Id	Name	Description	Alternate Names
		2 - Malicious Environment, Transactions or Input	
1A	Call to the unknown	An unrestricted CALL to an unknown address may lead to the execution of malicious code, e.g. the fallback function of the called contract. This vulnerability is a generalization of 1A1. Reentrancy, where the callee executes a call back to the caller.	Unexpected fallbackfunction , Malicious Fallback Function
1A1	Reentrancy	The callee calls the caller back before the initial call has been completed and relevant checks and state changes have been performed. The classical example is an Ether transfer to a contract, whose fallback function is specifically coded to call back the function emitting the original call.	Cross function race condition, race to empty, recursive-call vulnerability, reentrancy-with
1A1.1	Reentrancy with ETH transfer	This occurs when an external contract calls back into the contract that initiated the ETH transfer before the state changes (such as balance updates) are finalized. This can allow the attacker to withdraw more funds than intended.	Reentrancy Eth , Unsafe Credit Transfer
1A1.2	Reentrancy without ETH transfer	In this case, the contract does not transfer Ether directly but may still have a reentrancy vulnerability because of a recursive call, which might allow attackers to modify states or perform unintended operations.	Reentrancy No Eth , Unsafe System State Changes
1A1.3	Reentrancy with same effect	This form of reentrancy occurs when the reentrancy leads to multiple calls with the same effect	Reentrancy Benign
1A1.4	Reentrancy Vulnerability with Negative Events	This form of reentrancy involves an attacker triggering negative or destructive events (like causing a contract to fail or exploit the contract's vulnerability to cause undesirable outcomes such as loss of funds, or altered contract states).	Reentrancy Events
1B	Exact balance dependency	If the logic of a contract depends on maintaining the exact balance in its sign instead of querying the environment for the current balance, attackers can manipulate the contract by increasing its balance without triggering its code. This can be achieved by using the contract's address as the target of a SELFDESTRUCT operation or by initiating a reward, or by predicting the contract's address and sending Ether before the contract gets deployed.	Unexpected Ether, Manipulated balance, Unexpected Balance, Unexpected Ether balance, Unprotected Ether Balance, Balance equality, Strict Balance Equality, Improper Locking
1B1	Forcing Ether to contracts	Forcing Ether to contracts can be achieved by a) using the contract's address as the beneficiary of a SELFDESTRUCT operation or as the receiver of a mining reward.	Check on forced to receive balance, Force Sending Ether with this balance check in require(or assert(
1B2	Pre-sent Ether	Pre-sent Ether can be achieved by predicting the contract's address and sending Ether before the contract gets deployed.	
1C	Improper data validation	Failure to check data obtained from untrusted sources, such as external libraries or contracts, before using it.	Unsafe exterrnal data feeds
1D	Vulnerable Delegatecall	Performing a DELEGATECALL or CALLCODE instruction to an untrusted address. Code invoked via these types of calls is run in the caller's context. A malicious callee can manipulate the caller's state variables and fully control the balance.	Delegate call to Untrusted Callee , Delegate calls in secure Contracts, Unsafe delegatecall , Check or target address in delegatecall, Delegate call
		3 - Blockchain/Environment Dependency	
2A	Block State Dependency	The contracts gas, ether and block state variables or critical time-dependencies. Miners have control over these values and can manipulate them to a certain extent, or they may be subject to changes for other reasons.	Block State Dependency
2A1	Timestamp dependency	The block's timestamp should not be used for precise calculations or critical time-dependencies in the contract logic. Miners have control over this value and can manipulate it to a certain extent, or the value may be subject to changes for other reasons.	Time constraints, Time manipulation, Time restrictions, Block values as a proxy for time, Incorrect Use of Event Blockchain variables for Time
2A2	Block number dependency	Using the block number bears similar risks as using other block attributes like the timestamp and should not be used in critical operations, as miners can also manipulate the block number.	
