Secureum Bootcamp Epoch∞ - May RACE #6

May 17, 2022 / patrickd

This is a write-up of the Secureum Bootcamp Race 6 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 InSecureumLand contract.** This is the same contract you will see for all the 8 questions in this RACE. **InSecureumLand** is adapted from a well-known contract. The questions are below the shown contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.10;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC72
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.s
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol"
import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";
```

```
import "@openzeppelin/contracts/utils/cryptography/MerkleProo

contract InSecureumLand is ERC721Enumerable, Ownable, Reentra
    using SafeERC20 for IERC20;

    // attributes
    string private baseURI;
    address public operator;
    bool public publicSaleActive;
    uint256 public publicSaleStartTime;
    uint256 public publicSalePriceLoweringDuration;
    uint256 public publicSaleStartPrice;
    uint256 public publicSaleEndingPrice;
    uint256 public currentNumLandsMintedPublicSale;
    uint256 public mintIndexPublicSaleAndContributors;
    address public tokenContract;
    bool private isKycCheckRequired;
    bytes32 public kycMerkleRoot;
    uint256 public maxMintPerTx;
    uint256 public maxMintPerAddress;
    mapping(address => uint256) public mintedPerAddress;
    bool public claimableActive;
    bool public adminClaimStarted;
    address public alphaContract;
    mapping(uint256 => bool) public alphaClaimed;
    uint256 public alphaClaimedAmount;
    address public betaContract;
    mapping(uint256 => bool) public betaClaimed;
    uint256 public betaClaimedAmount;
    uint256 public betaNftIdCurrent;
    bool public contributorsClaimActive;
    mapping(address => uint256) public contributors;
    uint256 public futureLandsNftIdCurrent;
    address public futureMinter;
    Metadata[] public metadataHashes;
    bytes32 public keyHash;
    uint256 public fee;
    uint256 public publicSaleAndContributorsOffset;
    uint256 public alphaOffset;
    uint256 public betaOffset;
```

```
mapping(bytes32 => bool) public isRandomRequestForPublicSa
bool public publicSaleAndContributorsRandomnessRequested;
bool public ownerClaimRandomnessRequested;

// constants
uint256 immutable public MAX_LANDS;
uint256 immutable public MAX_LANDS_WITH_FUTURE;
uint256 immutable public MAX_ALPHA_NFT_AMOUNT;
uint256 immutable public MAX_BETA_NFT_AMOUNT;
uint256 immutable public MAX_PUBLIC_SALE_AMOUNT;
uint256 immutable public RESERVED_CONTRIBUTORS_AMOUNT;
uint256 immutable public MAX_FUTURE_LANDS;
uint256 constant public MAX_MINT_PER_BLOCK = 150;

// structs
struct LandAmount {
    uint256 alpha;
    uint256 beta;
    uint256 publicSale;
    uint256 future;
}
struct ContributorAmount {
    address contributor;
    uint256 amount;
}
struct Metadata {
    bytes32 metadataHash;
    bytes32 shuffledArrayHash;
    uint256 startIndex;
    uint256 endIndex;
}
struct ContractAddresses {
    address alphaContract;
    address betaContract;
    address tokenContract;
}

// modifiers
modifier whenPublicSaleActive() {
    require(publicSaleActive, "Public sale is not active");
```

```
    _;
}

modifier whenContributorsClaimActive() {
    require(contributorsClaimActive, "Contributors Claim is
    _;
}

modifier whenClaimableActive() {
    require(claimableActive && !adminClaimStarted, "Claimab
    _;
}

modifier checkMetadataRange(Metadata memory _landMetadata){
    require(_landMetadata.endIndex < MAX_LANDS_WITH_FUTURE,
    _;
}

modifier onlyContributors(address _contributor){
    require(contributors[_contributor] >= 0, "Only contribu
    _;
}

modifier onlyOperator() {
    require(operator == msg.sender , "Only operator can ca
    _;
}

modifier onlyFutureMinter() {
    require(futureMinter == msg.sender , "Only futureMinter
    _;
}

modifier checkFirstMetadataRange(uint256 index, uint256 sta
    if(index == 0){
        require(startIndex == 0, "For first metadata range
        require(endIndex == MAX_LANDS - 1, "For first metad
    }
    _;
}
```

