Secureum Bootcamp Epoch∞ - June RACE #7

July 5, 2022 / patrickd

This is a write-up of the Secureum Bootcamp Race 7 Quiz of Epoch Infinity with explanations.
For fairness it was published after submissions to it were closed.

This quiz had a strict time limit of 16 minutes for 8 questions, no pause.
Choose all and *only* correct answers.
Syntax highlighting was omitted since the original quiz did not have any either.

Note: All 8 questions in this RACE are based on the InSecureumApe contract. This is the same contract you will see for all the 8 questions in this RACE. InSecureumApe is adapted from a well-known contract. The question is below the shown contract.

```
pragma solidity ^0.7.0;

import "https://github.com/OpenZeppelin/openzeppelin-contracts
import "https://github.com/OpenZeppelin/openzeppelin-contracts
import "https://github.com/OpenZeppelin/openzeppelin-contracts

contract InSecureumApe is ERC721, Ownable {
    using SafeMath for uint256;

    string public IA_PROVENANCE = "";
uint256 public startingIndexBlock;

uint256 public startingIndex;

uint256 public constant apePrice = 80000000000000000; //(

uint public constant maxApePurchase = 20;

uint256 public MAX_APES;

bool public saleIsActive = false;

uint256 public REVEAL_TIMESTAMP;

constructor(string memory name, string memory symbol, uint
    MAX_APES = maxNftSupply;
    REVEAL_TIMESTAMP = saleStart + (86400 * 9);
}

function withdraw() public onlyOwner {
    uint balance = address(this).balance;
    msg.sender.transfer(balance);
}

function reserveApes() public onlyOwner {
    uint supply = totalSupply();
    uint i;
    for (i = 0; i < 30; i++) {
        _safeMint(msg.sender, supply + i);
    }
}

function setRevealTimestamp(uint256 revealTimeStamp) publi
    REVEAL_TIMESTAMP = revealTimeStamp;
}

function setProvenanceHash(string memory provenanceHash) p
    IA_PROVENANCE = provenanceHash;
}
```

```
    function setBaseURI(string memory baseURI) public onlyOwne
        _setBaseURI(baseURI);
    }

    function flipSaleState() public onlyOwner {
        saleIsActive = !saleIsActive;
    }

    function mintApe(uint numberOfTokens) public payable {
        require(saleIsActive, "Sale must be active to mint Ape
        require(numberOfTokens < maxApePurchase, "Can only min
        require(totalSupply().add(numberOfTokens) <= MAX_APES,
        require(apePrice.mul(numberOfTokens) <= msg.value, "Et

        for(uint i = 0; i < numberOfTokens; i++) {
            uint mintIndex = totalSupply();
            if (totalSupply() < MAX_APES) {
                _safeMint(msg.sender, mintIndex);
            }
        }

        // If we haven't set the starting index and this is ei
        // the end of pre-sale, set the starting index block
        if (startingIndexBlock == 0 && (totalSupply() == MAX_A
            startingIndexBlock = block.number;
        }
    }

    function setStartingIndex() public {
        require(startingIndex == 0, "Starting index is already
        require(startingIndexBlock != 0, "Starting index block

        startingIndex = uint(blockhash(startingIndexBlock)) %
        if (block.number.sub(startingIndexBlock) > 255) {
            startingIndex = uint(blockhash(block.number - 1))
        }
        if (startingIndex == 0) {
            startingIndex = startingIndex.add(1);
        }
    }

    function emergencySetStartingIndexBlock() public onlyOwne
        require(startingIndex == 0, "Starting index is alread
        startingIndexBlock = block.number;
    }
}
```

> “The mint price of an InSecureumApe is:”

— 1 of 8

- ☐ A. 0.0008 ETH
- ☐ B. 0.008 ETH
- ☐ C. 0.08 ETH
- ☑ D. 0.8 ETH

▼ Solution

Correct is D

We can see the price is determined by the `apePrice` constant in wei, by which the number of tokens to mint are multiplied by.

The inline comment claims it to be `//0.08 ETH` but, knowing that ethereum has 18 decimals, we can check and realize the price actually 0.8 eth.

The https://eth-toolbox.com/ website offers a quick way to convert between these denominations.

It would've been a lot better if the code made use of denominations, this would've made the code much more readable and likely prevented the issue: `0.08 ether`.

> “The security concern(s) with InSecureumApe access control is/are”

— 2 of 8

- ✓ A. Owner can arbitrarily pause public minting of InSecureumApes
- ✓ B. Owner can arbitrarily mint InSecureumApes
- ✓ C. Single-step ownership change
- ✓ D. Missing event emits in and time-delayed effects of owner functions

▼ Solution

Correct is A, B, C, D

The `saleIsActive` state variable is checked within `mintApe()` can be toggled via the `flipSaleState()` by the owner at any time, without delay or warning.