2A3	Block.blh dependency	Blockhash usage vulnerability results from the usage, in critical operations, of blockhash variables. A malicious miner can manipulate this value.	Blockhash usage
2B	Transaction-ordering dependency	The logic of a contract should not depend on the order in which a block's transactions are executed, as miners can compose the block in any order and may do so deliberately for malicious reasons. Moreover, the state of a contract may change between the time a transaction is submitted to the network and the time it is included in a block, or the transaction, even when submitted later, may be prioritized by miners.	Event-ordering bugs, Incorrect Function Call Order
2B1	Front-running attack	Suppose a player guesses the answer and sends it. An attacker can check the transaction pool (mempool) and "steal" the answer. The attacker can send an equivalent transaction with a much higher gas price than the original one. Because of the higher gas, the second transaction will be processed first, causing a front-running attack.	Race condition
2B2	Transfer Pre-Condition Dependent on Transaction Order	In the case of this vuln-existing, the order in which transactions are executed influences a pre-condition that guards the execution of the transfer. This influence may entirely or only result in, for instance, a transaction not being executed at all	
2B3	Transfer Amount Depending on Transaction Order	This issue refers to the case where the value of the variable that stores or determines an amount of a digital asset (to be transferred) is modified before it is sent to the recipient's transaction ordering within a block. The amount may be changed due to the effect of multiple transactions being grouped in a block and executed in a specific order having the effect of producing unexpected changes in the value being transferred	
2B4	Transfer Recipient Depending on Transaction Order	In the case of this defect, the transfer recipient is modified before the said event due to transaction ordering within a block. As an example, if the intended recipient address is stored as a storage variable and a transfer is to executed based on this address, there is a chance the address may be changed or overwritten by another transaction prior to the transfer	
2C	Bad random number generation	Random numbers are important for applications like lotteries and games but are difficult to generate. Using global blockchain variables, such as timestamps, as seed is open to manipulation by malicious miners who can influence global variables. Moreover, other transactions in the same block or other contracts involved in the same transaction have access to the same global variables and may guess the random number.	Generating Randomness, Bad random ness, Weak Sources of Randomness from Chain Attributes, random number, Weak PRNG
2C1	Randomness Using Block Hash	So neither, randomness is required in operations, and block hash can be used for this purpose. However, it can be manipulated like a timestamp because the miners can predict the same.	
2D	Leakage of confidential information	All transactions and the state of all contracts are visible to all miners and the world outside of the blockchain, even if the visibility of variables is set to private or internal in the source code. Therefore, encryption and commitment schemes are important to protect secret information.	Lack of transaction privacy , Private information leakage, Nothing is secret, Confidentially failure, Keeping secrets, Data exposure, On-chain random manipulation, Secrecy failure, Exposed secret, Unencrypted private data on-chain, Secrets
2E	Unpredictable state	Calling contracts on-chain in the business logic of a contract can lead to unpredictable effects. They can self-destruct, be updated (alter, and so on) and known vulnerabilities or, more simply, be an error when returning to them. The results are as predictable effects.	Unsafe External Library Call, External Library Calling
2E1	Dynamic Library	This vulnerability happens when a smart contract depends on a potentially compromisable (other smart contracts). Therefore, the attacker can change the contract logic if the contracts the core contracts.	
2E2	References to destroyed contracts	In the context of Ethereum smart contracts, a destroyed contract refers to a contract that has been terminated or self-destructed via the selfdestruct function, also known as suicide in earlier versions of Solidity. When a contract is destroyed, its code and state are erased from the blockchain, and any remaining Ether (ETH) in the contract is sent to a specified address.	
2E3	Using components with known vulnerabilities	Using existing libraries and open-source codes in a newly created project is a very useful practice to avoid reinventing the wheel. However, open-source codes and libraries are not safe and might be vulnerable to a public exploit. Therefore, even if the newly created smart contracts safe, those added codes may introduce hidden vulnerabilities	Use of Malicious Libraries
2E4	External Contract Referencing	Solidity allows programmers to use any address to refer to a specific contract, enabling attacker contracts to concoct malicious code by providing an incorrect address to a smart contract constructor as a parameter. The misuse of address parameters in the contructor underscores the importance of address parameter avoidance. Smart contracts often need to reuse other contracts' code by referencing their addresses. For instance, a developer may initialize an external library in their contract's constructor, with the library's address provided during deployment. While this approach may seem legitimate and pass a code audit, providing the wrong address during deployment can create a security vulnerability. In such cases, a malicious contract (such as a honey pot) could be called from the incorrect address, posing a significant risk.	Misuse of address parameters in the con structs , Value assigned to contract address
2F	Unsafe Non-Blockchain External Call	Like external command execution, smart contracts also enable access to external files, but it cannot be assured that the nodes will receive identical results.	