```
// events
event LandPublicSaleStart(
    uint256 indexed _saleDuration,
    uint256 indexed _saleStartTime
);
event LandPublicSaleStop(
    uint256 indexed _currentPrice,
    uint256 indexed _timeElapsed
);
event ClaimableStateChanged(bool indexed claimableActive);
event ContributorsClaimStart(uint256 _timestamp);
event ContributorsClaimStop(uint256 _timestamp);
event StartingIndexSetPublicSale(uint256 indexed _starting
event StartingIndexSetAlphaBeta(uint256 indexed _alphaOffs
event PublicSaleMint(address indexed sender, uint256 index

constructor(string memory name, string memory symbol,
    ContractAddresses memory addresses,
    LandAmount memory amount,
    ContributorAmount[] memory _contributors,
    address _vrfCoordinator, address _linkTokenAddress,
    bytes32 _vrfKeyHash, uint256 _vrfFee,
    address _operator
) ERC721(name, symbol) VRFConsumerBase(_vrfCoordinator, _l
    alphaContract = addresses.alphaContract;
    betaContract = addresses.betaContract;
    tokenContract = addresses.tokenContract;

    MAX_ALPHA_NFT_AMOUNT = amount.alpha;
    MAX_BETA_NFT_AMOUNT = amount.beta;
    MAX_PUBLIC_SALE_AMOUNT = amount.publicSale;
    MAX_FUTURE_LANDS = amount.future;

    betaNftIdCurrent = amount.alpha; //beta starts after a
    mintIndexPublicSaleAndContributors = amount.alpha + amo

    uint256 tempSum;
    for(uint256 i; i<_contributors.length; ++i){
        contributors[_contributors[i].contributor] = _contr
```

```
            tempSum += _contributors[i].amount;
        }
        RESERVED_CONTRIBUTORS_AMOUNT = tempSum;
        MAX_LANDS = amount.alpha + amount.beta + amount.publics
        MAX_LANDS_WITH_FUTURE = MAX_LANDS + amount.future;
        futureLandsNftIdCurrent = MAX_LANDS; //future starts a
        keyHash  = _vrfKeyHash;
        fee = _vrfFee;
        operator = _operator;
    }

    function _baseURI() internal view override returns (string
        return baseURI;
    }

    function setBaseURI(string memory uri) external onlyOperat
        baseURI = uri;
    }

    function setOperator(address _operator) external onlyOwner
        operator = _operator;
    }

    function setMaxMintPerTx(uint256 _maxMintPerTx) external o
        maxMintPerTx = _maxMintPerTx;
    }

    function setMaxMintPerAddress(uint256 _maxMintPerAddress) e
        maxMintPerAddress = _maxMintPerAddress;
    }

    function setKycCheckRequired(bool _isKycCheckRequired) exte
        isKycCheckRequired = _isKycCheckRequired;
    }

    function setKycMerkleRoot(bytes32 _kycMerkleRoot) external
        kycMerkleRoot = _kycMerkleRoot;
    }

    // Public Sale Methods
```

```
function startPublicSale(
    uint256 _publicSalePriceLoweringDuration,
    uint256 _publicSaleStartPrice,
    uint256 _publicSaleEndingPrice,
    uint256 _maxMintPerTx,
    uint256 _maxMintPerAddress,
    bool _isKycCheckRequired
) external onlyOperator {
    require(!publicSaleActive, "Public sale has already beg

    publicSalePriceLoweringDuration = _publicSalePriceLower
    publicSaleStartPrice = _publicSaleStartPrice;
    publicSaleEndingPrice = _publicSaleEndingPrice;
    publicSaleStartTime = block.timestamp;
    publicSaleActive = true;

