# Secureum Bootcamp Audit Findings 101 Quiz

*November 28, 2021 / patrickd*

*This is a writeup of the Secureum Bootcamp Audit Findings 101 Quiz with solutions. For fairness it was published after submissions to it were closed.*

*The quiz consisted of 8 questions with an overall strict timelimit of 16 minutes.* ***All questions are concerning the same snippet of code. No syntax highlighting or indentation was used in the original quiz, so it was skipped here as well. Make sure to read code comments carefully.*** *The ordering of the questions was randomized, so the numbering here won't match with the numbering elsewhere.*

[**Note**: *All 8 questions in this quiz are based on the InSecureumDAO contract snippet. This is the same contract snippet you will see for all the 8 questions in this quiz.* The `InSecureumDAO` contract snippet illustrates some basic functionality of a Decentralized Autonomous Organization (DAO) which includes the opening of the DAO for memberships, allowing users to join as members by depositing a membership fee, creating proposals for voting, casting votes, etc. Assume that all the other functionality (that is not shown or represented by ...) is implemented correctly.]

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contract
```

```
import "https://github.com/OpenZeppelin/openzeppelin-contract

contract InSecureumDAO is Pausable, ReentrancyGuard {

    // Assume that all functionality represented by ... belo

    address public admin;
    mapping (address => bool) public members;
    mapping (uint256 => uint8[]) public votes;
    mapping (uint256 => uint8) public winningOutcome;
    uint256 memberCount = 0;
    uint256 membershipFee = 1000;

    modifier onlyWhenOpen() {
        require(address(this).balance > 0, 'InSecureumDAO: Th
        _;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin);
        _;
    }

    modifier voteExists(uint256 _voteId) {
        // Assume this correctly checks if _voteId is present
        ...
        _;
    }

    constructor (address _admin) {
        require(_admin == address(0));
        admin = _admin;
    }

    function openDAO() external payable onlyAdmin {
        // Admin is expected to open DAO by making a notional
        ...
    }

    function join() external payable onlyWhenOpen nonReentran
```

```
        require(msg.value == membershipFee, 'InSecureumDAO: I
        members[msg.sender] = true;
        ...
    }

    function createVote(uint256 _voteId, uint8[] memory _poss
        votes[_voteId] = _possibleOutcomes;
        ...
    }

    function castVote(uint256 _voteId, uint8 _vote) external
        ...
    }

    function getWinningOutcome(uint256 _voteId) public view r
        ...
    }

    function setMembershipFee(uint256 _fee) external onlyAdmi
        membershipFee = _fee;
    }

    function removeAllMembers() external onlyAdmin {
        delete members[msg.sender];
    }

