# VSkill Audit Report

Version 2.0

*Luo Yingjie*

March 9, 2025

# VSkill Audit Report

Luo Yingjie

March 9, 2025

Prepared by: Luo Yingjie Lead Auditors:

- Luo Yingjie

Assisting Auditors:

- None

## Table of Contents

## 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 |
| | 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

- Commit Hash: 02ceaa15a2f32ac00ffbb693996cddffa909bd57

### Scope

```
./src/
#-- Distribution.sol
#-- Relayer.sol
#-- Staking.sol
#-- VSkillUser.sol
#-- VSkillUserNft.sol
#-- Verifier.sol
#-- interfaces
```

```
|   #-- IAutomationRegistrar.sol
|   #-- IRelayer.sol
#-- library
|   #-- OracleLib.sol
|   #-- PriceCoverter.sol
|   #-- StructDefinition.sol
#-- optimizedGas
    #-- RelayerYul.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 | 4 |
| Medium | 1 |
| Low | 2 |
| Info | 7 |
| Gas Optimizations | 2 |
| Total | 16 |

# Findings

## High

### [H-1] Miss stable price check in `PriceConverter` library

**Description:**

The function `PriceConverter::getChainlinkDataFeedLatestAnswer` calls `latestRoundData` instead of `staleCheckLatestRoundData` from `OracleLib`. Although `staleCheckLatestRoundData` is available, it is not used, leading to a risk of processing outdated or incorrect price data.

```
(
    /* uint80 roundID */,
    int answer,
    /*uint startedAt*/,
    /*uint timeStamp*/,
    /*uint80 answeredInRound*/
@> ) = priceFeed.latestRoundData();
```

**Impact:**

If the `Chainlink` oracle experiences issues and provides outdated or incorrect price data, contracts depending on this function for ETH/USD conversion could process stale prices, potentially leading to severe financial losses.

**Proof of Concept:**

Check the following test case in `test/v2/auditTests/ProofOfCodes.t.sol`:

Proof of Code

```
function testUnstablePriceFeedRevert() external {
        uint256 ethAmount = 1 ether;
        vm.expectRevert(OracleLib.OracleLib__StalePrice.selector);
        ethAmount.correctConvertEthToUsd(maliciousMockAggregator);
        // However, the current price feed will not revert
        uint256 outdatedData = ethAmount.convertEthToUsd(
            maliciousMockAggregator
        );
        console.log("This is the outdated data: ", outdatedData);
    }
```

The `maliciousMockAggregator` is a mock price feed returning outdated price data:

```
/// @notice This is the malicious part, we return the previous round as the
 ↪  latest round
    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        return (
            uint80(latestRound),
            getAnswer[latestRound],
            getStartedAt[latestRound],
            getTimestamp[latestRound],
@>          uint80(latestRound) - 1
        );
    }
```

Using the correct function `CorrectPriceConverter::correctConvertEthToUsd`, it reverts due to outdated price detection:

```
function staleCheckLatestRoundData(
        AggregatorV3Interface chainlinkFeed
    ) public view returns (uint80, int256, uint256, uint256, uint80) {
        (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        ) = chainlinkFeed.latestRoundData();

@>      if (updatedAt == 0 || answeredInRound < roundId) {
            revert OracleLib__StalePrice();
        }
        uint256 secondsSince = block.timestamp - updatedAt;
        if (secondsSince > TIMEOUT) revert OracleLib__StalePrice();

        return (roundId, answer, startedAt, updatedAt, answeredInRound);
```

```
    }
```

Here, answeredInRound is less than roundId, triggering a revert:

```
 function getChainlinkDataFeedLatestAnswer(
        AggregatorV3Interface priceFeed
    ) internal view returns (int) {
        // prettier-ignore
        (
            /* uint80 roundID */,
            int answer,
            /*uint startedAt*/,
            /*uint timeStamp*/,
            /*uint80 answeredInRound*/
@>      ) = priceFeed.staleCheckLatestRoundData();

        return (answer * int(DECIMALS)) / int(PRICE_FEED_DECIMALS);
    }
```

However, using the current function PriceConverter::convertEthToUsd, it does not revert and instead returns outdated price data:

```
[38464] ProofOfCodes::testUnstablePriceFeedRevert()
    ├─ [0] VM::expectRevert(custom error 0xc31eb0e0: ġ  :)
    │   └─ ← [Return]
    ├─ [12672] OracleLib::staleCheckLatestRoundData() [delegatecall]
    │   ├─ [9199] MaliciousMockV3Aggregator::latestRoundData() [staticcall]
    │   │   └─ ← [Return] 1, 2000, 1, 1, 0
    │   └─ ← [Revert] OracleLib__StalePrice()
    ├─ [9199] MaliciousMockV3Aggregator::latestRoundData() [staticcall]
    │   └─ ← [Return] 1, 2000, 1, 1, 0
    ├─ [0] console::log("This is the outdated data: ", 20000000000000
    ↪  [2e13]) [staticcall]
    │   └─ ← [Stop]
    └─ ← [Stop]
```

**Recommended Mitigation:**

Modify PriceConverter::getChainlinkDataFeedLatestAnswer to use staleCheck-LatestRoundData from OracleLib instead of latestRoundData. This ensures that stale or invalid price data is not used.