2F1	Unsafe External Web Service Call	So neither, contracts may opt to streamline the development process and reduce effort by reusing external web services through API calls. Nevertheless, this approach can potentially lead to security issues if the returned values vary across different peer nodes. This vulnerability is also known in literature as Web service.	web service
2F2	Unsafe External Command Execution	Ensuring external contracts within smart contracts is possible. however, it does not ensure consistent results across all peer nodes.	system command execution
2F3	Unsafe External File Access	Like external command execution, smart contracts also enable access to external files, but it cannot be assured that the nodes will receive identical results.	External File Accessing
2G	Cross Channel Invocation	Certain blockchain platforms like Hyperledger Fabric permit contracts to call each other. However, when two contracts interact through different channels, a cross channel invocation occurs.	cross channel chaincode invocation
		3 - Exception & Error Handling Disorders	
3A	Unchecked low-level call/send return values	Low-level calls do not raise an exception on failure but return a boolean status. Failing to check the return status and to handle errors adequately may lead to critical vulnerabilities.	Unhandled exception, Improper handling of exceptions , Improper Checks if Low-Level Call Return Value, Mishandled Exception, Unchecked external call, Check on external call, Improper Check of External Call Return Value, Improper Exception Handling of External Calls
3A1	Unchecked call	The call function is used to send Ether or execute low-level function calls. If the return value of call (a boolean indicating success) is not checked, the contract might behave as if the call succeeded even if it failed.	Unchecked call return values
3A2	Unchecked send	Function send is an obsolete way to transfer Ether that forwards only 2300 gas. Like call, it returns a boolean, and if this is not checked, you don't know whether the transfer succeeded.	Unchecked unsafe send
3A3	Unchecked transfer	In older versions (both v0.4.0), transfer was considered safe. However, in certain contexts (e.g., when catch blocks are used), its behavior could be misinterpreted or ignored.	Improper Check on Transfer Credit
3B	Unexpected throw or revert	Calls to contracts may fail and throw exceptions or revert the transaction. Not handling these situations adequately can lead to critical issues, such as DoS vulnerabilities.	Improper Use of Exception Handling Functions
3B1	Mishandled out-of-gas exceptions	Calling another contract with too little gas will cause an out-of-gas exception, possibly resulting in an inevitability if handled inadequately. A typical situation is the transfer of funds with Solidity's send method. The amount of 2300 gas supplied with the call will lead to an out-of-gas exception if the callee executes more than a few simple instructions.	Gasless send (subcategory of 3B)
3C	Assert, require or revert violation	Solidity's error handling via assert, require, and revert (or its bytecode counterparts) must be used correctly to avoid abnormal function behavior.	Assert violation, Assertion failure, Requirement violation, Requirement violation by the called smart contract, Require on input variable, Revert require, Exception state
3C1	Wrong Selection of Guard Function	Assert is a Solidity function, which is recommended to be used only in the development phase. Intentionally, the programmer inserts the function at a specific point in the program where a bug is suspected. If running the program results in gas depletion, the suspicion is confirmed. Thus, this defect refers to the cases in which the assert function is implemented with the wrong purpose, not having the expected effect. In more severe cases, in which the programmer forgets to remove it from the code or does not replace it with require, the impact of this defect can be serious	
3D	Send instead of transfer	The choice of ether transfer function affects not only the transfer of gas but also the execution of the called (sender) contract because call, send and transfer/send/call return an exception in case of a failed transfer. In contrast, transfer/send throw an exception if the transaction is unsuccessful. This makes the use of send instead of transfer a vulnerability.	Wrong use of Transfer Credit Function
		4 - Denial of Service	
4A	Greedy contract	A smart contract can access assets through an external library, which may self-destruct accidentally or maliciously or lack the implementation of functions for asset withdrawal, providing a trap deposit functions.	Locked Ether, Freezing Ether
4A1	Locked Money	Smart contracts that implement payable functions but no withdrawal ones are vulnerable because they can lock money without the ability to get it back, resulting in the transaction ether being locked in the contract.	Unreachable Payable Function
4A2	Frozen Ether	If a contract does not have a way to access its assets, their accidental or maliciously self-destruction of the library contract may render the assets inaccessible.	
