Secureum Bootcamp Security Pitfalls & Best Practices 201 Quiz

November 14, 2021 / patrickd

This is a writeup of the Secureum Bootcamp Security Pitfalls & Best Practices 201 Quiz containing solutions and references to the provided study material.
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 *InSecureumToken* contract shown below.
This is the same contract you see for all the 8 questions in this quiz.
The *InSecureumToken* contract implements a token contract which allows users to buy tokens by depositing Ether (at a certain conversion ratio), and transfer tokens.]

```
pragma solidity 0.7.0;
```

```
contract InSecureumToken {

mapping(address => uint) private balances;

uint public decimals = 10**18; // decimals of the token
uint public totalSupply; // total supply
uint MAX_SUPPLY = 100 ether; // Maximum total supply

event Mint(address indexed destination, uint amount);

function transfer(address to, uint amount) public {
    // save the balance in local variables
    // so that we can re-use them multiple times
    // without paying for SLOAD on every access
    uint balance_from = balances[msg.sender];
    uint balance_to = balances[to];
    require(balance_from >= amount);
    balances[msg.sender] = balance_from - amount;
    balances[to] = safeAdd(balance_to, amount);
}

/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @dev Users can send more ether than token to be bought, t
function buy(uint desired_tokens) public payable {
    // Check if enough ether has been sent
    uint required_wei_sent = (desired_tokens / 10) * decimals
    require(msg.value >= required_wei_sent);

    // Mint the tokens
    totalSupply = safeAdd(totalSupply, desired_tokens);
    balances[msg.sender] = safeAdd(balances[msg.sender], desi
    emit Mint(msg.sender, desired_tokens);
}

/// @notice Add two values. Revert if overflow
function safeAdd(uint a, uint b) pure internal returns(uint)
    if (a + b < a) {
        revert();
    }
    return a + b;
```

```
}
}
```

> “To avoid lock of funds, the following feature(s) MUST be implemented before contract deployment”
>
> — 1 of 8

- [ ] A. A `transferFrom()` function
- [ ] B. A `burn()` function
- [x] C. A way to withdraw/exchange/use Ether from the contract
- [ ] D. None of the above

▼ Solution

Correct is C.

> Locked Ether: Contracts that accept Ether via payable functions but without withdrawal mechanisms will lock up that Ether. Remove payable attribute or add withdraw function.

from point 29 of Security Pitfalls & Best Practices 101 - by Secureum

> “Which of the following assertion(s) is/are true (without affecting the security posture of the contract)?”
>
> — 2 of 8

- [ ] A. `buy()` does not need `payable` keyword
- [ ] B. `balances` must be `private`
- [x] C. `transfer()` can be `external`
- [x] D. `safeAdd()` can be `public`

▼ Solution

Correct is C, D. `buy()` cannot function without being `payable`. There's no reason the visibility of `balances` needs to be `private`. `transfer()` can be

`external` since it's not called internally. `safeAdd()` can be `public` since it is a pure function.

> "The total supply is limited by"

— 3 of 8

- [ ] A. 10**18
- [ ] B. 100 * 10**18
- [ ] C. 100
- [x] D. None of the above

▼ Solution

Correct is D. It would be B, but `MAX_SUPPLY` isn't actually used anywhere in the code.

> "The following issue(s) is/are present in the codebase"

— 4 of 8

- [ ] A. An integer underflow allows one to drain Ether
- [x] B. Unsafe rounding allows one to receive new tokens for free
- [ ] C. A division by zero allows one to trap/freeze the system
- [ ] D. None of the above

▼ Solution

Correct is B. It's impossible to get any Ether out of this contract, so no draining either. It divides `desired_tokens` first and only then multiplies by the decimals, this causes any amount of tokens below 10 to result in 0 `required_wei_sent`. There are no divisions here that could allow a division by 0.

> "The following issue(s) is/are present in the codebase"

— 5 of 8

- [ ] A. A front-running allows one to pay less than expected for tokens
- [ ] B. A lack of access control allows one to receive tokens for free
- [x] C. Incorrect balance update allows one to receive new tokens for free
- [ ] D. None of the above

▼ Solution

Correct is C. No requests made before/after a function call would be able to change the token price. All of the functions are intended to be used by users, so no "access control" would be possible without excluding users. A user can send all of their tokens to themselve, which will double their balance due to the pre-loaded variable reuse.

> “The following issue(s) is/are present in the codebase”

— 6 of 8

- [ ] A. A reentrancy allows one to drain Ether
- [ ] B. A reentrancy allows one to drain the tokens
- [ ] C. A reentrancy allows one to receive new tokens for free
- [x] D. None of the above

▼ Solution

Correct is D. No reentrancies are possible since no external calls are made.

> “The following issue(s) is/are present in the codebase”

— 7 of 8

- [ ] A. An integer overflow allows one to drain Ether
- [ ] B. An integer overflow allows one to receive new tokens for free
- [ ] C. An integer overflow allows one to trap/freeze the system
- [x] D. None of the above

▼ Solution

Correct is D. While it is indeed possible to exploit an overflow at the multiplication (`(desired_tokens / 10) * decimals`), it doesn't allow you to receive FREE tokens (although it makes them a bargain).

> "The InSecureumToken contract strictly follows the specification of"
>
> — 8 of 8

- [ ] A. ERC20
- [ ] B. ERC777
- [ ] C. ERC721
- [x] D. None of the above

▼ Solution

Correct is D. Since decimals was defined as `uint` (same as `uint256`) and not as `uint8` as ERC20 or ERC777 standardized, it can't strictly be either of them. And since this is clearly a fungible token contract, it can't be ERC721.

In Blockchain Tags Ethereum, Secureum Bootcamp

← Secureum Bootcamp Audit Techniq… Damn Vulnerable DeFi v2 - part #1: Set…