    maxMintPerTx = _maxMintPerTx;
    maxMintPerAddress = _maxMintPerAddress;
    isKycCheckRequired = _isKycCheckRequired;
    emit LandPublicSaleStart(publicSalePriceLoweringDuratio
}

function stopPublicSale() external onlyOperator whenPublicS
    emit LandPublicSaleStop(getMintPrice(), getElapsedSaleT
    publicSaleActive = false;
}

function getElapsedSaleTime() private view returns (uint256
    return publicSaleStartTime > 0 ? block.timestamp - publ
}

function getMintPrice() public view whenPublicSaleActive re
    uint256 elapsed = getElapsedSaleTime();
    uint256 price;
    if(elapsed < publicSalePriceLoweringDuration) {
        // Linear decreasing function
        price =
            publicSaleStartPrice -
                ( ( publicSaleStartPrice - publicSaleEndingP
    } else {
```

```
            price = publicSaleEndingPrice;
        }
        return price;
    }

    function mintLands(uint256 numLands, bytes32[] calldata me
        require(numLands > 0, "Must mint at least one beta");
        require(currentNumLandsMintedPublicSale + numLands <= M
        require(numLands <= maxMintPerTx, "numLands should not
        require(numLands + mintedPerAddress[msg.sender] <= maxM
        if(isKycCheckRequired) {
            require(MerkleProof.verify(merkleProof, kycMerkleR
        } else {
            require(msg.sender == tx.origin, "Minting from sma
        }

        uint256 mintPrice = getMintPrice();
        IERC20(tokenContract).safeTransferFrom(msg.sender, addr
        mintedPerAddress[msg.sender] += numLands;
        emit PublicSaleMint(msg.sender, numLands, mintPrice);
        mintLandsCommon(numLands, msg.sender);
    }

    function mintLandsCommon(uint256 numLands, address recipie
        for (uint256 i; i < numLands; ++i) {
            _safeMint(recipient, mintIndexPublicSaleAndContribu
        }
    }

    function withdraw() external onlyOwner {
        uint256 balance = address(this).balance;
        if(balance > 0){
            Address.sendValue(payable(owner()), balance);
        }
        balance = IERC20(tokenContract).balanceOf(address(this)
        if(balance > 0){
            IERC20(tokenContract).safeTransfer(owner(), balance
        }
    }
```

```
// Alpha/Beta Claim Methods
function flipClaimableState() external onlyOperator {
    claimableActive = !claimableActive;
    emit ClaimableStateChanged(claimableActive);
}

function nftOwnerClaimLand(uint256[] calldata alphaTokenIds
    require(alphaTokenIds.length > 0 || betaTokenIds.length
    require(alphaTokenIds.length + betaTokenIds.length <= M

    alphaClaimLand(alphaTokenIds);
    betaClaimLand(betaTokenIds);
}

function alphaClaimLand(uint256[] calldata alphaTokenIds) p
    for(uint256 i; i < alphaTokenIds.length; ++i){
        uint256 alphaTokenId = alphaTokenIds[i];
        require(!alphaClaimed[alphaTokenId], "ALPHA NFT alr
        require(ERC721(alphaContract).ownerOf(alphaTokenId)

        alphaClaimLandByTokenId(alphaTokenId);
    }
}

function alphaClaimLandByTokenId(uint256 alphaTokenId) priv
    alphaClaimed[alphaTokenId] = true;
    ++alphaClaimedAmount;
    _safeMint(msg.sender, alphaTokenId);
}

function betaClaimLand(uint256[] calldata betaTokenIds) pri
    for(uint256 i; i < betaTokenIds.length; ++i){
        uint256 betaTokenId = betaTokenIds[i];
        require(!betaClaimed[betaTokenId], "BETA NFT alread
        require(ERC721(betaContract).ownerOf(betaTokenId) =

        betaClaimLandByTokenId(betaTokenId);
    }
}
```

```
function betaClaimLandByTokenId(uint256 betaTokenId) privat
    betaClaimed[betaTokenId] = true;
    ++betaClaimedAmount;
    _safeMint(msg.sender, betaNftIdCurrent++);
}

