Secureum Bootcamp Security Pitfalls & Best Practices 101 Quiz

November 8, 2021 / patrickd

This is a writeup of the Secureum Bootcamp Security Pitfalls & Best Practices 101 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 16 questions with an overall strict timelimit of 16 minutes. ***No syntax highlighting 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.*

> "The use of pragma in the given contract snippet"
>
> — 1 of 16

```
pragma solidity ^0.6.0;

contract test {
   // Assume other required functionality is correctly imple
   // Assume this contract can work correctly without modifi
}
```

- [ ] A. Is incorrect and will cause a compilation error
- [x] B. Allows testing with 0.6.11 but accidental deployment with buggy 0.6.5
- [x] C. Is illustrative of risks from using a much older Solidity version (assume current version is 0.8.9)
- [ ] D. None of the above

▼ Solution

Correct is B, C.

> Unlocked pragma: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in pragma solidity 0.5.10) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs.

from point 2 of Security Pitfalls & Best Practices 101 - by Secureum

> There were bugs that were fixed in versions between 0.6.5 and 0.6.11 which means those fixes were absent in 0.6.5. Choice B was about these aspects.

from Rajeev on Secureum Discord Server

> Illustrative means "serving as an example or explanation." So the use of ^0.6.0 when the latest available version is 0.8.9 (as mentioned in choice C) is an example of using a much older compiler version when newer versions with bug fixes and more security features e.g. built-in overflow checks in ^0.8.0 are available.

from Rajeev on Secureum Discord Server

> “The given contract snippet has”

— 2 of 16

```
pragma solidity 0.8.4;

contract test {
```

```
    // Assume other required functionality is correctly implem

    function kill() public {
       selfdestruct(payable(0x0));
    }
}
```

- [x] A. Unprotected call to selfdestruct allowing anyone to destroy this contract
- [x] B. Dangerous use of zero address leading to burning of contract balance
- [ ] C. A compiler error because of the use of the kill reserved keyword
- [ ] D. None of the above

▼ Solution

Correct is A, B.

> Unprotected call to selfdestruct: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions.

from point 6 of Security Pitfalls & Best Practices 101 - by Secureum

> 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

> Reserved Keywords: These keywords are reserved in Solidity. They might become part of the syntax in the future: after, alias, apply, auto, case, copyof, default, define, final, immutable, implements, in, inline, let, macro, match, mutable, null, of, partial, promise, reference, relocatable, sealed, sizeof, static, supports, switch, typedef, typeof, unchecked

from point 131 of Solidity 201 - by Secureum

> “The given contract snippet has”
>
> — 3 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    modifier onlyAdmin() {
        // Assume this is correctly implemented
        _;
    }

    function transferFunds(address payable recipient, uint am
        recipient.transfer(amount);
   }
}
```

- [x] A. Missing use of onlyAdmin modifier on transferFunds
- [ ] B. Missing return value check on transfer
- [x] C. Unprotected withdrawal of funds
- [ ] D. None of the above

▼ Solution

Correct is A, C.

> Incorrect access control: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic.

from point 4 of Security Pitfalls & Best Practices 101 - by Secureum

> Unprotected withdraw function: Unprotected (external/public) function calls sending Ether/tokens to user-controlled addresses may allow users to withdraw unauthorized funds.

from point 5 of Security Pitfalls & Best Practices 101 - by Secureum

> `transferFunds()` clearly lets anyone withdraw any amount to any address. The only hint in the Q is the `onlyAdmin` modifier. While some other access control may also have been acceptable, the focus is on the code snippet provided and hence (A).

from Rajeev on Secureum Discord Server

> transfer (unlike send) does not return a success/failure return value. It reverts on failure. So there is nothing to be checked. Note that ERC20's transfer() returns a boolean which should be checked

from Rajeev on Secureum Discord Server

> “In the given contract snippet”
>
> — 4 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    mapping (uint256 => address) addresses;
    bool check;

    modifier onlyIf() {
        if (check) {
            _;
```

```
        }
    }

    function setAddress(uint id, address addr) public {
        addresses[id] = addr;
    }

    function getAddress(uint id) public onlyIf returns (addre
        return addresses[id];
    }
}
```

- [x] A. getAddress returns the expected addresses if check is true
- [x] B. getAddress returns zero address if check is false
- [ ] C. getAddress reverts if check is false
- [ ] D. None of the above

▼ Solution

Correct is A, B. If you have any doubts, give it a try in Remix!

