Secureum Bootcamp Solidity 201 Quiz

October 31, 2021 / patrickd

This is a writeup of the Secureum Bootcamp Solidity 201 Quiz containing solutions and references to the provided study material.
For fairness it was published after submissions to it were closed.

The quiz consisted of 32 questions with a strict timelimit of 16 minutes. The ordering of the questions was randomized, so the numbering here won't match with the numbering elsewhere.

> " OpenZeppelin SafeERC20 is generally considered safer to use than ERC20 because "

— 1 of 32

- ☐ A. It adds integer overflow/underflow checks
- ☑ B. It adds return value/data checks
- ☐ C. It adds pause/unpause capability
- ☐ D. It adds race-conditon checks

▼ Solution

Correct is B.

> OpenZeppelin SafeERC20: Wrappers around ERC20 operations that throw on failure when the token contract implementation returns false. Tokens that return no value and instead revert or throw on failure are also supported with non-reverting calls assumed to be successful. Adds safeTransfer,

> safeTransferFrom, safeApprove, safeDecreaseAllowance, and safeIncreaseAllowance.

from point 149 of Solidity 201 - by Secureum

> “ The OpenZeppelin library that provides `onlyOwner` modifier ”

— 2 of 32

- [✓] A. Is Ownable
- [ ] B. Provides role based access control
- [✓] C. Provides a function to renounce ownership
- [ ] D. None of the above

▼ Solution

Correct is A, C. See the source code on GitHub

> OpenZeppelin Ownable: provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with transferOwnership. This module is used through inheritance. It will make available the modifier onlyOwner, which can be applied to your functions to restrict their use to the owner.

from point 154 of Solidity 201 - by Secureum

> “ OpenZeppelin ECDSA ”

— 3 of 32

- [ ] A. Implements functions for signature creation & verification
- [ ] B. Is susceptible to signature malleability
- [ ] C. Both A & B
- [✓] D. Neither A nor B

▼ Solution

Correct is D. See the source code on GitHub

> OpenZeppelin ECDSA: provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via web3.eth.sign, and are a 65 byte array (of type bytes in Solidity) arranged the following way: `[[v (1)], [r (32)], [s (32)]]`. The data signer can be recovered with ECDSA.recover, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix `'\x19Ethereum Signed Message:\n'`, so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use toEthSignedMessageHash.

from point 166 of Solidity 201 - by Secureum

> Externally Owned Accounts (EOA) can sign messages with their associated private keys, but currently contracts cannot.

from point 168.1 of Solidity 201 - by Secureum

> " Which of the following is/are true about Solidity compiler 0.8.0? "

— 4 of 32

- [x] A. ABI coder v2 is made the default
- [ ] B. No opt-out primitives for default checked arithmetic
- [x] C. Failing to `assert` returns the gas left instead of consuming all gas
- [x] D. Exponentiation is made right associative

▼ Solution

Correct is A, C, D.

> ABI coder v2 is activated by default. You can choose to use the old behaviour using `pragma abicoder v1;`. The pragma `pragma experimental ABIEncoderV2;` is still valid, but it is deprecated and has no effect. If you want to be explicit, please use `pragma abicoder v2;` instead.

from Solidity v0.8.0 Breaking Semantic Changes, point 142.2 of Solidity 201 - by Secureum

> Arithmetic operations revert on underflow and overflow. You can use `unchecked` to use the previous wrapping behaviour.

from Solidity v0.8.0 Breaking Semantic Changes, point 142.1 of Solidity 201 - by Secureum

> Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the invalid opcode but instead the revert opcode. More specifically, they will use error data equal to a function call to Panic(uint256) with an error code specific to the circumstances. This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing require.

from Solidity v0.8.0 Breaking Semantic Changes, point 142.4 of Solidity 201 - by Secureum

> Exponentiation is right associative, i.e., the expression a**b**c is parsed as a**(b**c). Before 0.8.0, it was parsed as (a**b)**c. This is the common way to parse the exponentiation operator.

from Solidity v0.8.0 Breaking Semantic Changes, point 142.3 of Solidity 201 - by Secureum

> “ EVM memory ”

— 5 of 32

- [x] A. Is linear and byte-addressable
- [x] B. Is reserved by Solidity until 0x7f
- [ ] C. Can be accessed in bytes using MLOAD8/MSTORE8
- [ ] D. Is non-volatile or persistent

▼ Solution

Correct is A, B.