```
(
    /* uint80 roundID */,
    int answer,
```

```
    /*uint startedAt*/,
    /*uint timeStamp*/,
    /*uint80 answeredInRound*/
- ) = priceFeed.latestRoundData();
+ ) = priceFeed.staleCheckLatestRoundData();
```

### [H-2] `Verifier` locks Ether without a withdraw function.

**Description:**

The `Verifier` contract does not implement a proper withdrawal function for accidentally sent Ether. Since the contract inherits from `Staking`, which defines a `receive()` function but does not handle unintended Ether deposits, any Ether sent to the contract will be permanently locked.

In `Staking` contract:

```
@>  receive() external payable {}

    fallback() external payable {
        stake();
    }
```

This issue arises because the `Verifier` contract does not provide a way for the owner or any user to retrieve mistakenly sent Ether. The only way to withdraw Ether from the contract is `withdrawStakeAndLoseVerifier()` and `withdrawReward()` which can only withdraw the eth that is either staked or the reward.

In `Verifier` contract:

```
 function withdrawStakeAndLoseVerifier() public onlyInitialized {
        if (s_verifierToInfo[msg.sender].unhandledRequestCount > 0) {
            revert Verifier__ExistUnhandledEvidence();
        }
@>      super.withdrawStake();
    }

 function withdrawReward() public onlyInitialized {
        s_verifierToInfo[msg.sender].reward = 0;
        (bool success, ) = msg.sender.call{
@>          value: s_verifierToInfo[msg.sender].reward
        }("");
        if (!success) {
            revert Staking__WithdrawFailed();
```

```
        }

        emit Withdrawn(msg.sender, s_verifierToInfo[msg.sender].reward);
    }
```

In `Staking` contract:

```
function withdrawStake() internal onlyVerifier {
        s_verifierCount -= 1;
        s_addressToIsVerifier[msg.sender] = false;
        delete s_verifierToInfo[msg.sender];

@>      (bool success, ) = msg.sender.call{value: STAKE_ETH_AMOUNT}("");
        if (!success) {
            revert Staking__WithdrawFailed();
        }

        emit LoseVerifier(msg.sender);
        emit Withdrawn(msg.sender, STAKE_ETH_AMOUNT);
    }
```

**Impact:**

- Users who send Ether to the `Verifier` contract by mistake cannot withdraw it.
- The contract may accumulate unintended funds that are forever locked.
- If users expect a refund mechanism, they may experience financial loss due to the missing withdrawal functionality.

**Proof of Concept:**

Check the following test case in `test/v2/auditTests/ProofOfCodes.t.sol`:

Proof of Code

```
function testVerifierLocksEth() external {
        address ethLocker = makeAddr("ethLocker");
        deal(ethLocker, 10 ether);
        vm.prank(ethLocker);
        (bool success, ) = address(verifier).call{value: 1 ether}("");
        assertEq(success, true);

        // This will fail because it will trigger the `stake()` function
        // which will revert if the user has not provided the correct amount
         ↪  of eth
```

```
        // Even if someone send the correct amount of eth, he will be a
         ↪  verifier and can withdraw the eth back
        vm.prank(ethLocker);
        (bool fallbackSuccess, ) = address(verifier).call{value: 1 ether}(
            "Give my Eth back!"
        );
        assertEq(fallbackSuccess, false);
    }
```

- This test sends 1 ETH to the `Verifier` contract.
- There is no function to withdraw this ETH.
- As a result, the funds are irretrievable.

**Recommended Mitigation:**

1. Override the `receive()` function in the `Verifier` contract and store the received Ether in a separate variable. And implement a withdrawal function that allows the owner to send the locked Ether back to the sender.

In `Staking` contract:

```
-   receive() external payable {}
+   receive() external payable virtual {}
```

In `Verifier` contract:

```
+   mapping(address accidentLocker => uint256 lockedEth) private
 ↪  s_accidentLockerToEth;

+   receive() external payable override {
+        s_accidentLockerToEth[msg.sender] += msg.value;
+    }

+   sendAccidentEth(address to) external onlyOwner {
+       if(s_accidentLockerToEth[to] <=0 ) {
+           revert Verifier__NoAccidentEth();
+       }
+       s_accidentLockerToEth[to] = 0;
+       (bool success, ) = to.call{value: s_accidentLockerToEth[to]}("");
+       if (!success) {
+           revert Verifier__SendAccidentEthFailed();
+       }
+       emit AccidentEthSent(to, s_accidentLockerToEth[to]);
+    }
```

### [H-3] Incomplete Verifier Deletion Allows Removed Verifiers to Be Selected

**Description:**

In the `Verifier::withdrawStakeAndLoseVerifier` function, the current implementation does not fully remove a verifier from all relevant mappings and arrays. Specifically, while the verifier is removed from `s_verifierToInfo`, they are still present in `s_skillDomainToVerifiersWithinSameDoma` As a result, they can still be selected as verifiers even after withdrawing their stake as in `Relayer` contract, the `Relayer::assignEvidenceToVerifiers` function selects the verifiers based on this `s_skillDomainToVerifiersWithinSameDomain` mapping:

In `Verifier` contract:

```
    function withdrawStakeAndLoseVerifier() public onlyInitialized {
        if (s_verifierToInfo[msg.sender].unhandledRequestCount > 0) {
            revert Verifier__ExistUnhandledEvidence();
        }
@>      super.withdrawStake();
    }
```

We only called the `withdrawStake()` function as below:

```
 function withdrawStake() internal onlyVerifier {
@>      s_verifierCount -= 1;
@>      s_addressToIsVerifier[msg.sender] = false;
@>      delete s_verifierToInfo[msg.sender];

