METATRUST

Security Assessment for

QAMarketplace 2

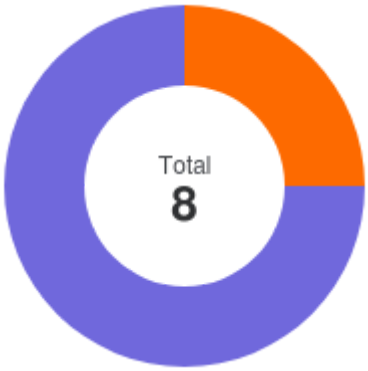
March 28, 2025

Executive Summary

Overview	
Project Name	QAMarketplace 2
Codebase URL	https://github.com/XNect/QAMarketplace
Scan Engine	Security Analyzer
Scan Time	2025/03/28 08:00:00
Commit Id	a2d5f958e224fc07c21017b07b011246eff4637a

Total	
Critical Issues	0
High risk Issues	2
Medium risk Issues	0
Low risk Issues	6
Informational Issues	0

Critical Issues	The issue can cause large economic losses, large-scale data disorder, loss of control of authority management, failure of key functions, or indirectly affect the correct operation of other smart contracts interacting with it.
High Risk Issues	The issue puts a large number of users' sensitive information at risk or is reasonably likely to lead to catastrophic impacts on clients' reputations or serious financial implications for clients and users.
Medium Risk Issues	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk Issues	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational Issue	The issue does not pose an immediate risk but is relevant to security best practices or Defence in Depth.



Critical Issues	0%	0
High risk Issues	25%	2
Medium risk Issues	0%	0
Low risk Issues	75%	6
Informational Issues	0%	0

Summary of Findings

MetaScan security assessment was performed on **March 28, 2025 08:00:00** on project **QAMarketplace 2** with the repository on branch **default branch**. The assessment was carried out by scanning the project's codebase using the scan engine **Security Analyzer**. There are in total **8** vulnerabilities / security risks discovered during the scanning session, among which **2** high risk vulnerabilities, **6** low risk vulnerabilities,

ID	Description	Severity	Alleviation
MSA-001	DOS attack when distributing reward	High risk	Fixed
MSA-002	Unable to call the `reInitialize()` function	High risk	Fixed
MSA-003	Centralization Risk	Low risk	Acknowledged
MSA-004	The `submitQuestion()` function missing check the `_questionId`, `_askerId`, and `_answererId`	Low risk	Acknowledged
MSA-005	The centralized function `withdraw()` may result in the contract malfunctional	Low risk	Acknowledged
MSA-006	The `addAdditionalReward()` function does check the relation between `_uid` and `msg.sender`	Low risk	Fixed
MSA-007	The `answererEarnings` for un-registered users would be greater than exptected	Low risk	Acknowledged
MSA-008	The check condition in the `addAdditionalReward()` function is possible wrong	Low risk	Acknowledged

Findings

High risk (2)

1. DOS attack when distributing reward

High risk | Security Analyzer

The `_handleQAReward()` function iterate the addresses of the QA reward pool to distribute reward to each address:

```
function _handleQAReward(
        for (uint256 i = 0; i < qaRewardPool.rewards.length; i++) {
            address paymentAddress = qaRewardPool.paymentAddresses[i];
            ...
            _rewardAsker(uid, paymentAddress, refundValue); //@here
            ...
        }
    }

function _rewardAsker(
    string memory _askerId,
    address _askerAddress,
    uint256 _reward
) internal {
    _safeTransfer(_askerAddress, _reward);
    emit AskerRewarded(_askerId, _askerAddress, _reward);
}

function _safeTransfer(address to, uint256 amount) private {
    require(to != address(0), "Invalid address");
    (bool success, ) = payable(to).call{value: amount}("");
    require(success, "Transfer failed");
}
```

The reward is the native token and sent to each ask address.