> Incorrect modifier: If a modifier does not execute _ or revert, the function using that modifier will return the default value causing unexpected behavior.

from point 8 of Security Pitfalls & Best Practices 101 - by Secureum

> 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

> “The security concern(s) in the given contract snippet is/are”
>
> — 5 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    modifier onlyAdmin {
     // Assume this is correctly implemented
        _;
    }

    function delegate (address addr) external {  addr.delegat
    }
}
```

- [x] A. Potential controlled delegatecall risk
- [x] B. delegatecall return value is not checked
- [x] C. delegate() may be missing onlyAdmin modifier
- [x] D. delegate() does not check for contract existence at addr

▼ Solution

Correct is A, B, C, D.

> Controlled delegatecall: delegatecall() or callcode() to an address controlled by the user allows execution of malicious contracts in the context of the caller’s state. Ensure trusted destination addresses for such calls.

from point 12 of Security Pitfalls & Best Practices 101 - by Secureum

> Return values of low-level calls: Ensure that return values of low-level calls (call/callcode/delegatecall/send/etc.) are checked to avoid unexpected failures.

from point 37 of Security Pitfalls & Best Practices 101 - by Secureum

> Incorrect access control: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic.

from point 4 of Security Pitfalls & Best Practices 101 - by Secureum

> Account existence check for low-level calls: Low-level calls call/delegatecall/staticcall return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed.

from point 38 of Security Pitfalls & Best Practices 101 - by Secureum

> “The vulnerability/vulnerabilities present in the given contract snippet is/are”
>
> — 6 of 16

```
pragma solidity 0.7.0;
import "@openzeppelin/contracts/security/ReentrancyGuard.sol'
// which works with 0.7.0

contract test {

 // Assume other required functionality is correctly implemer
 // For e.g. users have deposited balances in the contract
 // Assume nonReentrant modifier is always applied

    mapping (address => uint256) balances;

    function withdraw(uint256 amount) external nonReentrant {
        msg.sender.call{value: amount}("");
        balances[msg.sender] -= amount;
    }
}
```

- [ ] A. Reentrancy
- [x] B. Integer underflow leading to wrapping
- [x] C. Missing check on user balance in withdraw()
- [ ] D. All of the above

▼ Solution

Correct is B, C. The code in this question was unintentionally missing inheritance from the ReentrancyGuard Contract. While there's a lot of discussion about the correct meaning of the term "underflow", this is how it is used in the Solidity Documentation and other related literature.

> Reentrancy vulnerabilities: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards.

from point 13 of Security Pitfalls & Best Practices 101 - by Secureum

> Integer overflow/underflow: Not using OpenZeppelin's SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. Solc v0.8.0 introduced default overflow/underflow checks for all arithmetic operations.

from point 19 of Security Pitfalls & Best Practices 101 - by Secureum

> I hope that the comment `// Assume nonReentrant modifier is always applied` implied that the intent was to apply the modifier "correctly" (i.e. with successful compilation 😅), which further implies reentrancy risk mitigation i.e. A is not a correct choice. Also, if A were to also be correct then that would again result in the "All of the above" ambiguity (A+B+C or D or A+B+C+D) which I have consciously tried to avoid. If that confused folks, apologies — sorry about that. I've tried to compile all snippets (in Remix) to avoid such silly mistakes, but not sure how this one slipped through.

from Rajeev on Secureum Discord Server

> “The security concern(s) in the given contract snippet is/are”
>
> — 7 of 16

```
pragma solidity 0.8.4;

contract test {

 // Assume other required functionality is correctly implemer

    uint256 private constant secret = 123;

    function diceRoll() external view returns (uint256) {
        return (((block.timestamp * secret) % 6) + 1);
    }
}
```

- [ ] A. diceRoll() visibility should be public instead of external
- [x] B. The private variable secret is not really hidden from users
- [x] C. block.timestamp is an insecure source of randomness
- [ ] D. Integer overflow

▼ Solution

Correct is B, C.

> Private on-chain data: Marking variables private does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain.

from point 16 of Security Pitfalls & Best Practices 101 - by Secureum

> Weak PRNG: PRNG relying on block.timestamp, now or blockhash can be influenced by miners to some extent and should be avoided.

from point 17 of Security Pitfalls & Best Practices 101 - by Secureum

> Making it public in this case should not affect gas given that there are no function arguments to copy over (if there were parameters/arguments, making it public would increase gas). Even otherwise, making it public from external should not affect the attack surface of the contract because it will only further allow (trusted) contract functions to call it.

from Rajeev on Secureum Discord Server

> E.: The logic of diceRoll() is broken as it returns only 1 or 4 🙂

from lukasz.glen on Secureum Discord Server

> “The security concern(s) in the given contract snippet is/are”
>
> — 8 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple
    // Contract admin set to deployer in constructor (not sho
    address admin;

    modifier onlyAdmin {
        require(tx.origin == admin);
        _;
    }

    function emergencyWithdraw() external payable onlyAdmin {
        msg.sender.transfer(address(this).balance);
    }
}
```

