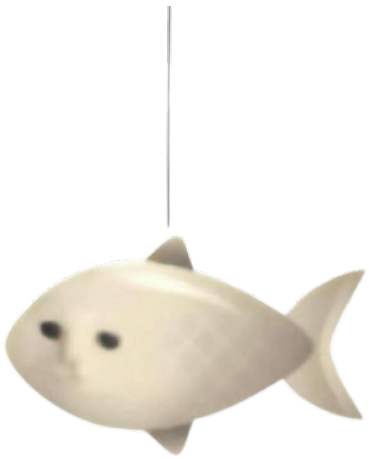

VSkill Audit Report

Version 1.0

Luo Yingjie

December 13, 2024

VSkill Audit Report

Luo Yingjie

December 13, 2024

Prepared by: Luo Yingjie Lead Auditors:

- Luo Yingjie

Assisting Auditors:

- None

Table of Contents

    * [H-2] No restrictions in `Distribution::distributionRandomNumberForVerifiers` function, anyone can directly call it, drain the subscription Link tokens
    * [H-3] The way delete verifier in `Staking::_removeVerifier` function is not correct, it will ruin the way we fetch verifiers
    * [H-4] No restrictions in `VSkillUser::earnUserNft` function, anyone can directly call it by passing an approved evidence parameter to mint NFTs, ruin the verification process
    * [H-5] The same verifier can call multiple times `Verifier::provideFeedback` function to dominate the evidence status, ruin the verification process
    * [H-6] The same verifier can call multiple times `Verifier::provideFeedback` function and violate the `Verifier::_earnRewardsOrGetPenalized` function for `DIFFERENTOPINION` status
    * [H-7] If the `Verifier::provideFeedback` function makes the same evidence with `DIFFERENTOPINION` status for more than once, the `statusApproveOrNot` array will be popped when it's empty, ruin the verification process
    * [H-8] Verifier will lose all the stake when reputation is less than `LOWEST_REPUTATION` and get penalized again
  - Medium
    * [M-1] No bounds check in `Verifier::checkUpkeep` for the `s_evidences` array, can cause DoS attack as the array grows
    * [M-2] The two for loops in `Verifier::_verifiersWithinSameDomain` function can cause DoS attack, as the `s_verifiers` array grows
    * [M-3] The for loop in `Verifier::_selectedVerifiersAddressCallback` function can cause DoS attack, as the `s_verifiers` array grows
    * [M-4] Using memory variables to update the status of evidence in `Verifier::_assignEvidenceToSelectedVerifier` function, will drain the `Chainlink Automation` service
  - Low
    * [L-1] The check condition in `VSkillUser::checkFeedbackOfEvidence` is wrong, user will be reverted due to the return statement, not custom error message
    * [L-2] Not checking the stability of the price feed in `PriceConverter::getChainlinkDataFeed` may lead to wrong conversion
    * [L-3] No validation check in `Verifier::updateSkillDomains` function, verifier can update the skill domain to whatever value they want
    * [L-4] User can call the `VSkillUserNft::mintUserNft` with non-exist skill domain
    * [L-5] Invalid `tokenId` will result in blank `imageUri` in `VSkillUserNft::tokenURI`

function

- Informational
  - [I-1] Best follow the CEI in `Staking::withdrawStake` function
  - [I-2] Solidity pragma should be specific, not wide
  - [I-3] `public` functions not used internally could be marked `external`
  - [I-4] Define and use `constant` variables instead of using literals
  - [I-5] PUSH0 is not supported by all chains
  - [I-6] Modifiers invoked only once can be shoe-horned into the function
  - [I-7] Unused Custom Error
  - [I-8] Costly operations inside loops.
  - [I-9] State variable could be declared constant
  - [I-10] State variable could be declared immutable
  - [I-11] It's best to use the most up-to-date version of `Chainlink VRF`
  - [I-12] Centralization Risk for trusted owners
  - [I-13] The `Verifier::provideFeedback` function is too long, making the maintenance difficult
  - [I-14] The first verifier who submit the evidence will be rewarded more than the following verifiers
- Gas
  - [G-1] Custom error message include a constant `Staking::minStakeUsdAmount` and `VSkillUser::submittedFeeInUsd` which costs more gas
  - [G-2] There are two functions work the same in `Staking` contract and is a waste of gas
  - [G-3] Double check in `Verifier::_earnRewardsOrGetPenalized` function, spend unnecessary gas
  - [G-4] Compute the same `Verifier::keccak256(abi.encodePacked(evidenceIpfsHash` costs unnecessary gas

About LUO YINGJIE

LUO YINGJIE is a blockchain developer and security researcher. With massive experience in the blockchain, he has audited numerous projects and has a deep understanding of the blockchain ecosystem. ….(add more about the auditor)

Protocol Summary

VSkill (VeriSkill) is a decentralized platform for verifying skills. It leverages blockchain technology to create a transparent and trustworthy system for skill verification. The platform involves three key roles:

1. Users: Submit evidence to be verified.
2. Verifiers: Stake money to become verifiers, review evidence, and decide on skill verification.
3. Owner: Can modify submission fees and supported skills for verification.

Disclaimer

The Luo Yingjie 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
Likelihood	High	H	H/M	M
	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

- Commit Hash: cfc7fb2dad65a4902e83aa9673c4d4ed1e7f1448

Scope

```
./src/
#-- nft
    #-- VSkillUserNft.sol
#-- oracle
    #-- Distribution.sol
#-- staking
    #-- Staking.sol
#-- user
    #-- VSkillUser.sol
#-- utils
    #-- interface
        #-- VerifierInterface.sol
    #-- library
        #-- PriceCoverter.sol
        #-- StructDefinition.sol
#-- verifier
    #-- Verifier.sol
```

Roles

- Users: Submit evidence to be verified.
- Verifiers: Stake money to become verifiers, review evidence, and decide on skill verification.
- Owner: Can modify submission fees and supported skills for verification.

Executive Summary

Add some notes of how the audit went, types of issues found, etc.

We spend X hours with Y auditors using Z tools, etc.

Issues found

Severity	Number of issues found
High	8
Medium	4
Low	5
Info	14
Gas Optimizations	4
Total	35

Findings

High

[H-1] No restrictions in `VSKillUserNft::mintUserNft` function, anyone can directly call and mint NFTs

Description:

In the `VSKillUserNft` contract, the `mintUserNft` function is public and no checks are performed to ensure that only the `VSKill` contract can call this function. This means that anyone can call this function and mint NFTs.

Impact:

Users do not need to pay any fees and being verified before minting NFTs. This can lead to a large number of NFTs being minted by anyone. Ruin the verification process.

Proof of Concept:

Add the following test case to `VerifierTest.t.sol` file:

Proof of Code

```
function testAnyoneCanMintUserNftWithoutEnrollingProtocol() external {
        address randomUser = makeAddr("randomUser");
        string memory skillDomainRandomUserWants = SKILL_DOMAINS[0];
        vm.prank(randomUser);
        verifier.mintUserNft(skillDomainRandomUserWants);
```

```
        assert(verifier.getTokenCounter() == 1);
    }
```

Recommended Mitigation:

Add `openzeppelin` access control like `Ownable` to ensure that only the `VSKill` contract can call the `mintUserNft` function.

Then add the modifier `onlyOwner` to the `mintUserNft` function.

```
-   function mintUserNft(string memory skillDomain) public {
+   function mintUserNft(string memory skillDomain) public onlyOwner {
        _safeMint(msg.sender, s_tokenCounter);
        s_tokenIdToSkillDomain[s_tokenCounter] = skillDomain;
        s_tokenCounter++;

        emit MintNftSuccess(s_tokenCounter - 1, skillDomain);
    }
```

[H-2] No restrictions in `Distribution::distributionRandomNumberForVerifiers` function, anyone can directly call it, drain the subscription Link tokens

Description:

In the `Distribution` contract, the `distributionRandomNumberForVerifiers` function is public and no checks are performed to ensure that only the `VSKill` contract can call this function.

```
 function distributionRandomNumberForVerifiers(
        address requester,
        StructDefinition.VSkillUserEvidence memory ev
@>  ) public {
        s_requestId = s_vrfCoordinator.requestRandomWords(
            s_keyHash,
            s_subscriptionId,
            s_requestConfirmations,
            s_callbackGasLimit,
            s_numWords
        );

        s_requestIdToContext[s_requestId] = StructDefinition
            .DistributionVerifierRequestContext(requester, ev);

        emit RequestIdToContextUpdated(
            s_requestId,
```

```
            s_requestIdToContext[s_requestId]
        );
    }
```

Impact:

This means that anyone can call this function and drain the subscription Link tokens, since each time request is made, it will consume the `Chainlink` Link tokens.

Proof of Concept:

Add the following test case to `VerifierTest.t.sol` file:

Proof of Code

```
event RequestIdToContextUpdated(
        uint256 indexed requestId,
        StructDefinition.DistributionVerifierRequestContext context
    );

    using StructDefinition for
↪   StructDefinition.DistributionVerifierRequestContext;

    function testAnyoneCanMakeRequestToVRF() external {
        address randomUser = makeAddr("randomUser");
        StructDefinition.VSkillUserEvidence
            memory dummyEvidence = StructDefinition.VSkillUserEvidence(
                randomUser,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );

        StructDefinition.DistributionVerifierRequestContext
            memory context = StructDefinition
                .DistributionVerifierRequestContext(randomUser,
                ↪  dummyEvidence);

        vm.expectEmit(true, false, false, false, address(verifier));
        emit RequestIdToContextUpdated(1, context);
        vm.prank(randomUser);
        verifier.distributionRandomNumberForVerifiers(
            randomUser,
```

```
            dummyEvidence
        );
    }
```

Recommended Mitigation:

Same as the first issue, add `openzeppelin` access control like `Ownable` to ensure that only the `VSKill` contract can call the `distributionRandomNumberForVerifiers` function.