        (bool success, ) = msg.sender.call{value: STAKE_ETH_AMOUNT}("");
        if (!success) {
            revert Staking__WithdrawFailed();
        }

        emit LoseVerifier(msg.sender);
        emit Withdrawn(msg.sender, STAKE_ETH_AMOUNT);
    }
```

As we can find, the `s_skillDomainToVerifiersWithinSameDomain` mapping is not updated in the `withdrawStakeAndLoseVerifier` function.

However, in `Relayer` contract, which handles the verifier selection, the `assignEvidenceToVerifiers` function selects verifiers based on the `s_skillDomainToVerifiersWithinSameDomain` mapping:

```
 function assignEvidenceToVerifiers() external onlyOwner {
        .
        .
        .
        for (uint256 i = 0; i < length; i++) {
            uint256 requestId = s_unhandledRequestIds[i];
            uint256[]
                memory randomWordsWithinRange =
↪   s_requestIdToRandomWordsWithinRange[
                    requestId
                ];
@>          address[] memory verifiersWithinSameDomain = i_verifier
@>              .getSkillDomainToVerifiersWithinSameDomain(
@>                i_vSkillUser.getRequestIdToEvidence(requestId).skillDomain
@>              );
        .
        .
        .
    }
```

**Impact:**

- Even though the verifier is no longer active, they can still be randomly assigned new verification tasks.
- This can disrupt the verification process and potentially allow an unqualified verifier to be assigned tasks.
- It introduces inconsistencies in the system, where the internal state does not accurately reflect the verifier's status.

**Proof of Concept:**

1. A verifier calls `withdrawStakeAndLoseVerifier()`.
2. The contract removes them from `s_verifierToInfo` but does not remove them from `s_skillDomainToVerifiersWithinSameDomain`.
3. The function `assignEvidenceToVerifiers` still selects verifiers based on `s_skillDomainToVerifie` which includes the removed verifier.
4. The removed verifier may still be assigned verification tasks and ruin this protocol.

Check the following test case in `test/v2/auditTests/ProofOfCodes.t.sol`:

Proof of Code

```
function testVerifierDeletionIsNotComplete() external {
        address verifierAddress = makeAddr("verifier");
```

```
        deal(verifierAddress, 1 ether);

        vm.startPrank(verifierAddress);
        verifier.stake{value: verifier.getStakeEthAmount()}();
        verifier.addSkillDomain("Blockchain");

        uint256 verifierWithinSameDomainLength = verifier
            .getSkillDomainToVerifiersWithinSameDomainLength("Blockchain");

        verifier.withdrawStakeAndLoseVerifier();
        uint256 verifierWithinSameDomainLengthAfterDeletion = verifier
            .getSkillDomainToVerifiersWithinSameDomainLength("Blockchain");

        vm.stopPrank();

        console.log(
            "Verifier within same domain length before deletion: ",
            verifierWithinSameDomainLength
        );
        console.log(
            "Verifier within same domain length after deletion: ",
            verifierWithinSameDomainLengthAfterDeletion
        );
        console.log("You can find that the deletion is not complete!");
        console.log(
            "Thus, even if the verifier is deleted, he can still be
             ↪  selected!!!"
        );
    }
```

**Recommended Mitigation:**

Ensure that when a verifier withdraws their stake, they are also removed from `s_skillDomainToVerifiersWith`
Modify `withdrawStakeAndLoseVerifier()` as follows:

```
function withdrawStakeAndLoseVerifier() public onlyInitialized {
        if (s_verifierToInfo[msg.sender].unhandledRequestCount > 0) {
            revert Verifier__ExistUnhandledEvidence();
        }
        super.withdrawStake();

+         string[] memory skillDomains =
 ↪  s_verifierToInfo[msg.sender].skillDomains;
+         uint256 length = skillDomains.length;
```

```
+           _removeVerifierFromSkillDomain(skillDomains, length, msg.sender);
    }
```

Where this `_removeVerifierFromSkillDomain` function can be designed as you wish, but it will be better create another variable like a mapping from the skill domain to the verifiers and remove the verifier from this mapping as this will be much gas efficient.

### [H-4] `Relayer::assignEvidenceToVerifiers` Can Revert if a Verifier Leaves Before Assignment

**Description:**

The `assignEvidenceToVerifiers` function uses pre-generated random indices (`randomWordsWithinRang` to assign verifiers. However, if a verifier withdraws their stake before this function is executed, the total number of verifiers in the domain will decrease, causing `randomWordsWithinRange` to contain out-of-range indices, leading to a revert.

Here the `randomWordsWithinRange` is generated by `Chainlink` nodes once they listened this `Distribution::RequestIdToRandomWordsUpdated` event:

```
    // Here we just store select randomWords in the range of the
     ↪  verifierWithinSameDomainLength
    // And store the requestId in the s_unhandledRequestIds array
    // As for the assignment, we will handle this in batches to reduce gas
     ↪  costs
    function performUpkeep(
        bytes calldata performData
    ) external override onlyForwarder {
        uint256 requestId = abi.decode(performData, (uint256));
        string memory skillDomain = i_vSkillUser
            .getRequestIdToEvidenceSkillDomain(requestId);
        uint256[] memory randomWords =
 ↪  i_distribution.getRandomWords(requestId);

        // As for the number of verifiers enough or not, since we only
         ↪  require 3 verifiers
        // At the very beginning of the project, we will make sure that the
         ↪  number of verifiers is enough for each skill domain(above 3)
        // we always has three verifiers in the community to make sure it's
         ↪  always enough
        // After that we will allow users to submit the evidence
@>      uint256 verifierWithinSameDomainLength = i_verifier
@>          .getSkillDomainToVerifiersWithinSameDomainLength(skillDomain);
```

