Secureum Bootcamp Epoch∞ - October RACE #10

October 3, 2022 / patrickd

This is a write-up of the Secureum Bootcamp Race 9 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 below contract. This is the same contract you will see for all the 8 questions in this RACE. The question is below the shown contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;
import { Ownable } from "@openzeppelin/contracts/access/Ownab

contract TestContract is Ownable {

    function Test1(uint n) external pure returns(uint) {
        return n + abi.decode(msg.data[msg.data.length-64:],
    }
```

```
function Test2(uint n) public view returns(uint) {
    bytes memory fcall = abi.encodeCall(TestContract.Test1
    bytes memory xtr = abi.encodePacked(uint(4),uint(5));
    bytes memory all = bytes.concat(fcall,xtr);
    (bool success, bytes memory data) = address(this).sta
    return abi.decode(data,(uint));
}

type Nonce is uint256;
struct Book { Nonce nonce;}

function NextBookNonce(Book memory x) public pure {
   x.nonce = Nonce.wrap(Nonce.unwrap(x.nonce) + 3);
}

function Test3(uint n) public pure returns (uint) {
  Book memory bookIndex;
  bookIndex.nonce = Nonce.wrap(7);
  for (uint i=0;i<n;i++) {
     NextBookNonce(bookIndex);
  }
  return Nonce.unwrap(bookIndex.nonce);
}

error ZeroAddress();
error ZeroAmount();
uint constant ZeroAddressFlag = 1;
uint constant ZeroAmountFlag = 2;

function process(address[] memory a, uint[] memory amount
    uint error;
    uint total;
    for (uint i=0;i<a.length;i++) {
        if (a[i] == address(0)) error |= ZeroAddressFlag;
        if (amount[i] == 0) error |= ZeroAmountFlag;
        total += amount[i];
    }
    if (error == ZeroAddressFlag) revert ZeroAddress();
    if (error == ZeroAmountFlag)  revert ZeroAmount();
    return total;
}

function Test4(uint n) public pure returns (uint) {
    address[] memory a = new address[](n+1);
    for (uint i=0;i<=n;i++) {
        a[i] = address(uint160(i));
    }
    uint[] memory amount = new uint[](n+1);
    for (uint i=0;i<=n;i++) {
        amount[i] = i;
    }
    return process(a,amount);
}

uint public totalMinted;
uint constant maxMinted = 100;
event minted(uint totalMinted,uint currentMint);

modifier checkInvariants() {
    require(!paused, "Paused");
    _;
    invariantCheck();
    require(!paused, "Paused");
}

function invariantCheck() public {
    if (totalMinted > maxMinted) // this may never happen
        pause();
}

bool public paused;
function pause() public {
    paused = true;
}
function unpause() public onlyOwner {
    paused = false;
}

