Solidity versions: Using very old versions of Solidity prevents benefits of bug fixes and newer security checks. Using the latest versions might make contracts susceptible to undiscovered compiler bugs. Consider using one of these versions: *0.7.5, 0.7.6 or 0.8.4* . (see here)

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. (see here)

Multiple Solidity pragma: It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks. (see here)

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. (see here and here)

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

Unprotected call to *selfdestruct*: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions. (see here)

Modifier side-effects: Modifiers should only implement checks and not make state changes and external calls which violates the checks-effects-interactions pattern. These side-effects may go unnoticed by developers/auditors because the modifier code is typically far from the function implementation. (see here)

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

Constructor names: Before *solc 0.4.22*, constructor names had to be the same name as the contract class containing it. Misnaming it wouldn't make it a constructor which has security implications. *Solc 0.4.22* introduced the *constructor* keyword. Until *solc 0.5.0*, contracts could have both old-style and new-style constructor names with the first defined one taking precedence over the second if both existed, which also led to security issues. *Solc 0.5.0* forced the use of the *constructor* keyword. (see here and here)

Void constructor: Calls to base contract constructors that are unimplemented leads to misplaced assumptions. Check if the constructor is implemented or remove call if not. (see here)

Implicit constructor callValue check: The creation code of a contract that does not define a constructor but has a base that does, did not revert for calls with non-zero callValue when such a constructor was not explicitly payable. This is due to a compiler bug introduced in *v0.4.5* and fixed in *v0.6.8*. Starting from Solidity 0.4.5 the creation code of contracts without explicit payable constructor is supposed to contain a callvalue check that results in contract creation reverting, if non-zero value is passed. However, this check was missing in case no explicit constructor was defined in a contract at all, but the contract has a base that does define a constructor. In these cases it is possible to send value in a contract creation transaction or using inline assembly without revert, even though the creation code is supposed to be non-payable. (see here)

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. (see here)

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. (see here)

ERC777 callbacks and reentrancy: ERC777 tokens allow arbitrary callbacks via hooks that are called during token transfers. Malicious contract addresses may cause reentrancy on such callbacks if reentrancy guards are not used. (see here)

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. (see here and here)

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. (see here)

Weak PRNG: PRNG relying on *block.timestamp*, *now* or *blockhash* can be influenced by miners to some extent and should be avoided. (see here)

Block values as time proxies: *block.timestamp* and *block.number* are not good proxies (i.e. representations, not to be confused with smart contract proxy/implementation pattern) for time because of issues with synchronization, miner manipulation and changing block times. (see here)

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. (see here and here)

Divide before multiply: Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. (see here)

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. (see here)

ERC20 *approve()* race condition: Use *safeIncreaseAllowance()* and *safeDecreaseAllowance()* from OpenZeppelin's *SafeERC20* implementation to prevent race conditions from manipulating the allowance amounts. (see here)

Signature malleability: The *ecrecover* function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin's ECDSA library. (see here, here and here)

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

Incorrect return values for ERC721
***ownerOf()*:** Contracts compiled with *solc* >= *0.4.22* interacting with ERC721 *ownerOf()* that returns a *bool* instead of *address* type will revert. Use OpenZeppelin's ERC721 contracts. (see here)

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. (see here and here)

fallback* vs *receive(): Check that all precautions and subtleties of *fallback*/*receive* functions related to visibility, state mutability and Ether transfers have been considered. (see here and here)

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. (see here)

Locked Ether: Contracts that accept Ether via *payable* functions but without withdrawal mechanisms will lock up that Ether. Remove *payable* attribute or add withdraw function. (see here)

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. (see here)

Contract check: Checking if a call was made from an Externally Owned Account (EOA) or a contract account is typically done using *extcodesize* check which may be circumvented by a contract during construction when it does not have source code available. Checking if *tx.origin* == *msg.sender* is another option. Both have implications that need to be considered. (see here)

Deleting a *mapping* within a *struct*: Deleting a *struct* that contains a *mapping* will not delete the *mapping* contents which may lead to unintended consequences. (see here)

Tautology or contradiction: Tautologies (always true) or contradictions (always false) indicate potential flawed logic or redundant checks. e.g. *x >= 0* which is always true if *x* is *uint*. (see here)

Boolean constant: Use of Boolean constants (*true*/*false*) in code (e.g. conditionals) is indicative of flawed logic. (see here)

Boolean equality: Boolean variables can be checked within conditionals directly without the use of equality operators to *true*/*false*. (see here)

State-modifying functions: Functions that modify state (in assembly or otherwise) but are labelled *constant*/*pure*/*view* revert in *solc >=0.5.0* (work in prior versions) because of the use of *STATICCALL* opcode. (see here)

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. (see here)

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. (see here)

Dangerous shadowing: Local variables, state variables, functions, modifiers, or events with names that shadow (i.e. override) builtin Solidity symbols e.g. *now* or other declarations from the current scope are misleading and may lead to unexpected usages and behavior. (see here)