> EVM Memory: EVM memory is linear and can be addressed at byte level and accessed with MSTORE/MSTORE8/MLOAD instructions. All locations in memory are initialized as zero.

from point 125 of Solidity 201 - by Secureum

> Reserved Memory: Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) [...]: `0x60 - 0x7f (32 bytes)`: zero slot (The zero slot is used as initial value for dynamic memory arrays and should never be written to)

from point 127 of Solidity 201 - by Secureum

> “ Dappsys provides ”

— 6 of 32

- [x] A. A proxy implementation
- [ ] B. A floating-point implementation with wad & ray
- [x] C. A flexible authorization implementation
- [ ] D. All of the above

▼ Solution

Correct is A, C.

> Dappsys DSProxy: implements a proxy deployed as a standalone contract which can then be used by the owner to execute code.

from point 193 of Solidity 201 - by Secureum

> Dappsys DSMath: provides arithmetic functions for the common numerical primitive types of Solidity. You can safely add, subtract, multiply, and divide uint numbers without fear of integer overflow. You can also find the minimum and maximum of two numbers. Additionally, this package provides arithmetic functions for two new higher level numerical concepts called wad (18 decimals) and ray (27 decimals). These are used to represent **fixed-point decimal numbers**. A wad is a decimal number with 18 digits of precision and a ray is a decimal number with 27 digits of precision.

from point 194 of Solidity 201 - by Secureum

> Dappsys DSAuth: Provides a flexible and updatable auth pattern which is completely separate from application logic.

from point 195 of Solidity 201 - by Secureum

> “ Libraries are contracts ”

— 7 of 32

- [✓ A. That cannot have state variables]
- [✓ B. That cannot be inherited]
- C. That always require a delegatecall
- [✓ D. That are not meant to receive Ether]

▼ Solution

Correct is A, B, D.

> Libraries: They are deployed only once at a specific address and their code is reused using the DELEGATECALL opcode. This means that if library functions are called, their code is executed in the context of the calling contract. They use the library keyword.

from point 103.3 of Solidity 201 - by Secureum

> Library Restrictions: In comparison to contracts, libraries are restricted in the following ways: They cannot have state variables, they cannot inherit nor be inherited, they cannot receive Ether.

from point 113 of Solidity 201 - by Secureum

> Library functions can only be called directly (i.e. without the use of DELEGATECALL) if they do not modify the state (i.e. if they are view or pure functions), because libraries are assumed to be stateless

from point 113.6 of Solidity 201 - by Secureum

> “ ERC20 `transferFrom(address sender, address recipient, uint256 amount)` (that follows the ERC20 spec strictly) ”
>
> — 8 of 32

- ☑ A. Transfers token amount from `sender` to `recipient`
- ☑ B. `sender` must have given caller (`msg.sender`) approval for at least `amount` or more
- ☐ C. Deducts amount from `sender`'s allowance
- ☑ D. Deducts amount from caller's (`msg.senders`'s) allowance

▼ Solution

Correct is A, B, D.

> Moves amount tokens from sender to recipient using the allowance mechanism. amount is then deducted from the caller’s allowance. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.

from point 73 of Solidity 201 - by Secureum

> “ ERC777 may be considered as an improved version of ERC20 because ”
>
> — 9 of 32

- ☑ A. Hooks allow reacting to token mint/burn/transfer
- ☑ B. It can help avoid separate `approve` and `transferFrom` transactions
- ☑ C. It can help prevent tokens getting stuck in contracts
- ☐ D. It removes reentrancy risk

▼ Solution

Correct is A, B, C.

> OpenZeppelin ERC777: Like ERC20, ERC777 is a standard for fungible tokens with improvements such as getting rid of the confusion around decimals,

> minting and burning with proper events, among others, but its killer feature is receive hooks. [...] A hook is simply a function in a contract that is called when tokens are sent to it, meaning accounts and contracts can react to receiving tokens. This enables a lot of interesting use cases, including atomic purchases using tokens (no need to do approve and transferFrom in two separate transactions), rejecting reception of tokens (by reverting on the hook call), redirecting the received tokens to other addresses, among many others. Furthermore, since contracts are required to implement these hooks in order to receive tokens, no tokens can get stuck in a contract that is unaware of the ERC777 protocol, as has happened countless times when using ERC20s.

from point 152 of Solidity 201 - by Secureum

> “ WETH is ”

— 10 of 32

- ☐ A. An ERC20 pre-compile for Wrapped Ether built into Ethereum protocol
- ☐ B. Warp Ether for super-fast Ether transfers
- ☐ C. Wrapped Ether to convert Ether into an ERC721 NFT
- ☑ D. None of the above

▼ Solution

Correct is D.