Then add the modifier `onlyOwner` to the `distributionRandomNumberForVerifiers` function.

```
function distributionRandomNumberForVerifiers(
        address requester,
        StructDefinition.VSkillUserEvidence memory ev
-   ) public {
+   ) public onlyOwner {
        s_requestId = s_vrfCoordinator.requestRandomWords(
            s_keyHash,
            s_subscriptionId,
            s_requestConfirmations,
            s_callbackGasLimit,
            s_numWords
        );

        s_requestIdToContext[s_requestId] = StructDefinition
            .DistributionVerifierRequestContext(requester, ev);

        emit RequestIdToContextUpdated(
            s_requestId,
            s_requestIdToContext[s_requestId]
        );
    }
```

[H-3] The way delete verifier in `Staking::_removeVerifier` function is not correct, it will ruin the way we fetch verifiers

Description:

In the `Staking` contract, the `_removeVerifier` function is implemented in the way below:

```
function _removeVerifier(address verifierAddress) internal {
        uint256 index = s_addressToId[verifierAddress] - 1;
        s_verifiers[index] = s_verifiers[s_verifierCount - 1];
```

```
        s_verifiers.pop();

        s_addressToId[verifierAddress] = 0;
        s_verifierCount--;

        emit LoseVerifier(verifierAddress);
    }
```

It first gets the index of the verifier, then removes the verifier by replacing the verifier with the last verifier in the array, and then pops the last verifier. This will leads to problem of out-of-range index when fetching verifiers.

How we fetch verifiers in other functions:

```
s_verifiers[s_addressToId[verifierAddress] - 1];
```

As shown, we use the id to get the index of the verifier, and then use the index to get the verifier. But the index will be changed every time a verifier is removed, and thus ruin the way we fetch verifiers in other functions.

Impact:

Every time a verifier is removed, the verifiers array will be re-ordered, and the index of the verifiers will be changed. This will lead to the problem of out-of-range index when fetching verifiers. And thus ruin the way we fetch verifiers in other functions.

Proof of Concept:

Let's say we have 5 verifiers in the array, and we remove the 3rd verifier. The array will be like this:

```
Before:
Verifier 1 2 3 4 5
Index    0 1 2 3 4
```

1. Get the index of the verifier to be removed, which is 3. The index is 2.
2. Replace the verifier with the last verifier in the array, and then pop the last verifier. The array will be like this:

```
After:
Verifier 1 2 5 4
Index    0 1 2 3
```

3. Now let's say I want to get the 5th verifier, the index is 4. But we only have 4 verifiers in the array, so the index is out-of-range. Which will lead to the problem of out-of-range index when fetching verifiers and ruin the way we fetch verifiers in other functions.

Recommended Mitigation:

There are several ways to fix this issue:

1. Instead of really moving the verifier, we can just set the verifier to be removed to `address(0)`, and then emit an event to notify that the verifier is removed. This way, the index of the verifiers will not be changed, and we can still fetch verifiers correctly. And as for the total number of verifiers, we can just decrease the total number of verifiers by 1.
2. We can use a mapping to store the verifiers, and then use the id to get the verifier. This way, the index of the verifiers will not be changed, and we can still fetch verifiers correctly.

[H-4] No restrictions in `VSkillUser::earnUserNft` function, anyone can directly call it by passing an approved evidence parameter to mint NFTs, ruin the verification process

Description:

In the `VSkillUser` contract, the `earnUserNft` function is public and no checks are performed to make sure that the evidence is approved by the verifier. This means that anyone can call this function by passing an approved evidence parameter to mint NFTs.

```
 function earnUserNft(
@>      StructDefinition.VSkillUserEvidence memory _evidence
@>  ) public virtual {
@>      if (
@>          _evidence.status !=
@>          StructDefinition.VSkillUserSubmissionStatus.APPROVED
        ) {
            revert VSkillUser__EvidenceNotApprovedYet(_evidence.status);
        }

        super.mintUserNft(_evidence.skillDomain);
    }
```

The malicious user can just pass an approved evidence parameter to mint NFTs, which will ruin the verification process.

Impact:

Users do not need to pay any fees and being verified before minting NFTs. This can lead to a large number of NFTs being minted by anyone. Ruin the verification process.

Proof of Concept:

Add the following test case to `./test/user/uint/VSkillUserTest.t.sol`:

Proof of Code

```
using StructDefinition for StructDefinition.VSkillUserSubmissionStatus;

    function testAnyoneCanMintAnNftWithoutSubmitAndGetVerified() external {
        StructDefinition.VSkillUserEvidence memory evidence =
↪   StructDefinition
            .VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAIN,
                StructDefinition.VSkillUserSubmissionStatus.APPROVED,
                new string[](0)
            );

        vm.prank(USER);
        vskill.earnUserNft(evidence);

        assertEq(vskill.getTokenCounter(), 1);
    }
```

Recommended Mitigation:

Add a modifier to ensure that only the verifier can call the `earnUserNft` function.

```
function earnUserNft(
        StructDefinition.VSkillUserEvidence memory _evidence
-   ) public virtual {
+   ) public virtual onlyOwner {
        if (
            _evidence.status !=
            StructDefinition.VSkillUserSubmissionStatus.APPROVED
        ) {
            revert VSkillUser__EvidenceNotApprovedYet(_evidence.status);
        }

        super.mintUserNft(_evidence.skillDomain);
    }
```

[H-5] The same verifier can call multiple times `Verifier::provideFeedback` function to dominate the evidence status, ruin the verification process

Description:

In the `Verifier` contract, the `provideFeedback` function is used to provide feedback for the evidence. However, the same selected verifier can just call multiple times this function to pass the `NUM_WORDS` checks and centralize the evidence status.

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
        .
        .
        .
        // get all the verifiers who provide feedback and call the function
         ↪  to earn rewards or get penalized

@>      if (
@>          s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
@>              .statusApproveOrNot
@>              .length < s_numWords
        ) {
            return;
        } else {
            address[] memory allSelectedVerifiers =
 ↪  s_evidenceIpfsHashToItsInfo[
                evidenceIpfsHash
            ].selectedVerifiers;
           uint256 allSelectedVerifiersLength = allSelectedVerifiers.length;
            StructDefinition.VSkillUserSubmissionStatus evidenceStatus =
 ↪  _updateEvidenceStatus(
                    evidenceIpfsHash,
                    user
                );
            for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
                _earnRewardsOrGetPenalized(
                    evidenceIpfsHash,
                    allSelectedVerifiers[i],
                    evidenceStatus
                );
            }
        }
    }
```

Impact:

The same verifier can call multiple times this function to pass the NUM_WORDS checks and centralize the evidence status. This will ruin the verification process.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
function testSelectedVerifierCanProvideMultipleFeedbacksCentral-
 ↪  izeTheEvidenceStatus()
        external
    {
        _createNumWordsNumberOfSameDomainVerifier(SKILL_DOMAINS);

        StructDefinition.VSkillUserEvidence memory ev = StructDefinition
            .VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );
        vm.startPrank(USER);
        verifier.submitEvidence{
            value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                AggregatorV3Interface(verifierConstructorParams.priceFeed)
            )
        }(ev.evidenceIpfsHash, ev.skillDomain);
        vm.stopPrank();

        vm.recordLogs();
        verifier._requestVerifiersSelection(ev);
        Vm.Log[] memory entries = vm.getRecordedLogs();
        bytes32 requestId = entries[0].topics[2];
        VRFCoordinatorV2Mock vrfCoordinatorMock = VRFCoordinatorV2Mock(
            verifierConstructorParams.vrfCoordinator
        );
        vm.pauseGasMetering();
        vm.recordLogs();
        vrfCoordinatorMock.fulfillRandomWords(
            uint256(requestId),
            address(verifier)
```

```
        );
        Vm.Log[] memory entriesOfFulfillRandomWords = vm.getRecordedLogs();
        bytes32 selectedVerifierOne =
↪   entriesOfFulfillRandomWords[1].topics[1];

        address selectedVerifierAddressOne = address(
            uint160(uint256(selectedVerifierOne))
        );

        uint256 selectedVerifierOneStakeBefore = verifier
            .getVerifierMoneyStakedInEth(selectedVerifierAddressOne);
        console.log(
            "Selected verifier one stake before: ",
            selectedVerifierOneStakeBefore
        );

        uint256 selectedVerifierOneReputationBefore = verifier
            .getVerifierReputation(selectedVerifierAddressOne);
        console.log(
            "Selected verifier one reputation before: ",
            selectedVerifierOneReputationBefore
        );

        // selectedVerifierOne call multiple times to provide feedback, then
         ↪  earn the rewards
        for (uint160 i = 0; i < NUM_WORDS; i++) {
            vm.prank(selectedVerifierAddressOne);
            verifier.provideFeedback(
                FEEDBACK_IPFS_HASH,
                IPFS_HASH,
                USER,
                false
            );
        }

        uint256 selectedVerifierOneStake =
↪   verifier.getVerifierMoneyStakedInEth(
            selectedVerifierAddressOne
        );

        uint256 selectedVerifierOneReputation =
↪   verifier.getVerifierReputation(
            selectedVerifierAddressOne
```

```
        );

        console.log(
            "Selected verifier one stake after: ",
            selectedVerifierOneStake
        );

        console.log(
            "Selected verifier one reputation after: ",
            selectedVerifierOneReputation
        );

        assert(selectedVerifierOneStake > selectedVerifierOneStakeBefore);
        assert(
            selectedVerifierOneReputation >
↪   selectedVerifierOneReputationBefore
        );
    }
```

Then run the test case:

```
forge test --mt testSelectedVerifierCanProvideMultipleFeedbacksCentral-
↪   izeTheEvidenceStatus -vv
```

Then you can find that, even though only `selectedVerifierOne` has provided the feedback, he is rewarded.

```
  Selected verifier one stake before:  10000000000000000
  Selected verifier one reputation before:  2
  Selected verifier one stake after:  10061753750000000
  Selected verifier one reputation after:  4
```

Recommended Mitigation:

Add some restrictions to ensure that the same verifier can only provide feedback once.