The ask address could be arbitrary caller, including a malicious contract, due to the `addAdditionalReward()` contract lacks checking if the `msg.sender` is an EOA or a contract:

```
function addAdditionalReward(
    uint256 _questionId,
    string calldata _uid
) external payable {
    require(registeredUIds[_uid], "UID is not registered");
    QASession memory q = questions[_questionId];
    require(q.paymentAddress != address(0), "Question does not exist");
    require(!q.resolved, "Question is already answered");
    require(!q.terminated, "Question is already canceled");
    uint256 reward = (q.reward * viewRewardPercentage) / 100;
    require(msg.value >= reward, "Reward must be greater than viewReward");
    QARewardPool storage qaRewardPool = qaRewardPools[_questionId];
qaRewardPool.paymentAddresses.push(msg.sender);
...
```

Thus, a malicious contarct can invoke the `addAdditionalReward()` function, and become a `paymentAddress`, and always refuse to receive the native token, like:

```
contract malicious contract{
  ...
  receive () external payable {
```

```
    revert("refuse to accept native token");
  }
}
```

As a result, all other `paymentAddress` will fail to receive reward, due to there is a malicious user(contract) in the payment addresses when the `_handleQAReward()` tries to distribute reward to every payment address.

File(s) Affected

contracts/QAMarketplaceV2.sol #452-464

```
452             for (uint256 i = 0; i < qaRewardPool.rewards.length; i++) {
453                 address paymentAddress = qaRewardPool.paymentAddresses[i];
454                 uint256 reward = qaRewardPool.rewards[i];
455                 string memory uid = qaRewardPool.uids[i];
456                 uint256 userFee = (reward * refundFee) / 100; // 10% fee
457                 uint256 refundValue = (reward * askerRefundPercentage) / 100; // 45% to asker
458                 uint256 rewardToAnswerer = reward - userFee - refundValue; // 45% to answerer
459                 _rewardAnswerer(_answerUid, rewardToAnswerer);
460                 _rewardAsker(uid, paymentAddress, refundValue);
461                 toAnswerer += rewardToAnswerer;
462                 toAsk += refundValue;
463                 fee += userFee;
464             }
```

Recommendation

Consider adding a `require` check to prevent the malicious contract call:

```
require(tx.origin == msg.sender, "Only EOA is allowed")
```

Alleviation Fixed

The team fixed this finding, in the commit d2806bc4f14a8e745a8b1d8f9bbd0a91a336d241.

2. Unable to call the `reInitialize()` function

High risk Security Analyzer

The `QAMarketplaceV2` contract is an implementation contract that used to update an on-chain contract.

The point is that the on-chain contract's implementation contract is supposed to be `QAMarketplace.sol` and its `initialize()` function is supposed to have already be invoked, it results in the `$._initialized` to be `1`, and then the same call to the modifier `initializer` will revert:

```
//QAMarketplace.sol
    function initialize(address _server) external initializer { //@here
        __ReentrancyGuard_init();
        __Ownable_init(msg.sender);
        server = _server;
        paused = false;
        _setDefaultParameters();
    }
```

```
//Initializable.sol
    modifier initializer() {
        // solhint-disable-next-line var-name-mixedcase
        InitializableStorage storage $ = _getInitializableStorage();

        // Cache values to avoid duplicated sloads
        bool isTopLevelCall = !$._initializing;
```

```
        uint64 initialized = $._initialized;

        // Allowed calls:
        // - initialSetup: the contract is not in the initializing state and no previous version was
        //                 initialized
        // - construction: the contract is initialized at version 1 (no reinitialization) and the
        //                 current contract is just being deployed
        bool initialSetup = initialized == 0 && isTopLevelCall;
        bool construction = initialized == 1 && address(this).code.length == 0;

        if (!initialSetup && !construction) {
            revert InvalidInitialization();
        }
        $._initialized = 1;  //@here
        if (isTopLevelCall) {
            $._initializing = true;
        }
        _;
        if (isTopLevelCall) {
            $._initializing = false;
            emit Initialized(1);
        }
    }
```

As a result, the same call to the modifier `initializer` on a same proxy address will revert, then, the `MAX_ASK_USER_COUNT` will always be 0, the last, the `addAdditionalReward()` function will always fail.