4B	Ether sent to an unpayable contract	Ether sent to an unpayable contract, for instance, to a logic, for authentication purposes instead of the address of the immediate caller, contracts situated in the call chain between its origin and the vulnerable contract gas access to the assets of the contract.	Transfer to unpayable address, Lack of Ether, Loss of Ether
4C	DoS with block gas limit reached	The total gas spent by a block's transactions is bounded by the block gas limit. Transactions exceeding this limit will not get processed.	DoS Costly Pattern and Loops, Check on gas Limit, Gas limit in loops, Extra Gas in loop , DoS with Block Gas Limit
4C1	Multiple calls in a single transaction	Performing calls inside a loop, for example, could create a DOS. If one call in the chain, the whole execution stops. So if there is a loop on the user address and one of them is a contract that implements a fallback function that always reverts, a DOS attack will occur.	Multiple calls, Transfer in loop, Nested Call
4C2	Array length manipulation	For older versions of solidity, it is possible to manipulate the "length" field of an array with standard arithmetic functions. A simple code to decrement the value of this field can bring the value to an underflow. "array.length--" can make the array size set to max int. This vulnerability can disable a contract.	Unsafe Array Length Assignment
4C3	Infinite Loop	In a loop's termination condition, an unintended input from a user or a comparison of integers of different sizes can cause the loop to exceed the block gas limit or before expensive to complete and result in transaction failures with non-terminating loops or unbounded mass operations errors.	Unbounded mass operation
4D	DoS by Exception inside loop	Operation inside loops that may throw exceptions and depend on external transactions to succeed are open to DoS attacks by malicious users.	DoS unsafe external influence, Failed gasleft causing out of gas exception, Improper Exception Handling in a Loop
4E	Insufficient gas griefing	Insufficient gas griefing occurs when a function call, typically to an external contract or address, is forwarded insufficient gas, preventing the called function from executing its logic. This can be exploited by attackers to block or disrupt operations, especially in systems that depend on sub-transactions, like fraud verification or payment distribution.	Incorrect Revert Implementation in a Loop
4F	Overpowered Role	In Solidity smart contracts, an "overpowered role" refers to a role with extensive control over a smart contract, like an administrator role with broad privileges such as modifying parameters, freezing accounts, or accessing sensitive data unchecked. Such roles pose risks like centralization, abuse, and security vulnerabilities. It's crucial for developers to design access controls to ensure roles are balanced and prevent any single role from having excessive power. This safeguards the integrity, security, and fairness of decentralized systems.	Check is overpowered role
4G	Missed to implement a fallback function	Using call.value(x) can be problematic when the recipient function is not explicitly specified in the call's arguments. By default, call invokes the default fallback function of the target contract. Suppose a malicious contract deliberately does not implement a fallback function. In this case, any deposit made to that not succeed, and the sender contract will not be able to function as intended due to failed transfers. This vulnerability can lead to DoS issues in the sender contract.	
4H	Improper locking during external calls	A vulnerable contract raises a lock mechanism in an erroneous manner, which may cause deadlocks. This may result, for instance, in the impossibility of executing transfers and eventually in Denial of Service	
4I	DoS with failed call	DoS with Failed Call occurs when a single failed external call causes the entire transaction or loop to revert or stop, effectively allowing a malicious participant to block functionality for everyone else.	unexpected throw DoS, DoS with unexpected revert throw, DoS by External contracts
4J	Unsatisfiable(1) in the require() Condition	If there is send in require logic, the logic may revert from recipient by return false, cause dos attack.	
		5 - Resource Consumption & Gas Related Issues	
5A	Gas costly loops	Unbounded and costly operations in loops should be avoided to avoid inefficiency and potential DoS.	Invariants in loops
5B	Gas costly pattern	Smart contract code should pre-evaluate inefficient operations and gas-costly patterns that consume more gas than necessary.	Inefficient Operation Sequence
5C	High gas consumption of variables data type or declaration	In Solidity, certain data types consume more gas than others, and a variable's visibility also influences gas consumption. Similarly, choosing inefficient access modifiers for functions, such as declaring a public instead of external when unnecessary, can increase resource usage. These characteristics can contribute to excessive gas costs.	