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
        _onlySelectedVerifier(evidenceIpfsHash, msg.sender);
+       _notProvideFeedbackYet(evidenceIpfsHash, msg.sender);
        .
```

```
        .
        .
    }
```

[H-6] The same verifier can call multiple times `Verifier::provideFeedback` function and violate the `Verifier::_earnRewardsOrGetPenalized` function for `DIFFERENTOPINION` status

Description:

In the `Verifier` contract, the `provideFeedback` function is used to provide feedback for the evidence. However, the same selected verifier can just call multiple times this function and violate the `_earnRewardsOrGetPenalized` function for `DIFFERENTOPINION` status.

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
        .
        .
        .
        if (
            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .statusApproveOrNot
                .length < s_numWords
        ) {
            return;
        } else {
            address[] memory allSelectedVerifiers =
 ↪  s_evidenceIpfsHashToItsInfo[
                evidenceIpfsHash
            ].selectedVerifiers;
           uint256 allSelectedVerifiersLength = allSelectedVerifiers.length;
            StructDefinition.VSkillUserSubmissionStatus evidenceStatus =
 ↪  _updateEvidenceStatus(
                    evidenceIpfsHash,
                    user
                );
@>          for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
@>              _earnRewardsOrGetPenalized(
```

```
                    evidenceIpfsHash,
                    allSelectedVerifiers[i],
                    evidenceStatus
                );
            }
        }
    }

function _earnRewardsOrGetPenalized(
        string memory evidenceIpfsHash,
        address verifierAddress,
        StructDefinition.VSkillUserSubmissionStatus evidenceStatus
    ) internal {
        .
        .
        .
        // DIFFERENTOPINION
@>      else {
@>          s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
@>              .statusApproveOrNot
@>              .pop();

            return;
        }
    }
```

Here is how the bug happens:

1. The first verifier provides feedback with `true` value for three times.
2. The second verifier provides feedback with `false` value.
3. Once the second verifier provided the feedback, it will reach the line to pop the `statusApproveOrNot` array since already enough feedbacks are provided.
4. Then since there are only two verifiers provide the feedback, the `statusApproveOrNot` array will not be empty. Instead it will be length of 1.
5. Then when the third verifier provides feedback, he will not be able to trigger the line for `_earnRewardsOrGetPenalized` function, because now only the array length is only 2, not enough to meet the NUM_WORDS checks.

Impact:

The same verifier can call multiple times this function and violate the `_earnRewardsOrGetPenalized` function and violates the other verifiers' rewards.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
function testStatusApprovedOrNotArrayWillBePoppedEvenWhenEmpty() external {
        _createNumWordsNumberOfSameDomainVerifier(SKILL_DOMAINS);

        StructDefinition.VSkillUserEvidence memory ev = StructDefinition
            .VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );
        vm.startPrank(USER);
        verifier.submitEvidence{
            value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                AggregatorV3Interface(verifierConstructorParams.priceFeed)
            )
        }(ev.evidenceIpfsHash, ev.skillDomain);
        vm.stopPrank();

        vm.recordLogs();
        verifier._requestVerifiersSelection(ev);
        Vm.Log[] memory entries = vm.getRecordedLogs();
        bytes32 requestId = entries[0].topics[2];
        VRFCoordinatorV2Mock vrfCoordinatorMock = VRFCoordinatorV2Mock(
            verifierConstructorParams.vrfCoordinator
        );
        vm.pauseGasMetering();
        vm.recordLogs();
        vrfCoordinatorMock.fulfillRandomWords(
            uint256(requestId),
            address(verifier)
        );
        Vm.Log[] memory entriesOfFulfillRandomWords = vm.getRecordedLogs();
        bytes32 selectedVerifierOne =
↪   entriesOfFulfillRandomWords[1].topics[1];
        bytes32 selectedVerifierTwo =
↪   entriesOfFulfillRandomWords[2].topics[1];

        address selectedVerifierAddressOne = address(
            uint160(uint256(selectedVerifierOne))
```

```
        );
        address selectedVerifierAddressTwo = address(
            uint160(uint256(selectedVerifierTwo))
        );

        for (uint160 i = 0; i < 3; i++) {
            vm.prank(selectedVerifierAddressOne);
            verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER,
↪  true);
        }
        bool[] memory statusApproveOrNot = verifier
            .getEvidenceToStatusApproveOrNot(IPFS_HASH);
        console.log("Status approve or not: ", statusApproveOrNot.length);

        vm.prank(selectedVerifierAddressTwo);
       verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, false);

        bool[] memory statusApproveOrNot1 = verifier
            .getEvidenceToStatusApproveOrNot(IPFS_HASH);
       console.log("Status approve or not 1: ", statusApproveOrNot1.length);

        assert(statusApproveOrNot1.length != 0);
    }
```

Then run the commands below:

```
forge test --mt testStatusApprovedOrNotArrayWillBePoppedEvenWhenEmpty -vv
```

And you will see the console output:

```
  Status approve or not:  3
  Status approve or not 1:  1
```

Recommended Mitigation:

Same as the previous issue, add some restrictions to ensure that the same verifier can only provide feedback once.

[H-7] If the `Verifier::provideFeedback` function makes the same evidence with `DIFFERENTOPINION` status for more than once, the `statusApproveOrNot` array will be popped when it's empty, ruin the verification process

Description:

In the `Verifier` contract, the `provideFeedback` function has the logic below:

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
        .
        .
        .
        if (
            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .statusApproveOrNot
                .length < s_numWords
        ) {
            return;
        } else {
            address[] memory allSelectedVerifiers =
 ↪  s_evidenceIpfsHashToItsInfo[
                evidenceIpfsHash
            ].selectedVerifiers;
@>          uint256 allSelectedVerifiersLength = allSelectedVerifiers.length;
            StructDefinition.VSkillUserSubmissionStatus evidenceStatus =
 ↪  _updateEvidenceStatus(
                    evidenceIpfsHash,
                    user
                );

@>          for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
                _earnRewardsOrGetPenalized(
                    evidenceIpfsHash,
                    allSelectedVerifiers[i],
                    evidenceStatus
                );
            }
        }
    }
```

It will have the for loop for every ever selectedVerifiers to call the `_earnRewardsOrGetPenalized` function. However, in the function we have the logic below for `DIFFERENTOPINION` status condition:

```
 function _earnRewardsOrGetPenalized(
        string memory evidenceIpfsHash,
```

```
        address verifierAddress,
        StructDefinition.VSkillUserSubmissionStatus evidenceStatus
    ) internal {
        .
        .
        .
        // DIFFERENTOPINION
        else {
@>          s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
@>              .statusApproveOrNot
@>              .pop();

            return;
        }
    }
```

Here we will pop the `statusApproveOrNot` array for the selectedVerifiers, but what the bug is, in the for loop we are popping the array for every selectedVerifiers ever, that way, if the same evidence has `DIFFERENTOPINION` status for more than once, the `statusApproveOrNot` array will be popped when it's empty.

Impact:

When the evidence finally get approved or rejected by the last verifier, this function will revert, And the evidence will be ruined, and the verification process will be ruined.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
function testIfMoreThanOneTimeDifferentOpinionWillRevert() external {
        uint256 numOfVerifiersWithinOneEvidence = 200;
        address[] memory verifierWithinSameDomain = new address[](
            numOfVerifiersWithinOneEvidence
        );
        for (
            uint160 i = 1;
            i < uint160(numOfVerifiersWithinOneEvidence + 1);
            i++
        ) {
            address verifierAddress = address(i);
            vm.deal(verifierAddress, INITIAL_BALANCE);
            _becomeVerifierWithSkillDomain(verifierAddress, SKILL_DOMAINS);
```

```
            verifierWithinSameDomain[i - 1] = verifierAddress;
        }

        StructDefinition.VSkillUserEvidence memory ev = StructDefinition
            .VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );

        vm.startPrank(USER);
        verifier.submitEvidence{
            value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                AggregatorV3Interface(verifierConstructorParams.priceFeed)
            )
        }(ev.evidenceIpfsHash, ev.skillDomain);
        vm.stopPrank();

        vm.recordLogs();
        verifier._requestVerifiersSelection(ev);
        Vm.Log[] memory entries = vm.getRecordedLogs();
        bytes32 requestId = entries[1].topics[1];
        VRFCoordinatorV2Mock vrfCoordinatorMock = VRFCoordinatorV2Mock(
            verifierConstructorParams.vrfCoordinator
        );
        vm.pauseGasMetering();
        vm.recordLogs();
        vrfCoordinatorMock.fulfillRandomWords(
            uint256(requestId),
            address(verifier)
        );
        Vm.Log[] memory entriesOfFulfillRandomWords = vm.getRecordedLogs();
        bytes32 selectedVerifierOne =
↪   entriesOfFulfillRandomWords[1].topics[1];
        bytes32 selectedVerifierTwo =
↪   entriesOfFulfillRandomWords[2].topics[1];
        bytes32 selectedVerifierThree =
↪   entriesOfFulfillRandomWords[3].topics[
            1
        ];
        address selectedVerifierAddressOne = address(
```

```
            uint160(uint256(selectedVerifierOne))
        );
        address selectedVerifierAddressTwo = address(
            uint160(uint256(selectedVerifierTwo))
        );
        address selectedVerifierAddressThree = address(
            uint160(uint256(selectedVerifierThree))
        );

        vm.prank(selectedVerifierAddressOne);
        verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, true);

        vm.prank(selectedVerifierAddressTwo);
       verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, false);
        bool[] memory statusApproveOrNot = verifier
            .getEvidenceToStatusApproveOrNot(IPFS_HASH);

        console.log("Status approve or not: ", statusApproveOrNot.length);

        vm.prank(selectedVerifierAddressThree);
       verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, false);

        vm.recordLogs();
        verifier._requestVerifiersSelection(ev);
        Vm.Log[] memory finalEntries = vm.getRecordedLogs();
        bytes32 finalRequestId = finalEntries[1].topics[1];
        VRFCoordinatorV2Mock finalVrfCoordinatorMock = VRFCoordinatorV2Mock(
            verifierConstructorParams.vrfCoordinator
        );
        vm.pauseGasMetering();
        vm.recordLogs();
        finalVrfCoordinatorMock.fulfillRandomWords(
            uint256(finalRequestId),
            address(verifier)
        );
        Vm.Log[] memory finalEntriesOfFulfillRandomWords =
↪   vm.getRecordedLogs();
        bytes32 finalSelectedVerifierOne =
↪   finalEntriesOfFulfillRandomWords[1]
            .topics[1];
        bytes32 finalSelectedVerifierTwo =
↪   finalEntriesOfFulfillRandomWords[2]
            .topics[1];
```