Dangerous state variable shadowing: Shadowing state variables in derived contracts may be dangerous for critical variables such as contract owner (for e.g. where modifiers in base contracts check on base variables but shadowed variables are set mistakenly) and contracts incorrectly use base/shadowed variables. Do not shadow state variables. (see here)

Pre-declaration usage of local variables: Usage of a variable before its declaration (either declared later or in another scope) leads to unexpected behavior in *solc < 0.5.0* but *solc >= 0.5.0* implements C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes. (see here)

Costly operations inside a loop: Operations such as state variable updates (use SSTOREs) inside a loop cost a lot of gas, are expensive and may lead to out-of-gas errors. Optimizations using local variables are preferred. (see here)

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. (see here)

DoS with block gas limit: Programming patterns such as looping over arrays of unknown size may lead to DoS when the gas cost of execution exceeds the block gas limit. (see here)

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

Unindexed event parameters: Parameters of certain events are expected to be indexed (e.g. ERC20 Transfer/Approval events) so that they're included in the block's bloom filter for faster access. Failure to do so might confuse off-chain tooling looking for such indexed events. (see here)

Incorrect event signature in libraries: Contract types used in events in libraries cause an incorrect event signature hash. Instead of using the type `address` in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.8*. (see here)

Dangerous unary expressions: Unary expressions such as $x =+ 1$ are likely errors where the programmer really meant to use $x += 1$. Unary + operator was deprecated in *solc v0.5.0*. (see here)

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. (see here)

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. (see here and here)

assert()/require() state change: Invariants in *assert()* and *require()* statements should not modify the state per best practices. (see here)

***require()* vs *assert()*:** *require()* should be used for checking error conditions on inputs and return values while *assert()* should be used for invariant checking. Between *solc 0.4.10* and *0.8.0*, *require()* used *REVERT* (*0xfd*) opcode which refunded remaining gas on failure while *assert()* used INVALID (*0xfe*) opcode which consumed all the supplied gas. (see here)

Deprecated keywords: Use of deprecated functions/operators such as *block.blockhash()* for *blockhash()*, *msg.gas* for *gasleft(), throw* for *revert()*, *sha3()* for *keccak256()*, *callcode()* for *delegatecall(), suicide()* for *selfdestruct(), constant* for *view* or *var* for *actual type name* should be avoided to prevent unintended errors with newer compiler versions. (see here)

Function default visibility: Functions without a visibility type specifier are *public* by default in *solc < 0.5.0*. This can lead to a vulnerability where a malicious user may make unauthorized state changes. *solc >= 0.5.0* requires explicit function visibility specifiers. (see here)

Incorrect inheritance order: Contracts inheriting from multiple contracts with identical functions should specify the correct inheritance order i.e. more general to more specific to avoid inheriting the incorrect function implementation. (see here)

Missing inheritance: A contract might appear (based on name or functions implemented) to inherit from another interface or abstract contract without actually doing so. (see here)

Insufficient gas griefing: Transaction relayers need to be trusted to provide enough gas for the transaction to succeed. (see here)

Modifying reference type parameters: Structs/Arrays/Mappings passed as arguments to a function may be by value (memory) or reference (storage) as specified by the data location (optional before *solc 0.5.0*). Ensure correct usage of memory and storage in function parameters and make all data locations explicit. (see here)

Arbitrary jump with function type variable: Function type variables should be carefully handled and avoided in assembly manipulations to prevent jumps to arbitrary code locations. (see here)

Hash collisions with multiple variable length arguments: Using *abi.encodePacked()* with multiple variable length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in *abi.encodePacked()*, use fixed length arrays or use *abi.encode()*. (see here and here)

Malleability risk from dirty high order bits: Types that do not occupy the full 32 bytes might contain "dirty higher order bits" which does not affect operation on types but gives different results with *msg.data*. (see here)

Incorrect shift in assembly: Shift operators (*shl(x, y)*, *shr(x, y)*, *sar(x, y)*) in Solidity assembly apply the shift operation of *x* bits on *y* and not the other way around, which may be confusing. Check if the values in a shift operation are reversed. (see here)

Assembly usage: Use of EVM assembly is error-prone and should be avoided or double-checked for correctness. (see here)

Right-To-Left-Override control character (U+202E): Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract. U+202E character should not appear in the source code of a smart contract. (see here)

Constant state variables: Constant state variables should be declared constant to save gas. (see here)

Similar variable names: Variables with similar names could be confused for each other and therefore should be avoided. (see here)

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. (see here and here)

Uninitialized storage pointers: Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to vulnerabilities. *Solc 0.5.0* and above disallow such pointers. (see here)

Uninitialized function pointers in constructors: Calling uninitialized function pointers in constructors of contracts compiled with *solc* versions *0.4.5-0.4.25* and *0.5.0-0.5.7* lead to unexpected behavior because of a compiler bug. (see here)

