Secureum Bootcamp Epoch∞ - July RACE #8

August 2, 2022 / patrickd

This is a write-up of the Secureum Bootcamp Race 8 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 following ERC721 implementation. This is the same contract you will see for all the 8 questions in this RACE. The implementation is adapted from a well-known contract. The question is below the shown contract.

```
pragma solidity >=0.8.0;

abstract contract ERC721 {

    event Transfer(address indexed from, address indexed to, u

    event Approval(address indexed owner, address indexed spe

    event ApprovalForAll(address indexed owner, address indexe
string public name;

string public symbol;

function tokenURI(uint256 id) public view virtual returns

mapping(uint256 => address) internal _ownerOf;

mapping(address => uint256) internal _balanceOf;

function ownerOf(uint256 id) public view virtual returns (
    require((owner = _ownerOf[id]) != address(0), "NOT_MI
}

function balanceOf(address owner) public view virtual ret
    require(owner != address(0), "ZERO_ADDRESS");

    return _balanceOf[owner];
}

mapping(uint256 => address) public getApproved;

mapping(address => mapping(address => bool)) public isAppr

constructor(string memory _name, string memory _symbol) {
    name = _name;
    symbol = _symbol;
}

function approve(address spender, uint256 id) public virt
    address owner = _ownerOf[id];

    require(msg.sender == owner || isApprovedForAll[owner

    getApproved[id] = spender;

    emit Approval(owner, spender, id);
}
```

```
    function setApprovalForAll(address operator, bool approve
        isApprovedForAll[msg.sender][operator] = approved;

        emit ApprovalForAll(msg.sender, operator, approved);
    }

    function transferFrom(
        address from,
        address to,
        uint256 id
    ) public virtual {
        require(to != address(0), "INVALID_RECIPIENT");

        require(
            msg.sender == from || isApprovedForAll[from][msg.
            "NOT_AUTHORIZED"
        );

        unchecked {
            _balanceOf[from]--;

            _balanceOf[to]++;
        }

        _ownerOf[id] = to;

        delete getApproved[id];

        emit Transfer(from, to, id);
    }

    function safeTransferFrom(
        address from,
        address to,
        uint256 id
    ) public virtual {
        transferFrom(from, to, id);

        require(
            to.code.length == 0 ||
                ERC721TokenReceiver(to).onERC721Received(msg.s
                ERC721TokenReceiver.onERC721Received.selector,
            "UNSAFE_RECIPIENT"
        );
    }

    function safeTransferFrom(
        address from,
        address to,
        uint256 id,
        bytes calldata data
    ) public virtual {
        transferFrom(from, to, id);

        require(
            to.code.length == 0 ||
                ERC721TokenReceiver(to).onERC721Received(msg.s
                ERC721TokenReceiver.onERC721Received.selector,
            "UNSAFE_RECIPIENT"
        );
    }

    function supportsInterface(bytes4 interfaceId) public view
        return
            interfaceId == 0x01ffc9a7 || // ERC165 Interface I
            interfaceId == 0x80ac58cd || // ERC165 Interface I
            interfaceId == 0x5b5e139f; // ERC165 Interface ID
    }

    function _mint(address to, uint256 id) internal virtual {
        require(to != address(0), "INVALID_RECIPIENT");

        require(_ownerOf[id] == address(0), "ALREADY_MINTED"),

        unchecked {
            _balanceOf[to]++;
        }

        _ownerOf[id] = to;
        emit Transfer(address(0), to, id);
    }

    function _burn(uint256 id) external virtual {
        address owner = _ownerOf[id];

        require(owner != address(0), "NOT_MINTED");

        unchecked {
            _balanceOf[owner]--;
        }

        delete _ownerOf[id];

        delete getApproved[id];

        emit Transfer(owner, address(0), id);
    }

    function _safeMint(address to, uint256 id) internal virtu
        _mint(to, id);

        require(
            to.code.length == 0 ||
                ERC721TokenReceiver(to).onERC721Received(msg.
                ERC721TokenReceiver.onERC721Received.selector
            "UNSAFE_RECIPIENT"
        );
    }

    function _safeMint(
        address to,
        uint256 id,
        bytes memory data
    ) internal virtual {
        _mint(to, id);

        require(
            to.code.length == 0 ||
                ERC721TokenReceiver(to).onERC721Received(msg.
                    ERC721TokenReceiver.onERC721Received.selector,
                "UNSAFE_RECIPIENT"
            );
        }
    }

abstract contract ERC721TokenReceiver {
    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external virtual returns (bytes4) {
        return ERC721TokenReceiver.onERC721Received.selector;
    }
}
```

> "The security concern(s) addressed explicitly in _mint include"

— 1 of 8

- [x] A. Prevent minting to zero address
- [x] B. Prevent reminting of NFTs
- [x] C. Transparency by emitting event
- [ ] D. None of the above

▼ Solution

Correct is A, B, C

The `_mint()` function addresses both A and B with the first two requires. Also C is correct since the emission of the Transfer event allows for easy tracking of mints and therefore transparency.

