Assignment 3 – Report for P1 and P2

Mingzhen JIANG 21108128

P1

Testing for Q1:

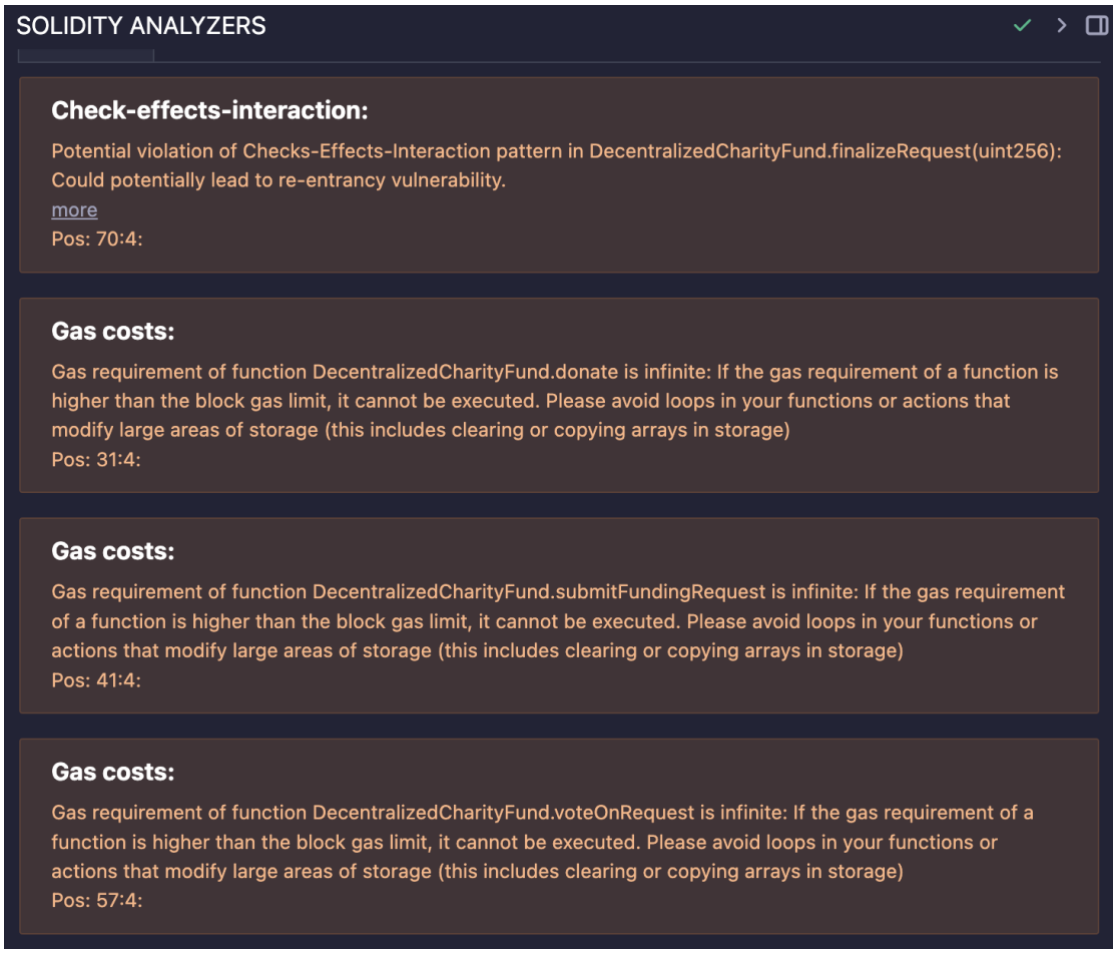
```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("DecentralizedCharityFund", function () {
    let CharityFund, charityFund, owner, donor1, donor2, project1, project2;

    beforeEach(async function () {
        [owner, donor1, donor2, project1, project2] = await ethers.getSigners();

        // Deploy the contract
        const CharityFundFactory = await ethers.getContractFactory("DecentralizedCharityFund");
        charityFund = await CharityFundFactory.deploy();
    });

    it("Should allow users to donate and update their balances", async function () {
        await charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") });
        const donorBalance = await charityFund.donorBalances(donor1.address);

        expect(donorBalance).to.equal(ethers.utils.parseEther("1"));
    });

    it("Should emit Donated event on donation", async function () {
        await expect(charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") }))
            .to.emit(charityFund, "Donated")
            .withArgs(donor1.address, ethers.utils.parseEther("1"));
    });

    it("Should allow submitting funding requests", async function () {
        await charityFund
            .connect(owner)
            .submitFundingRequest(project1.address, ethers.utils.parseEther("2"), "Support Education");

        const [projectAddress, requestedAmount, description] = await charityFund.fundingRequests(0);

        expect(projectAddress).to.equal(project1.address);
        expect(requestedAmount).to.equal(ethers.utils.parseEther("2"));
        expect(description).to.equal("Support Education");
    });

    it("Should allow users to vote on funding requests", async function () {
        await charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") });

        await charityFund
            .connect(owner)
            .submitFundingRequest(project1.address, ethers.utils.parseEther("2"), "Support Education");

        await charityFund.connect(donor1).voteOnRequest(0);
        const requestVotes = (await charityFund.fundingRequests(0)).votes;

        expect(requestVotes).to.equal(ethers.utils.parseEther("1"));
    });

    it("Should prevent double voting", async function () {
        await charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") });

        await charityFund
            .connect(owner)
            .submitFundingRequest(project1.address, ethers.utils.parseEther("2"), "Support Education");

        await charityFund.connect(donor1).voteOnRequest(0);

        await expect(charityFund.connect(donor1).voteOnRequest(0)).to.be.revertedWith("You have already voted.");
    });