> WETH: WETH stands for Wrapped Ether. For protocols that work with ERC-20 tokens but also need to handle Ether, WETH contracts allow converting Ether to its ERC-20 equivalent WETH (called wrapping) and vice-versa (called unwrapping). WETH can be created by sending ether to a WETH smart contract where the Ether is stored and in turn receiving the WETH ERC-20 token at a 1:1 ratio. This WETH can be sent back to the same smart contract to be “unwrapped” i.e. redeemed back for the original Ether at a 1:1 ratio. The most widely used WETH contract is WETH9 which holds more than 7 million Ether for now.

from point 198 of Solidity 201 - by Secureum

> “ OpenZeppelin SafeCast ”

— 11 of 32

- ☐ A. Prevents underflows while downcasting
- ☑ B. Prevents overflows while downcasting
- ☐ C. Prevents underflows while upcasting
- ☐ D. Prevents overflows while upcasting

▼ Solution

Correct is B.

> OpenZeppelin SafeCast: Wrappers over Solidity's uintXX/intXX casting operators with added overflow checks. Downcasting from uint256/int256 in Solidity does not revert on overflow. This can easily result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. `SafeCast` restores this intuition by reverting the transaction when such an operation overflows.

from point 177 of Solidity 201 - by Secureum

> “ OpenZeppelin ERC20Pausable ”

— 12 of 32

- ☑ A. Adds ability to pause token transfers
- ☑ B. Adds ability to pause token minting and burning
- ☐ C. Provides modifiers `whenPaused` and `whenNotPaused`
- ☐ D. None of the above

▼ Solution

Correct is A, B. Not C, because it inherits these modifiers from Pausable and doesn't implement them

> OpenZeppelin ERC20Pausable: ERC20 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the

> end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug.

from point 148 Extensions of Solidity 201 - by Secureum

> OpenZeppelin Pausable: provides an emergency stop mechanism using functions pause and unpause that can be triggered by an authorized account. This module is used through inheritance. It will make available the modifiers whenNotPaused and whenPaused, which can be applied to the functions of your contract. Only the functions using the modifiers will be affected when the contract is paused or unpaused.

from point 156 of Solidity 201 - by Secureum

> “ CREATE2 ”

— 13 of 32

- ☐ A. Deploys two contracts proxy and implementation concurrently
- ☑ B. Deploys contract at an address that can be predetermined
- ☑ C. Uses a salt and contract `creationCode`
- ☐ D. None of the above

▼ Solution

Correct is B, C.

> OpenZeppelin Create2: makes usage of the CREATE2 EVM opcode easier and safer. CREATE2 can be used to compute in advance the address where a smart contract will be deployed [...].

from point 163 of Solidity 201 - by Secureum

> `deploy(uint256 amount, bytes32 salt, bytes bytecode) → address`: Deploys a contract using CREATE2.

from point 163.1 of Solidity 201 - by Secureum

> “ Name collision error with inheritance happens when the following pairs have the same name within a contract ”

— 14 of 32

- [✓ A. Function & modifier]
- [✓ B. Function & event]
- [✓ C. Function & function]
- D. Event & modifier

▼ Solution

Correct is A, B, D.

> Name Collision Error: It is an error when any of the following pairs in a contract have the same name due to inheritance: 1) a function and a modifier 2) a function and an event 3) an event and a modifier.

from point 112 of Solidity 201 - by Secureum

> “ OpenZeppelin's (role-based) AccessControl library ”

— 15 of 32

- A. Provides support only for two specific: Owner and User
- [✓ B. Provides support for different roles with different authorization levels]
- [✓ C. Provides support for granting and revoking roles]
- D. None of the above

▼ Solution

Correct is B, C.

> OpenZeppelin AccessControl: provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts. Roles can be used to represent a set of permissions. hasRole is used to restrict access to a function call. Roles can be granted and revoked

> dynamically via the grantRole and revokeRole functions which can only be called by the role's associated admin accounts.

from point 155 of Solidity 201 - by Secureum

> “ OpenZeppelin's proxy implementations ”

— 16 of 32

- [x] A. Typically have a proxy contract and an implementation contract
- [x] B. Use delegatecalls from proxy to implementation
- [ ] C. Cannot support upgradable proxies
- [ ] D. None of the above

▼ Solution

Correct is A, B.