```
        bytes32 finalSelectedVerifierThree =
↪   finalEntriesOfFulfillRandomWords[3]
            .topics[1];
        address finalSelectedVerifierAddressOne = address(
            uint160(uint256(finalSelectedVerifierOne))
        );
        address finalSelectedVerifierAddressTwo = address(
            uint160(uint256(finalSelectedVerifierTwo))
        );
        address finalSelectedVerifierAddressThree = address(
            uint160(uint256(finalSelectedVerifierThree))
        );

        vm.prank(finalSelectedVerifierAddressOne);
        verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, true);

        vm.prank(finalSelectedVerifierAddressTwo);
       verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, false);

        vm.expectRevert();
        vm.prank(finalSelectedVerifierAddressThree);
       verifier.provideFeedback(FEEDBACK_IPFS_HASH, IPFS_HASH, USER, false);
    }
```

Here we have the same evidence be `DIFFERENTOPINION` status for twice, and the `statusApproveOrNot` array will be popped when it's empty.

Then run the test case:

```
forge test --mt testIfMoreThanOneTimeDifferentOpinionWillRevert -vvvv
```

You can get the logs below:

```
│   ├─ emit EvidenceStatusUpdated(user: user:
↪  [0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], evidenceIpfsHash:
↪  0x4ec31a7244ef446c1acb5ded1a805b85118d0f808bcb005219f73857ca57896a,
↪  status: 4)
│   └─ ← [Revert] panic: called `.pop()` on an empty array (0x31)
```

And yes, we indeed see the error `panic: called `.pop()` on an empty array`.

Recommended Mitigation:

Refactor the section of how to rewards the verifiers, maybe add the `Chainlink Automation` to listen for the event when the evidence is finally approved or rejected, and then rewards the verifiers.

Then, in the `_earnRewardsOrGetPenalized` function, we can pop the array only three times each time reach the `DIFFERENTOPINION` status.

[H-8] Verifier will lose all the stake when reputation is less than `LOWEST_REPUTATION` and get penalized again

Description:

In the `Verifier` contract, the `_penalizeVerifiers` function have the following logic to handle the penalty:

```
function _penalizeVerifiers(address verifiersAddress) internal {
        if (
            s_verifiers[s_addressToId[verifiersAddress] - 1].reputation >
            LOWEST_REPUTATION
        ) {
            .
            .
            .
        } else {
            .
            .
            .
@>          uint256 verifierStakedMoneyInEth = verifierToBeRemoved
@>              .moneyStakedInEth;

@>          super._penalizeVerifierStakeToBonusMoney(
@>              verifiersAddress,
@>              verifierStakedMoneyInEth
@>          );

            super._removeVerifier(verifiersAddress);

            emit LoseVerifier(verifierToBeRemoved.verifierAddress);
        }
    }
```

As the logic shows, if the reputation is less than `LOWEST_REPUTATION`, the verifier will be switched to the `else` branch, and then the `_penalizeVerifierStakeToBonusMoney` function will be called to penalize the verifier, but the penalty amount is the same as the verifier's stake!!!

Impact:

Verifier will lose all the stake when reputation is less than `LOWEST_REPUTATION` if get penalized again!!!

Recommended Mitigation:

There are several ways to consider:

1. Change the penalty amount to a reasonable amount, like `MIN_USD_AMOUNT`
2. Set a upper limit for the stake, like `MAX_STAKE`, and force the verifier to withdraw the stake when exceed the limit.

Medium

[M-1] No bounds check in `Verifier::checkUpkeep` for the `s_evidences` array, can cause DoS attack as the array grows

Description:

In the `Verifier` contract, the `checkUpkeep` function is called by `chainlink` nodes to automatically check the state of the evidence and distribute the evidence to verifiers. However, there is no bounds check for the `s_evidences` array.

```
function checkUpkeep(
        bytes calldata /* checkData */
    )
        external
        view
        override
        returns (bool upkeepNeeded, bytes memory performData)
    {
        // if the evidence status is `submitted` or `differentOpinion`, this
         ↪  function will return true
        uint256 length = s_evidences.length;

@>      for (uint256 i = 0; i < length; i++) {
            if (
                s_evidences[i].status ==
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED ||
                s_evidences[i].status ==
                StructDefinition.VSkillUserSubmissionStatus.DIFFERENTOPINION
            ) {
                upkeepNeeded = true;
                performData = abi.encode(s_evidences[i]);
```

```
                return (upkeepNeeded, performData);
            }
        }
        upkeepNeeded = false;
        return (upkeepNeeded, "");
    }
```

Impact:

As the array grows, the function will consume more and more gas, and can cause a DoS attack. Then no evidence will be distributed to verifiers, ruin the verification process.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
function testCheckUpKeepWillCostMoreGasAsTheEvidencesGrows() external {
        StructDefinition.VSkillUserEvidence
            memory dummyEvidence = StructDefinition.VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );

        vm.prank(USER);
        verifier.submitEvidence{
            value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                AggregatorV3Interface(verifierConstructorParams.priceFeed)
            )
        }(dummyEvidence.evidenceIpfsHash, dummyEvidence.skillDomain);

        uint256 gasBefore = gasleft();
        vm.prank(USER);
        verifier.checkUpkeep("");
        uint256 gasAfter = gasleft();
        uint256 gasCost = gasBefore - gasAfter;
        console.log("Gas cost for 1 evidence: ", gasCost);

        for (uint160 i = 0; i < 1000; i++) {
            vm.pauseGasMetering();
            verifier.submitEvidence{
```

```
                value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                  AggregatorV3Interface(verifierConstructorParams.priceFeed)
                )
            }(dummyEvidence.evidenceIpfsHash, dummyEvidence.skillDomain);
        }

        vm.resumeGasMetering();

        uint256 gasBefore2 = gasleft();
        vm.prank(USER);
        verifier.checkUpkeep("");
        uint256 gasAfter2 = gasleft();
        uint256 gasCost2 = gasBefore2 - gasAfter2;
        console.log("Gas cost for 1001 evidence: ", gasCost2);

        assert(gasCost2 > gasCost);

        for (uint160 i = 0; i < 10000; i++) {
            vm.pauseGasMetering();
            verifier.submitEvidence{
                value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                  AggregatorV3Interface(verifierConstructorParams.priceFeed)
                )
            }(dummyEvidence.evidenceIpfsHash, dummyEvidence.skillDomain);
        }

        vm.resumeGasMetering();

        uint256 gasBefore3 = gasleft();
        vm.prank(USER);
        verifier.checkUpkeep("");
        uint256 gasAfter3 = gasleft();
        uint256 gasCost3 = gasBefore3 - gasAfter3;
        console.log("Gas cost for 11001 evidence: ", gasCost3);

        assert(gasCost3 > gasCost2);
    }
```

Then run the commands below:

```
forge test --mt testCheckUpKeepWillCostMoreGasAsTheEvidencesGrows -vv
```

And you will find that the gas cost for 11001 evidence is much higher than the gas cost for 1 evidence. 13192 - 7253 = 5939, almost two times higher.

```
  Gas cost for 1 evidence:  7253
  Gas cost for 1001 evidence:  7798
  Gas cost for 11001 evidence:  13192
```

Recommended Mitigation:

Instead of this custom logic automation, try to use the `Chainlink log triggering` to trigger the `checkUpkeep` function. Once someone submits the evidence, emit an event to trigger the `checkUpkeep` function.

[M-2] The two for loops in `Verifier::_verifiersWithinSameDomain` function can cause DoS attack, as the `s_verifiers` array grows

Description:

In the `Verifier` contract, the `_verifiersWithinSameDomain` function has the following two for loops:

```
function _verifiersWithinSameDomain(
        string memory skillDomain
    ) public view returns (address[] memory, uint256 count) {
        uint256 length = s_verifiers.length;

        uint256 verifiersWithinSameDomainCount = 0;

@>      for (uint256 i = 0; i < length; i++) {
            if (s_verifiers[i].skillDomains.length > 0) {
                uint256 skillDomainLength =
 ↪  s_verifiers[i].skillDomains.length;
@>              for (uint256 j = 0; j < skillDomainLength; j++) {
                    if (
                        keccak256(
                            abi.encodePacked(s_verifiers[i].skillDomains[j])
                        ) == keccak256(abi.encodePacked(skillDomain))
                    ) {
                        verifiersWithinSameDomainCount++;
                        break; // No need to check other domains for this
                         ↪  verifier
                    }
                }
            }
        }
```

```
        address[] memory verifiersWithinSameDomain = new address[](
            verifiersWithinSameDomainCount
        );

        uint256 verifiersWithinSameDomainIndex = 0;

@>      for (uint256 i = 0; i < length; i++) {
            if (s_verifiers[i].skillDomains.length > 0) {
                uint256 skillDomainLength =
 ↪  s_verifiers[i].skillDomains.length;
@>              for (uint256 j = 0; j < skillDomainLength; j++) {
                    if (
                        keccak256(
                            abi.encodePacked(s_verifiers[i].skillDomains[j])
                        ) == keccak256(abi.encodePacked(skillDomain))
                    ) {
                        verifiersWithinSameDomain[
                            verifiersWithinSameDomainIndex
                        ] = s_verifiers[i].verifierAddress;
                        verifiersWithinSameDomainIndex++;
                        break; // No need to check other domains for this
                         ↪  verifier
                    }
                }
            }
        }

        return (verifiersWithinSameDomain, verifiersWithinSameDomainCount);
    }
```

The first for loop is used to count the number of verifiers within the same domain, and the second for loop is used to get the verifiers within the same domain.