```
        if (verifierWithinSameDomainLength < NUM_WORDS) {
            emit Relayer__NotEnoughVerifierForThisSkillDomainYet();
            return;
        }
        // get the randomWords within the range of the
         ↪  verifierWithinSameDomainLength
        // here the length is just 3, no worries about DoS attack
@>      for (uint8 i = 0; i < NUM_WORDS; i++) {
@>          randomWords[i] = randomWords[i] % verifierWithinSameDomainLength;
@>      }
        // check the current randomWords is unique or not
        // if not, we will need to modify the randomWords a bit
        // since the length is only 3, we can afford to do this
        randomWords = _makeRandomWordsUnique(
            randomWords,
            verifierWithinSameDomainLength
        );
@>      s_requestIdToRandomWordsWithinRange[requestId] = randomWords;
        s_unhandledRequestIds.push(requestId);
        emit Relayer__UnhandledRequestIdAdded(s_unhandledRequestIds.length);
    }
```

However, the actual assignment is done in the function `assignEvidenceToVerifiers` which will be called daily by the owner from the comments:

```
@>  // These functions will be called daily by the owner, the automation
 ↪  process will be set on the browser

    // This will be a very gas cost function as it will assign all the
     ↪  evidence to the verifiers
    // Try to reduce the gas cost as much as possible
    // We are the only one (owner or the profit beneficiary) who can call
     ↪  this function

    // once the verifiers are assigned, the verifier will get the
     ↪  notification on the frontend(listening to the event)
    // and they can start to provide feedback to the specific evidence

    // set the assigned verifiers as the one who can change the evidence
     ↪  status
    function assignEvidenceToVerifiers() external onlyOwner {
        uint256 length = s_unhandledRequestIds.length;
```

```
        // if there is no unhandled request, we will return so that we don't
         ↪  waste gas
        if (length == 0) {
            return;
        }
        // update the batch number
        s_batchToProcessedRequestIds[s_batchProcessed] =
 ↪  s_unhandledRequestIds;
        s_batchToDeadline[s_batchProcessed] = block.timestamp + DEADLINE;
        s_batchProcessed++;

        // the length can be very large, but we will monitor the event to
         ↪  track the length and avoid DoS attack
        for (uint256 i = 0; i < length; i++) {
            uint256 requestId = s_unhandledRequestIds[i];
@>          uint256[]
@>              memory randomWordsWithinRange =
 ↪  s_requestIdToRandomWordsWithinRange[
@>                  requestId
@>              ];

@>          address[] memory verifiersWithinSameDomain = i_verifier
@>              .getSkillDomainToVerifiersWithinSameDomain(
@>                i_vSkillUser.getRequestIdToEvidence(requestId).skillDomain
@>              );
            for (uint8 j = 0; j < randomWordsWithinRange.length; j++) {
@>              s_requestIdToVerifiersAssigned[requestId].push(
@>                  verifiersWithinSameDomain[randomWordsWithinRange[j]]
@>              );
                i_verifier.setVerifierAssignedRequestIds(
                    requestId,
                    verifiersWithinSameDomain[randomWordsWithinRange[j]]
                );
                i_verifier.addVerifierUnhandledRequestCount(
                    verifiersWithinSameDomain[randomWordsWithinRange[j]]
                );
                // only 7 days allowed for the verifiers to provide feedback
                i_vSkillUser.setDeadline(requestId, block.timestamp +
 ↪  DEADLINE);
            }
        }
        delete s_unhandledRequestIds;
```

```
        emit Relayer__EvidenceAssignedToVerifiers();
    }
```

As we can find in the code, it uses the most up-to-date `verifiersWithinSameDomain` array, but the `randomWordsWithinRange` is generated before the assignment, which can be greater than the current range if any verifier leaves the system before the assignment. This can cause a revert in the function.(out-of-range index)

**Impact:**

- If a verifier leaves before the assignment, the function will attempt to access an out-of-range index, causing a revert.
- This can halt the verification process entirely, leading to a denial-of-service (DoS) scenario.
- Since verifiers are assigned evidence randomly, this bug can be exploited by malicious actors to prevent evidence processing.

**Proof of Concept:**

Check the following test case in `test/v2/auditTests/ProofOfCodes.t.sol`:

Proof of Code

```
function testVerifierLeavesBeforeAssignedWillRevert() external {
        (address[] memory selectedVerifiers, ) = _setUpForRelayer();

        // Before assigning the evidence to the verifiers, one of them leaves
        vm.startPrank(selectedVerifiers[0]);
        verifier.withdrawStakeAndLoseVerifier();
        vm.stopPrank();

        vm.startPrank(relayer.owner());
        // Here it will not revert because the verifier deletion process is
        ↪  not complete
        // But if we complete this process, it will revert
        relayer.assignEvidenceToVerifiers();
        vm.stopPrank();

        console.log("Current verifiers length: ",
        ↪  verifier.getVerifierCount());
        console.log(
            "But we can find that even though the verifier has left, he can
            ↪  still be selected!"
        );
    }
```

And here is the log:

```
├─ [0] VM::startPrank(ECRecover:
↪  [0x0000000000000000000000000000000000000001])
│    └─ ← [Return]
├─ [40437] Verifier::withdrawStakeAndLoseVerifier()
│    ├─ [3000] ECRecover::fallback{value: 100000000000000000}()
│    │    └─ ← [Return]
│    ├─ emit LoseVerifier(verifier: ECRecover:
↪  [0x0000000000000000000000000000000000000001])
│    ├─ emit Withdrawn(staker: ECRecover:
↪  [0x0000000000000000000000000000000000000001], amount:
↪  100000000000000000 [1e17])
│    └─ ← [Stop]
├─ [0] VM::stopPrank()
│    └─ ← [Return]
├─ [2363] Relayer::owner() [staticcall]
│    └─ ← [Return] DefaultSender:
↪  [0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38]
├─ [0] VM::startPrank(DefaultSender:
↪  [0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38])
│    └─ ← [Return]
├─ [392915] Relayer::assignEvidenceToVerifiers()
│    ├─ [5065] VSkillUser::getRequestIdToEvidence(1) [staticcall]
│    │    └─ ← [Return] VSkillUserEvidence({ submitter:
↪  0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D, cid: "cid", skillDomain:
↪  "Blockchain", status: 0, statusApproveOrNot: [false, false, false],
↪  feedbackCids: [], deadline: 1 })
│    ├─ [2102]
↪  Verifier::getSkillDomainToVerifiersWithinSameDomain("Blockchain")
↪  [staticcall]
│    │    └─ ← [Return] [0x0000000000000000000000000000000000000001,
↪  0x0000000000000000000000000000000000000002,
↪  0x0000000000000000000000000000000000000003]
│    ├─ [43017] Verifier::setVerifierAssignedRequestIds(1, RIPEMD-160:
↪  [0x0000000000000000000000000000000000000003])
│    │    └─ ← [Stop]
│    ├─ [20901] Verifier::addVerifierUnhandledRequestCount(RIPEMD-160:
↪  [0x0000000000000000000000000000000000000003])
│    │    └─ ← [Stop]
│    ├─ [746] VSkillUser::setDeadline(1, 604801 [6.048e5])
│    │    └─ ← [Stop]
```

```
    │   ├─ [43017] Verifier::setVerifierAssignedRequestIds(1, ECRecover:
    ↪ [0x0000000000000000000000000000000000000001])
    │   │   └─ ← [Stop]
    │   ├─ [20901] Verifier::addVerifierUnhandledRequestCount(ECRecover:
    ↪ [0x0000000000000000000000000000000000000001])
    │   │   └─ ← [Stop]
    │   ├─ [746] VSkillUser::setDeadline(1, 604801 [6.048e5])
    │   │   └─ ← [Stop]
    │   ├─ [43017] Verifier::setVerifierAssignedRequestIds(1, SHA-256:
    ↪ [0x0000000000000000000000000000000000000002])
    │   │   └─ ← [Stop]
    │   ├─ [20901] Verifier::addVerifierUnhandledRequestCount(SHA-256:
    ↪ [0x0000000000000000000000000000000000000002])
    │   │   └─ ← [Stop]
    │   ├─ [746] VSkillUser::setDeadline(1, 604801 [6.048e5])
    │   │   └─ ← [Stop]
    │   ├─ emit Relayer__EvidenceAssignedToVerifiers()
    │   └─ ← [Stop]
    ├─ [0] VM::stopPrank()
```

We can find that the verifier with the address `0x000...1` has left the system, but he can still be selected for the assignment.

**Recommended Mitigation:**

Modify `assignEvidenceToVerifiers` to validate each `randomWordsWithinRange` entry before using it. If a random index is out of range due to verifier withdrawal, regenerate it using a random salt provided by the owner to ensure unpredictability.

```
- function assignEvidenceToVerifiers() external onlyOwner {
+ function assignEvidenceToVerifiers(uint256 salt) external onlyOwner {
        uint256 length = s_unhandledRequestIds.length;
        // if there is no unhandled request, we will return so that we don't
 ↪  waste gas
        if (length == 0) {
            return;
        }
        // update the batch number
        s_batchToProcessedRequestIds[s_batchProcessed] =
 ↪  s_unhandledRequestIds;
        s_batchToDeadline[s_batchProcessed] = block.timestamp + DEADLINE;
        s_batchProcessed++;
```

```
        // the length can be very large, but we will monitor the event to
↪   track the length and avoid DoS attack
        for (uint256 i = 0; i < length; i++) {
            uint256 requestId = s_unhandledRequestIds[i];
            uint256[]
                memory randomWordsWithinRange =
↪   s_requestIdToRandomWordsWithinRange[
                    requestId
                ];

            address[] memory verifiersWithinSameDomain = i_verifier
                .getSkillDomainToVerifiersWithinSameDomain(
                  i_vSkillUser.getRequestIdToEvidence(requestId).skillDomain
                );

+           uint256 maxRandomWord = _getMaxValueFromArray(
+               randomWordsWithinRange
+           );

+           if (maxRandomWord >= verifiersWithinSameDomain.length) {
+               // if out of range, we will regenerate the randomWords
+               randomWordsWithinRange = _regenerateRandomWords(
+                   randomWordsWithinRange,
+                   verifiersWithinSameDomain.length,
+                   salt
+               );
+           }

            for (uint8 j = 0; j < randomWordsWithinRange.length; j++) {
                s_requestIdToVerifiersAssigned[requestId].push(
                    verifiersWithinSameDomain[randomWordsWithinRange[j]]
                );
                i_verifier.setVerifierAssignedRequestIds(
                    requestId,
                    verifiersWithinSameDomain[randomWordsWithinRange[j]]
                );
                i_verifier.addVerifierUnhandledRequestCount(
                    verifiersWithinSameDomain[randomWordsWithinRange[j]]
                );
                // only 7 days allowed for the verifiers to provide feedback
                i_vSkillUser.setDeadline(requestId, block.timestamp +
↪   DEADLINE);
            }
```