// Contributors Claim Methods
function startContributorsClaimPeriod() onlyOperator exter
    require(!contributorsClaimActive, "Contributors claim 
    contributorsClaimActive = true;
    emit ContributorsClaimStart(block.timestamp);
}

function stopContributorsClaimPeriod() onlyOperator externa
    contributorsClaimActive = false;
    emit ContributorsClaimStop(block.timestamp);
}

function contributorsClaimLand(uint256 amount, address rec
    require(amount > 0, "Must mint at least one land");
    require(amount <= MAX_MINT_PER_BLOCK, "amount should n
    mintLandsCommon(amount, recipient);
}

function claimUnclaimedAndUnsoldLands(address recipient) e
    claimUnclaimedAndUnsoldLandsWithAmount(recipient, MAX_M
}

function claimUnclaimedAndUnsoldLandsWithAmount(address re
    require (publicSaleStartTime > 0 && !claimableActive &&
        "Cannot claim the unclaimed if claimable or public 
    require(maxAmount <= MAX_MINT_PER_BLOCK, "maxAmount ca
    require(alphaClaimedAmount < MAX_ALPHA_NFT_AMOUNT || be
                || mintIndexPublicSaleAndContributors < MAX

    uint256 totalMinted;
    adminClaimStarted = true;
    //claim beta
    if(betaClaimedAmount < MAX_BETA_NFT_AMOUNT) {
        uint256 leftToBeMinted = MAX_BETA_NFT_AMOUNT - beta
```

```
        uint256 toMint = leftToBeMinted < maxAmount ? leftT
            maxAmount; //take the min

        uint256 target = betaNftIdCurrent + toMint;
        for(; betaNftIdCurrent < target; ++betaNftIdCurrent
            ++betaClaimedAmount;
            ++totalMinted;
            _safeMint(recipient, betaNftIdCurrent);
        }
    }

    //claim alpha
    if(alphaClaimedAmount < MAX_ALPHA_NFT_AMOUNT) {
        uint256 leftToBeMinted = MAX_ALPHA_NFT_AMOUNT - alp
        uint256 toMint = maxAmount < leftToBeMinted + total
                        maxAmount :
                        leftToBeMinted + totalMinted; //sur

        uint256 lastAlphaNft = MAX_ALPHA_NFT_AMOUNT - 1;
        for(uint256 i; i <= lastAlphaNft && totalMinted < t
            if(!alphaClaimed[i]){
                ++alphaClaimedAmount;
                ++totalMinted;
                alphaClaimed[i] = true;
                _safeMint(recipient, i);
            }
        }
    }

    //claim unsold
    if(mintIndexPublicSaleAndContributors < MAX_LANDS){
        uint256 leftToBeMinted = MAX_LANDS - mintIndexPubli
        uint256 toMint = maxAmount < leftToBeMinted + total
                        maxAmount :
                        leftToBeMinted + totalMinted; //sur

        for(; mintIndexPublicSaleAndContributors < MAX_LAND
                ++totalMinted;
                _safeMint(recipient, mintIndexPublicSaleAnd
        }
```

```
    }
}

//future
function setFutureMinter(address _futureMinter) external or
    futureMinter = _futureMinter;
}

function mintFutureLands(address recipient) external onlyFu
    mintFutureLandsWithAmount(recipient, MAX_MINT_PER_BLOCK
}

function mintFutureLandsWithAmount(address recipient, uint2
    require(maxAmount <= MAX_MINT_PER_BLOCK, "maxAmount car
    require(futureLandsNftIdCurrent < MAX_LANDS_WITH_FUTURE
    for(uint256 claimed; claimed < maxAmount && futureLands
        _safeMint(recipient, futureLandsNftIdCurrent++);
    }
}