}
```

> “Based on the comments and code shown in the `InSecureumDAO` snippet”
>
> — 1 of 8

- [x] A. DAO is meant to be opened only by the `admin` by making an Ether deposit to the contract
- [x] B. DAO can be opened by anyone by making an Ether deposit to the contract

- [x] C. DAO requires an exact payment of `membershipFee` to join the DAO
- [ ] D. None of the above

▼ Solution

**Correct is A, B, C**. While the payable `openDAO()` function is protected by the correctly implemented `onlyAdmin` modifier, it is always possible to force send Ether into a contract via `selfdestruct()`. The `onlyWhenOpen()` modifier only checks for the contracts own balance which can be bypassed by doing that. The payable `join()` function indeed checks for the `msg.value` to exactly match `membershipFee`.

> “Based on the comments and code shown in the `InSecureumDAO` snippet”
>
> — 2 of 8

- [ ] A. Guarded launch via circuit breakers has been implemented correctly for all state modifying functions
- [ ] B. Zero-address check(s) has/have been implemented correctly
- [ ] C. All critical privileged-role functions have events emitted
- [x] D. None of the above

▼ Solution

**Correct is D**. All state modifying functions, that can be accessed by users other than admin, are indeed correctly "protected" by the `onlyWhenOpen` modifier, but that modifier is, as explained in the previous answer, not correctly implemented itself. The only zero-address check is made during construction, but it's currently ensuring that the admin will always be the zero-address - it's therefore doing the opposite of a correctly implemented zero-address-check. There are several functions that sound more than critical enough to have events (eg. `removeAllMembers`), but this contract isn't using events at all.

> “Reentrancy protection only on `join()` (assume it’s correctly specified) indicates that”
>
> — 3 of 8

- [ ] A. Only `payable` functions require this protection because of handling `msg.value`
- [x] B. `join()` likely makes untrusted external call(s) but not the other contract functions
- [ ] C. Both A and B
- [ ] D. Neither A nor B

▼ Solution

**Correct is B**. A simply sounds like nonesense. Since it says that we should assume that reentrancy protection has been used correctly, and a reentrancy vulnerability requires making untrusted external calls, we can assume that it is at least likely, although not certain, that other functions do not.

> “Access control on `msg.sender` for DAO membership is required in”
>
> — 4 of 8

- [x] A. `createVote()` to prevent non-members from creating votes
- [x] B. `castVote()` to prevent non-members from casting votes
- [ ] C. `getWinningOutcome()` to prevent non-members from viewing winning outcomes
- [ ] D. None of the above

▼ Solution

**Correct is A, B**. It wouldn't make much sense to pay a membership fee if you are allowed to create and cast votes without it. There's no clear reason to prevent non-members from accessing winning outcomes though, since they'd be publicly readable on the blockchain anyway.

> “A commit/reveal scheme (a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden from others, with the ability to reveal the committed value later) is relevant for”
>
> — 5 of 8

- [ ] A. `join()` to not disclose `msg.sender` while joining the DAO
- [ ] B. `createVote()` to not disclose the possible outcomes during creation
- [x] C. `castVote()` to not disclose the vote being cast
- [ ] D. All the above

▼ Solution

**Correct is C**. It's not possible to hide the `msg.sender` using a simple commit/reveal scheme and it wouldn't make much sense to try, since there's no clear advantage from temporarily hiding your membership. It also wouldn't make much sense to not disclose possible outcomes of a new vote, unless you want to make your members vote blindly on the options. It does make sense to hide what you are voting for until voting closes, since this makes it impossible to calculate how many members voted for a specific option and how many fake/sibyl members you'd exactly need to create in order to manipulate the vote.

> “Security concern(s) from missing input validation(s) is/are present in”
>
> — 6 of 8

- [x] A. `createVote()` for duplicate `_voteId`
- [ ] B. `castVote()` for existing `_voteId`
- [ ] C. `getWinningOutcome()` for existing `_voteId`
- [x] D. `setMembershipFee()` for sanity/threshold checks on `_fee`

▼ Solution

**Correct is A, D**. The `createVote()` function currently allows overwriting existing votes by specifying a previously used `_voteId`. It would probably be better to use an array instead of a mapping here and simply push new votes into it. `castVote()` has a modifier in place ensuring that a vote can only be cast on existing votes. Since function body should be assumed as correctly implemented, we should also assume there are no security concerns in regards to validation either. Without sanity/threshold checks when setting fees in `setMembershipFee()`, admins could practically close the DAO off, preventing new members from joining, which could certainly be considered a security concern for the protocol.

> “`removeAllMembers()` function”

— 7 of 8

- [x] A. Will not work as expected to remove all the members from the DAO
- [ ] B. Will work as expected to remove all the members from the DAO
- [x] C. Is a critical function missing an event emission
- [ ] D. None of the above

▼ Solution

**Correct is A, C**. What the function actually does is only removing the admin as member from the DAO, he'd still stay the admin though. Properly implementing this function would actually be rather difficult, since a simple delete on the mapping variable without specifying a key would not actually delete any of its values. Assuming it would work as advertised by its name you can certainly say it would be a critical function that should emit an event, which it currently does not.

> “`InSecureumDAO` will NOT be susceptible to something like the 2016 “DAO exploit””

— 8 of 8

- [ ] A. Because it derives from `ReentrancyGuard.sol` which protects all contract functions by default
- [x] B. Only if it does not have a withdraw Ether function vulnerable to reentrancy and makes no external calls
- [ ] C. Because Ethereum protocol was fixed after the DAO exploit to prevent such exploits
- [ ] D. Because Solidity language was fixed after the DAO exploit to prevent such exploits

▼ Solution

**Correct is B**. The 2016 "DAO exploit" was indeed a reentrancy issue caused by an external call within an Ether withdrawal function. But simply inheriting from `ReentrancyGuard.sol` will not prevent them since you actually have to apply the `nonReentrant` modifier to relevant functions. There have indeed been some efforts to prevent reentrancy issues with changes made in Ethereum and Solidity, but none of them can be considered a "fix". A recurrence of the 2016 "DAO exploit"

is indeed still possible, although more unlikely since, thanks to all the attention it got, this anti-pattern is now widely known and rarely found anymore during audits.

In Blockchain Tags Ethereum, Secureum Bootcamp

← Secureum Bootcamp Audit Findings… Secureum Bootcamp Audit Techniques…