(a) Describe the (intended) basic functionality of this contract.

Solution:
The creator of the contract specifies the amount of candidates which can be voted for. The amount of votes for each candidate are stored in the `candidates` array. Now accounts can buy votes with the designated `buyVotes()` function. One vote costs one Ether and if an account sends an odd amount of Ether, they get a refund of the money they payed too much. The votes are stored in the `remainingVotes` array and there is no limit to how many votes one account can buy. An account with votes can then call the `vote()` function to give a certain amount of their remaining votes to one candidate. This function can be called multiple times to vote for multiple candidates. If an account bought more votes than they actually wanted to give to candidates, they can convert their votes back into Ether by calling the `payoutVotes()` function. Finally, the contract creator can end the auction at any time with the `endVoting()` function. Afterwards, none of the other functions can be called anymore and the creator gets all funds that were sent to the contract during the auction.

(b) What is the problem with the contract? You can assume that the contract creator is benign and does not e.g. end the voting prematurely. If you are having trouble to see the bug, copy the code into the Remix IDE and try calling some functions.

Solution:
Anyone can vote without having to buy votes because the check in line 27 is not working as intended. The check is supposed to verify that an account has enough votes. However, as `remainingVotes[msg.sender]` and `amountOfVotes` are both unsigned integer variables, the difference between them is also unsigned. Therefore, it can never be less than 0 and the check is always true. What is more, because the amount of votes are subtracted from the remaining votes in the next line, this variable can underflow and its value can be enormously large as a result. An attacker could use this circumstance to get a refund of the votes of all other accounts using the `payoutVotes()` function.

(c) Which Solidity idiom is commonly used to prevent this type of error?

Solution:
Over- and underflow bugs can be prevented by using the SafeMath library.

(d) Rewrite the respective code to make the contract work as intended. How exactly does the idiom prevent the error?

Solution:
After including the SafeMath library in the code and adding the line `using SafeMath for uint256;` at the beginning of the contract, line 27 can be rewritten as follows: `require(remainingVotes[msg.sender].sub(amountOfVotes) >= 0);` Of course, all other usages of mathematical operators in the contract should be rewritten as well.
Applying the SafeMath library fixes the bug because the `sub()` function throws if `amountOfVotes` is larger than `remainingVotes[msg.sender]`.