File(s) Affected

contracts/QAMarketplaceV2.sol #137-139

```
137     function reInitialize() external initializer {
138         MAX_ASK_USER_COUNT = 10;
139     }
```

Recommendation

Consider using the `reinitializer(2)` for the `function initialize(address _server) external` function.

```
    modifier reinitializer(uint64 version) {
        // solhint-disable-next-line var-name-mixedcase
        InitializableStorage storage $ = _getInitializableStorage();

        if ($._initializing || $._initialized >= version) {
            revert InvalidInitialization();
        }
        $._initialized = version;
        $._initializing = true;
        _;
        $._initializing = false;
        emit Initialized(version);
    }
```

Alleviation Fixed

The team fixed this finding, in the commit d2806bc4f14a8e745a8b1d8f9bbd0a91a336d241.

Low risk (6)

1. Centralization Risk

Low risk | Security Analyzer

In the `QAMarketplaceV2` contract, the owner has the privilege of the following functions:

- `setMinReward`: Set the minimum reward value for questions;
- `setQuestionFee`: Set the fee percentage for asking a question;
- `setRefundFee`: Set the fee percentage for refunding a question;
- `setServer`: Set the address of the authorized server;
- `setViewFee`: Set the fee percentage for viewing a question;
- `setAskerRefundPercentage`: Set the percentage of the refund given to the asker;
- `setAnswererRefundPercentage`: Set the percentage of the refund given to the answerer;
- `setViewRewardPercentage`: Set the percentage of the viewing reward;
- `setAskerViewRewardPercentage`: Set the percentage of the viewing reward given to the asker;
- `setAnswererViewRewardPercentage`: Set the percentage of the viewing reward given to the answerer;
- `pause`: Pause the contract;
- `unpause`: Unpause the contract;
- `setMaxUserCount`: Set the maximum number of additional users allowed to contribute to a question.

File(s) Affected

contracts/QAMarketplaceV2.sol #10-10

```
contract QAMarketplaceV2 is
```

Recommendation

Consider implementing a decentralized governance mechanism or a multi-signature scheme that requires consensus among multiple parties before pausing or unpausing the contract. This can help mitigate the centralization risk associated with a single owner controlling critical contract functions. Alternatively, you can provide a clear justification for the centralization aspect and ensure that users are aware of the potential risks associated with a single point of control.

Alleviation Acknowledged

The team acknowledged this finding.

2. The `submitQuestion()` function missing check the `_questionId`, `_askerId`, and `_answererId`

Low risk | Security Analyzer

The `submitQuestion()` function submits question that contains the `_questionId`, `_askerId`, and `_answererId`:

```
function submitQuestion(
    uint256 _questionId,
    string calldata _askerId,
    string calldata _answererId,
    string calldata _questionContent,
    uint256 _minReward,
    uint256 _expiryTimestamp,
    bytes calldata _signature
) external payable whenNotPaused {
...
    QASession memory q = QASession({
        id: _questionId,  //@here
        askerId: _askerId,
        answererId: _answererId,
        questionContent: _questionContent,
        reward: msg.value,
        paymentAddress: msg.sender,
        resolved: false,
```

```
            terminated: false,
            creationTimestamp: block.timestamp,
            expiryTimestamp: _expiryTimestamp
        });
        questions[_questionId] = q;
```

However, the function does not check the `_questionId`, `_askerId`, and `_answererId`. As a result, the non-exist user id may be used as the `_askerId` or the `_answererId`, which results in the fund loss.

The question id, `_questionId`, is not checked, so, the same question id may be used for the different question and results in the old question malfunctional.

File(s) Affected

Recommendation

The `_askerId` and the `_answererId` should be exist and the `_questionId` should not be used before.

Alleviation Acknowledged

The team responded that verifying the signature of the user's parameters and the server's signature can solve the problem of parameter legitimacy.