```
        }
        delete s_unhandledRequestIds;

        emit Relayer__EvidenceAssignedToVerifiers();
    }


+ function _regenerateRandomWords(
+       uint256[] memory currentRandomWords,
+       uint256 verifierLength,
+       uint256 salt
+   ) private pure returns (uint256[] memory) {
+       uint256 length = currentRandomWords.length;
+       for (uint256 i = 0; i < length; i++) {
+           if (currentRandomWords[i] >= verifierLength) {
+               // Generate a new valid index using a combination of salt and
 ↪  the original random word
+                currentRandomWords[i] =
+                    uint256(
+                        keccak256(
+                            abi.encodePacked(salt, currentRandomWords[i], i)
+                        )
+                    ) %
+                    verifierLength;
+           }
+       }
+       return currentRandomWords;
+   }


+ function _getMaxValueFromArray(
+       uint256[] memory array
+   ) private pure returns (uint256) {
+       uint256 max = 0;
+       for (uint256 i = 0; i < array.length; i++) {
+           if (array[i] > max) {
+               max = array[i];
+           }
+       }
+       return max;
+   }
```

## Medium

### [M-1] Lack of Getter Function for Verifiers to Retrieve Their Assigned Request IDs

**Description:**

The `Verifier` contract assigns verifiers to evidence requests but does not provide a way for them to retrieve their assigned request IDs. This omission forces verifiers to rely on off-chain tracking or event logs, which may be inefficient or unreliable.

Currently, assigned request IDs are stored in `s_verifierToInfo[msg.sender].assignedRequestIds` but cannot be accessed through a public or external function.

**Impact:**

- Verifiers cannot directly check which requests they have been assigned.
- This forces reliance on event logs, which may not be accessible in some cases.
- Makes it harder for verifiers to track pending tasks, increasing the likelihood of missing deadlines and penalties.

**Recommended Mitigation:**

Implement a public getter function for verifiers to check their assigned request IDs.

```
+ function getVerifierAssignedRequestIds(
+        address verifier
+    ) external view returns (uint256[] memory) {
+        return s_verifierToInfo[verifier].assignedRequestIds;
+    }
```

## Low

### [L-1] Incorrect Custom Error in `VSkillUser::_calledByVerifierContract`

**Description:** The `_calledByVerifierContract` function incorrectly uses the `VSkillUser__NotRelayer` error when reverting. The intended check is for the `Verifier` contract, not the `Relayer`.

```
 function _calledByVerifierContract() internal view {
@>      if (msg.sender != Relayer(i_relayer).getVerifierContractAddress()) {
@>          revert VSkillUser__NotRelayer();
        }
    }
```

The error message should clearly indicate that the caller is not the `Verifier` contract, not that it's not the `Relayer`.

**Impact:**

- Misleading error message: Developers debugging contract interactions may misunderstand the issue.
- Harder troubleshooting: Incorrect error messages can make it difficult to determine the real cause of a failed transaction.

**Recommended Mitigation:**

Replace the incorrect error message with a more appropriate one:

```
+   error VSkillUser__NotVerifierContract();
function _calledByVerifierContract() internal view {
    if (msg.sender != Relayer(i_relayer).getVerifierContractAddress()) {
-       revert VSkillUser__NotRelayer(); // Incorrect
+       revert VSkillUser__NotVerifierContract(); // Correct
    }
}
```

### [L-2] Potential DoS Risk Due to Excessive Assignments in `Relayer::assignEvidenceToVerifiers`

**Description:**

The `assignEvidenceToVerifiers` function assigns verifiers to evidence requests by randomly selecting verifiers from `verifiersWithinSameDomain`. However, the function does not impose a cap on the number of assignments a single verifier can receive, leading to potential overload.

In `Relayer` contract:

```
 function assignEvidenceToVerifiers() external onlyOwner {
        .
        .
        .
        for (uint256 i = 0; i < length; i++) {
            uint256 requestId = s_unhandledRequestIds[i];
            uint256[]
                memory randomWordsWithinRange =
↪   s_requestIdToRandomWordsWithinRange[
                    requestId
                ];
```

```
@>          address[] memory verifiersWithinSameDomain = i_verifier
@>              .getSkillDomainToVerifiersWithinSameDomain(
@>                i_vSkillUser.getRequestIdToEvidence(requestId).skillDomain
@>              );
@>          for (uint8 j = 0; j < randomWordsWithinRange.length; j++) {
@>              s_requestIdToVerifiersAssigned[requestId].push(
@>                  verifiersWithinSameDomain[randomWordsWithinRange[j]]
@>              );
@>              i_verifier.setVerifierAssignedRequestIds(
@>                  requestId,
@>                  verifiersWithinSameDomain[randomWordsWithinRange[j]]
@>              );
@>              i_verifier.addVerifierUnhandledRequestCount(
@>                  verifiersWithinSameDomain[randomWordsWithinRange[j]]
@>              );
@>              // only 7 days allowed for the verifiers to provide feedback
@>              i_vSkillUser.setDeadline(requestId, block.timestamp +
 ↪  DEADLINE);
@>          }
@>      }
        .
        .
        .
    }
```

And if verifier cannot provide the feedback within deadline, they will be a punishment for verifiers:

```
 // punish is lose the verifier, not the same as penalize
    function punishVerifier(
        address verifier
    ) public onlyInitialized onlyRelayer {
        // take all the stake out and remove the verifier
        s_addressToIsVerifier[verifier] = false;
        s_verifierCount -= 1;
        // what about the stake money? The money will be collected by the
         ↪  staking contract and will be used to reward the verifiers who
         ↪  provide feedback
        // also those rewards will be used to reward the verifiers who
         ↪  provide feedback
        s_reward += super.getStakeEthAmount();
        s_reward += s_verifierToInfo[verifier].reward;
        delete s_verifierToInfo[verifier];

        emit LoseVerifier(verifier);
```

```
    }
```

There is a risk that a verifier is overloaded with too many requests, preventing them from providing timely feedback. This can lead to:

1. Denial of Service (DoS): If a verifier has an excessive number of pending requests, they might be unable to process all of them before deadlines expire.
2. Unfair Stake Loss: Since verifiers must provide feedback before a deadline, failing to do so results in penalties, potentially leading to the loss of their stake.
3. Imbalanced Workload: If certain domains have fewer verifiers, some might receive significantly more requests than others, worsening the issue.

**Impact:**

- A single verifier could be assigned an excessive number of requests.
- If they cannot provide feedback on all requests before deadlines, they will be penalized unfairly.
- If a skill domain has too few verifiers, they could be overloaded, creating a bottleneck in the verification process.
- This may lead to unintended stake losses and discourage participation.

**Proof of Concept:**

1. Deploy the contract and register a small number of verifiers in a specific domain.
2. Submit a large number of evidence requests.
3. Observe that the few available verifiers receive an excessive number of assignments.
4. If verifiers fail to meet deadlines, they are penalized, even if the workload is too high.

**Recommended Mitigation:**

1. Enforce an Assignment Cap: Before assigning a request, check if a verifier is already handling too many requests.
2. Ensure Fair Distribution:
    - If a verifier is overloaded, select another one from the pool.
    - If all verifiers in a domain are overloaded, prioritize those with the fewest assignments.

## Informational

### [I-1] Incorrect File Name: `PriceCoverter` Should Be `PriceConverter`

**Description:**

The filename for the `PriceConverter` library is currently **`PriceCoverter.sol`**, which appears to be a typo. The correct spelling should be **`PriceConverter.sol`** to align with the library's actual name declared in the contract.

**Impact:**

- This typo may cause **import errors** or confusion when referencing the library in other contracts.
- Developers might unintentionally reference the wrong file name, leading to deployment or compilation issues.
- Consistency across file names and contract/library names is a best practice for maintainability and readability.

**Recommended Mitigation:**

Rename the file from **`PriceCoverter.sol`** to **`PriceConverter.sol`** to match the library name and maintain consistency.

```
- PriceCoverter.sol
+ PriceConverter.sol
```

Additionally, ensure all import statements in other contracts reflect the correct filename:

```
- import {PriceConverter} from "./PriceCoverter.sol";
+ import {PriceConverter} from "./PriceConverter.sol";
```

#### [I-2] Unused Custom Error

**Description:**

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

- Found in src/VSkillUser.sol Line: 22

  ```
  error VSkillUser__EvidenceNotApprovedYet(
  ```

- Found in src/VSkillUserNft.sol Line: 31

  ```
  error VSkillUserNft__NotSkillHandler();
  ```

#### [I-3] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

**Description:**

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). Unless there is a compelling reason, `abi.encode` should be preferred. If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

2 Found Instances

- Found in src/VSkillUserNft.sol Line: 139

```
abi.encodePacked(
```

- Found in src/VSkillUserNft.sol Line: 143

```
abi.encodePacked(
```

#### [I-4] Excess Submission Fee Taken

**Description:**

In the `VSkillUser::submitEvidence` function, if a user submits more than the required `s_submissionFeeInUsd`, the excess amount is not refunded.

```
 function submitEvidence(
        string memory cid,
        string memory skillDomain
    ) public payable onlyInitialized {
@>      if (msg.value.convertEthToUsd(i_priceFeed) < s_submissionFeeInUsd) {
            revert VSkillUser__NotEnoughSubmissionFee();
        }
        .
        .
        .
```

This behavior may be intentional, allowing the contract owner to use the excess funds at their discretion. However, users may expect a refund instead of their excess payment being treated as profit.

**Impact:**

- Unexpected fund retention: Users might not realize that excess ETH is not refunded.
- Potential user dissatisfaction: Some users might assume they will only pay the exact required submission fee.
- Owner discretion: While the owner can withdraw the funds and possibly return them, this is not guaranteed by the contract.

**Proof of Concept:**

Check the following test case in `test/v2/auditTests/ProofOfCodes.t.sol`:

Proof of Code

If a user sends more than the required submission fee:

```
function testUserCanSendMoreThanRequiredForSubmission() external {
        vm.prank(USER);
        vSkillUser.submitEvidence{
            value: vSkillUser.getSubmissionFeeInEth() + 1e18
        }("cid", "Blockchain"); // We know that Blockchain is a valid skill
↪  domain from the config

        uint256 evidenceLength = vSkillUser.getEvidences().length;
        assertEq(evidenceLength, 1);
    }
```

Here is the log:

```
├─ [15010] VSkillUser::getSubmissionFeeInEth() [staticcall]
│   ├─ [8991] MockV3Aggregator::latestRoundData() [staticcall]
│   │   └─ ← [Return] 1, 200000000000 [2e11], 1, 1, 1
│   └─ ← [Return] 2500000000000000 [2.5e15]
├─ [532533] VSkillUser::submitEvidence{value:
↪  1002500000000000000}("cid", "Blockchain")
│   ├─ [991] MockV3Aggregator::latestRoundData() [staticcall]
│   │   └─ ← [Return] 1, 200000000000 [2e11], 1, 1, 1
│   ├─ [124693] Distribution::distributionRandomNumberForVerifiers()
│   │   ├─ [115438]
↪  VRFCoordinatorV2_5Mock::requestRandomWords(RandomWordsRequest({
↪  keyHash:
↪  0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc,
↪  subId:
↪  71559651348530707092126447838144209774845187901096159534269265783461099618592
↪  [7.155e76], requestCon-
↪  firmations: 3, callbackGasLimit: 500000 [5e5], numWords: 3, extraArgs:
↪  0x92fd133800000000000000000000000000000000000000000000000000000000000000000
↪  }))
```