> OpenZeppelin Proxy: This abstract contract provides a fallback function that delegates all calls to another contract using the EVM instruction delegatecall. We refer to the second contract as the implementation behind the proxy, and it has to be specified by overriding the virtual _implementation function.

from point 185 of Solidity 201 - by Secureum

> OpenZeppelin ERC1967Proxy: implements an upgradeable proxy. It is upgradeable because calls are delegated to an implementation address that can be changed.

from point 186 of Solidity 201 - by Secureum

> “ Which of the following is/are true for a function `f` that has a modifier `m`? ”

— 17 of 32

- ☐ A. Function `f` cannot have another modifier because every function function can have at most one modifier
- ☑ B. Function `f`'s code is inlined at the point of '_' within modifer `m`
- ☐ C. Function `f` reverts if '_' is not executed in the modifier `m`
- ☐ D. None of the above

▼ Solution

Correct is B.

> Function Modifiers: They can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function's control flow continues after the "_" in the preceding modifier. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The _ symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

from point 22 of Solidity 101 - by Secureum

> " Zero address check is typically recommended because "

— 18 of 32

- ☐ A. The use of zero address for transfers will trigger an EVM exception
- ☑ B. Ether/tokens sent to zero address will be inaccessible
- ☐ C. Ether/tokens sent to zero address can be accessed by anyone
- ☐ D. Address 0 is the Ethereum Masternode account and is forbidden for access

▼ Solution

Correct is B.

> Zero Address Check: address(0) which is 20-bytes of 0's is treated specially in Solidity contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign

> transactions). Therefore zero addresses should be used with care and checks should be implemented for user-supplied address parameters.

from point 144 of Solidity 201 - by Secureum

> “ Solidity supports ”

— 19 of 32

- [x] A. Multiple inheritance
- [x] B. Polymorphism
- [ ] C. Contract overloading
- [x] D. Function overloading

▼ Solution

Correct is A, B, D. No such things as C.

> Solidity supports multiple inheritance including polymorphism

from point 102 of Solidity 201 - by Secureum

> Function Overloading: A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading.”

from point 25 of Solidity 101 - by Secureum

> “ OpenZeppelin ERC721 ”

— 20 of 32

- [x] A. Implements the NFT standard
- [x] B. `safeTransferFrom(..)` checks for zero-addresses
- [ ] C. `approve(..)` is susceptible to race-condition just like ERC20
- [x] D. `setApprovalForAll(address operator, bool _approved)` approves/removes `operator` for all of caller's tokens

▼ Solution

Correct is A, B, D. Not C, since approval is not susceptible since approval can only given or taken away for a single token, so changing approval doesn't allow stealing more than was already approved.

> OpenZeppelin ERC721: Implements the popular ERC721 Non-Fungible Token Standard.

from point 151 of Solidity 201 - by Secureum

> `safeTransferFrom(..)`: Safely transfers tokenId token from from to to, checking first that contract recipients are aware of the ERC721 protocol to prevent tokens from being forever locked. Requirements: 1) from cannot be the zero address [...]

from point 151.4 of Solidity 201 - by Secureum

> `setApprovalForAll(address operator, bool _approved)`: Approve or remove operator as an operator for the caller. Operators can call transferFrom or safeTransferFrom for any token owned by the caller.

from point 151.7 of Solidity 201 - by Secureum

> " For `contract A {uint256 i; bool b1; bool b2; address a1;}` the number of storage slots used is: "

— 21 of 32

- ☐ A. 4
- ☐ B. 3
- ☑ C. 2
- ☐ D. 1

▼ Solution

Correct is C. The uint256 takes a full slot, the bools (each 1 byte) and the address (20 bytes) can packed into the same slot

> Storage Layout: State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0

from point 115 of Solidity 201 - by Secureum

> Storage Layout Packing: For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules: [...] If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot

from point 116 of Solidity 201 - by Secureum

> “ Which of the following is/are generally true about storage layouts? ”
>
> — 22 of 32

- [x] A. The number of slots used for a contract depends on the ordering of state variable declarations
- [x] B. The slots for struct elements are consecutive
- [x] C. The slot `s` for dynamic array contains the length with individual elements stored consecutively in slots starting at `keccak256(s)`
- [ ] D. The slot `s` for mapping is empty with individual values stored consecutively in slots starting at `keccak(h(k).s)` where `k` is the first key and `h` is a hash function that depends on type of `k`

▼ Solution

Correct is A, B, C. For mappings the slots are unique for each key, they're not consecutive.