3. The centralized function `withdraw()` may result in the contract malfunctional

Low risk | Security Analyzer

The `withdraw()` function allows the owner withdraw all the native token to a specified address. The point is that what if there are still some user rewards pending to be distributed for askers and answerers, if the owner withdraws all the native tokens, then, the contract can not work as expected due to lack of funds to distribute reward.

File(s) Affected

Recommendation

When a question is processed or viewed, there is a fee charged for the owner. Accumulating the fee into a variable, like `totalFee`, and only withdraw fee under the `totalFee`.

Alleviation Acknowledged

The team acknowledged this finding.

4. The `addAdditionalReward()` function does check the relation between `_uid` and `msg.sender`

Low risk | Security Analyzer

The `addAdditionalReward()` function does check if the `_uid` matches with the `msg.sender`, which may result in the reward being distributed wrongly if the user send wrong `_uid` by mistake.

File(s) Affected

contracts/QAMarketplaceV2.sol #4-428

```
import {Initializable} from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import {OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import {MessageHashUtils} from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import {ECDSA} from "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import {ReentrancyGuardUpgradeable} from "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgr

contract QAMarketplaceV2 is
    Initializable,
    OwnableUpgradeable,
    ReentrancyGuardUpgradeable
{
    bool public paused;
    uint8 public questionFee;
    uint8 public refundFee;
    uint8 public viewFee;

    uint8 public askerRefundPercentage;
    uint8 public answererRefundPercentage;

    uint8 public viewRewardPercentage;
    uint8 public askerViewRewardPercentage;
    uint8 public answererViewRewardPercentage;

    uint256 public MIN_REWARD;

    address public server;

    struct QASession {
        uint256 id;
        string askerId;
        string answererId;
        string questionContent;
        uint256 reward;
        address paymentAddress;
        bool resolved;
        bool terminated;
        uint256 creationTimestamp;
        uint256 expiryTimestamp;
    }

    // store the mapping relationship between address and twitterId
    mapping(address => string) public addressToUid;
    mapping(string => address) public uidToAddress;

    // record if the address is registered
    mapping(address => bool) public registeredAddresses;
    mapping(string => bool) public registeredUIds;

    // if the answerer is answered, the reward will be stored in the mapping
    mapping(string => uint256) public answererEarnings;
    mapping(string => uint256) public pendingRewards;

    // store the questions
    mapping(uint256 => QASession) public questions;

    uint8 public MAX_ASK_USER_COUNT;
    struct QARewardPool {
```

```
        uint256 totalReward;
        address[] paymentAddresses;
        uint256[] rewards;
        string[] uids;
        uint256 userCount;
    }

    mapping(uint256 => QARewardPool) public qaRewardPools;

    event Registered(string indexed uid, address indexed user);
    event QuestionSubmitted(
        uint256 indexed questionId,
        string askerId,
        string answererId,
        address paymentAddress,
        uint256 reward,
        uint256 expiryTimestamp
    );
    event QuestionProcessed(
        uint256 indexed questionId,
        string askerId,
        string answererId,
        uint256 rewardToAnswerer,
        uint256 fee
    );
    event QuestionExpired(
        uint256 indexed questionId,
        string askerId,
        string answererId,
        uint256 refundToAsker,
        uint256 rewardToAnswerer,
        uint256 fee
    );
    event QuestionViewed(
        uint256 indexed questionId,
        string viewerId,
        string askerId,
        string answererId,
        uint256 rewardToAsker,
        uint256 rewardToAnswerer,
        uint256 fee
    );

    event AskerRewarded(string indexed askerId, address to, uint256 reward);

    event AnswererRewarded(
        string indexed answererId,
        address to,
        uint256 reward
    );

    event AdditionalRewardAdded(
        uint256 indexed questionId,
        string uid,
        uint256 reward,
        uint256 creationTimestamp
    );

    modifier onlyServer() {
```

```
        require(msg.sender == server, "You are not the server");
        _;
    }

    modifier whenNotPaused() {
        require(!paused, "Contract is paused");
        _;
    }

    function initialize(address _server) external initializer {
        __ReentrancyGuard_init();
        __Ownable_init(msg.sender);
        server = _server;
        paused = false;
        _setDefaultParameters();
    }

    function reInitialize() external initializer {
        MAX_ASK_USER_COUNT = 10;
    }

    function _setDefaultParameters() private {
        questionFee = 10;
        refundFee = 10;
        viewFee = 10;
        askerRefundPercentage = 45;
        answererRefundPercentage = 45;
        viewRewardPercentage = 10;
        askerViewRewardPercentage = 45;
        answererViewRewardPercentage = 45;
        MIN_REWARD = 0.01 ether;
    }

    function register(
        string calldata uid,
        address user,
        uint256 expirationTime,
        bytes calldata signature
    ) external whenNotPaused {
        // check if the address is already registered
        require(!registeredAddresses[user], "Address Already Used");
        require(!registeredUIds[uid], "UID Already Used");

        // check if the expierTime is greater than current timestamp
        require(
            expirationTime > block.timestamp,
            "ExpirationTime must be greater than current timestamp"
        );

        // get the eth signed message hash
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(
            abi.encode(uid, user, expirationTime)
        );
        // recover the signer address from the signature
        address signer = ECDSA.recover(ethSignedMessageHash, signature);

        // check if the signer is the authorized server address
        require(signer == server, "Invalid Signature");