```

```
64
65      it("Should allow finalizing a request if it has enough votes", async function () {
66          await charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") });
67          await charityFund.connect(donor2).donate({ value: ethers.utils.parseEther("1") });
68
69          await charityFund
70              .connect(owner)
71              .submitFundingRequest(project1.address, ethers.utils.parseEther("1.5"), "Support Education");
72
73          await charityFund.connect(donor1).voteOnRequest(0);
74          await charityFund.connect(donor2).voteOnRequest(0);
75
76          const projectInitialBalance = await ethers.provider.getBalance(project1.address);
77
78          await charityFund.connect(owner).finalizeRequest(0);
79
80          const projectFinalBalance = await ethers.provider.getBalance(project1.address);
81          const requestFinalized = (await charityFund.fundingRequests(0)).finalized;
82
83          expect(requestFinalized).to.equal(true);
84          expect(projectFinalBalance.sub(projectInitialBalance)).to.equal(ethers.utils.parseEther("1.5"));
85      });
86
87      it("Should prevent finalizing a request without enough votes", async function () {
88          await charityFund.connect(donor1).donate({ value: ethers.utils.parseEther("1") });
89
90          await charityFund
91              .connect(owner)
92              .submitFundingRequest(project1.address, ethers.utils.parseEther("2"), "Support Education");
93
94          await charityFund.connect(donor1).voteOnRequest(0);
95
96          await expect(charityFund.connect(owner).finalizeRequest(0)).to.be.revertedWith(
97              "Not enough votes to approve."
98          );
99      });
100 });
101
```

Testing result for Q1:

```
running test1.js ...
RUNS test1.js....
 DecentralizedCharityFund
   ✓ Should allow users to donate and update their balances (55 ms)
   ✓ Should emit Donated event on donation (50 ms)
   ✓ Should allow submitting funding requests (60 ms)
   ✓ Should allow users to vote on funding requests (93 ms)
null
   ✓ Should prevent double voting (78 ms)
   ✓ Should allow finalizing a request if it has enough votes (154 ms)
null
   ✘ Should prevent finalizing a request without enough votes (79 ms)
     - Expected: Transaction reverted with Not enough votes to approve..
     + Received: Error: Error from IDE : data out-of-bounds (length=0, offset=32, code=BUFFER_OVERRUN, version=abi/5.7.0)
     Message: Expected transaction to be reverted with Not enough votes to approve., but other exception was thrown: Error:
Error from IDE : data out-of-bounds (length=0, offset=32, code=BUFFER_OVERRUN, version=abi/5.7.0)
Passed: 6
Failed: 1
```

Solidity analyzers are used to verify DecentralizedCharityFund contract for vulnerabilities:



After deployment, the verified contract address:

https://sepolia.etherscan.io/address/0x870f80823772b3Ef098844A852dDfBeec1061776

Testing for Q2:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("FanEngagementSystem", function () {
  let RewardToken;
  let FanEngagementSystem;
  let rewardToken;
  let fanEngagementSystem;
  let owner;
  let fan1;
  let fan2;

  beforeEach(async function () {
    [owner, fan1, fan2] = await ethers.getSigners();

    // Deploy contracts
    RewardToken = await ethers.getContractFactory("RewardToken");
    FanEngagementSystem = await ethers.getContractFactory("FanEngagementSystem");

    fanEngagementSystem = await FanEngagementSystem.deploy();
    await fanEngagementSystem.deployed();

    // Get RewardToken address from FanEngagementSystem
    const rewardTokenAddress = await fanEngagementSystem.rewardToken();
    rewardToken = RewardToken.attach(rewardTokenAddress);
```

```
  describe("Token Earning and Loyalty Tiers", function () {
  it("Should earn tokens and update loyalty tier", async function () {
    await fanEngagementSystem.earnTokens(
      fan1.address,
      200,
      "Watch Game",
      "proof123"
    );

    expect(await rewardToken.balanceOf(fan1.address)).to.equal(200);
    expect(await fanEngagementSystem.getFanLoyaltyTier(fan1.address))
      .to.equal("Bronze");

    await fanEngagementSystem.earnTokens(
      fan1.address,
      400,
      "Season Ticket",
      "proof456"
    );

    expect(await rewardToken.balanceOf(fan1.address)).to.equal(600);
    expect(await fanEngagementSystem.getFanLoyaltyTier(fan1.address))
      .to.equal("Gold");
  });

  it("Should store activity history", async function () {
    await fanEngagementSystem.earnTokens(
      fan1.address,
      100,
      "Watch Game",
      "proof123"
    );