The `reserveApes()` function allows the owner to mint arbitrary amounts of tokens at any time even bypassing the `MAX_APES` maximum supply config set during construction.

The `transferOwnership()` function inherited from OpenZeppelin's Ownable contract only ensures that ownership is not transferred to the zero-address, but it can be transferred in a single step to any other potentially invalid address.

None of the functions using the `onlyOwner` modifier emit events or have any sort of time-delay for their action, due to this users can suffer from unwanted surprises that are difficult to monitor for.

> “The security concern(s) with InSecureumApe constructor is/are”
>
> — 3 of 8

- ✓ A. Missing sanity/threshold check on maxNftSupply
- ✓ B. Missing sanity/threshold check on saleStart
- ✓ C. Potential integer overflow
- ☐ D. None of the above

▼ Solution

Correct is A, B, C

None of the mentioned parameters are sanity/threshold checked which would allow accidental deployment with incorrect parameters that could be noticed too late, after money has already gone into the contract.

Unlike in Solidity 0.8.x, integer overflows aren't automatically checked for in this version, so an extremely high `saleStart` value could indeed cause an integer overflow, although unlikely for sane values. The best practice is to use a SafeMath library here.

> “The total number of InSecureumApes that can ever be minted is”
>
> — 4 of 8

- ☐ A. maxApePurchase
- ☐ B. MAX_APES
- ☐ C. MAX_APES + 30
- ☑ D. type(uint256).max

▼ Solution

Correct is D

Since the `reserveApes()` function allows the owner to arbitrarily mint tokens without checking the `MAX_APES` variable, it's possible to mint as many tokens as the totalSupply variable can hold, which is the maximum value an uint256 can have.

> “The public minting of InSecureumApes”
>
> — 5 of 8

- ☐ A. Must be paid the exact amount in Ether
- ☑ B. May be performed 19 NFTs at a time
- ☑ C. Uses _safeMint to prevent locked/stuck NFTs
- ☐ D. None of the above

▼ Solution

Correct is B, C

The amount doesn't need to be paid exactly, more can be sent but shouldn't since any above this amount is kept by the protocol and not sent back.

The contract doesn't correctly check how many tokens can be minted at a time, it should be `numberOfTokens <= maxApePurchase` to allow 20 as described.

The contract indeed uses the `_safeMint()` function that'll ensure that if the receiver is a contract, it must correctly implement the `onERC721Received()` function, proving that the receiver is capable of handling NFTs and that they won't be stuck after receiving them.

> "The security concerns with InSecureumApe is/are"
>
> — 6 of 8

- [✓] A. Use of a floating pragma and an older compiler version
- [ ] B. Oracle price manipulation
- [✓] C. Reentrancy allowing bypass of maxApePurchase check
- [ ] D. None of the above

▼ Solution

Correct is A, C

The best practice is to avoid floating pragmas for contracts to ensure that they're always tested with the same Solidity version throughout the entire development cycle until deployment.

The contract does not make use of any oracles.

Since `_safeMint()` is used and calls `onERC721Received()` on receiving contracts, a NFT receiver can indeed call back into the `mintApe()` function and bypass how many tokens can be minted within a single transaction. But this check can be bypassed by simply repeatedly calling `mintApe()` from a custom contract since the function doesn't ensure that only EOAs can call it.

> "The starting index determination"
>
> — 7 of 8

- [✓] A. Is meant to randomize NFT reveal post-mint
- [✓] B. Can be triggered by the owner at any time
- [ ] C. May be triggered only 9 days after sale start
- [✓] D. Accounts for the fact that EVM only stores previous 256 block hashes

▼ Solution

Correct is A, B, D

You can read about how this is used for post-mint reveal randomization in this article.

The 9-day delay of the `REVEAL_TIMESTAMP` variable can be overriden at any point in time, it can also be triggered earlier if the totalSupply matches `MAX_APES` exactly, or be triggered at any time by the owner via `emergencySetStartingIndexBlock()`.

It accounts for the block hash access limitation by falling back to using the hash of the previous block instead.

> "Potential gas optimization(s) in InSecureumApe is/are"
>
> — 8 of 8

- [✓] A. Caching of storage variables
- [✓] B. Avoiding initializations of variables to default values of their types
- [✓] C. Use of immutables
- [ ] D. None of the above

▼ Solution

Correct is A, B, C

Whenever storage variables are read from multiple times, they should be cached in memory to safe gas. This is missing for `MAX_APES` in `mintApe()` and `startingIndexBlock` in `setStartingIndex()`.

All state variables are zero-initialized by default, therefore there's no need to manually set `saleIsActive` to false, for example.

The state variable `MAX_APES` is only set once during construction and should be immutable to save gas

In Blockchain Tags Ethereum, Secureum Bootcamp

← Secureum Bootcamp Epoch∞ - July … Damn Vulnerable DeFi V2 - #13 Junior …