```

```
        // // check if the user is the same as the msg.sender
        // require(user == msg.sender, "Address is not valid");

        // store the mapping relationship
        addressToUid[user] = uid;
        uidToAddress[uid] = user;
        registeredAddresses[user] = true;
        registeredUIds[uid] = true;

        // claim the reward if there is any
        uint256 reward = pendingRewards[uid];
        if (reward > 0) {
            pendingRewards[uid] = 0;
            _rewardAnswerer(uid, reward);
            emit AnswererRewarded(uid, user, reward);
        }

        emit Registered(uid, user);
    }

    function submitQuestion(
        uint256 _questionId,
        string calldata _askerId,
        string calldata _answererId,
        string calldata _questionContent,
        uint256 _minReward,
        uint256 _expiryTimestamp,
        bytes calldata _signature
    ) external payable whenNotPaused {
        require(
            msg.value >= MIN_REWARD,
            "Reward must be greater than MIN_REWARD"
        );
        require(
            _expiryTimestamp > block.timestamp,
            "ExpirationTime must be greater than current timestamp"
        );
        require(
            bytes(_questionContent).length <= 2000,
            "Question must be greater than 0 and less than 2000 characters"
        );

        // check if the signature is valid
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(
            abi.encode(
                _questionId,
                _askerId,
                _answererId,
                _questionContent,
                _minReward,
                _expiryTimestamp
            )
        );
        address signer = ECDSA.recover(ethSignedMessageHash, _signature);
        require(signer == server, "Invalid Signature");

        // check if the reward is greater than the minAnswerReward
        require(
            msg.value >= _minReward,
```

```
            "Reward must be greater than minReward"
        );

        QASession memory q = QASession({
            id: _questionId,
            askerId: _askerId,
            answererId: _answererId,
            questionContent: _questionContent,
            reward: msg.value,
            paymentAddress: msg.sender,
            resolved: false,
            terminated: false,
            creationTimestamp: block.timestamp,
            expiryTimestamp: _expiryTimestamp
        });
        questions[_questionId] = q;

        QARewardPool storage qaRewardPool = qaRewardPools[_questionId];
        qaRewardPool.totalReward = msg.value;
        qaRewardPool.paymentAddresses.push(msg.sender);
        qaRewardPool.rewards.push(msg.value);
        qaRewardPool.uids.push(_askerId);
        qaRewardPool.userCount++;

        emit QuestionSubmitted(
            _questionId,
            _askerId,
            _answererId,
            msg.sender,
            msg.value,
            _expiryTimestamp
        );
    }

    function processQuestions(
        uint256[] calldata _questionIds,
        uint256[] calldata _answerTimestamps
    ) external onlyServer whenNotPaused {
        require(
            _questionIds.length == _answerTimestamps.length,
            "QuestionIds and AnswerTimestamps must have the same length"
        );

        for (uint256 i = 0; i < _questionIds.length; i++) {
            uint256 questionId = _questionIds[i];
            uint256 answerTimestamp = _answerTimestamps[i];

            QASession storage q = questions[questionId];
            require(q.paymentAddress != address(0), "Question does not exist");
            require(!q.resolved, "Question is already answered");
            require(!q.terminated, "Question is already canceled");

            if (q.expiryTimestamp <= answerTimestamp) {
                processExpiredQuestion(questionId); // cancelled reward sharing logic
                continue;
            }

            q.resolved = true;