function Test5( uint n) public checkInvariants(){
    totalMinted += n;
```

```
            emit minted(n,totalMinted);
        }
    }
```

> “Which statements are true in *Test1()*?”
>
> — 1 of 8

- [x] A. The function does not use all supplied extra data
- [x] B. The function can revert due to an underflow
- [x] C. The function can revert due to an overflow
- [ ] D. The function accesses memory which it should not

▼ Solution

Correct is A, B, C

Answer A seems a bit confusing when looking at *Test1()* alone, but seeing the *xtr* variable of *Test2()* brings some clarity: The *Test1()* function signature expects one uint to be passed, but then within the function body it loads 64 bytes directly from calldata. *Test2()* then shows how the function is intended to be called by concatenating extra data to the ABI encoded calldata. It adds two more uint types which together are 64 bytes of extra data. But then in the *abi.decode* only the first uint from extra data is actually decoded and used.

Both B and C are true since a Solidity version (^0.8.0) is used, that automatically checks for integer over/underflows and reverts when these happen. In this specific case, an overflow could happen when parameter *n* or the number supplied from extra data are large enough to wrap. The underflow can happen when the overall supplied calldata is smaller than 64 bytes, making the subtraction within the slicing parameters fail.

You could argue that accessing *msg.data* directly should be avoided when possible. But this doesn't access memory but read-only calldata. Therefore no memory is accessed that should not be.

> “Which statements are true in *Test2()*?”

— 2 of 8

- [x] A. Result of *encodePacked* is deterministic
- [ ] B. *abi.decode* always succeeds
- [x] C. It calls function *Test1()* without any problem
- [ ] D. It should use *uint256* instead of *uint*

▼ Solution

Correct is A, C

Deterministic means that you should always get the same predictable output for a given input. As such, *encodePacked* always encodes passed data the same way.

Test2's *abi.decode* will only succeed if no error happens in *Test1()*. If *Test1()* reverts the returned data would not contain a decodable uint but error data. One way to cause this to happen would be supplying a number *n* that causes an overflow. The best practice is to check the *success* boolean before attempting to decode the returned data.

Answer D leaves some room for interpretation. *uint* is an alias of *uint256* and there should not be an issue using it here. But it's a common best practice to avoid the shorter alias and instead use the longer-named version of the type. While this is generally considered to improve readability, I'd argue that consistency (always using the same type) is more important.

> “Which statements are true in *NextBookNonce()*?”

— 3 of 8

- [ ] A. *wrap* and *unwrap* cost additional gas
- [x] B. It is safe to use *unchecked*
- [x] C. There is something suspicious about the increment value
- [ ] D. It could have used *x.nonce++*

▼ Solution

Correct is B, C

The calls to *wrap* and *unwrap* are basically telling Solidity whether it should treat a certain variable as being of a custom type (*Nonce*) or of its native type (*uint256*). This switch is basically just syntactic sugar for handling types within Solidity, the EVM will know nothing of these type switches and no additional gas will be used by doing so.

Using an *unchecked* block in this function would omit Solidity's over/underflow handling. Especially in the context of a Nonce (Number used only once), you don't want integer values to wrap and overflow to values that were once used before. But usually, a nonce is only increased by such a small value that exploiting this would be very expensive. In this specific case, the function is pure and the nonce is not stored, so whether it's safe to use unchecked block will depend on the function being used correctly.

Answer C sounds rather ominous but it's simply pointing out that Nonces are commonly increased by one and not by such a weird number as 3.

Arithmetic operations cannot be executed on custom types without unwrapping the number first.

> “Which statements are true in *Test3()*?”
>
> — 4 of 8

- [✓] A. *bookIndex.nonce* is incremented in the loop
- [ ] B. *bookIndex.nonce* cannot be incremented because *NextBookNonce* is pure
- [✓] C. *i*++ can be made *unchecked*
- [ ] D. *memory* can be changed to *storage* without any other changes

▼ Solution

Correct is A, C

Both A and B should be clear from reading the code.

The increment of *i* within the loop can indeed be made *unchecked* since it won't be able to overflow no matter what is supplied as *n*.

The *memory* location can't simply be changed to *storage* without various further changes such as assigning it to a specific storage slot before being able to make use of it.

> "Which statements are true In *Test4()*?"
>
> — 5 of 8

- [ ] A. The function always reverts with *ZeroAddress()*
- [ ] B. The function always reverts with *ZeroAmount()*
- [x] C. The function never reverts with *ZeroAddress()*
- [x] D. The function never reverts with *ZeroAmount()*

▼ Solution

Correct is C, D

The first array elements of both *a* and *amounts* will always be zero-like. Both *1* for *ZeroAddress* and *2* for *ZeroAmount* will be OR-combined resulting in 3. Once this value is set as an error, further iterations will not influence it. After the loop has finished, this error value is not checked for and instead, the function returns the *total* without reverting.

> "Which statements are true in *Test5()*?"
>
> — 6 of 8

- [ ] A. modifier *checkInvariants* will pause the contract if too much is minted
- [x] B. modifier *checkInvariants* will never pause the contract
- [ ] C. modifier *checkInvariants* will always pause the contract
- [x] D. There are more efficient ways to handle the *require*

▼ Solution

Correct is B, D

While the *checkInvariants* modifier does intend to pause the contract if too much is minted, it'll be unable to ever do so since this will be reverted by the second *require* call.

A single call to *require* would fix this issue and also be more efficient.

> “Which statements are true about the *owner*?”
>
> — 7 of 8

- ✓ A. The *owner* is initialized
- ☐ B. The *owner* is not initialized
- ☐ C. The *owner* cannot be changed
- ✓ D. The *owner* can be changed

▼ Solution

Correct is A, D

Although not visible here, the *owner* is indeed initialized by the constructor inherited from *Ownable*, which also comes with functions allowing to change the *owner* at a later point.

> “Which statements are true in *Test5()* and related functions?”
>
> — 8 of 8

- ✓ A. *pause* is unsafe
- ☐ B. *unpause* is unsafe
- ☐ C. The *emit* is done right
- ✓ D. The *emit* is done wrong

▼ Solution

Correct is A, D

The *pause* function is missing the *onlyOwner* modifier allowing anyone to arbitrarily pause the contract.

The *minted* event's parameters appear to be in the wrong order.

In Blockchain Tags Ethereum, Secureum Bootcamp

Secureum Bootcamp Epoch∞ - August …