Impact:

As the `s_verifiers` array grows, these two for loops can cause a DoS attack.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
    function testDoSHappenWhenTooMuchVerifiers() external {
        vm.pauseGasMetering();
        uint256 numOfVerifiersWithinOneEvidence = 100;
```

```
address[] memory verifierWithinSameDomain = new address[](
    numOfVerifiersWithinOneEvidence
);
for (
    uint160 i = 1;
    i < uint160(numOfVerifiersWithinOneEvidence + 1);
    i++
) {
    address verifierAddress = address(i);
    vm.deal(verifierAddress, INITIAL_BALANCE);
    _becomeVerifierWithSkillDomain(verifierAddress, SKILL_DOMAINS);
    verifierWithinSameDomain[i - 1] = verifierAddress;
}
vm.resumeGasMetering();

uint256 gasBefore = gasleft();
verifier._verifiersWithinSameDomain(SKILL_DOMAINS[0]);
uint256 gasAfter = gasleft();
uint256 gasCost = gasBefore - gasAfter;

console.log("Gas cost for 100 verifiers: ", gasCost);

vm.pauseGasMetering();
uint256 numOfVerifiersWithinOneEvidence2 = 1000;
address[] memory verifierWithinSameDomain2 = new address[](
    numOfVerifiersWithinOneEvidence2
);
for (
    uint160 i = 1;
    i < uint160(numOfVerifiersWithinOneEvidence2 + 1);
    i++
) {
    address verifierAddress = address(i);
    vm.deal(verifierAddress, INITIAL_BALANCE);
    _becomeVerifierWithSkillDomain(verifierAddress, SKILL_DOMAINS);
    verifierWithinSameDomain2[i - 1] = verifierAddress;
}
vm.resumeGasMetering();

uint256 gasBefore2 = gasleft();
verifier._verifiersWithinSameDomain(SKILL_DOMAINS[0]);
uint256 gasAfter2 = gasleft();
uint256 gasCost2 = gasBefore2 - gasAfter2;
```

```
        console.log("Gas cost for 1000 verifiers: ", gasCost2);

        assert(gasCost2 > gasCost);

        vm.pauseGasMetering();
        uint256 numOfVerifiersWithinOneEvidence3 = 100000;
        address[] memory verifierWithinSameDomain3 = new address[](
            numOfVerifiersWithinOneEvidence3
        );
        for (
            uint160 i = 1;
            i < uint160(numOfVerifiersWithinOneEvidence3 + 1);
            i++
        ) {
            address verifierAddress = address(i);
            vm.deal(verifierAddress, INITIAL_BALANCE);
            _becomeVerifierWithSkillDomain(verifierAddress, SKILL_DOMAINS);
            verifierWithinSameDomain3[i - 1] = verifierAddress;
        }
        vm.resumeGasMetering();

        vm.expectRevert();
        verifier._verifiersWithinSameDomain(SKILL_DOMAINS[0]);

        console.log("Revert due to DoS!");
    }
```

When we set the num of verifiers to 100000, this function reverts!

And you can run the command below to check the logs see the gas cost for `100` and `1000` verifiers calling this function:

```
forge test --mt testDoSHappenWhenTooMuchVerifiers -vv
```

Output:

```
  Gas cost for 100 verifiers:  472882
  Gas cost for 1000 verifiers:  4898941
  Revert due to DoS!
```

Here 100 costs `472882`, 1000 costs `4898941`, almost 10 times expensive gas cost!

Recommended Mitigation:

Consider use a map to store the skill domains to the verifiers and when select the verifiers, only a certain amount of verifiers will be first selected as the participants in the final selection.

[M-3] The for loop in `Verifier::_selectedVerifiersAddressCallback` function can cause DoS attack, as the `s_verifiers` array grows

Description:

Same as above but in the `Verifier::_selectedVerifiersAddressCallback` function:

```
 function _selectedVerifiersAddressCallback(
        StructDefinition.VSkillUserEvidence memory ev,
        uint256[] memory randomWords
    )
        public
        enoughNumberOfVerifiers(ev.skillDomain)
        returns (address[] memory)
    {
        address[] memory selectedVerifiers = new address[](s_numWords);

        (
            address[] memory verifiersWithinSameDomain,
            uint256 verifiersWithinSameDomainCount
        ) = _verifiersWithinSameDomain(ev.skillDomain);

        uint256 totalReputationScore = 0;
@>      for (uint256 i = 0; i < verifiersWithinSameDomainCount; i++) {
            totalReputationScore += s_verifiers[
                s_addressToId[verifiersWithinSameDomain[i]] - 1
            ].reputation;
        }

        uint256[] memory selectedIndices = new
 ↪  uint256[](totalReputationScore);

        uint256 selectedIndicesCount = 0;

@>      for (uint256 i = 0; i < verifiersWithinSameDomainCount; i++) {
            uint256 reputation = s_verifiers[
                s_addressToId[verifiersWithinSameDomain[i]] - 1
            ].reputation;
            for (uint256 j = 0; j < reputation; j++) {
                selectedIndices[selectedIndicesCount] = i;
```

```
                selectedIndicesCount++;
            }
        }

        for (uint256 i = 0; i < s_numWords; i++) {
            uint256 randomIndex = randomWords[i] % totalReputationScore;
            selectedVerifiers[i] = verifiersWithinSameDomain[
                selectedIndices[randomIndex]
            ];
        }

        _updateSelectedVerifiersInfo(ev.evidenceIpfsHash,
         ↪  selectedVerifiers);

        _assignEvidenceToSelectedVerifier(ev, selectedVerifiers);

        return selectedVerifiers;
    }
```

Impact:

As the `s_verifiers` array grows, this for loop can cause a DoS attack.

Recommended Mitigation:

Same as M-2, as only a certain amount of verifiers will be first selected as the participants in the final selection. The `verifiersWithinSameDomainCount` will be limited to a certain amount.

[M-4] Using memory variables to update the status of evidence in `Verifier::_assignEvidenceToSelectedVerifier` function, will drain the `Chainlink Automation` service

Description:

In the `Verifier` contract, the `_assignEvidenceToSelectedVerifier` use a memory variable as the evidence to update its `status`, which will not update the status of the evidence in the storage.

```
function _assignEvidenceToSelectedVerifier(
@>      StructDefinition.VSkillUserEvidence memory ev,
        address[] memory selectedVerifiers
    ) internal {
        .
        .
```

```
        .
@>      ev.status = StructDefinition.VSkillUserSubmissionStatus.INREVIEW;
        emit EvidenceStatusUpdated(
            ev.submitter,
            ev.evidenceIpfsHash,
            ev.status
        );
    }
```

Impact:

As a result, the evidence status will not be updated in the storage and remains as SUBMITTED, which will trigger the ‘checkUpkeep function being called by `Chainlink Automation` node, cost the money and drain the service.

Proof of Concept:

Add the following test case to `./test/verifier/uint/VerifierTest.t.sol`:

Proof of Code

```
function testEvidenceStatusNotUpdateAfterDistributedToVerifiers() external
 ↪  {
        _createNumWordsNumberOfSameDomainVerifier(SKILL_DOMAINS);

        StructDefinition.VSkillUserEvidence memory ev = StructDefinition
            .VSkillUserEvidence(
                USER,
                IPFS_HASH,
                SKILL_DOMAINS[0],
                StructDefinition.VSkillUserSubmissionStatus.SUBMITTED,
                new string[](0)
            );
        vm.startPrank(USER);
        verifier.submitEvidence{
            value: verifier.getSubmissionFeeInUsd().convertUsdToEth(
                AggregatorV3Interface(verifierConstructorParams.priceFeed)
            )
        }(ev.evidenceIpfsHash, ev.skillDomain);
        vm.stopPrank();

        vm.recordLogs();
        verifier._requestVerifiersSelection(ev);
        Vm.Log[] memory entries = vm.getRecordedLogs();
        bytes32 requestId = entries[0].topics[2];
```

```
        VRFCoordinatorV2Mock vrfCoordinatorMock = VRFCoordinatorV2Mock(
            verifierConstructorParams.vrfCoordinator
        );
        vm.pauseGasMetering();
        vrfCoordinatorMock.fulfillRandomWords(
            uint256(requestId),
            address(verifier)
        );

        StructDefinition.VSkillUserSubmissionStatus status = verifier
            .getEvidenceStatus(USER, 0);
        assert(uint256(status) != uint256(SubmissionStatus.INREVIEW));
        console.log("Evidence status: ", uint256(status));
    }
```

Recommended Mitigation:

Change the ev variable to the storage variable to update the status of the evidence in the storage. Or just pass some parameters which can be used to get the storage evidence.

Low

[L-1] The check condition in `VSkillUser::checkFeedbackOfEvidence` is wrong, user will be reverted due to the return statement, not custom error message

Description:

In the `VSkillUser` contract, the `checkFeedbackOfEvidence` function is checking the index of user evidence validity. However, the check condition is wrong.

```
 function checkFeedbackOfEvidence(
        uint256 indexOfUserEvidence
    ) public view virtual returns (string[] memory) {
@>      if (indexOfUserEvidence >= s_evidences.length) {
            revert VSkillUser__EvidenceIndexOutOfRange();
        }

@>      return
            s_addressToEvidences[msg.sender][indexOfUserEvidence]
                .feedbackIpfsHash;
    }
```

This check checks the index of the user evidence with the overall length of the evidences. However, the index should be checked with the length of the user evidences.

Impact:

If user input the index of the evidence that is out of range, the user will be reverted due to the return statement, not custom error message.

Proof of Concept:

Add the following test case to `./test/user/uint/VSkillUserTest.t.sol`:

Proof of Code

```
function testCheckFeedbackOfEvidenceAlwaysRevertAsLongAsMoreThanT-
 ↪  woUsersSubmitEvidence()
        external
    {
        vm.startPrank(USER);
        vskill.submitEvidence{value: SUBMISSION_FEE_IN_ETH}(
            IPFS_HASH,
            SKILL_DOMAIN
        );
        vm.stopPrank();

        address anotherUser = makeAddr("anotherUser");
        vm.deal(anotherUser, INITIAL_BALANCE);
        vm.startPrank(anotherUser);
        vskill.submitEvidence{value: SUBMISSION_FEE_IN_ETH}(
            IPFS_HASH,
            SKILL_DOMAIN
        );
        vm.stopPrank();

        vm.prank(USER);
        vm.expectRevert();
        vskill.checkFeedbackOfEvidence(1);
    }
```

Then run this test case:

```
forge test --mt testCheckFeedbackOfEvidenceAlwaysRevertAsLongAsMoreThanT-
 ↪  woUsersSubmitEvidence -vvvv
```

And you will find that the test case revert with the error:

```
├─ [697] VSkillUser::checkFeedbackOfEvidence(1) [staticcall]
│    └─ ← [Revert] panic: array out-of-bounds access (0x32)
```

This `array out-of-bounds access (0x32)` error is not our custom error `VSkillUser__EvidenceIndexOutOfRange()` and is from the return statement.

Recommended Mitigation:

Just change the check condition to check the index of the user evidence with the length of the user evidences.