```

```
            uint256 answerReward = 0;
            uint256 fee = 0;
            QARewardPool storage qaRewardPool = qaRewardPools[questionId];
            if (qaRewardPool.totalReward > 0) {
                (, answerReward, fee) = _handleQAReward(
                    questionId,
                    q.answererId,
                    true
                );
            } else {
                //compatible old version
                answerReward = (q.reward * (100 - questionFee)) / 100; // 90% to answerer
                fee = q.reward - answerReward; // 10% fee
                _rewardAnswerer(q.answererId, answerReward); // reward the answerer
            }

            emit QuestionProcessed(
                questionId,
                q.askerId,
                q.answererId,
                answerReward,
                fee
            );
        }
    }

    function processExpiredQuestion(uint256 _questionId) public whenNotPaused {
        QASession storage q = questions[_questionId];
        require(!q.resolved, "Question is answered");
        require(!q.terminated, "Question is already canceled");
        require(
            q.expiryTimestamp <= block.timestamp,
            "Question is not expired"
        );

        q.terminated = true;

        uint256 toAsk = 0;
        uint256 toAnswerer = 0;
        uint256 fee = 0;
        QARewardPool storage qaRewardPool = qaRewardPools[_questionId];
        if (qaRewardPool.totalReward > 0) {
            (toAsk, toAnswerer, fee) = _handleQAReward(
                _questionId,
                q.answererId,
                false
            );
        } else {
            //compatible old version
            fee = (q.reward * refundFee) / 100; // 10% fee
            toAsk = (q.reward * askerRefundPercentage) / 100; // 45% to asker
            toAnswerer = q.reward - fee - toAsk; // 45% to answerer
            _rewardAnswerer(q.answererId, toAnswerer);
            _rewardAsker(q.askerId, q.paymentAddress, toAsk);
        }

        emit QuestionExpired(
            _questionId,
            q.askerId,
```

```
            q.answererId,
            toAsk,
            toAnswerer,
            fee
        );
    }

    function viewQuestion(
        uint256 _questionId,
        string calldata _viewerId
    ) external payable whenNotPaused nonReentrant {
        QASession memory q = questions[_questionId];
        require(q.paymentAddress != address(0), "Question does not exist");
        require(q.resolved || q.terminated, "Question is not processed");
        uint256 minReward = (q.reward * viewRewardPercentage) / 100;
        require(
            msg.value >= minReward,
            "Value must be greater than viewReward"
        );

        uint256 fee = (msg.value * viewFee) / 100; // 10% fee
        uint256 rewardToAskers = (msg.value * askerViewRewardPercentage) / 100; // 45% to asker
        uint256 rewardToAnswerer = msg.value - rewardToAskers - fee; // 45% to answerer

        QARewardPool storage qaRewardPool = qaRewardPools[_questionId];
        if (qaRewardPool.totalReward > 0) {
            uint256 denominator = qaRewardPool.totalReward;
            for (uint256 i = 0; i < qaRewardPool.rewards.length; i++) {
                uint256 reward = qaRewardPool.rewards[i];
                address paymentAddress = qaRewardPool.paymentAddresses[i];
                string memory uid = qaRewardPool.uids[i];
                uint256 rewardToAsker = (rewardToAskers * reward) / denominator;
                _rewardAsker(uid, paymentAddress, rewardToAsker);
            }
        } else {
            //compatible old version
            _rewardAsker(q.askerId, q.paymentAddress, rewardToAskers);
        }

        _rewardAnswerer(q.answererId, rewardToAnswerer);
        emit QuestionViewed(
            _questionId,
            _viewerId,
            q.askerId,
            q.answererId,
            rewardToAskers,
            rewardToAnswerer,
            fee
        );
    }

    function addAdditionalReward(
        uint256 _questionId,
        string calldata _uid
    ) external payable {
        require(registeredUIds[_uid], "UID is not registered");
        QASession memory q = questions[_questionId];
        require(q.paymentAddress != address(0), "Question does not exist");
        require(!q.resolved, "Question is already answered");
```

```
        require(!q.terminated, "Question is already canceled");
        uint256 reward = (q.reward * viewRewardPercentage) / 100;
        require(msg.value >= reward, "Reward must be greater than viewReward");
        QARewardPool storage qaRewardPool = qaRewardPools[_questionId];
        require(
            qaRewardPool.userCount <= MAX_ASK_USER_COUNT,
            "Additional reward count must be less than MAX_ASK_USER_COUNT"
        );

        qaRewardPool.totalReward += msg.value;
        qaRewardPool.paymentAddresses.push(msg.sender);
        qaRewardPool.rewards.push(msg.value);
        qaRewardPool.uids.push(_uid);
        qaRewardPool.userCount++;
```