- [ ] A. Incorrect use of transfer() instead of using send()
- [x] B. Potential man-in-the-middle attack on admin address authentication
- [ ] C. Assumption on contract balance might cause a revert
- [x] D. Missing event for critical emergencyWithdraw() function

▼ Solution

Correct is B, D. Neither transfer nor send are recommended anymore.

> Avoid transfer()/send() as reentrancy mitigations: Although transfer() and send() have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use call() instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection.

from point 15 of Security Pitfalls & Best Practices 101 - by Secureum

> Dangerous usage of tx.origin: Use of tx.origin for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use msg.sender instead.

from point 30 of Security Pitfalls & Best Practices 101 - by Secureum

> [Regarding C.:] 0 transfers should not revert 😅. Even if they did, in this context, it wouldn't be considered a "security" concern because there would be nothing to withdraw and so a revert wouldn't be a concern w.r.t. any locked funds as such.

from Rajeev on Secureum Discord Server

> Missing events: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain.

from point 45 of Security Pitfalls & Best Practices 101 - by Secureum

> “The given contract snippet is vulnerable because of”

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    uint256 private constant MAX_FUND_RAISE = 100 ether;
    mapping (address => uint256) contributions;

    function contribute() external payable {
        require(address(this).balance != MAX_FUND_RAISE);
        contributions[msg.sender] += msg.value;
    }
}
```

- [ ] A. Integer overflow leading to wrapping
- [ ] B. Overly permissive function visibility of contribute()
- [ ] C. Incorrect use of msg.sender
- [x] D. Use of strict equality (!=) may break the MAX_FUND_RAISE constraint

▼ Solution

Correct is D. Visibility of external or public is required for a function to be payable. This use of message sender is very common and correct.

> Dangerous strict equalities: Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using >= or <= instead of == for such variables depending on the contract logic.

from point 28 of Security Pitfalls & Best Practices 101 - by Secureum

> Unexpected Ether and this.balance: A contract can receive Ether via payable functions, selfdestruct(), coinbase transaction or pre-sent before creation. Contract logic depending on this.balance can therefore be manipulated.

from point 26 of Security Pitfalls & Best Practices 101 - by Secureum

> Given the compiler version, even if there is an attempted integer overflow at runtime, it will revert before overflowing (because of inbuilt checks) with an exception but will not wrap. So A is not a vulnerability. While this is true in general, this snippet cannot be overflowed because of its dependence on msg.value.

from Rajeev on Secureum Discord Server

> “In the given contract snippet, the require check will”
>
> — 10 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    function callMe (address target) external {
        (bool success, ) = target.call("");
        require(success);
    }
}
```

- [ ] A. Pass only if target is an existing contract address
- [x] B. Pass for a non-existent contract address
- [ ] C. Pass always
- [ ] D. Fail always

▼ Solution

Correct is B.

> Account existence check for low-level calls: Low-level calls call/delegatecall/staticcall return true even if the account called is non-existent

> (per EVM design). Account existence must be checked prior to calling if needed.

from point 38 of Security Pitfalls & Best Practices 101 - by Secureum

> “The security concern(s) in the given contract snippet is/are”
>
> — 11 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple
    // Assume admin is set correctly to contract deployer in
    address admin;

    function setAdmin (address _newAdmin) external {
        admin = _newAdmin;
    }
}
```

- [x] A. Missing access control on critical function
- [x] B. Missing zero-address validation
- [x] C. Single-step change of critical address
- [x] D. Missing event for critical function

▼ Solution

Correct is A, B, C, D.

> Incorrect access control: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic.

from point 4 of Security Pitfalls & Best Practices 101 - by Secureum

> Missing zero address validation: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

from point 49 of Security Pitfalls & Best Practices 101 - by Secureum

> Critical address change: Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e. claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible.

from point 50 of Security Pitfalls & Best Practices 101 - by Secureum

> Missing events: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain.

from point 45 of Security Pitfalls & Best Practices 101 - by Secureum

> “The security concern(s) in the given contract snippet is/are”
>
> — 12 of 16