```
function checkFeedbackOfEvidence(
        uint256 indexOfUserEvidence
    ) public view virtual returns (string[] memory) {
-       if (indexOfUserEvidence >= s_evidences.length) {
+      if (indexOfUserEvidence >= s_addressToEvidences[msg.sender].length) {
            revert VSkillUser__EvidenceIndexOutOfRange();
        }

        return
            s_addressToEvidences[msg.sender][indexOfUserEvidence]
                .feedbackIpfsHash;
    }
```

[L-2] Not checking the stability of the price feed in `PriceConverter::getChainlinkDataFeedLatestAnswer`, may lead to wrong conversion

Description:

In the `PriceConverter` library, the `getChainlinkDataFeedLatestAnswer` function is used to get the latest price feed from the `Chainlink` oracle. However, the stability of the price feed is not checked.

Impact:

This might lead to wrong conversion, if the price feed is not stable.

Recommended Mitigation:

Rewrite the `getChainlinkDataFeedLatestAnswer` function to check the stability of the price feed for a certain period of time.

[L-3] No validation check in `Verifier::updateSkillDomains` function, verifier can update the skill domain to whatever value they want

Description:

In the `Verifier` contract, the `updateSkillDomains` function is used to update the skill domains. However, there is no validation check for the skill domain value.

```
function updateSkillDomains(
        string[] memory newSkillDomains
    ) external isVeifier {
@>      s_verifiers[s_addressToId[msg.sender] - 1]
            .skillDomains = newSkillDomains;
        emit VerifierSkillDomainUpdated(msg.sender, newSkillDomains);
    }
```

Impact:

The verifier can update the skill domain to whatever value they want, which might be hard to manage.

Recommended Mitigation:

Add a validation check for the skill domain value.

```
function updateSkillDomains(
        string[] memory newSkillDomains
    ) external isVeifier {
+       _validSkillDomains(newSkillDomains);
        s_verifiers[s_addressToId[msg.sender] - 1]
            .skillDomains = newSkillDomains;
        emit VerifierSkillDomainUpdated(msg.sender, newSkillDomains);
    }
```

This `_validSkillDomains` function should check if the user input skill domains exist in the predefined skill domains.

[L-4] User can call the `VSkillUserNft::mintUserNft` with non-exist skill domain

Description:

In the `VSkillUserNft` contract, the `mintUserNft` function does not check if the skill domain exists.

```
@>  function mintUserNft(string memory skillDomain) public {
        _safeMint(msg.sender, s_tokenCounter);
        s_tokenIdToSkillDomain[s_tokenCounter] = skillDomain;
        s_tokenCounter++;

        emit MintNftSuccess(s_tokenCounter - 1, skillDomain);
    }
```

Impact:

The user can call the `mintUserNft` function with a non-exist skill domain, generate a lot of no-sense NFTs.

Proof of Concept:

Add the following test case to `./test/nft/uint/VSkillUserNftTest.t.sol`:

Proof of Code

```
function testUserCanMintANonExistentSkillDomain() external {
        vm.prank(USER);
        vskillUserNft.mintUserNft("non-existent-skill-domain");
        uint256 tokenCounter = vskillUserNft.getTokenCounter();
        assertEq(tokenCounter, 1);
    }
```

Recommended Mitigation:

Add the validation check for the skill domain value.

```
 function mintUserNft(string memory skillDomain) public {
+       for (uint256 i = 0; i < s_skillDomains.length; i++) {
+           if (
+               keccak256(abi.encodePacked(s_skillDomains[i])) ==
+               keccak256(abi.encodePacked(skillDomain))
+           ) {
+               break;
+           }
+           if (i == s_skillDomains.length - 1) {
+               revert VSkillUserNft__SkillDomainNotFound(skillDomain);
+           }
+       }
        _safeMint(msg.sender, s_tokenCounter);
        s_tokenIdToSkillDomain[s_tokenCounter] = skillDomain;
        s_tokenCounter++;

        emit MintNftSuccess(s_tokenCounter - 1, skillDomain);
    }
```

[L-5] Invalid `tokenId` will result in blank `imageUri` in `VSkillUserNft::tokenURI` function

Description:

In the `VSkillUserNft` contract, the `tokenURI` function has no validation check for the `tokenId`.

```
function tokenURI(
        uint256 tokenId
    ) public view override returns (string memory) {
@>      string memory skillDomain = s_tokenIdToSkillDomain[tokenId];
@>     string memory imageUri = s_skillDomainToUserNftImageUri[skillDomain];

        return
            string(
                abi.encodePacked(
                    _baseURI(),
                    Base64.encode(
                        bytes( // bytes casting actually unnecessary as
                         ↪  'abi.encodePacked()' returns a bytes
                            abi.encodePacked(
                                '{"name":"',
                                name(),
                               '", "description":"Proof of capability of the
                                ↪  skill", ',
                                '"attributes": [{"trait_type": "skill",
                                ↪  "value": 100}], "image":"',
                                imageUri,
                                '"}'
                            )
                        )
                    )
                )
            );
    }
```

Impact:

If the user input an invalid `tokenId`, the `imageUri` will be blank.

Proof of Concept:

Add the following test case to `./test/nft/uint/VSkillUserNftTest.t.sol`:

Proof of Code

```
 function testInvalidTokenIdWillReturnBlankString() external view {
        string memory skillDomain = vskillUserNft.tokenURI(100);
        assertEq(
            skillDomain,
```

```
            ↪  "data:application/json;base64,eyJuYW1lIjoiVlNraWxsVXNlck5mdCIsICJkZXNjcm
        );
    }
```

You can copy the `data:application...` to your browser and you will find the output like this:

```
{
  "name": "VSkillUserNft",
  "description": "Proof of capability of the skill",
  "attributes": [
    {
      "trait_type": "skill",
      "value": 100
    }
  ],
  "image": ""
}
```

The image is blank.

Recommended Mitigation:

Add the validation check for the `tokenId`.

```
error VSkillUserNft__InvalidTokenId(uint256 tokenId);

function tokenURI(
        uint256 tokenId
    ) public view override returns (string memory) {
+       if (_ownerOf(tokenId) == address(0)) {
+           revert VSkillUserNft__InvalidTokenId(tokenId);
+       }
        string memory skillDomain = s_tokenIdToSkillDomain[tokenId];
       string memory imageUri = s_skillDomainToUserNftImageUri[skillDomain];

        return
            string(
                abi.encodePacked(
                    _baseURI(),
                    Base64.encode(
                        bytes( // bytes casting actually unnecessary as
 ↪  'abi.encodePacked()' returns a bytes
                            abi.encodePacked(
                                '{"name":"',
```

```
                                name(),
                               '", "description":"Proof of capability of the
↪  skill", ',
                                '"attributes": [{"trait_type": "skill",
↪  "value": 100}], "image":"',
                                imageUri,
                                '"}'
                            )
                        )
                    )
                )
            );
    }
```

Informational

[I-1] Best follow the CEI in `Staking::withdrawStake` function

Description:

In the `Staking` contract, the `withdrawStake` function not follow the `Checks-Effects-Interactions` pattern, it sends the money first and then updates the state.

```
function withdrawStake(uint256 amountToWithdrawInEth) public virtual {
        .
        .
        .
@>      (bool success, ) = msg.sender.call{value: amountToWithdrawInEth}("");
        if (!success) {
            revert Staking__WithdrawFailed();
        }

@>      s_verifiers[s_addressToId[msg.sender] - 1]
            .moneyStakedInEth -= amountToWithdrawInEth;
        .
        .
        .
    }
```

However, The verifier cannot withdraw their stakes multiple times and drain the protocol because the state change will be reverted if the moneyStakedInEth is negative.

Impact:

It's best to follow the best practices, which can avoid this kind psuedo-reentrancy attack.

Proof of Concept:

In the `test/staking/uint/StakingTest.t.sol` file, add the following test case:

Proof of Code

```
function testVerifierCanDrainTheProtocolByReenterWithdrawStakeFunction()
        external
    {
        vm.startPrank(USER);
        staking.stake{value: 2 * MIN_ETH_AMOUNT}();
        vm.stopPrank();

        MaliciousUser attacker = new MaliciousUser(staking);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, MIN_ETH_AMOUNT);

        uint256 balanceBefore = address(staking).balance;
        uint256 balanceBeforeAttacker = address(attacker).balance;
        console.log("Balance before attack: ", balanceBefore);
        console.log("Attacker balance before attack: ",
         ↪  balanceBeforeAttacker);

        vm.expectRevert();
        vm.startPrank(attackUser);
        attacker.hack{value: MIN_ETH_AMOUNT}();
        vm.stopPrank();

        uint256 balanceAfter = address(staking).balance;
        uint256 balanceAfterAttacker = address(attacker).balance;
        console.log("Balance after attack: ", balanceAfter);
       console.log("Attacker balance after attack: ", balanceAfterAttacker);
    }
```

And also this contract:

```
contract MaliciousUser {
    using PriceConverter for uint256;

    Staking staking;
    // Only works on anvil local chain, since we know the price is 2000 USD
     ↪  per ETH
    uint256 MIN_ETH_AMOUNT = 1e16;
```

```
    constructor(Staking _stakingContract) {
        staking = _stakingContract;
    }

    function hack() external payable {
        staking.stake{value: MIN_ETH_AMOUNT}();
        staking.withdrawStake(MIN_ETH_AMOUNT);
    }

    receive() external payable {
        if (address(staking).balance >= MIN_ETH_AMOUNT) {
            staking.withdrawStake(MIN_ETH_AMOUNT);
        }
    }
}
```

Then run the test case:

```
forge test --mt
 ↪  testVerifierCanDrainTheProtocolByReenterWithdrawStakeFunction -vvvvv
```

You will see the logs like below:

```
  │   ├─ [42884] Staking::withdrawStake(10000000000000000 [1e16])
    │   │   ├─ [34747] MaliciousUser::receive{value: 10000000000000000}()
    │   │   │   ├─ [33743] Staking::withdrawStake(10000000000000000 [1e16])
    │   │   │   │   ├─ [25452] MaliciousUser::receive{value:
     ↪  10000000000000000}()
    │   │   │   │   │   ├─ [24460] Staking::withdrawStake(10000000000000000
     ↪  [1e16])
    │   │   │   │   │   │   ├─ [396] MaliciousUser::receive{value:
     ↪  10000000000000000}()
    │   │   │   │   │   │   │   └─ ← [Stop]
    │   │   │   │   │   │   ├─ emit Withdrawn(staker: MaliciousUser:
     ↪  [0x2e234DAe75C793f67A35089C9d99245E1C58470b], amount:
     ↪  10000000000000000 [1e16])
    │   │   │   │   │   │   ├─ emit VerifierStakeUpdated(verifier:
     ↪  MaliciousUser: [0x2e234DAe75C793f67A35089C9d99245E1C58470b],
     ↪  previousAmountInEth: 10000000000000000 [1e16], newAmountInEth: 0)
    │   │   │   │   │   │   ├─ [993] MockV3Aggregator::latestRoundData()
     ↪  [staticcall]
    │   │   │   │   │   │   │   └─ ← [Return] 1, 200000000000 [2e11], 1, 1, 1
    │   │   │   │   │   │   ├─ emit LoseVerifier(verifier: MaliciousUser:
     ↪  [0x2e234DAe75C793f67A35089C9d99245E1C58470b])
```

```
    │   │   │   │   │   │   └─ ← [Stop]
    │   │   │   │   │   └─ ← [Stop]
    │   │   │   │   └─ ← [Revert] panic: arithmetic underflow or overflow
    ↪  (0x11)
    │   │   │   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
    │   │   └─ ← [Revert] Staking__WithdrawFailed()
    │   └─ ← [Revert] Staking__WithdrawFailed()
```

And also you can see the console value:

```
Balance before attack:  20000000000000000
Attacker balance before attack:  0
Balance after attack:  20000000000000000
Attacker balance after attack:  0
```

We didn't see the attacker's balance increase, which means the attacker cannot drain the protocol by reentering the `withdrawStake` function.

Recommended Mitigation:

Just follow the some best practices.

[I-2] Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.24;`, use `pragma solidity 0.8.24;`

6 Found Instances

- Found in src/nft/VSkillUserNft.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/oracle/Distribution.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/staking/Staking.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/user/VSkillUser.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/utils/interface/VerifierInterface.sol Line: 3

```
pragma solidity ^0.8.24;
```

- Found in src/verifier/Verifier.sol Line: 2

```
pragma solidity ^0.8.24;
```

[I-3] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

28 Found Instances

- Found in src/nft/VSkillUserNft.sol Line: 54

```
function mintUserNft(string memory skillDomain) public {
```

- Found in src/nft/VSkillUserNft.sol Line: 68

```
function tokenURI(
```

- Found in src/nft/VSkillUserNft.sol Line: 103

```
function _addMoreSkillsForNft(
```

- Found in src/oracle/Distribution.sol Line: 60

```
function distributionRandomNumberForVerifiers(
```

- Found in src/oracle/Distribution.sol Line: 116

```
function getRandomWords() public view returns (uint256[] memory) {
```

- Found in src/oracle/Distribution.sol Line: 120

```
function getRequestIdToContext(
```

- Found in src/oracle/Distribution.sol Line: 130

```
function getSubscriptionId() public view returns (uint64) {
```

- Found in src/oracle/Distribution.sol Line: 134

```
function getVrfCoordinator()
```

- Found in src/oracle/Distribution.sol Line: 142

```
function getKeyHash() public view returns (bytes32) {
```

- Found in src/oracle/Distribution.sol Line: 146

```
    function getCallbackGasLimit() public view returns (uint32) {
```

- Found in src/oracle/Distribution.sol Line: 150

```
    function getRequestConfirmations() public view returns (uint16) {
```

- Found in src/staking/Staking.sol Line: 97

```
    function withdrawStake(uint256 amountToWithdrawInEth) public
↪   virtual {
```

- Found in src/staking/Staking.sol Line: 189

```
    function addBonusMoneyForVerifier() public payable {
```

- Found in src/staking/Staking.sol Line: 378

```
    function getVerifierEvidenceIpfsHash(
```

- Found in src/staking/Staking.sol Line: 414

```
    function getBonusMoneyInEth() public view returns (uint256) {
```

- Found in src/user/VSkillUser.sol Line: 83

```
    function submitEvidence(
```

- Found in src/user/VSkillUser.sol Line: 143

```
    function checkFeedbackOfEvidence(
```

- Found in src/user/VSkillUser.sol Line: 161

```
    function earnUserNft(
```

- Found in src/user/VSkillUser.sol Line: 185

```
    function changeSubmissionFee(uint256 newFeeInUsd) public virtual
↪   onlyOwner {
```

- Found in src/user/VSkillUser.sol Line: 198

```
    function addMoreSkills(
```

- Found in src/verifier/Verifier.sol Line: 331

```
    function stake() public payable override {
```

- Found in src/verifier/Verifier.sol Line: 335

```
    function withdrawStake(uint256 amountToWithdrawInEth) public
↪   override {
```

- Found in src/verifier/Verifier.sol Line: 343

```
    function submitEvidence(
```

- Found in src/verifier/Verifier.sol Line: 350

```
    function checkFeedbackOfEvidence(
```

- Found in src/verifier/Verifier.sol Line: 356

```
    function earnUserNft(
```

- Found in src/verifier/Verifier.sol Line: 362

```
    function changeSubmissionFee(
```

- Found in src/verifier/Verifier.sol Line: 368

```
    function addMoreSkills(
```

- Found in src/verifier/Verifier.sol Line: 636

```
    function _selectedVerifiersAddressCallback(
```

[I-4] Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

2 Found Instances

- Found in src/utils/library/PriceCoverter.sol Line: 33

```
        return (ethAmount * uint256(ethPrice)) / 1e18;
```

- Found in src/utils/library/PriceCoverter.sol Line: 41

```
        return (usdAmount * 1e18) / uint256(ethPrice);
```

[I-5] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in

case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

8 Found Instances

- Found in src/nft/VSkillUserNft.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/oracle/Distribution.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/staking/Staking.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/user/VSkillUser.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/utils/interface/VerifierInterface.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/utils/library/PriceCoverter.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/utils/library/StructDefinition.sol Line: 3

  ```
  pragma solidity ^0.8.24;
  ```

- Found in src/verifier/Verifier.sol Line: 2

  ```
  pragma solidity ^0.8.24;
  ```

[I-6] Modifiers invoked only once can be shoe-horned into the function

1 Found Instances

- Found in src/verifier/Verifier.sol Line: 115

  ```
      modifier enoughNumberOfVerifiers(string memory skillDomain) {
  ```

[I-7] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

1 Found Instances

- Found in src/staking/Staking.sol Line: 23

```
error Staking__AlreadyVerifier();
```

[I-8] Costly operations inside loops.

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

4 Found Instances

- Found in src/nft/VSkillUserNft.sol Line: 42

```
for (uint256 i = 0; i < skillDomainLength; i++) {
```

- Found in src/verifier/Verifier.sol Line: 314

```
for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
```

- Found in src/verifier/Verifier.sol Line: 759

```
for (uint256 i = 0; i < s_numWords; i++) {
```

- Found in src/verifier/Verifier.sol Line: 870

```
for (uint256 i = 1; i < statusLength; i++) {
```

[I-9] State variable could be declared constant

State variables that are not updated following deployment should be declared constant to save gas. Add the `constant` attribute to state variables that never change.

2 Found Instances

- Found in src/oracle/Distribution.sol Line: 29

```
uint16 s_requestConfirmations = 3;
```

- Found in src/oracle/Distribution.sol Line: 30

```
uint32 s_numWords = 3;
```

[I-10] State variable could be declared immutable

State variables that are should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

5 Found Instances

- Found in src/oracle/Distribution.sol Line: 25

```
uint64 s_subscriptionId;
```

- Found in src/oracle/Distribution.sol Line: 26

```
VRFCoordinatorV2Interface s_vrfCoordinator;
```

- Found in src/oracle/Distribution.sol Line: 27

```
bytes32 s_keyHash;
```

- Found in src/oracle/Distribution.sol Line: 28

```
uint32 s_callbackGasLimit;
```

- Found in src/staking/Staking.sol Line: 54

```
AggregatorV3Interface internal s_priceFeed;
```

[I-11] It's best to use the most up-to-date version of `Chainlink VRF`

Description:

In the `Distribution` contract, we are using the `VRFCoordinatorV2` contract, which is the old version of the `Chainlink VRF`. It's best to use the most up-to-date version of version 2.5.

[I-12] Centralization Risk for trusted owners

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

Found Instances

- Found in src/user/VSkillUser.sol Line: 185