Recommendation

Consider using the `addressToUid` to check if the `_uid` matches the `msg.sender`

Alleviation Fixed

The team fixed this finding, in the commit d2806bc4f14a8e745a8b1d8f9bbd0a91a336d241.

5. The `answererEarnings` for un-registered users would be greater than exptected

Low risk Security Analyzer

The `_rewardAnswerer()` function records users' reward into the `pendingRewards[_uid]` if an user id is un-registered:

```
function _rewardAnswerer(string memory _uid, uint256 _reward) internal {
    answererEarnings[_uid] += _reward;    //@here
    address answererAddress = uidToAddress[_uid];
    if (answererAddress != address(0)) {
        _safeTransfer(answererAddress, _reward);
    } else {
        pendingRewards[_uid] += _reward;    //@here
    }
    emit AnswererRewarded(_uid, answererAddress, _reward);
}
```

Once the user id is registered, the pending reward will be distributed to the user soon:

```
function register(
    string calldata uid,
    address user,
    uint256 expirationTime,
    bytes calldata signature
) external whenNotPaused {
    ...
    // store the mapping relationship
    addressToUid[user] = uid;
    uidToAddress[uid] = user;

    // claim the reward if there is any
    uint256 reward = pendingRewards[uid];  //@here
    if (reward > 0) {
        pendingRewards[uid] = 0;
        _rewardAnswerer(uid, reward); //@here
        emit AnswererRewarded(uid, user, reward);
    }
```

```
        ...
    }
```

However, the register function calls the `_rewardAnswerer()` function for an un-registered user will repeatedly increase the `answererEarnings`, which results in the `answererEarnings` for the un-registered users being greater than expected once the users registered later.

File(s) Affected

Recommendation

In the `_rewardAnswerer()` function, consider only increasing the `answererEarnings` for the specified address when the `answererAddress` is not zero address, i.e., the user registered.

Alleviation Acknowledged

The team acknowledged this finding.

6. The check condition in the `addAdditionalReward()` function is possible wrong

Low risk Security Analyzer

The `addAdditionalReward()` function allows users to add reward for a question. The `require` check limit the user number of a QA reward pool:

```
    function addAdditionalReward(
        uint256 _questionId,
        string calldata _uid
    ) external payable {
    ...
        require(
            qaRewardPool.userCount <= MAX_ASK_USER_COUNT,
            "Additional reward count must be less than MAX_ASK_USER_COUNT"
        );
    ...
```

The `MAX_ASK_USER_COUNT` is 10 and implies that the max user number of a QA reward pool is 10, however, the above check condition, `qaRewardPool.userCount <= MAX_ASK_USER_COUNT`, will results in the actual max user number of a QA reward pool being 11, due to the `addAdditionalReward()` function can be invoked 10 times, plus the first user, which adds to 11.

