Secureum Bootcamp Epoch∞ - August RACE #9

August 31, 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 following contracts. You will see them for all the 8 questions in this RACE. The question is below the shown contract.

```
pragma solidity 0.8.7;

import "@openzeppelin/contracts/access/Ownable.sol";

// Assume the Proxy contract was deployed and no further trans

contract Mastercopy is Ownable {
    int256 public counter = 0;

    function increase() public onlyOwner returns (int256) {
        return ++counter;
```

```
    }

    function decrease() public onlyOwner returns (int256) {
        return --counter;
    }

}

contract Proxy is Ownable {
    mapping(bytes4 => address) public implementations;

    constructor() {
        Mastercopy mastercopy = new Mastercopy();
        implementations[bytes4(keccak256(bytes("counter()")))]
        implementations[Mastercopy.increase.selector] = addres
        implementations[Mastercopy.increase.selector] = addres
    }

    fallback() external payable {
        address implementation = implementations[msg.sig];

        assembly {
            // Copied without changes to the logic from OpenZe
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), implementation,
            returndatacopy(0, 0, returndatasize())
            switch result
                case 0 {
                    revert(0, returndatasize())
                }
                default {
                    return(0, returndatasize())
                }
        }
    }

    function setImplementationForSelector(bytes4 signature, ac
        implementations[signature] = implementation;
    }
```

```
}
```

> “The function signature is the first 4 bytes of the keccak hash which”

— 1 of 8

- [x] A. Includes the function name
- [x] B. Includes a comma separated list of parameter types
- [ ] C. Includes a comma separated list of return value types
- [x] D. Is generated only for public and external functions

▼ Solution

Correct is A, B, D

A function's signature is created by hashing its name and a comma separated list (no spaces) of the types of all its parameters. For example: *add(uint256,uint256)*.

The fact that the return value type isn't part of the signature is basically given away by the fact that the creation of the *counter()* function's signature doesn't mention *int256*.

Since it's used for calling external and public functions of a contract, only these functions need a signature to be called by. Internal and private functions can only be directly JUMPed to within the bytecode of the contract that contains them.

> “The Proxy contract is most similar to a”

— 2 of 8

- [ ] A. UUPS Proxy
- [ ] B. Beacon Proxy
- [x] C. Transparent Proxy
- [ ] D. Metamorphic Proxy

▼ Solution

Correct is C

A UUPS (or Universal Upgradeable Proxy Standard) would have it's upgradeability logic within the implementation which ensures there won't be function signature clashes. This is not the case here with the *setImplementationForSelector()* function being part of the Proxy.

A Beacon Proxy would ask another contract where it can find the implementation, this isn't the case here since the implementation address is managed and stored in the Proxy contract itself.

This makes it most similar to a Transparent Proxy.

A "Metamorphic" Proxy isn't really a thing. Contracts referred to being metamorphic usually achieve upgradeability not thanks to a proxy, but due to the fact that they can be re-deployed to the same address via CREATE2.

> “Gas will be saved with the following changes”
>
> — 3 of 8

- [x] A. Skipping initialization of *counter* variable
- [ ] B. Making *increase()* function external to avoid copying from calldata to memory
- [ ] C. Packing multiple implementation addresses into the same storage slot
- [ ] D. Moving the calculation of the *counter()* function's signature hash to a constant

▼ Solution

Correct is A

Avoiding initialization of state variables to zero can indeed save gas and are usually not necessary when deploying contracts to fresh addresses where all state variables will be zero-initialized by default.

If initialization in the Mastercopy contract would attempt setting a value different from 0 it wouldn't even have any effect, since it's not setting this value in the Proxy's state - this would be considered a bug.

The *increase()* function does not have any function parameters that are being copied from calldata to memory. Introducing this change would have no effect.

Addresses are too large (20 bytes) for multiple of them to be packed into a single storage slot (32 bytes).

Constants are basically placeholders in the bytecode for expressions that are filled during compile time. It would not make a difference whether the compiler fills them or whether we've already "filled" them by hand. It might however improve readability to do so.

> “Calling the *increase()* function on the Proxy contract will”
>
> — 4 of 8

- [ ] A. Will revert since the Proxy contract has no *increase()* function
- [x] B. Will revert for any other caller than the one that deployed the Proxy
- [x] C. Increases the integer value in the Proxy's storage slot located at index 1
- [ ] D. Delegate-call to the zero-address

▼ Solution

Correct is B, C

When the Proxy is called with the function signature for *increase()*, Solidity will call the *fallback()* function since the Proxy contract itself does not have a function with a matching signature.