// metadata
function loadLandMetadata(Metadata memory _landMetadata)
    external onlyOperator checkMetadataRange(_landMetadata)
    checkFirstMetadataRange(metadataHashes.length, _landMet
{
    metadataHashes.push(_landMetadata);
}

function putLandMetadataAtIndex(uint256 index, Metadata mem
    external onlyOperator checkMetadataRange(_landMetadata)
    checkFirstMetadataRange(index, _landMetadata.startIndex
{
    metadataHashes[index] = _landMetadata;
}

// randomness
function requestRandomnessForPublicSaleAndContributors() ex
    require(!publicSaleAndContributorsRandomnessRequested,
    publicSaleAndContributorsRandomnessRequested = true;
    requestId = requestRandomnessPrivate();
```

```
        isRandomRequestForPublicSaleAndContributors[requestId]
    }

    function requestRandomnessForOwnerClaim() external onlyOpe
        require(!ownerClaimRandomnessRequested, "Owner Claim O
        ownerClaimRandomnessRequested = true;
        requestId = requestRandomnessPrivate();
        isRandomRequestForPublicSaleAndContributors[requestId]
    }

    function requestRandomnessPrivate() private returns (bytes
        require(
            LINK.balanceOf(address(this)) >= fee,
            "Not enough LINK"
        );
        return requestRandomness(keyHash, fee);
    }