```
pragma solidity 0.8.4;

contract test {

    // Assume other required functionality is correctly imple

    address admin;
    address payable pool;

   constructor(address _admin) {
```

```
        admin = _admin;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function setPoolAddress(address payable _pool) external {
        pool = _pool;
    }

    function addLiquidity() payable external {
        pool.transfer(msg.value);
    }
}
```

- [x] A. Uninitialized pool storage variable which assumes setPoolAddress() will be called before addLiquidity()
- [ ] B. Incorrect use of modifier onlyAdmin on setPoolAddress()
- [x] C. Missing zero-address validation for _pool in setPoolAddress()
- [x] D. Transaction order dependence risk from admin front-running with pool address change

▼ Solution

Correct is A, C, D.

> Function invocation order: Externally accessible functions (external/public visibility) may be called in any order (with respect to other defined functions). It is not safe to assume they will only be called in the specific order that makes sense to the system design or is implicitly assumed in the code. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called before other system functions can be called.

from point 145 of Security Pitfalls & Best Practices 201 - by Secureum

> Uninitialized state/local variables: Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables.

from point 67 of Security Pitfalls & Best Practices 101 - by Secureum

> Missing zero address validation: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

from point 49 of Security Pitfalls & Best Practices 101 - by Secureum

> Transaction order dependence: Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic ERC20 approve() change can be front-run using this method. Do not make assumptions about transaction order dependence.

from point 21 of Security Pitfalls & Best Practices 101 - by Secureum

> “The security concern(s) in the given proxy-based implementation contract snippet is/are”
>
> — 13 of 16

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contract

contract test is Initializable {

    // Assume other required functionality is correctly imple

    address admin;
    uint256 rewards = 10;

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
```

```
    }

    function initialize (address _admin) external {
        require(_admin != address(0));
        admin = _admin;
    }

    function setRewards(uint256 _rewards) external onlyAdmin
        rewards = _rewards;
    }
}
```

- [ ] A. Imported contracts are not upgradeable
- [x] B. Multiple initialize() calls possible which allows anyone to reset the admin
- [x] C. rewards will be 0 in the proxy contract before setRewards() is called by it
- [ ] D. All the above

▼ Solution

Correct is B, C. There are no imported contracts that need to be made upgradable (by implementing Initializable).

> Import upgradeable contracts in proxy-based upgradeable contracts: Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors.

from point 97 of Security Pitfalls & Best Practices 101 - by Secureum

> Unprotected initializers in proxy-based upgradeable contracts: Proxy-based upgradeable contracts need to use public initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via initializer modifier from OpenZeppelin's Initializable library) is a must.

from point 95 of Security Pitfalls & Best Practices 101 - by Secureum

> Initializing state-variables in proxy-based upgradeable contracts: This should be done in initializer functions and not as part of the state variable declarations in which case they won't be set.

from point 96 of Security Pitfalls & Best Practices 101 - by Secureum

> "The security concern(s) in the given contract snippet is/are"
>
> — 14 of 16

```
pragma solidity 0.8.4;

import "https://github.com/OpenZeppelin/openzeppelin-contract

contract test {

    // Assume other required functionality is correctly imple

    address admin;
    address token;

    constructor(address _admin, address _token) {
        require(_admin != address(0));
        require(_token != address(0));
        admin = _admin;
        token = _token;
    }

    modifier onlyAdmin {
        require(msg.sender == admin);
        _;
    }

    function payRewards(address[] calldata recipients, uint25
        for (uint i; i < recipients.length; i++) {
            IERC20(token).transfer(recipients[i], amounts[i]
        }
```

```
        }
    }
```

- [x] A. Potential out-of-gas exceptions due to unbounded external calls within loop
- [ ] B. ERC20 approve() race condition
- [x] C. Unchecked return value of transfer() (assuming it returns a boolean/other value and does not revert on failure)
- [x] D. Potential reverts due to mismatched lengths of recipients and amounts arrays

▼ Solution

Correct is A, C, D. There's no guarantee that the passed arrays are of same length, so if one would be longer than the other one it can cause an Out Of Bounds error, which is why D is correct.