    const history = await fanEngagementSystem.getRewardHistory(fan1.address);
    expect(history[0]).to.equal("Watch Game");
  });
});
```

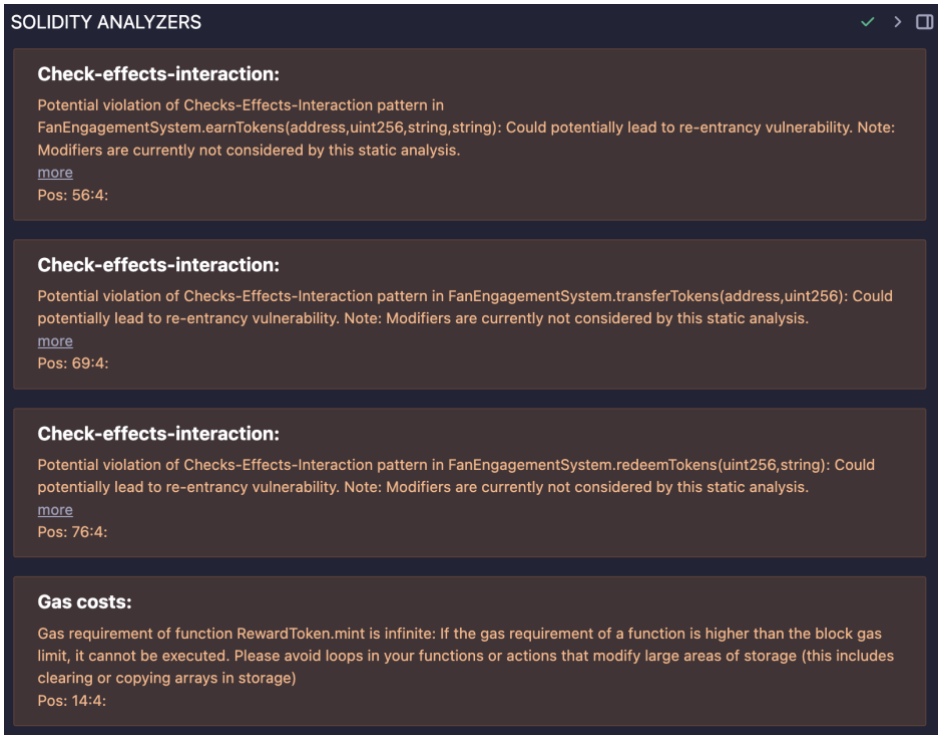
```
  describe("Token Transfers and Redemption", function () {
  beforeEach(async function () {
    await fanEngagementSystem.earnTokens(
      fan1.address,
      500,
      "Initial Balance",
      "proof"
    );
  });

  it("Should transfer tokens between fans", async function () {
    await rewardToken.connect(fan1).approve(fan1.address, 200);
    await fanEngagementSystem.connect(fan1).transferTokens(fan2.address, 200);

    expect(await rewardToken.balanceOf(fan1.address)).to.equal(300);
    expect(await rewardToken.balanceOf(fan2.address)).to.equal(200);
  });

  it("Should redeem tokens for rewards", async function () {
    await fanEngagementSystem.connect(fan1).redeemTokens(200, "VIP Access");

    expect(await rewardToken.balanceOf(fan1.address)).to.equal(300);
    expect(await fanEngagementSystem.totalTokens(fan1.address)).to.equal(300);
  });
});

```

```
92      describe("Proposal and Voting System", function () {
93      it("Should submit proposals", async function () {
94        await expect(fanEngagementSystem.submitProposal("New Reward Type"))
95          .to.emit(fanEngagementSystem, "ProposalSubmitted")
96          .withArgs(1, "New Reward Type");
97
98        expect(await fanEngagementSystem.proposalCount()).to.equal(1);
99      });
100
101     it("Should allow voting on proposals", async function () {
102       await fanEngagementSystem.submitProposal("New Reward Type");
103
104       await expect(fanEngagementSystem.connect(fan1).voteOnProposal(1))
105         .to.emit(fanEngagementSystem, "Voted")
106         .withArgs(1, fan1.address);
107     });
108   });
109 });
110
111
```

Testing result for Q2:

```
running test2.js ...
RUNS test2.js....
 Fan Engagement System
 Token Earning and Loyalty Tiers
   ✘ Should earn tokens and update loyalty tier (118 ms)
     - Expected: Gold
     + Received: Silver
     Message: expected 'Silver' to equal 'Gold'
   ✓ Should store activity history (54 ms)
 Token Transfers and Redemption
null
   ✘ Should transfer tokens between fans (52 ms)
     Message: Error from IDE : overflow [ See: https://links.ethers.org/v5-errors-NUMERIC_FAULT-overflow ] (fault="overflow", operation="toNumber",
value="1244527991663198812434494064928241768818607018 83", code=NUMERIC_FAULT, version=bignumber/5.7.0)
null
   ✘ Should redeem tokens for rewards (6 ms)
     Message: Error from IDE : overflow [ See: https://links.ethers.org/v5-errors-NUMERIC_FAULT-overflow ] (fault="overflow", operation="toNumber",
value="9809567294695738244752257651258519525595778842845", code=NUMERIC_FAULT, version=bignumber/5.7.0)
 Proposal and Voting System
   ✓ Should submit proposals (41 ms)
   ✓ Should allow voting on proposals (22 ms)
Passed: 3
Failed: 3
Time Taken: 1746 ms
```

Solidity analyzers are used to verify FanEngagementSystem contract for vulnerabilities:



After deployment, the verified contract address:

https://sepolia.etherscan.io/address/0xF4EF13a0c667B0dF23197106Df29ffBd491ddd0B

Testing code for Q3:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("RentalAgreementManagement", function () {
    let rentalManagement;
    let owner;
    let tenant;

    beforeEach(async function () {
        [owner, tenant] = await ethers.getSigners();
        const RentalAgreementManagement = await ethers.getContractFactory("RentalAgreementManagement");
        rentalManagement = await RentalAgreementManagement.deploy();
        await rentalManagement.deployed();
    });

    it("should allow the owner to create a rental agreement", async function () {
        const rentAmount = ethers.utils.parseEther("1.0");
        const duration = 30 * 24 * 60 * 60; // 30 days in seconds

        await rentalManagement.createAgreement(tenant.address, rentAmount, duration);

        const agreement = await rentalManagement.agreements(0);
        expect(agreement.tenant).to.equal(tenant.address);
        expect(agreement.rentAmount).to.equal(rentAmount);
        expect(agreement.duration).to.equal(duration);
        expect(agreement.isActive).to.be.true;
    });

    it("should emit an AgreementCreated event when an agreement is created", async function () {
        const rentAmount = ethers.utils.parseEther("1.0");
        const duration = 30 * 24 * 60 * 60; // 30 days in seconds

        await expect(rentalManagement.createAgreement(tenant.address, rentAmount, duration))
            .to.emit(rentalManagement, "AgreementCreated")
            .withArgs(0, tenant.address, rentAmount, duration);
    });
```

```
37
38      it("should allow the tenant to pay rent", async function () {
39          const rentAmount = ethers.utils.parseEther("1.0");
40          const duration = 30 * 24 * 60 * 60; // 30 days in seconds
41
42          await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
43
44          await expect(() => {
45              return rentalManagement.connect(tenant).payRent(0, { value: rentAmount });
46          }).to.changeEtherBalance(rentalManagement, rentAmount);
47
48          const agreement = await rentalManagement.agreements(0);
49          expect(agreement.totalPaid).to.equal(rentAmount);
50      });
51
52      it("should emit a RentPaid event when rent is paid", async function () {
53          const rentAmount = ethers.utils.parseEther("1.0");
54          const duration = 30 * 24 * 60 * 60; // 30 days in seconds
55
56          await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
57
58          await expect(rentalManagement.connect(tenant).payRent(0, { value: rentAmount }))
59              .to.emit(rentalManagement, "RentPaid")
60              .withArgs(0, tenant.address, rentAmount);
61      });
62
63      it("should not allow non-tenants to pay rent", async function () {
64          const rentAmount = ethers.utils.parseEther("1.0");
65          const duration = 30 * 24 * 60 * 60; // 30 days in seconds
66
67          await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
68
69          await expect(rentalManagement.connect(owner).payRent(0, { value: rentAmount }))
70              .to.be.revertedWith("Only the tenant can pay rent.");
71      });
```

