Secureum Bootcamp Audit Findings 201 Quiz

November 28, 2021 / patrickd

This is a writeup of the Secureum Bootcamp Audit Findings 201 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 InSecureumNFT contract snippet. This is the same contract snippet you will see for all the 8 questions in this quiz.* InSecureumNFT is a NFT project that aims to distribute CryptoSAFU NFTs to its community where most of them are fairdropped based on past contributions and a few are sold. CryptoSAFUs with lower IDs have more unique traits, may be valued higher and therefore require a random distribution for fairness. Assume that all strictly required ERC721 functionality (not shown) and any other required functionality (not shown) are implemented correctly. Only functionality specific to the sale and minting of NFTs is shown in this contract snippet.]

```
pragma solidity 0.8.0;

interface ERC721TokenReceiver {function onERC721Received(addr

// Assume that all strictly required ERC721 functionality (nc
// Assume that any other required functionality (not shown) i
contract InSecureumNFT {
    bytes4 internal constant MAGIC_ERC721_RECEIVED = 0x150b7a
    uint public constant TOKEN_LIMIT = 10; // 10 for testing,
    uint public constant SALE_LIMIT = 5; // 5 for testing, 13

    mapping (uint256 => address) internal idToOwner;
    uint internal numTokens = 0;
    uint internal numSales = 0;
    address payable internal deployer;
    address payable internal beneficiary;
    bool public publicSale = false;
    uint private price;
    uint public saleStartTime;
    uint public constant saleDuration = 13*13337; // 13337 bl
    uint internal nonce = 0;
    uint[TOKEN_LIMIT] internal indices;

    constructor(address payable _benificiary) {
        deployer = payable(msg.sender);
        beneficiary = _benificiary;
    }

    function startSale(uint _price) external {
        require(msg.sender == deployer || _price != 0, "Only
        price = _price;
        saleStartTime = block.timestamp;
        publicSale = true;
    }

    function isContract(address _addr) internal view returns
        uint256 size;
        assembly { size := extcodesize(_addr) }
```

```
        addressCheck = size > 0;
    }

    function randomIndex() internal returns (uint) {
        uint totalSize = TOKEN_LIMIT - numTokens;
        uint index = uint(keccak256(abi.encodePacked(nonce, m
        uint value = 0;
        if (indices[index] != 0) {
            value = indices[index];
        } else {
            value = index;
        }
        if (indices[totalSize - 1] == 0) {
            indices[index] = totalSize - 1;
        } else {
            indices[index] = indices[totalSize - 1];
        }
        nonce += 1;
        return (value + 1);
    }

    // Calculate the mint price
    function getPrice() public view returns (uint) {
        require(publicSale, "Sale not started.");
        uint elapsed = block.timestamp - saleStartTime;
        if (elapsed > saleDuration) {
            return 0;
        } else {
            return ((saleDuration - elapsed) * price) / saleD
        }
    }

    // SALE_LIMIT is 1337
    // Rest i.e. (TOKEN_LIMIT - SALE_LIMIT) are reserved for
    function mint() external payable returns (uint) {
        require(publicSale, "Sale not started.");
        require(numSales < SALE_LIMIT, "Sale limit reached.")
        numSales++;
        uint salePrice = getPrice();
        require((address(this)).balance >= salePrice, "Insuf
```

```
        if ((address(this)).balance >= salePrice) {
            payable(msg.sender).transfer((address(this)).bala
        }
        return _mint(msg.sender);
    }

    // TOKEN_LIMIT is 13337
    function _mint(address _to) internal returns (uint) {
        require(numTokens < TOKEN_LIMIT, "Token limit reachec
        // Lower indexed/numbered NFTs have rare traits and r
        // as more valuable by buyers => Therefore randomize
        uint id = randomIndex();
        if (isContract(_to)) {
            bytes4 retval = ERC721TokenReceiver(_to).onERC72
            idToOwner[id] = _to;
            numTokens = numTokens + 1;
            beneficiary.transfer((address(this)).balance);
            return id;
        }
    }
}
```

> "Missing zero-address check(s) in the contract"
>
> — 1 of 8

- [ ] A. May allow anyone to start the sale
- [x] B. May put the NFT sale proceeds at risk
- [ ] C. May burn the newly minted NFTs
- [ ] D. None of the above

▼ Solution

Correct is B. While the require statement in `startSale()` states that only the deployer may call the function AND the price needs to be not zero, the actual code uses OR which allows anyone to start the sale as long as they specify a valid price - but that can't be fixed by adding a zero-address check. All proceeds appear to be intended to go to the benificiary and since there's no validation of the `_benificiary` address when it is set during construction, a zero-address could indeed put the sale proceeds to risk. In the given code, the internal `_mint(_to)` function is always called with `msg.sender` as `_to` value which can't be a zero-address.