> Storage Layout & Ordering: Ordering of storage variables and struct members affects how they can be packed tightly. For example, declaring your storage variables in the order of uint128, uint128, uint256 instead of uint128, uint256, uint128, as the former will only take up two slots of storage whereas the latter will take up three.

from point 120 of Solidity 201 - by Secureum

> Storage Layout & Structs/Arrays: [...] The elements of structs and arrays are stored after each other, just as if they were given as individual values.

from point 117 of Solidity 201 - by Secureum

> Storage Layout for Dynamic Arrays: If the storage location of the array ends up being a slot p after applying the storage layout rules, this slot stores the number of elements in the array (byte arrays and strings are an exception). Array data is located starting at keccak256(p) and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes.

from point 122 of Solidity 201 - by Secureum

> Storage Layout for Mappings: For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations. The value corresponding to a mapping key k is located at keccak256(h(k) . p) where . is concatenation and h is a function that is applied to the key depending on its type [...]

from point 123 of Solidity 201 - by Secureum

> " Assuming all contracts C1, C2 and C3 define explicit constructors in `contract C1 is C2, C3` and both C2 and C3 don't inherit contracts, the number & order of constructor(s) executed is/are "

— 23 of 32

- ☐ A. One, that of C1
- ☑ B. Three, in the order C2, C3, C1
- ☐ C. One, that of C3
- ☐ D. Three, in the order C1, C2, C3

▼ Solution

Correct is B.

> Base Constructors: The constructors of all the base contracts will be called following the linearization rules.

from point 111 of Solidity 201 - by Secureum

> Storage Layout & Inheritance: For contracts that use inheritance, **the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract.** If allowed by the above rules, state variables from different contracts do share the same storage slot.

from point 118 of Solidity 201 - by Secureum

> “ OpenZeppelin SafeMath ”

— 24 of 32

- ☐ A. Prevents integer overflow/underflow at compile-time
- ☑ B. Is not required if using Solidity compiler version >= 0.8.0
- ☐ C. Both A & B
- ☐ D. Neither A nor B

▼ Solution

Correct is B. Not A, because it does not prevent them at compile- but at runtime. It can be argued it's not B since it's a recommendation and not a requirement.

> OpenZeppelin SafeMath: provides mathematical functions that protect your contract from overflows and underflows.

from point 175 of Solidity 201 - by Secureum

> Overflow/Underflow Check: Until Solidity version 0.8.0 which introduced checked arithmetic by default, arithmetic was unchecked and therefore susceptible to overflows and underflows which could lead to critical vulnerabilities. The recommended best-practice for such contracts is to use OpenZeppelin’s SafeMath library for arithmetic.

from point 146 of Solidity 201 - by Secureum

> “ Which of the following is/are not allowed? ”

— 25 of 32

- ☐ A. Function overriding
- ☐ B. Function overloading
- ☑ C. Modifier overloading
- ☐ D. Modifier overriding

▼ Solution

Correct is C.

> Function Overriding: Base functions can be overridden by inheriting contracts to change their behavior if they are marked as virtual. The overriding function must then use the override keyword in the function header.

from point 102.3 of Solidity 201 - by Secureum

> Function Overloading: A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading.”

from point 25 of Solidity 101 - by Secureum

> Modifier Overriding: Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers).

from point 110 of Solidity 201 - by Secureum

> “ Which of the following is/are true about abstract contracts and interfaces? ”

— 26 of 32

- ☑ A. Abstract contracts have at least one function undefined
- ☐ B. Interfaces can have some functions defined

- [✓] C. Unimplemented functions in abstract contracts need to be declared virtual
- [✓] D. All functions are implicitly virtual in interfaces

▼ Solution

Correct is A, C, D. Note that A isn't necessarily correct since abstract classes can have all functions defined.

> Abstract Contracts: Contracts need to be marked as abstract when at least one of their functions is not implemented. They use the abstract keyword.

from point 103.1 of Solidity 201 - by Secureum

> Interfaces: They cannot have any functions implemented.

from point 103.2 of Solidity 201 - by Secureum

> Virtual Functions: Functions without implementation have to be marked virtual outside of interfaces. In interfaces, all functions are automatically considered virtual. Functions with private visibility cannot be virtual.

from point 108 of Solidity 201 - by Secureum

> “ Storage layout ”

— 27 of 32

- [✓] A. Refers to the layout of state variables in storage
- [ ] B. Is organized in 256-byte slots
- [✓] C. Is packed for value types that use less than 32 bytes
- [✓] D. Always starts on a new slot for reference types

▼ Solution

Correct is A, C, D. Not B, because it's 256-bit slots, not Byte.