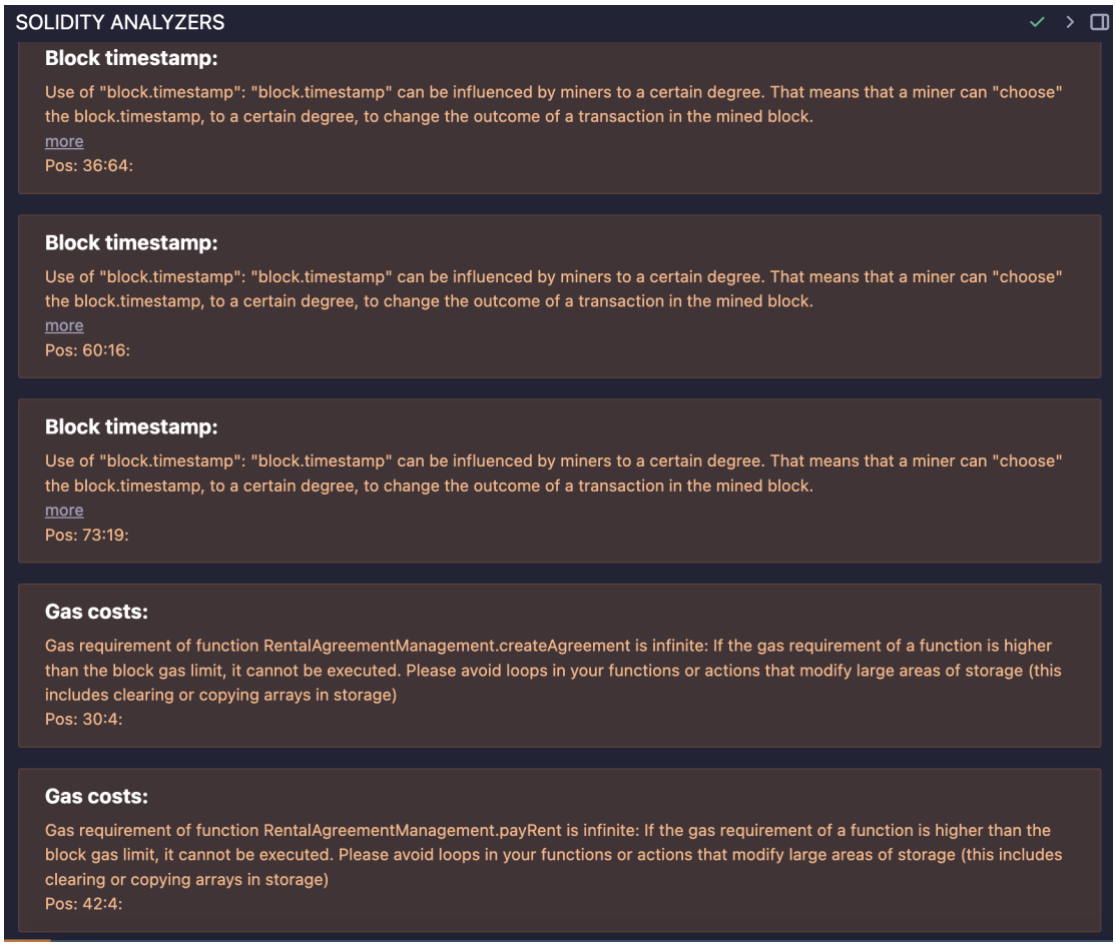
```
73      it("should allow the owner to terminate an agreement after its duration", async function () {
74          const rentAmount = ethers.utils.parseEther("1.0");
75          const duration = 1; // 1 second for testing
76
77          await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
78
79          // Wait for the agreement to expire
80          await new Promise(resolve => setTimeout(resolve, 2000));
81
82          await rentalManagement.terminateAgreement(0);
83
84          const agreement = await rentalManagement.agreements(0);
85          expect(agreement.isActive).to.be.false;
86      });
87
88      it("should emit an AgreementTerminated event when an agreement is terminated", async function () {
89          const rentAmount = ethers.utils.parseEther("1.0");
90          const duration = 1; // 1 second for testing
91
92          await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
93
94          // Wait for the agreement to expire
95          await new Promise(resolve => setTimeout(resolve, 2000));
96
97          await expect(rentalManagement.terminateAgreement(0))
98              .to.emit(rentalManagement, "AgreementTerminated")
99              .withArgs(0, owner.address);
100     });
```

```
102     it("should not allow te import ethers eement that is already terminated", async function () {
103         const rentAmount = ethers.utils.parseEther("1.0");
104         const duration = 1; // 1 second for testing
105
106         await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
107
108         // Wait for the agreement to expire
109         await new Promise(resolve => setTimeout(resolve, 2000));
110
111         await rentalManagement.terminateAgreement(0);
112
113         await expect(rentalManagement.terminateAgreement(0))
114             .to.be.revertedWith("Agreement is already terminated.");
115     });
116
117     it("should return the correct status of an agreement", async function () {
118         const rentAmount = ethers.utils.parseEther("1.0");
119         const duration = 1; // 1 second for testing
120
121         await rentalManagement.createAgreement(tenant.address, rentAmount, duration);
122
123         let status = await rentalManagement.getAgreementStatus(0);
124         expect(status).to.equal("Active");
125
126         // Wait for the agreement to expire
127         await new Promise(resolve => setTimeout(resolve, 2000));
128
129         status = await rentalManagement.getAgreementStatus(0);
130         expect(status).to.equal("Expired");
131
132         await rentalManagement.terminateAgreement(0);
133         status = await rentalManagement.getAgreementStatus(0);
134         expect(status).to.equal("Terminated");
135     });
136 });
```

Testing result for Q3:

```
running test3.js ...
RUNS test3.js....
 RentalAgreementManagement
   ✓ should allow the owner to create a rental agreement (34 ms)
   ✓ should emit an AgreementCreated event when an agreement is created (26 ms)
   ✘ should allow the tenant to pay rent (31 ms)
     Message: e.getAddress is not a function
   ✓ should emit a RentPaid event when rent is paid (52 ms)
null
   ✓ should not allow non-tenants to pay rent (44 ms)
   ✓ should allow the owner to terminate an agreement after its duration (2294 ms)
   ✓ should emit an AgreementTerminated event when an agreement is terminated (2794 ms)
null
   ✓ should not allow terminating an agreement that is already terminated (2855 ms)
   ✓ should return the correct status of an agreement (2852 ms)
Passed: 8
Failed: 1
Time Taken: 12430 ms
```

Solidity analyzers are used to verify DecentralizedCharityFund contract for vulnerabilities:



After deployment, the verified contract address:

https://sepolia.etherscan.io/address/0xbF7324BCFBc647960bf102Ba378B33CA0c8a3233

Testing codes for Q4:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("DecentralizedAuctionHouse", function () {
    let auctionHouse;
    let owner;
    let artist;
    let bidder1;
    let bidder2;

    beforeEach(async function () {
        [owner, artist, bidder1, bidder2] = await ethers.getSigners();
        const AuctionHouse = await ethers.getContractFactory("DecentralizedAuctionHouse");
        auctionHouse = await AuctionHouse.deploy();
        await auctionHouse.deployed();
    });

    it("should allow the artist to create an auction", async function () {
        const itemName = "Artwork 1";
        const reservePrice = ethers.utils.parseEther("1.0");
        const auctionDuration = 60 * 60; // 1 hour