The *fallback()* function will determine that, for this signature, it has stored the mastercontract's address as an implementation and will delegate-call it.

The Mastercontract's code will be execute in the context of the Proxy contract, meaning that the state being manipulated by the Mastercontract's code is that of the Proxy.

The function-selection logic of the Mastercontract will find that it indeed has a matching function signature belonging to *increase()* and will execute it.

The *increase()* function will increment the value of the counter state variable by one, who's index is at 1 because the first index is already reserved by Ownable's owner state variable.

This means that whatever value is currently located at the Proxy contract's storage slot with index 1 will be increased by one even if there's no variable called counter in the Proxy itself.

> “Calling the *decrease()* function on the Proxy contract will”
>
> — 5 of 8

- [ ] A. Will revert because it was not correctly registered on the proxy
- [ ] B. Will succeed and return the value of *counter* after it was decreased
- [ ] C. Will succeed and return the value of *counter* before it was decreased
- [x] D. Will succeed and return nothing

▼ Solution

Correct is D

When checking for the implementation address of the decrease() function's signature, the Proxy contract won't find one since it wasn't registered in the constructor like the increase() function was.

But that doesn't mean it'll revert, it'll instead get the default state value: The zero address.

Since no check is made to prevent *calls* when no matching signature is found in the *implementations* mapping, a delegate-call will be made to the zero address, and like all calls that are made to addresses that do not have runtime bytecode, this call will succeed without returning anything.

The EVM implicitly assumes that all bytecode ends with the STOP opcode, even if the STOP opcode isn't explicitly mentioned in the bytecode itself. So to the EVM an empty bytecode actually always contains one opcode: STOP - the opcode for stopping execution without errors.

> “Due to a storage clash between the Proxy and the Mastercopy contracts”
>
> — 6 of 8

- [ ] A. The Proxy's *implementations* would be overwritten by 0 during initialization of the Mastercopy
- [ ] B. The Proxy's *implementations* would be overwritten when the counter variable changes
- [ ] C. The Proxy's *implementations* variable's storage slot being overwritten causes a DoS
- [x] D. None of the above

▼ Solution

Correct is D

Mappings leave their assigned storage slot unused. The actual values of a mapping are stored at location's determined by hashing the mapping slot's index with the key.

That means that, even though the Proxy's *implementations* and the Mastercopy's *counter* variables make use of the same slot, they actually do not interfere with each other and nothing will happen when *counter*'s value changes.

> “The Proxy contract”
>
> — 7 of 8

- [ ] A. Won't be able to receive any ether when *calldatasize* is 0 due to a missing *receive()*
- [x] B. Will be the owner of the Mastercopy contract
- [ ] C. Has a storage clash in slot 0 which will cause issues with the current mastercopy
- [ ] D. None of the above

▼ Solution

Correct is B

Thanks to its payable *fallback()* function it'll still be able to receive ether without issues.

Ownable always initializes the owner with the msg.sender. When the Proxy deploys the Mastercopy contract, the Proxy will be the msg.sender and therefore become the owner of the mastercopy contract.

Both the Proxy contract and the Mastercopy contract first inherit from Ownable ensuring that the storage slot at index 0 will be used in the same manner on both contracts preventing any issues.

> “The *fallback()* function’s assembly block”
>
> — 8 of 8

- [ ] A. Can be marked as *"memory-safe"* for gas optimizations
- [x] B. The result of the delegate-call overwrites the the call parameters in memory
- [ ] C. Interferes with the Slot-Hash calculation for the implementations-mapping by overwriting the scratch space
- [ ] D. None of the above

▼ Solution

Correct is B

The assembly block doesn't respect Solidity's memory management and can't be considered to be *"memory-safe"*. And even if it did, this Solidity version does not have the option to mark assembly blocks as such yet, this was introduced with version 0.8.13.

The use of the *CALLDATACOPY* opcode will copy the full *CALLDATASIZE* to memory starting at offset 0. Then, after the delegate-call was finished, the use of the *RETURNDATACOPY* opcode will copy the full *RETURNDATASIZE* to memory, also starting at offset 0. This effectively means that the output will overwrite the input of the delegate-call.

The slot-hash calculation has already finished when the assembly block begins, therefore there should not be any interference by overwriting the sratch space that was used for it.

In Blockchain Tags Ethereum, Secureum Bootcamp

Secureum A-MAZE-X Stanford CTF →