatorUBIAmounts[distributionId][validatorUncmpPubkey] = 0;
        (bool success, ) = msg.sender.call{ value: amount }("");
        require(success, "UBIPool: failed to send UBI");
        totalPendingClaims -= amount;
    }
```

**Impact:** It does not make the function follow a standard way of preventing reentrancy attack by follow Check Effect Interactions(CEI) patterns although `nonReentrant` guard from the openzeppellin has been used to prevent reentrancy.

**Proof of Concepts:**

**Recommended Mitigation:** It is advised to follow CEI pattern wherever possible. Consider updating all the state changes first before making an external call to the other contract.

```
    function claimUBI(
        uint256 distributionId,
        bytes calldata validatorUncmpPubkey
    ) external nonReentrant
verifyUncmpPubkeyWithExpectedAddress(validatorUncmpPubkey, msg.sender) {
        uint256 amount = validatorUBIAmounts[distributionId]
[validatorUncmpPubkey];
        require(amount > 0, "UBIPool: no UBI to claim");
        validatorUBIAmounts[distributionId][validatorUncmpPubkey] = 0;
+       totalPendingClaims -= amount;
        (bool success, ) = msg.sender.call{ value: amount }("");
        require(success, "UBIPool: failed to send UBI");
-        totalPendingClaims -= amount;
    }
```

# Gas