        await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);

        const auction = await auctionHouse.auctions(0);
        expect(auction.artist).to.equal(artist.address);
        expect(auction.itemName).to.equal(itemName);
        expect(auction.reservePrice).to.equal(reservePrice);
        expect(auction.highestBid).to.equal(0);
        expect(auction.highestBidder).to.equal(ethers.constants.AddressZero);
        expect(auction.endTime).to.be.greaterThan(Math.floor(Date.now() / 1000));
        expect(auction.finalized).to.be.false;
    });

    it("should emit AuctionCreated event when an auction is created", async function () {
        const itemName = "Artwork 1";
        const reservePrice = ethers.utils.parseEther("1.0");
        const auctionDuration = 60 * 60; // 1 hour

        await expect(auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration))
            .to.emit(auctionHouse, "AuctionCreated")
            .withArgs(0, artist.address, itemName, reservePrice, auctionDuration + Math.floor(Date.now() / 1000));
    });

    it("should allow bidders to place bids", async function () {
        const itemName = "Artwork 1";
        const reservePrice = ethers.utils.parseEther("1.0");
        const auctionDuration = 60 * 60; // 1 hour

        await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);

        await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") });

        const auction = await auctionHouse.auctions(0);
        expect(auction.highestBid).to.equal(ethers.utils.parseEther("1.5"));
        expect(auction.highestBidder).to.equal(bidder1.address);
    });

    it("should emit BidPlaced event when a bid is placed", async function () {
        const itemName = "Artwork 1";
        const reservePrice = ethers.utils.parseEther("1.0");
        const auctionDuration = 60 * 60; // 1 hour

        await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);

        await expect(auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") }))
            .to.emit(auctionHouse, "BidPlaced")
            .withArgs(0, bidder1.address, ethers.utils.parseEther("1.5"));
    });
```

```
71      it("should refund the previous highest bidder when a new bid is placed", async function () {
72          const itemName = "Artwork 1";
73          const reservePrice = ethers.utils.parseEther("1.0");
74          const auctionDuration = 60 * 60; // 1 hour
75
76          await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
77
78          await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") });
79
80          // Bidder 2 places a higher bid
81          await expect(() => auctionHouse.connect(bidder2).placeBid(0, { value: ethers.utils.parseEther("2.0") }))
82              .to.changeEtherBalances([bidder1, auctionHouse], [ethers.utils.parseEther("1.5"), ethers.utils.parseEther("-1.5")]);
83
84          const auction = await auctionHouse.auctions(0);
85          expect(auction.highestBid).to.equal(ethers.utils.parseEther("2.0"));
86          expect(auction.highestBidder).to.equal(bidder2.address);
87      });
88
89      it("should not allow bids lower than the highest bid", async function () {
90          const itemName = "Artwork 1";
91          const reservePrice = ethers.utils.parseEther("1.0");
92          const auctionDuration = 60 * 60; // 1 hour
93
94          await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
95          await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") });
96
97          await expect(auctionHouse.connect(bidder2).placeBid(0, { value: ethers.utils.parseEther("1.0") }))
98              .to.be.revertedWith("Bid amount must be higher than current highest bid.");
99      });
100
101     it("should not allow bids below the reserve price", async function () {
102         const itemName = "Artwork 1";
103         const reservePrice = ethers.utils.parseEther("1.0");
104         const auctionDuration = 60 * 60; // 1 hour
105
106         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
107
108         await expect(auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("0.5") }))
109             .to.be.revertedWith("Bid must meet reserve price.");
110     });

112     it("should allow bidders to withdraw their bids", async function () {
113         const itemName = "Artwork 1";
114         const reservePrice = ethers.utils.parseEther("1.0");
115         const auctionDuration = 60 * 60; // 1 hour
116
117         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
118
119         await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") });
120
121         await expect(() => a any House.connect(bidder1).withdrawBid(0))
122             .to.changeEtherBalance(bidder1, ethers.utils.parseEther("1.5"));
123
124         const auction = await auctionHouse.auctions(0);
125         expect(auction.highestBid).to.equal(0);
126         expect(auction.highestBidder).to.equal(ethers.constants.AddressZero);
127     });
128
129     it("should not allow the highest bidder to withdraw their bid", async function () {
130         const itemName = "Artwork 1";
131         const reservePrice = ethers.utils.parseEther("1.0");
132         const auctionDuration = 60 * 60; // 1 hour
133
134         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
135
136         await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("1.5") });
137
138         await expect(auctionHouse.connect(bidder1).withdrawBid(0))
139             .to.be.revertedWith("Highest bidder cannot withdraw.");
140     });
```