Long number literals: Number literals with many digits should be carefully checked as they are prone to error. (see here)

Out-of-range enum: *Solc < 0.4.5* produced unexpected behavior with out-of-range enums. Check enum conversion or use a newer compiler. (see here)

Uncalled public functions: *Public* functions that are never called from within the contracts should be declared *external* to save gas. (see here)

Dead/Unreachable code: Dead code may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

Unused return values: Unused return values of function calls are indicative of programmer errors which may have unexpected behavior. (see here)

Unused variables: Unused state/local variables may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

Redundant statements: Statements with no effects that do not produce code may be indicative of programmer error or missing logic, which needs to be flagged for removal or addressed appropriately. (see here)

Storage array with signed Integers with ABIEncoderV2: Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array. This is due to a compiler bug introduced in *v0.4.7* and fixed in *v0.5.10*. (see here)

Dynamic constructor arguments clipped with ABIEncoderV2: A contract's constructor which takes structs or arrays that contain dynamically sized arrays reverts or decodes to invalid data when ABIEncoderV2 is used. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.9*. (see here)

Storage array with multiSlot element with ABIEncoderV2: Storage arrays containing structs or other statically sized arrays are not read properly when directly encoded in external function calls or in *abi.encode()*. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.10*. (see here)

Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2: Reading from calldata structs that contain dynamically encoded, but statically sized members can result in incorrect values. This is due to a compiler bug introduced in *v0.5.6* and fixed in *v0.5.11*. (see here)

Packed storage with ABIEncoderV2: Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using ABIEncoderV2. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.7*. (see here)

Incorrect loads with Yul optimizer and ABIEncoderV2: The Yul optimizer incorrectly replaces *MLOAD* and *SLOAD* calls with values that have been previously written to the load location. This can only happen if ABIEncoderV2 is activated and the experimental Yul optimizer has been activated manually in addition to the regular optimizer in the compiler settings. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.5.15*. (see here)

Array slice dynamically encoded base type with ABIEncoderV2: Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.0* and fixed in *v0.6.8*. (see here)

Missing escaping in formatting with ABIEncoderV2: String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.6.8*. (see here)

Double shift size overflow: Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values. Nested logical shift operations whose total shift size is *2**256* or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions. This happens when the optimizer is used and *evmVersion >= Constantinople*. This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.6*. (see here)

Incorrect byte instruction optimization: The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values. This can happen when performing index access on *bytesNN* types with a compile time constant value (not index) of 31 or when using the byte opcode in inline assembly. This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.7*. (see here)

Essential assignments removed with Yul Optimizer : The Yul optimizer can remove essential assignments to variables declared inside *for* loops when Yul's *continue* or *break* statement is used mostly while using inline assembly with *for* loops and *continue* and *break* statements. This is due to a compiler bug introduced in *v0.5.8*/*v0.6.0* and fixed in *v0.5.16*/*v0.6.1*. (see here)

Private methods overridden: While private methods of base contracts are not visible and cannot be called directly from the derived contract, it is still possible to declare a function of the same name and type and thus change the behaviour of the base contract's function. This is due to a compiler bug introduced in *v0.3.0* and fixed in *v0.5.17*. (see here)

Tuple assignment multi stack slot components: Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values. This is due to a compiler bug introduced in *v0.1.6* and fixed in *v0.6.6*. (see here)

Dynamic array cleanup: When assigning a dynamically sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out. This is due to a compiler bug fixed in *v0.7.3*. (see here)

Empty byte array copy: Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data. This is due to a compiler bug fixed in *v0.7.4*. (see here)

Memory array creation overflow: The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption. This is due to a compiler bug introduced in *v0.2.0* and fixed in *v0.6.5*. (see here)

Calldata *using for*: Function calls to internal library functions with calldata parameters called via "*using for*" can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.9* and fixed in *v0.6.10*. (see here)

Free function redefinition: The compiler does not flag an error when two or more free functions (functions outside of a contract) with the same name and parameter types are defined in a source unit or when an imported free function alias shadows another free function with a different name but identical parameter types. This is due to a compiler bug introduced in *v0.7.1* and fixed in *v0.7.2*. (see here)

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. (see here and here)

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. (see here)

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. (see here)

Avoid *selfdestruct* or *delegatecall* in proxy-based upgradeable contracts: This will cause the logic contract to be destroyed and all contract instances will end up delegating calls to an address without any code. (see here)

State variables in proxy-based upgradeable contracts: The declaration order/layout and type/mutability of state variables in such contracts should be preserved exactly while upgrading to prevent critical storage layout mismatch errors. (see here)

Function ID collision between proxy/implementation in proxy-based upgradeable contracts: Malicious proxy contracts may exploit function ID collision to invoke unintended proxy functions instead of delegating to implementation functions. Check for function ID collisions. (see here and here)

Function shadowing between proxy/contract in proxy-based upgradeable contracts:
Shadow functions in proxy contract prevent functions in logic contract from being invoked. (see here)