> "Given that lower indexed/numbered CryptoSAFU NFTs have rarer traits (and are considered more valuable as commented in `_mint`), the implementation of `InSecureumNFT` is susceptible to the following exploits"
>
> — 2 of 8

- [x] A. Buyers can repeatedly mint and revert until they receive desired NFT
- [x] B. Buyers can generate addresses to mint until they receive desired NFT
- [x] C. Miners can manipulate `block.timestamp` to facilitate minting of desired NFT
- [ ] D. None of the above

▼ Solution

Correct is A, B, C. The index of a CryptoSAFU NFT depends on a `nonce` that increases after every mint and has an `internal` visibility preventing contracts to read its current value easily, which would allow them to predict an index for the current block. But a prediction is not necessary since a contract can simply call `_mint()` repeatedly every block and revert if the result is not desired, ensuring a refund. The `msg.sender` is indeed also a variable for the "random" index generation, although it's very effective exploiting it, since you'd still have to pay the full price for each of those attempts because the nonce will change after each buy. There's also no need to generate a new address, you can just keep buying using the same address until you receive the desired NFT. A miner would indeed be able to pre-calculate a desirable index off-chain by picking a specific `block.timestamp` and adding their mint-transaction to the beginning of their block.

> "The `getPrice()` function"
>
> — 3 of 8

- [x] A. Is expected to reduce the mint price over time after sale starts
- [x] B. Allows free mints after ~13337 blocks from when `startSale()` is called
- [ ] C. Visibility should be changed to `external`
- [ ] D. None of the above

▼ Solution

Correct is A, B. The price is multiplied by `(saleDuration - elapsed)` and while `saleDuration` stays constant, `elapsed` will increase over time, making the multiplicator value and therefore the price lower over time. There's indeed a possibility for a free mint when `saleDuration` and `elapsed` have exactly the same value, which is not a very likely scenario though. Once `elapsed` is larger than `saleDuration` the subtraction will won't underflow since that is handled by `if (elapsed > saleDuration)`. Since this function is called internally, it wouldn't make much sense to change its visibility to `external`.

> “`InSecureumNFT` contract is”
>
> — 4 of 8

- [ ] A. Not susceptible to reentrancy given the absence of external contract calls
- [x] B. Not susceptible to integer overflow/wrapping given the compiler version used and the absence of `unchecked` blocks
- [x] C. Susceptible to reentrancy during minting
- [ ] D. Perfectly safe for production

▼ Solution

Correct is B, C. There are multiple external contract calls. The compiler version and absence of `unchecked` blocks should indeed prevent integer overflows/wrapping, but instead will cause reverts which could lead to Denial of Service. The fact that `mint()` keeps checking the current balance instead of the actual `msg.value` and that the `onERC721Received` hook is called before the balance is transferred to the beneficiary, can indeed be exploited using a reentrancy attack.

> “Assuming `InSecureumNFT` contract is deployed in production (i.e. live for users) on mainnet without any changes to shown code”
>
> — 5 of 8

- [x] A. Use of evident test configuration will cause fewer NFTs to be minted than expected in production
- [x] B. Illustrates the lack of best-practice for test parameterization to be removed or kept separate from production code
- [ ] C. It will behave as documented in code to mint the expected number of NFTs in production
- [ ] D. None of the above

▼ Solution

Correct is A, B. Multiple comments throughout the code show a discrepancy between the configuration expected in production and the actual configuration that is currently implemented. A much better way to do it, would be parameterization by setting these values during construction, which allows using the same code without changes for both mainnet and testnets.

> “The function `startSale()`”
>
> — 6 of 8

- [x] A. May be successfully called/executed by anyone
- [x] B. May be successfully called/executed with `_price` of 0
- [x] C. Must be called for minting to happen successfully
- [ ] D. None of the above

▼ Solution

Correct is A, B, C. While the require statement in `startSale()` states that only the deployer may call the function AND the price needs to be not zero, the actual code uses OR which allows anyone to start the sale as long as they specify a valid price. This also means that a price of 0 can be successful if the caller is the deployer. The `mint()` function requires `publicSale` to be true, which can only happen by calling `startSale()`.

> “The minting of NFTs”
>
> — 7 of 8

- [ ] A. Requires an exact amount of ETH to be paid by the buyer
- [x] B. Refunds excess ETH paid by buyer back to the buyer
- [x] C. Transfers the NFT `salePrice` to the `beneficiary` address
- [x] D. May be optimized to prevent any zero ETH transfers in its refund mechanism

▼ Solution

Correct is B, C, D. Thanks to the refund mechanism after the actual price has been determined, the buyer does not need to send an exact amount of ETH. After refunding the buyer, what is left in the contracts balance and sent to the benificiary should indeed be the `salePrice`. The refund mechanism can be optimized by skipping transfers when the current balance equals the price exactly.

> “The NFT sale”
>
> — 8 of 8

- [x] A. May be restarted by anyone any number of times
- [ ] B. Can be started exactly once by deployer
- [x] C. Is missing an additional check on publicSale
- [x] D. Is missing an event emit in startSale

▼ Solution

Correct is A, C, D. `startSale()` is not checking whether `publicSale` is already true, allowing `saleStartTime` to be reset and also overwriting the `price` and can indeed be called by anyone since the authentication can be bypassed by simply specifying a `_price` unequal to 0. The start of the sale would certainly be a good point to log an event, but events are currently completely missing from the contract.

In Blockchain Tags Ethereum, Secureum Bootcamp

← Damn Vulnerable DeFi V2 - #5 The R… Secureum Bootcamp Audit Findings 10…