```
    │   │   │   ├─ emit RandomWordsRequested(keyHash:
    ↪  0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc,
    ↪  requestId: 1, preSeed: 100, subId:
    ↪  71559651348530707092126447838144209774845187901096159534269265783461099618592
    ↪  [7.155e76], minimumRequestCon-
    ↪  firmations: 3, callbackGasLimit: 500000 [5e5], numWords: 3, extraArgs:
    ↪  0x92fd133800000000000000000000000000000000000000000000000000000000000000,
    ↪  sender: Distribution: [0x50EEf481cae4250d252Ae577A09bF514f224C6C4])
    │   │   │   └─ ← [Return] 1
    │   │   ├─ emit VerifierDistributionRequested(requestId: 1)
    │   │   └─ ← [Return] 1
    │   ├─ emit VSkillUser__EvidenceSubmitted(submitter: ProofOfCodes:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496])
```

- The contract does not refund the extra ETH.
- Instead, the excess amount is retained for the owner.
- Users have no way to retrieve the excess funds themselves.

**Recommended Mitigation:**

If the intention is only charge the exact amount of ETH, modify the function to check for `!=` instead of `<`, also modify the error message to reflect the new condition.

```
 function submitEvidence(
        string memory cid,
        string memory skillDomain
    ) public payable onlyInitialized {
-       if (msg.value.convertEthToUsd(i_priceFeed) < s_submissionFeeInUsd) {
+        if (msg.value.convertEthToUsd(i_priceFeed) != s_submissionFeeInUsd)
 ↪  {
-           revert VSkillUser__NotEnoughSubmissionFee();
+           revert VSkillUser__IncorrectSubmissionFee();
        }
        .
        .
        .
```

If the current behavior is intended, add a clear comment and update documentation to inform users:

```
 function submitEvidence(
        string memory cid,
        string memory skillDomain
    ) public payable onlyInitialized {
```

```
+       // The submission fee is fixed and any excess amount will be
 ↪  retained by the contract owner as profit.
        if (msg.value.convertEthToUsd(i_priceFeed) < s_submissionFeeInUsd) {
            revert VSkillUser__NotEnoughSubmissionFee();
        }
```

### [I-5] Unused Variable `Verifier::i_priceFeed`

**Description:**

The `Verifier` contract declares an variable `i_priceFeed` which is never used in the contract.:

```
AggregatorV3Interface private immutable i_priceFeed;
```

**Impact:**

- Wasted deployment gas: Assigning an immutable variable consumes gas at deployment.
- Increased bytecode size: Keeping unused variables increases contract size without providing functionality.
- Code maintainability: Unused variables can cause confusion and reduce code clarity.

**Recommended Mitigation:**

Remove it to optimize gas usage and contract size:

```
- AggregatorV3Interface private immutable i_priceFeed;
```

### [I-6] Unused Event `Verifier::Verifier__LoseVerifier`

**Description:**

The `Verifier` contract defines an event `Verifier__LoseVerifier`, but it is never emitted anywhere in the contract:

```
event Verifier__LoseVerifier(address indexed verifier);
```

**Impact:**

- Wasted gas on deployment: Declaring an unused event increases contract size and deployment costs.
- Reduces code clarity: Unused events may mislead developers into thinking they are relevant when they are not.

**Recommended Mitigation:**

Remove the unused event

```
- event Verifier__LoseVerifier(address indexed verifier);
```

#### [I-7]: Unsafe Casting

**Description:**

Downcasting int/uints in Solidity can be unsafe due to the potential for data loss and unintended behavior.When downcasting a larger integer type to a smaller one (e.g., uint256 to uint128), the value may exceed the range of the target type,leading to truncation and loss of significant digits. Use OpenZeppelin's SafeCast library to safely downcast integers.

1 Found Instances

- Found in src/Relayer.sol Line: 468

```
        return address(uint160(xorResult));
```

### Gas

#### [G-1] Use big data storage(Contract bytecode) to store the NFT image uris

**Description:**

In `VSkillUserNft` contract, the NFT image URIs are stored in storage slot, which is expensive in terms of gas costs. It is recommended to store the image URIs in contract bytecode, which is cheaper and more efficient.

For more information head to simple-big-data-storage.

#### [G-2] Redundant `tx.origin` Check in `onlyRelayer` Modifier

**Description:**

The `VSkillUser::onlyRelayer` modifier includes a redundant check:

```
@>  if (msg.sender != i_relayer || tx.origin != owner()) {
        revert VSkillUser__NotRelayer();
    }
    _;
```

Since the contract's design ensures that only the relayer contract can call certain functions by owner which is set at the deployment, checking `tx.origin` against `owner()` is unnecessary. Even if the

`tx.origin` is not the owner, it's a deployment issue, not a security issue. The second check adds unnecessary gas costs without providing additional security.

**Recommended Mitigation:**

Remove the redundant `tx.origin` check to optimize gas usage:

```
modifier onlyRelayer() {
-   if (msg.sender != i_relayer || tx.origin != owner()) {
+   if (msg.sender != i_relayer) {
        revert VSkillUser__NotRelayer();
    }
    _;
}
```