```
function changeSubmissionFee(uint256 newFeeInUsd) public virtual
↪ onlyOwner {
```

[I-13] The `Verifier::provideFeedback` function is too long, making the maintenance difficult

Description:

In the `Verifier` contract, the `provideFeedback` function is as follows:

Code

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
        // can the same verifier call multiple time of this function? Yes,
         ↪  the verifier can call multiple times
        // Any impact? The verifier will be rewarded or penalized multiple
         ↪  times
        // @written audit-high the verifier can call multiple times of this
         ↪  function and pass the check for the if statement, the judgement
         ↪  will be centralized!!!
        _onlySelectedVerifier(evidenceIpfsHash, msg.sender);
        StructDefinition.VSkillUserEvidence[]
            memory userEvidences = s_addressToEvidences[user];
        uint256 length = userEvidences.length;
        uint256 currentEvidenceIndex;
        for (uint256 i = 0; i < length; i++) {
            if (
                keccak256(
                    abi.encodePacked(userEvidences[i].evidenceIpfsHash)
                ) == keccak256(abi.encodePacked(evidenceIpfsHash))
            ) {
                currentEvidenceIndex = i;
                break;
            }
        }


 ↪  s_addressToEvidences[user][currentEvidenceIndex].feedbackIpfsHash.push(
            feedbackIpfsHash
        );

        s_verifiers[s_addressToId[msg.sender] - 1].feedbackIpfsHash.push(
```

```
            feedbackIpfsHash
        );

        emit FeedbackProvided(
            StructDefinition.VerifierFeedbackProvidedEventParams({
                verifierAddress: msg.sender,
                user: user,
                approved: approved,
                feedbackIpfsHash: feedbackIpfsHash,
                evidenceIpfsHash: evidenceIpfsHash
            })
        );

        // @audit-info separate the rest of the function into another
         ↪  function, this one is too long
        if (approved) {
            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .statusApproveOrNot
                .push(true);
           emit EvidenceToStatusApproveOrNotUpdated(evidenceIpfsHash, true);

            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .allSelectedVerifiersToFeedbackStatus[msg.sender] = true;
            emit EvidenceToAllSelectedVerifiersToFeedbackStatusUpdated(
                msg.sender,
                evidenceIpfsHash,
                true
            );
        } else {
            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .statusApproveOrNot
                .push(false);
            emit EvidenceToStatusApproveOrNotUpdated(evidenceIpfsHash,
↪   false);

            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .allSelectedVerifiersToFeedbackStatus[msg.sender] = false;
            emit EvidenceToAllSelectedVerifiersToFeedbackStatusUpdated(
                msg.sender,
                evidenceIpfsHash,
                false
            );
        }
```

```
        // get all the verifiers who provide feedback and call the function
         ↪  to earn rewards or get penalized

        // what if the evidenceIpfsHash is reassigned to other verifiers?
         ↪  The statusApproveOrNot length is reseted or not???
        // hold on, the check for the if statement will be passed if the same
         ↪  verifier just call multiple times of this function
        // And it will trigger the _earnRewardsOrGetPenalized function, any
         ↪  impact??
        // Yeah, the verifier can call multiple times of this function, and
         ↪  the verifier will be rewarded or penalized multiple times
        if (
            s_evidenceIpfsHashToItsInfo[evidenceIpfsHash]
                .statusApproveOrNot
                .length < s_numWords
        ) {
            return;
        } else {
            address[] memory allSelectedVerifiers =
↪   s_evidenceIpfsHashToItsInfo[
                evidenceIpfsHash
            ].selectedVerifiers;
           uint256 allSelectedVerifiersLength = allSelectedVerifiers.length;
            StructDefinition.VSkillUserSubmissionStatus evidenceStatus =
↪   _updateEvidenceStatus(
                    evidenceIpfsHash,
                    user
                );

            // @written audit-high the statusApproveOrNot array will call the
             ↪  .pop() function while empty with this setup of
             ↪  allSelectedVerifiersLength
            // when the evidence is different opinion for more than once.
            for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
                _earnRewardsOrGetPenalized(
                    evidenceIpfsHash,
                    allSelectedVerifiers[i],
                    evidenceStatus
                );
            }
        }
    }
```

Impact:

The maintainance of the contract will be difficult.

Recommended Mitigation:

Separate some logic into another function.

[I-14] The first verifier who submit the evidence will be rewarded more than the following verifiers

Description:

In the `Verifier::_rewardVerifiers` function, the logic is as follows:

```
function _rewardVerifiers(address verifiersAddress) internal {
        .
        .
        .
@>      uint256 rewardAmountInEth = (super.getBonusMoneyInEth() *
@>          currentReputation) /
@>          HIGHEST_REPUTATION /
@>          BONUS_DISTRIBUTION_NUMBER;

        super._rewardVerifierInFormOfStake(verifiersAddress,
         ↪  rewardAmountInEth);
    }
```

As the `rewardAmountInEth` is calculated based on the each time `super.getBonusMoneyInEth()`, thus the first verifier who submit the evidence will be rewarded more than the following verifiers since the `super.getBonusMoneyInEth()` will be decreased.

Gas

[G-1] Custom error message include a constant `Staking::minStakeUsdAmount` and `VSkillUser::submittedFeeInUsd` which costs more gas

Description:

In the `Staking` contract, the `Staking__NotEnoughStakeToBecomeVerifier` custom error message includes a constant `minStakeUsdAmount`. In the `VSkillUser` contract, the `VSkillUser__NotEnoughSubmissionFee` custom error message includes a constant `submittedFeeInUsd`.

```
 error Staking__NotEnoughStakeToBecomeVerifier(
        uint256 currentStakeUsdAmount,
@>      uint256 minStakeUsdAmount
    );

 error VSkillUser__NotEnoughSubmissionFee(
        uint256 requiredFeeInUsd,
@>      uint256 submittedFeeInUsd
    );
```

Impact:

This is not necessary and costs more gas.

[G-2] There are two functions work the same in `Staking` contract and is a waste of gas

Description:

In the `Staking` contract, these two functions `addBonusMoneyForVerifier` and `_addBonusMoney` are the same logic:

```
function addBonusMoneyForVerifier() public payable {
        s_bonusMoneyInEth += msg.value;
        emit BonusMoneyUpdated(
            s_bonusMoneyInEth - msg.value,
            s_bonusMoneyInEth
        );
    }

function _addBonusMoney(uint256 amountInEth) internal {
        s_bonusMoneyInEth += amountInEth;
        emit BonusMoneyUpdated(
            s_bonusMoneyInEth - amountInEth,
            s_bonusMoneyInEth
        );
    }
```

Impact:

This is a waste of gas.

[G-3] Double check in `Verifier::_earnRewardsOrGetPenalized` function, spend unnecessary gas

Description:

In the `Verifier` contract, the `_earnRewardsOrGetPenalized` function is only called by the `Verifier::provideFeedback` function and is marked as `internal`. However, it has also check for `_onlySelectedVerifier` function

```
function provideFeedback(
        string memory feedbackIpfsHash,
        string memory evidenceIpfsHash,
        address user,
        bool approved
    ) external {
@>      _onlySelectedVerifier(evidenceIpfsHash, msg.sender);
        .
        .
        .
            for (uint256 i = 0; i < allSelectedVerifiersLength; i++) {
@>              _earnRewardsOrGetPenalized(
                    evidenceIpfsHash,
                    allSelectedVerifiers[i],
                    evidenceStatus
                );
            }
        }
    }

function _earnRewardsOrGetPenalized(
        string memory evidenceIpfsHash,
        address verifierAddress,
        StructDefinition.VSkillUserSubmissionStatus evidenceStatus
    ) internal {
@>      _onlySelectedVerifier(evidenceIpfsHash, verifierAddress);
        .
        .
        .
    }
```

Impact:

This double check will cost unnecessary gas.

Recommended Mitigation:

Remove the `_onlySelectedVerifier` check in the `_earnRewardsOrGetPenalized` function.

```
function _earnRewardsOrGetPenalized(
        string memory evidenceIpfsHash,
        address verifierAddress,
        StructDefinition.VSkillUserSubmissionStatus evidenceStatus
    ) internal {
-       _onlySelectedVerifier(evidenceIpfsHash, verifierAddress);
        .
        .
        .
    }
```

[G-4] Compute the same `Verifier::keccak256(abi.encodePacked(evidenceIpfsHash))` costs unnecessary gas

Description:

In the `Verifier` contract, the `_onlySelectedVerifier` function computes the same `keccak256(abi.encodePacked(evidenceIpfsHash))` in each loop iteration. Also in the `_updateEvidenceStatus` function, the same `keccak256(abi.encodePacked(evidenceIpfsHash))` is computed in each loop iteration.

```
 function _onlySelectedVerifier(
        string memory evidenceIpfsHash,
        address verifierAddress
    ) internal view isVeifier {
        uint256 length = s_verifiers[s_addressToId[verifierAddress] - 1]
            .evidenceIpfsHash
            .length;
        for (uint256 i = 0; i < length; i++) {
            if (
                keccak256(
                    abi.encodePacked(
                        s_verifiers[s_addressToId[verifierAddress] - 1]
                            .evidenceIpfsHash[i]
                    )
@>              ) == keccak256(abi.encodePacked(evidenceIpfsHash))
            ) {
                return;
```

```
            }
        }
        revert Verifier__NotSelectedVerifier();
    }

function _updateEvidenceStatus(
        string memory evidenceIpfsHash,
        address user
    ) internal returns (StructDefinition.VSkillUserSubmissionStatus) {
        .
        .
        .
        for (uint256 i = 0; i < length; i++) {
            if (
                keccak256(
                    abi.encodePacked(
                        s_addressToEvidences[user][i].evidenceIpfsHash
                    )
@>              ) == keccak256(abi.encodePacked(evidenceIpfsHash))
            ) {
                currentEvidenceIndex = i;
                break;
            }
        }
        .
        .
        .
    }
```

Impact:

Since the `evidenceIpfsHash` is the same in each loop iteration, this will cost unnecessary gas.

Recommended Mitigation:

Use the memory variable to store the `keccak256(abi.encodePacked(evidenceIpfsHash))` value.

```
 function _onlySelectedVerifier(
        string memory evidenceIpfsHash,
        address verifierAddress
    ) internal view isVeifier {
+       bytes32 evidenceHash = keccak256(abi.encodePacked(evidenceIpfsHash));
        uint256 length = s_verifiers[s_addressToId[verifierAddress] - 1]
            .evidenceIpfsHash
```

```
            .length;
        for (uint256 i = 0; i < length; i++) {
            if (
                keccak256(
                    abi.encodePacked(
                        s_verifiers[s_addressToId[verifierAddress] - 1]
                            .evidenceIpfsHash[i]
                    )
                ) == evidenceHash
            ) {
                return;
            }
        }
        revert Verifier__NotSelectedVerifier();
    }

function _updateEvidenceStatus(
        string memory evidenceIpfsHash,
        address user
    ) internal returns (StructDefinition.VSkillUserSubmissionStatus) {
+       bytes32 evidenceHash = keccak256(abi.encodePacked(evidenceIpfsHash));
        .
        .
        .
        for (uint256 i = 0; i < length; i++) {
            if (
                keccak256(
                    abi.encodePacked(
                        s_addressToEvidences[user][i].evidenceIpfsHash
                    )
                ) == evidenceHash
            ) {
                currentEvidenceIndex = i;
                break;
            }
        }
        .
        .
        .
    }
```