1. Describe your experience writing Solidity code. What was easy, what was hard, what surprised you?

   Writing solidity code is much similar to writing JavaScript code. The grammar is straightforward, but it doesn't provide a lot of library codes to facilitate my coding. For example, when I want to traverse the keys in the mapping, I have to maintain another array to keep the keys, which is a lot annoying. It doesn't have float data type, which might be a trouble when the float arithmetic is needed. Just like JavaScript, it provides some useful globals to help fulfill magic operations. Most importantly, I have to pay special attention to the number of operations used in order to avoid wasting gas.

2. Describe the gas costs for the operation of dao.sol

   The gas used (unit) for delegateCurator: 28359 deposit: 73020 withdraw: 4600000 createProposal: 62325 vote: 4600000

3. What types of attack would the contract dao.sol need to consider and defend against?

   Reentrancy, Integer Overflow and Underflow, DoS with revert, DoS with Block Gas Limit.

4. Does your implementation in dao.sol defend against these attacks?

   The reentrancy attack is actually the race-to-empty problem we are dealing with, it could be avoided by finishing all internal work first, and only then calling the external function. I use uint256 to try to minimize the effect of Integer Overflow and Underflow, but this is unavoidable unless the increase and decrease of the integer is controlled by some administers. The Dos with revert is not a problem since we are not reverting any payment in the fallback function. I did't try to defend this attack except that I made sure that there is no unbounded loop in the contract.

5. Describe the attack which exploits race.sol and how someone can steal from totalBalance.

   The race-to-empty problem takes use of the logic in the code that the user's balance is not updated until the very end of the function such that the defending code cannot work properly, thus anyone successfully used this bug can withdraw the balance over and over again. They can simply call the withdraw function in the fallback function because the withdraw function will call the fallback function again, which will cause infinite layers of call stack.

6. These attacks are similar to what happened to the actual DAO, but not exactly the same. What happened in the actual DAO?

In the actual DAO attack, they used the split function to repeatedly call the withdraw function instead of using the fallback function. The basic idea is this: propose a split. Execute the split. When the DAO goes to withdraw your reward, call the function to execute a split before that withdrawal finishes. The function will start running without updating your balance such that the attacker could transfer more tokens than they should be able to into their child DAO.

7. Read about and describe the stalking attack which the DAO enabled, and the DAO rebuttal

   The "stalking attack" is as follows. A malicious DAO Token Holder will vote "yes" in any split proposals where the amount of tokens at stake is less than the amount of tokens the malicious DAO Token Holder is comfortable losing. If the victim does not have a majority of the votes and calls `splitDAO()' to create their new DAO anyway, a malicious attacker can also call it and they will both end up in the same child DAO with the victim being the Curator but not having the majority of NewDAO tokens. From this point on both the victim and the attacker are locked in a stalemate. The attacker will need the victim to add them to the whitelist so they can get ETH out of the DAO, and the victim will need the attacker to fail to vote on their proposal so they can get their ETH out of The DAO (the assumed purpose of splitting in the first place).
   The prevention of the attack is quite straightforward. Before calling splitDAO() make sure that none else apart from you has voted in this proposal. If someone else did, then make another new Curator proposal and see if they follow you there. If they do then don’t call splitDAO(). Otherwise, do so in the last block where it is possible.