```
142     it("should finalize the auction and transfer funds to the artist", async function () {
143         const itemName = "Artwork 1";
144         const reservePrice = ethers.utils.parseEther("1.0");
145         const auctionDuration = 1; // 1 second for testing
146 
147         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
148 
149         await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("2.0") });
150 
151         // Wait for auction to end
152         await new Promise(resolve => setTimeout(resolve, 2000));
153 
154         await expect(() => auctionHouse.connect(artist).finalizeAuction(0))
155             .to.changeEtherBalance(artist, ethers.utils.parseEther("2.0"));
156 
157         const auction = await auctionHouse.auctions(0);
158         expect(auction.finalized).to.be.true;
159     });
160 
161     it("should not allow finalizing an auction that has not ended", async function () {
162         const itemName = "Artwork 1";
163         const reservePrice = ethers.utils.parseEther("1.0");
164         const auctionDuration = 60 * 60; // 1 hour
165 
166         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
167 
168         await expect(auctionHouse.connect(artist).finalizeAuction(0))
169             .to.be.revertedWith("Auction is still active.");
170     });
171 
172     it("should not allow finalizing an auction that has already been finalized", async function () {
173         const itemName = "Artwork 1";
174         const reservePrice = ethers.utils.parseEther("1.0");
175         const auctionDuration = 1; // 1 second for testing
176 
177         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
178 
179         await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("2.0") });
180 
181         // Wait for auction to end
182         await new Promise(resolve => setTimeout(resolve, 2000));
183 
184         await auctionHouse.connect(artist).finalizeAuction(0);
185 
186         await expect(auctionHouse.connect(artist).finalizeAuction(0))
187             .to.be.revertedWith("Auction has already been finalized.");
188     });
189 
190     it("should                                                    ction () {
191         const
192         const                                                    her("1.0");
193         const auctionDuration = 60 * 60; // 1 hour
194 
195         await auctionHouse.connect(artist).createAuction(itemName, reservePrice, auctionDuration);
196 
197         await auctionHouse.connect(bidder1).placeBid(0, { value: ethers.utils.parseEther("2.0") });
198 
199         const [name, highestBid, endTime, finalized] = await auctionHouse.getAuctionDetails(0);
200         expect(name).to.equal(itemName);
201         expect(highestBid).to.equal(ethers.utils.parseEther("2.0"));
202         expect(endTime).to.be.greaterThan(Math.floor(Date.now() / 1000));
203         expect(finalized).to.be.false;
204     });
205 });
```

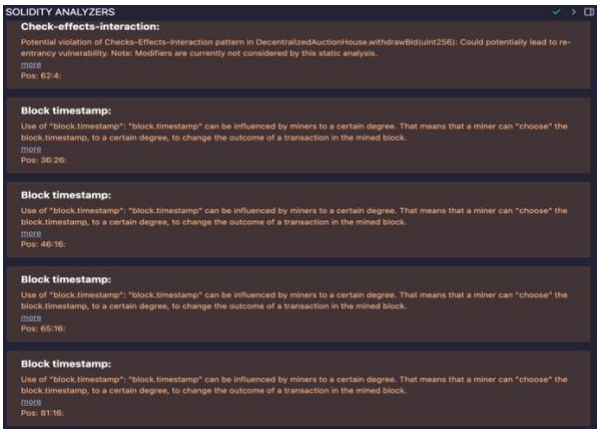
```
const auctionDuration: number
const auctionDuration = 60 * 60;
```

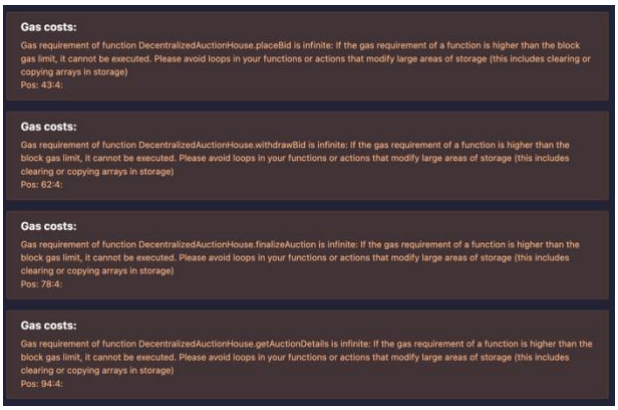
Testing result for Q4:

```
running test4.js ...
RUNS test4.js....
 DecentralizedAuctionHouse
   ✘ should allow the artist to create an auction (39 ms)
     Message: expected 1733399294 to be a number or a date
   ✓ should emit AuctionCreated event when an auction is created (46 ms)
   ✓ should allow bidders to place bids (66 ms)
   ✓ should emit BidPlaced event when a bid is placed (38 ms)
   ✘ should refund the previous highest bidder when a new bid is placed (28 ms)
     Message: e.getAddress is not a function
null
   ✓ should not allow bids lower than the highest bid (65 ms)
null
   ✓ should not allow bids below the reserve price (39 ms)
null
   ✘ should allow bidders to withdraw their bids (55 ms)
     Message: cannot estimate gas; transaction may fail or may require manual gas limit [ See:
https://links.ethers.org/v5-errors-UNPREDICTABLE_GAS_LIMIT ] (error={}, method="estimateGas", transaction=
{"from":"0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db","to":"0x6EcFfd954aCd876C35A81DF3aA6F6a1e303f0CEe","data":"0x0ea
af4c8000000000000000000000000000000000000000000000000000000000000000","accessList":null},
code=UNPREDICTABLE_GAS_LIMIT, version=providers/5.7.2)
```