> EVM Storage: Storage is a key-value store that maps 256-bit words to 256-bit words and is accessed with EVM’s SSTORE/SLOAD instructions. All locations in storage are initialized as zero.

from point 114 of Solidity 201 - by Secureum

> Storage Layout Packing: For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules: [...] Value types use only as many bytes as are necessary to store them

from point 116 of Solidity 201 - by Secureum

> " Which of the following EVM instruction(s) do(es) not touch EVM storage? "
>
> — 28 of 32

- ☐ A. SLOAD
- ☑ B. MSTORE8
- ☐ C. SSTORE
- ☑ D. SWAP

▼ Solution

Correct is B, D.

> EVM Storage: Storage is a key-value store that maps 256-bit words to 256-bit words and is accessed with EVM's SSTORE/SLOAD instructions. All locations in storage are initialized as zero.

from point 114 of Solidity 201 - by Secureum

> EVM Memory: EVM memory is linear and can be addressed at byte level and accessed with MSTORE/MSTORE8/MLOAD instructions. All locations in memory are initialized as zero.

from point 125 of Solidity 201 - by Secureum

> Stack is made up of 1024 256-bit elements. EVM instructions can operate with the top 16 stack elements. Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. PUSH, POP, SWAP, DUP etc.

from point 60 of Ethereum 101 - by Secureum

> “ Proxied contracts ”

— 29 of 32

- [ ] A. Should use constructors in implementation contract to initialize the proxy's state variables
- [✓] B. Should use an external/public `initialize()` function
- [✓] C. Should have their `initialize()` function called only once
- [ ] D. All of the above

▼ Solution

Correct is B, C.

> OpenZeppelin Initializable: aids in writing upgradeable contracts, or any kind of contract that will be deployed behind a proxy. Since a proxied contract cannot have a constructor, it is common to move constructor logic to an external initializer function, usually called initialize. It then becomes necessary to protect this initializer function so it can only be called once. The initializer modifier provided by this contract will have this effect.
>
> To avoid leaving the proxy in an uninitialized state, the initializer function should be called as early as possible by providing the encoded function call as the _data argument. When used with inheritance, manual care must be taken to not invoke a parent initializer twice, or to ensure that all initializers are idempotent. This is not verified automatically as constructors are by Solidity.

from point 192 of Solidity 201 - by Secureum

> “ OpenZeppelin's ReentrancyGuard library mitigates reentrancy risk in a contract ”

— 30 of 32

- ☐ A. For all its functions by simply deriving/inheriting from it
- ☑ B. Only for functions that apply the `nonReentrant` modifier
- ☐ C. By enforcing a checks-effects-interactions pattern in its functions
- ☐ D. None of the above

▼ Solution

Correct is B.

> OpenZeppelin ReentrancyGuard: prevents reentrant calls to a function. Inheriting from ReentrancyGuard will make the nonReentrant modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

from point 73 of Solidity 201 - by Secureum

> “ EVM inline assembly has ”

— 31 of 32

- ☑ A. Its own language Yul
- ☐ B. Safety checks just like Solidity
- ☑ C. Access to all variables in the contract and function where present
- ☑ D. References to variables as their addresses not values

▼ Solution

Correct is A, C, D.

> Inline Assembly: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it. The language used for inline assembly in Solidity is called Yul.

from point 132 of Solidity 201 - by Secureum

> Inline Assembly Access to External Variables, Functions and Libraries: You can access Solidity variables and other identifiers by using their name. Local variables of value type are directly usable in inline assembly. Local variables

> that refer to memory/calldata evaluate to the address of the variable in memory/calldata and not the value itself [...]

from point 133 of Solidity 201 - by Secureum

> “ If OpenZeppelin's `isContract(address)` returns false for an address then ”
>
> — 32 of 32

- [ ] A. Address is guaranteed to not be a contract
- [x] B. Codesize at address is 0 at time of invocation
- [ ] C. Both A & B
- [ ] D. Neither A nor B

▼ Solution

Correct is B.

> Returns true if account is a contract. It is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract. Among others, isContract will return false for the following types of addresses: 1) an externally-owned account 2) a contract in construction 3) an address where a contract will be created 4) an address where a contract lived, but was destroyed

from point 159.1 of Solidity 201 - by Secureum

In Blockchain Tags Ethereum, Secureum Bootcamp

← Secureum Bootcamp Security Pitfal… Secureum Bootcamp Solidity 101 Quiz …