> Calls inside a loop: Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded.

from point 43 of Security Pitfalls & Best Practices 101 - by Secureum

> ERC20 transfer() does not return boolean: Contracts compiled with solc >= 0.4.22 interacting with such functions will revert. Use OpenZeppelin's SafeERC20 wrappers.

from point 24 of Security Pitfalls & Best Practices 101 - by Secureum

> This is ERC20 token transfer and not Ether transfer (which throws on failure). ERC20 transfer is typically expected to return a boolean but non-ERC20-conforming tokens may return nothing or even revert which is typically why SafeERC20 is recommended.

from Rajeev on Secureum Discord Server

> “The vulnerability/vulnerabilities present in the given contract snippet is/are”
>
> — 15 of 16

```
pragma solidity 0.8.4;
import "https://github.com/OpenZeppelin/openzeppelin-contract

contract test {

 // Assume other required functionality is correctly implemer
 // For e.g. users have deposited balances in the contract

    mapping (address => uint256) balances;

    function withdrawBalance() external {
        msg.sender.call{value: balances[msg.sender]}("");
        balances[msg.sender] = 0;
    }
}
```

- [x] A. Reentrancy
- [ ] B. Integer overflow leading to wrapping
- [ ] C. Integer underflow leading to wrapping
- [ ] D. None of the above

▼ Solution

Correct is A.

> Reentrancy vulnerabilities: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards.

from point 13 of Security Pitfalls & Best Practices 101 - by Secureum

> Integer overflow/underflow: Not using OpenZeppelin’s SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or

> unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. Solc v0.8.0 introduced default overflow/underflow checks for all arithmetic operations.

from point 19 of Security Pitfalls & Best Practices 101 - by Secureum

> “The security concern(s) in the given contract snippet is/are”
>
> — 16 of 16

```
pragma solidity 0.8.4;

contract test {
// Assume other required functionality is correctly implement
// Assume that hash is the hash of a message computed elsewhe
// Assume that the contract does not make use of chainID or r

function verify(address signer, bytes32 memory hash, bytes32
   return signer == ecrecover(hash, sigV, sigR, sigS);
}
```

- [x] A. Signature malleability risk of ecrecover
- [x] B. Missing use of nonce in message hash may allow replay attacks across transactions
- [x] C. Missing use of chainID in message hash may allow replay attacks across chains
- [x] D. Missing zero-address check for ecrecover return value may allow invalid signatures

▼ Solution

Correct is A, B, C, D.

> Signature malleability: The ecrecover function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin’s

> ECDSA library.

from point 23 of Security Pitfalls & Best Practices 101 - by Secureum

> Insufficient Signature Information: The vulnerability occurs when a digital signature is valid for multiple transactions, which can happen when one sender (say Alice) sends money to multiple recipients through a proxy contract (instead of initiating multiple transactions). In the proxy contract mechanism, Alice can send a digitally signed message off-chain (e.g., via email) to the recipients, similar to writing personal checks in the real world, to let the recipients withdraw money from the proxy contract via transactions. To assure that Alice does approve a certain payment, the proxy contract verifies the validity of the digital signature in question. However, if the signature does not give the due information (e.g., nonce, proxy contract address), then a malicious recipient can replay the message multiple times to withdraw extra payments. This vulnerability was first exploited in a replay attack against smart contracts [36]. This vulnerability can be prevented by incorporating the due information in each message, such as a nonce value and timestamps

from point 3.1.13 of A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses

> Indistinguishable Chains: This vulnerability was first observed from the cross-chain replay attack when Ethereum was divided into two chains, namely, ETH and ETC [10]. Recall that Ethereum uses ECDSA to sign transactions. Prior to the hard fork for EIP-155 [7], each transaction consisted of six fields (i.e., nonce, recipient, value, input, gasPrice, and gasLimit). However, the digital signatures were not chain-specific, because no chain-specific information was even known back then. As a consequence, a transaction created for one chain can be reused for another chain. This vulnerability has been eliminated by incorporating chainID into the fields.

from point 3.2.1 of A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses

> `ecrecover() returns (address)`: recover the address associated with the public key from elliptic curve signature or return zero on error.

from of Mathematical and Cryptographic Functions – Solidity Documentation

In Blockchain Tags Ethereum, Secureum Bootcamp

← Damn Vulnerable DeFi v2 - part #1: … Secureum Bootcamp Solidity 201 Quiz …