For the old version question, the `addAdditionalReward()` function can be invoked 11 times, due to the `qaRewardPool.userCount` starts from 0.

File(s) Affected

contracts/QAMarketplaceV2.sol #137-139

```
137     function reInitialize() external initializer {
138         MAX_ASK_USER_COUNT = 10;
139     }
```

contracts/QAMarketplaceV2.sol #419-422

```
419         require(
420             qaRewardPool.userCount <= MAX_ASK_USER_COUNT,
421             "Additional reward count must be less than MAX_ASK_USER_COUNT"
422         );
```

Recommendation

Consider updating the check condtion as below:

```
function addAdditionalReward(
    uint256 _questionId,
    string calldata _uid
) external payable {
...
    require(
        qaRewardPool.userCount < MAX_ASK_USER_COUNT,//@here
        "Additional reward count must be less than MAX_ASK_USER_COUNT"
    );
...
```

Alleviation Acknowledged

The team acknowledged this finding.

Disclaimer

This report is governed by the stipulations (including but not limited to service descriptions, confidentiality, disclaimers, and liability limitations) outlined in the Services Agreement, or as detailed in the scope of services and terms provided to you, the Customer or Company, within the context of the Agreement. The Company is permitted to use this report only as allowed under the terms of the Agreement. Without explicit written permission from MetaTrust, this report must not be shared, disclosed, referenced, or depended upon by any third parties, nor should copies be distributed to anyone other than the Company.

It is important to clarify that this report neither endorses nor disapproves any specific project or team. It should not be viewed as a reflection of the economic value or potential of any product or asset developed by teams or projects engaging MetaTrust for security evaluations. This report does not guarantee that the technology assessed is completely free of bugs, nor does it comment on the business practices, models, or legal compliance of the technology's creators.

This report is not intended to serve as investment advice or a tool for investment decisions related to any project. It represents a thorough assessment process aimed at enhancing code quality and mitigating risks inherent in cryptographic tokens and blockchain technology. Blockchain and cryptographic assets inherently carry ongoing risks. MetaTrust's role is to support companies and individuals in their security diligence and to reduce risks associated with the use of emerging and evolving technologies. However, MetaTrust does not guarantee the security or functionality of the technologies it evaluates.

MetaTrust's assessment services are contingent on various dependencies and are continuously evolving. Accessing or using these services, including reports and materials, is at your own risk, on an as-is and as-available basis. Cryptographic tokens are novel technologies with inherent technical risks and uncertainties. The assessment reports may contain inaccuracies, such as false positives or negatives, and unpredictable outcomes. The services may rely on multiple third-party layers.

All services, labels, assessment reports, work products, and other materials, or any results from their use, are provided "as is" and "as available," with all faults and defects, without any warranty. MetaTrust expressly disclaims all warranties, whether express, implied, statutory, or otherwise, including but not limited to warranties of merchantability, fitness for a particular purpose, title, non-infringement, and any warranties arising from course of dealing, usage, or trade practice. MetaTrust does not guarantee that the services, reports, or materials will meet specific requirements, be error-free, or be compatible with other software, systems, or services.

Neither MetaTrust nor its agents make any representations or warranties regarding the accuracy, reliability, or currency of any content provided through the services. MetaTrust is not liable for any content inaccuracies, personal injuries, property damages, or any loss resulting from the use of the services, reports, or materials.

Third-party materials are provided "as is," and any warranty concerning them is strictly between the Customer and the third-party owner or distributor. The services, reports, and materials are intended solely for the Customer and should not be relied upon by others or shared without MetaTrust's consent. No third party or representative thereof shall have any rights or claims against MetaTrust regarding these services, reports, or materials.

The provisions and warranties of MetaTrust in this agreement are exclusively for the Customer's benefit. No third party has any rights or claims against MetaTrust regarding these provisions or warranties. For clarity, the services, including any assessment reports or materials, should not be used as financial, tax, legal, regulatory, or other forms of advice.