> "The security concerns in _burn include"

— 2 of 8

- ☑ A. Anyone can arbitrarily burn NFTs
- ☐ B. Potential integer underflow because of unchecked
- ☐ C. Incorrect emission of event
- ☐ D. None of the above

▼ Solution

Correct is A

It appears that the `_burn()` function was intended to be internal (based on the underscore prefix) but is actually external which allows for A.

Answer B is not a concern thanks to the ownership check ensuring that it cannot happen.

The emission of the event follows the event declaration and therefore C is not a concern either.

> "The security concern(s) addressed explicitly in _safeMint include"
>
> — 3 of 8

- ☑ A. Validating if the recipient is an EOA
- ☐ B. Ensuring that the recipient can only be an EOA
- ☑ C. Validating if the recipient is an ERC721 aware contract
- ☐ D. None of the above

▼ Solution

Correct is A, C

This function ensures that if (A) the recipient is an EOA the mint functions normally thanks to the `to.code.length == 0` check, but if (C) the recipient is a contract (non-EOA) it must be "ERC721 aware" by implementing the `ERC721TokenReceiver` interface.

> “Function approve”

— 4 of 8

- [x] A. Allows the NFT owner to approve a spender
- [ ] B. Allows the NFT spender to approve an operator
- [x] C. Allows the NFT operator to approve a spender
- [ ] D. None of the above

▼ Solution

Correct is A, C

The require shows that only (A) the NFT owner and (C) the operator that the owner gave access to manage all their NFTs have the ability to approve spenders. A spender cannot approve other spenders and especially not operators.

> “Function setApprovalForAll”

— 5 of 8

- [ ] A. Approves msg.sender to manage operator’s NFTs
- [ ] B. Gives everyone approval to manage msg.sender’s NFTs
- [ ] C. Revokes everyone’s approvals to manage msg.sender’s NFTs
- [x] D. None of the above

▼ Solution

Correct is D

The `setApprovalForAll()` function authorizes an address (called the operator) to manage all of the owner's NFTs in the contract. A, B and C are therefore incorrect.

> “The security concern(s) in transferFrom include”

— 6 of 8

- ✓ A. Allowing the msg.sender to transfer any NFT
- ✓ B. NFTs potentially stuck in recipient contracts
- ✓ C. Potential integer underflow
- ☐ D. None of the above

▼ Solution

Correct is A, B, C

The `transferFrom()` function does not check ownership of the NFT. This allows any `msg.sender` to overwrite the current owner, basically allowing a transfer of any NFT.

The `safeTransferFrom()` function ensures that NFTs will not be stuck in recipient contracts that don't communicate that they are able to handle them. This issue still exists for the normal `transferFrom()` function though for backwards compatability reasons.

Due to the missing ownership check, it's possible for the balance of the sender to underflow.

> “Which of the following is/are true?”
>
> — 7 of 8

- ✓ A. NFT ownership is tracked by _ownerOf
- ✓ B. NFT balance is tracked by _balanceOf
- ✓ C. NFT approvals are tracked by getApproved
- ✓ D. NFT operator can transfer all of owner’s NFTs

▼ Solution

Correct is A, B, C, D

The variables `_ownerOf`, `_balanceOf` and `getApproved` indeed keep track of the mentioned values.

And NFT operators are by definition able to transfer all NFTs of the owners that elected them to be their operators.

> “ERC721 recognizes the following role(s)”
>
> — 8 of 8

- [✓] A. Owner
- [✓] B. Spender (Approved address)
- [✓] C. Operator
- [ ] D. None of the above

▼ Solution

Correct is A, B, C

This is quite apparent from ERC721 implementation parameter names. They can also be found in the EIP721 spec.

In Blockchain Tags Ethereum, Secureum Bootcamp

← Paradigm CTF 2021 - Smart Contrac… Secureum Bootcamp Epoch∞ - June R…