```
   ✓ should not allow the highest bidder to withdraw their bid (61 ms)
   ✘ should finalize the auction and transfer funds to the artist (2649 ms)
     - Expected: 2000000000000000000
     + Received: 1286940000000000
     Message: Expected "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2" to change balance by 2000000000000000000 wei,
but it has changed by 1286940000000000 wei
null
   ✓ should not allow finalizing an auction that has not ended (47 ms)
null
   ✓ should not allow finalizing an auction that has already been finalized (2530 ms)
   ✘ should return auction details (53 ms)
     Message: expected 1733399302 to be a number or a date
Passed: 8
Failed: 5
Time Taken: 8561 ms
```

Solidity analyzers are used to verify DecentralizedAuctionHouse contract for vulnerabilities:





After deployment, the verified contract address:

https://sepolia.etherscan.io/address/0xc6da6c6A8C215ADBc521eeEd0C945D93F112Fd1e

P2

MZKKToken (ERC20)

The MZKKToken will have the following features:

- **Minting**: Only the owner can mint new tokens.
- **Burning**: Token holders can burn their tokens.
- **Capped Supply**: The maximum supply is set to 1 million tokens.

MZKKNFT (ERC721)

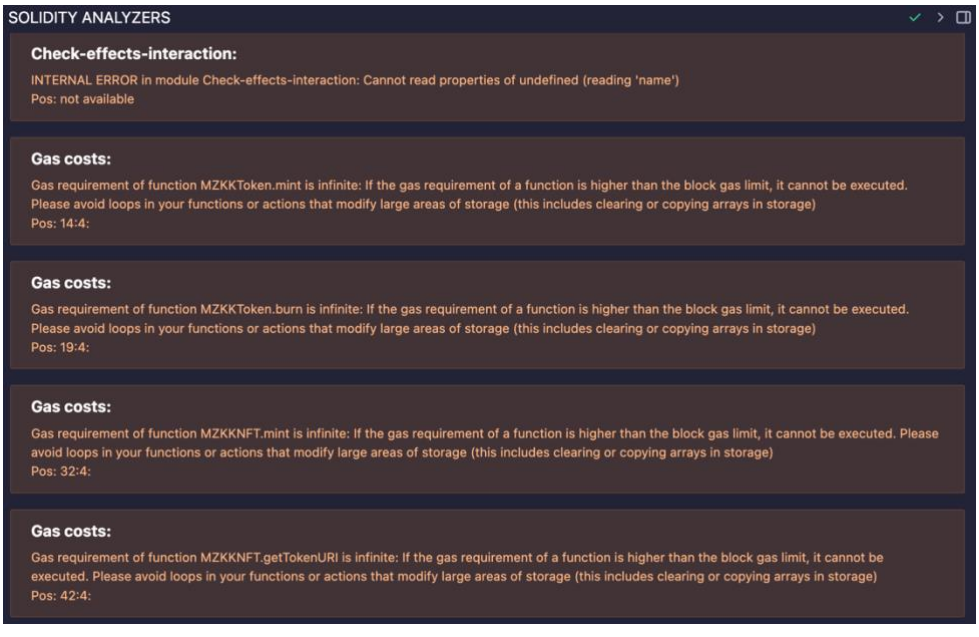
The MZKKNFT will have the following features:

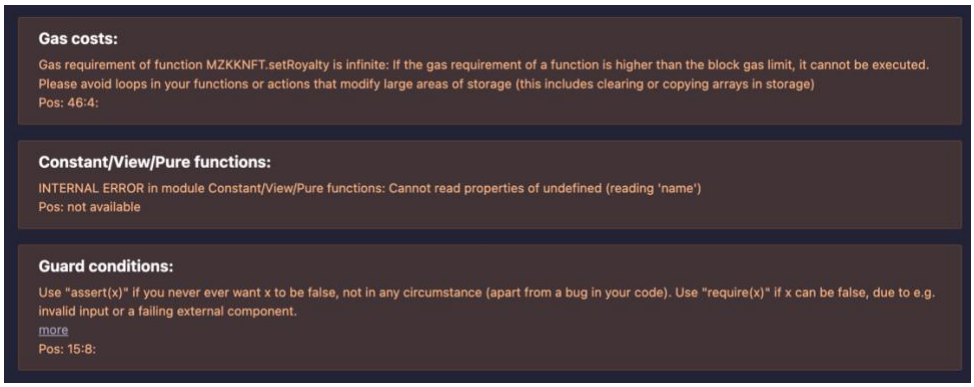
- **Minting with URI**: Each NFT will have a unique URI for metadata.
- **Transfer Restrictions**: Only the owner can mint new NFTs.
- **Royalty Payments**: The contract allows setting royalty fees for the original creator.

Before deploying, the vulnerabilities are detected as:

```
running testMzkk.js ...
RUNS testMzkk.js....
 MZKKToken
   ✓ should mint tokens (29 ms)
   ✘ should not exceed max supply (30 ms)
     - Expected: Transaction reverted.
     + Received: Transaction NOT reverted.
     Message: Expected transaction to be reverted
   ✓ should allow token holders to burn tokens (56 ms)
 MZKKNFT
   ✓ should mint an NFT with a URI (25 ms)
   ✓ should set royalty for a creator (22 ms)
Passed: 4
Failed: 1
Time Taken: 1320 ms
```

Vulnerabilities:





After deployment, the verified contract address is:

https://sepolia.etherscan.io/address/0x84030698cb02D41796503b43a36f61F25422FFF5