    function fulfillRandomness(bytes32 requestId, uint256 rand
        if(isRandomRequestForPublicSaleAndContributors[request
            publicSaleAndContributorsOffset = (randomness % (MA
            emit StartingIndexSetPublicSale(publicSaleAndContr
        } else {
            alphaOffset = (randomness % MAX_ALPHA_NFT_AMOUNT);
            betaOffset = (randomness % MAX_BETA_NFT_AMOUNT);
            emit StartingIndexSetAlphaBeta(alphaOffset, betaOf
        }
    }
}
```

> “The security concern(s) with InSecureumLand is/are”
>
> — 1 of 8

- [x] A. Single-step ownership change
- [ ] B. Incorrectly implemented KYC check using Merkle proofs
- [x] C. Missing time-delayed change of critical parameters

- [ ] D. Accidentally sent Ether gets locked in contract

▼ Solution

Correct is A, C.

A. Ownership management is inherited from OpenZeppelin's `Ownable` abstract contract, which only allows for single-step ownership change. If the ownership is mistakenly changed to an incorrect address, it could be permanently lost.

B. Contract appears to correctly make use of OpenZeppelin's `MerkleProof` library for KYC purposes.

C. Considering attributes like `operator` a critical parameter, it can indeed be argued that a time-delay would improve the contract's security.

D. Contract owner is be able to call the `withdraw()` function to extract any accidentally sent ether.

> "The security concern(s) with InSecureumLand setOperator() is/are"
>
> — 2 of 8

- [x] A. Missing zero-address validation
- [x] B. Missing event emission
- [ ] C. Incorrect modifier
- [ ] D. None of the above

▼ Solution

Correct is A, B.

A. There's indeed no check for zero-addresses, which could accidentally lead to no one being the operator. This would have little impact though, since the owner is able to correct the mistake by calling the function again.

B. There's also no event emitted when the operator is changed. This makes monitoring the contract for critical changes difficult.

C. Assuming that the intention is that only the owner should be able to update the operator, there seems to be no problem with the modifier that was chosen.

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

- [x] A. Minting could exceed max supply
- [ ] B. Minting could exceed `maxMintPerTx`
- [ ] C. Minting could exceed `maxMintPerAddress`
- [ ] D. None of the above

▼ Solution

Correct is A.

A. While the function checks `currentNumLandsMintedPublicSale` for whether the maximum supply has been exceeded, it doesn't actually ever increase this variable after minting. So it'll be possible to continue minting beyond the `MAX_PUBLIC_SALE_AMOUNT` value.

B. The `maxMintPerTx` value appears to be correctly checked against the `numLands` parameter.

C. The `maxMintPerAddress` value appears to be correctly checked against the overall amount of tokens that'll have been minted by the sender.

> “Missing threshold check(s) on parameter(s) is/are a concern in”
>
> — 4 of 8

- [ ] A. `mintLands`
- [x] B. `startPublicSale`
- [x] C. `contributorsClaimLand`
- [ ] D. None of the above

▼ Solution

Correct is B, C.

The `startPublicSale` should have some sanity checks for passed parameters like `_publicSaleStartPrice` and `_publicSaleEndingPrice`, especially since these cannot be corrected once set. The `contributorsClaimLand` function doesn't ensure the `amount` parameter, of how many tokens should be claimed for the contributor, is actually lower or equal to the amount of tokens they should be able to claim according to `contributors[msg.sender]`. It also doesn't update this amount allowing the contributor to claim the same amount multiple times.

> "The security concern(s) with InSecureumLand contributors claim functions is/are"
>
> — 5 of 8

- [ ] A. Anyone can call `startContributorsClaimPeriod`
- [ ] B. Anyone can call `stopContributorsClaimPeriod`
- [x] C. Anyone can call `contributorsClaimLand`
- [ ] D. None of the above

▼ Solution

Correct is C.

The first two functions can only be called by the operator. The `contributorsClaimLand` function appears to be only callable by contributors. But when looking at the `onlyContributors` modifier, callers are considered contributors even when they have not made any contribution (`contributors[_contributor] >= 0`). This error effectively allows anyone to call the `contributorsClaimLand` function.

> "The security concern(s) with InSecureumLand random number usage is/are"
>
> — 6 of 8

- [ ] A. It depends on miner-influenceable block.timestamp
- [ ] B. It depends on miner-influenceable blockhash
- [x] C. It depends on deprecated Chainlink VRF v1
- [ ] D. None of the above

▼ Solution

Correct is C.

It doesn't make use of miner-influenceable values for randomness. But it does indeed make use of a deprecated version of Chainlink's VRF. Projects should aim to make use of the most recent stable version of their dependencies before deployment.

> “The documentation/readability concern(s) with InSecureumLand is/are”
>
> — 7 of 8

- [ ] A. Stale comments
- [x] B. Missing NatSpec
- [x] C. Minimal inlined comments
- [ ] D. None of the above

▼ Solution

Correct is B, C.

There are no NatSpec comments at all, and the few inline comments that exist mostly just repeat what the code already states instead of explaining what is going on and what is the intention.

> “Potential gas optimization(s) (after appropriate security considerations) in InSecureumLand is/are”
>
> — 8 of 8

- [x] A. Removing `nonReentrant` modifier if mint addresses are known to be EOA
- [x] B. Using `_mint` instead of `_safeMint` if mint addresses are known to be EOA
- [x] C. Using unchecked in for loop increments
- [ ] D. None of the above

▼ Solution

Correct is A, B, C.

A. By checking `msg.sender == tx.origin` it can be known that the mint address, which is the mint function's caller, is an EOA, an account with a keypair and no bytecode. With that, transfer hooks are certain to not be triggered which means that `nonReentrant` can safely be omitted in this case.

B. If the mint address is known to be an EOA, `_mint` can be directly called, instead of `_safeMint` which first checks whether the receiver implements the hook function `onERC721Received`. An EOA has no bytecode, which means it's not possible that it implements this interface. A check like this also isn't necessary for EOAs since tokens cannot get stuck in them as they could in contracts.

C. Most loops increment with `++i` for which the solidity compiler will add overflow checks that cost additional gas. But the loop conditions (eg. `i < alphaTokenIds.length`) already ensures no overflow can happen. Therefore using an unchecked block around the increment can reduce gas cost by removing the unnecessary check. To implement this would require using a different loop though, since adding an unchecked block around the `for` loops primary expression would cause a compiler error.

In Blockchain Tags Ethereum, Secureum Bootcamp

← More EVM Puzzles - Part 1 Fuzzing for Memory Bugs in Solidity →