5C1	Byte Array	In Solidity, the data type byte[] can take more gas than a byte array due to padding rules.	Variable type, Bytes Variables Risk , Wrong Type in Variable Declaration
5C2	In-exact state variables is not declared constant	Invariants not declared constant consume more gas than necessary.	Declaration of invariant state variable , Missing Constant Modifier in Variable Declaration
5D	High gas consumption function type	Using the external function type in place of public, instead of extern, require more gas on deployment than necessary.	Function declaration public instead of external
5E	Underpriced opcodes	Contracts executing numerous cheap yet spurious opcodes may consume large amounts of computing resources, as the resource consumption is not reflected in the gas cost, which may lead to DoS attacks.	Improper Gas Requirements Checking
5F	Transfer of all the gas	Solidity provides three built-in functions to trade ether: transfer(), send(), and call(). Along with others, transfer() and send() transfer the Ethereum-specified minimum required 2300 units of gas for the receiver contract to process the transaction. The call() function is the one recommended by Solidity but does not provide parameters with the amount of the gas the transaction provides to the recipient, transferring all the gas. Gas transfer validity assures that if a specific gas amount is required to be transferred, then add.call.value(x)() is substituted with that amount, otherwise, transfer(x) or preferably add.call.value(x)() is used.	
		6 - Authentication & Access Control Vulnerabilities	
6A	Prodigal contract	Using msg.sender might not always be appropriate, for instance, if a contract is called by another contract to refund a transaction to a specific investor account specified in the parameters. In such cases, relying on msg.sender means that the calling contract could receive the refund instead of the intended recipient specified in the parameters. However, specifying the recipient in the parameters can also be problematic. If any arbitrary contract can initiate the transaction, it might provide an address of a contract that never submitted any funds, thus making it ineligible for a refund. This scenario describes a "prodigal contract," which can incorrectly send refunds to arbitrary contracts. To address these issues, it is crucial to validate the transfer address. Ensure that while authenticating a transaction, tx.origin is not used, and if recipient addresses are provided in the parameters, they must be verified against a list of eligible contracts. If the recipient address is not provided, default to using msg.sender.	
6A1	Authorization via transactio n origin	If a contract uses the external address in lieu of the transaction, tx.origin, for authentication purposes instead of the address of the immediate caller, contracts situated in the call chain between tx.origin and the vulnerable contract gain access to the assets of the contract.	Authenticate through tx.origin , Transaction state dependency , Transaction origin use, Dangerous Usage of tx.origin , Wrong Caller Identification
6B	Access control management	Functions that are supposed to be executed by only users with specific roles are not protected. Therefore, any user can execute them. For example, functions that perform withdrawals or execute dangerous functions like SELFDESTRUCT should be protected and only executed by users with the right permissions.	
6B1	Unprotected selfdestruction	Insufficient access control to a SELFDESTRUCT instruction may lead to a contract's accidental or malicious destruction.	Unexpected kill, Destructable contract, Suicidal contract, Unsafe suicide, Check on suicide functionality, Unprotected Contract Destruction , Missing verification for program termination , Flaky Multisig
6B2	Unauthorized Ether withdrawal	Insufficient access control to a contract's withdrawal functionality can lead to asset theft.	Unprotected Ether withdrawal , Check on balance withdrawal, Ether leak, Unprotected Transfer Value
6B3	Initializing Method without Permission Check	This vulnerability occurs when a smart contract's initialization function, typically used to set key variables like the owner or admin, can be called by anyone because it lacks access control. This usually happens in proxy-based or upgradeable contracts that do not use constructors and instead rely on initialization functions.	
6B4	Missing msg.sender Check for transferFrom()	This vulnerability arises when a smart contract allows anyone to invoke the transferFrom() function without verifying that the caller (msg.sender) is authorized to transfer tokens or funds on behalf of someone else.	Arbitrary from in transferFrom() without msg.sender Check
6B5	Sending Ether to Arbitrary Destinations	When an secure function "call" is used ether, the target address should be carefully checked, and an arbitrary address may lead to exploitability.	
6B6	Unauthorized accessibility due to wrong function or state variable visibility	In Solidity, functions inadvertently declared public/external will expose an entry point that may lead to unauthorized access by malicious users.	Visibility of exposed functions, Function or state variable default visibility, Function State visibility error, Unspecified visibility level, Bad visibility, Default Visibility, No restricted write, Implicit visibility level, No-public variables accessed by public/external functions, Default/unintended/erroneous visibility
6B6.1	Function default visibility	In Solidity, if no visibility is explicitly declared for a function, it defaults to public in versions prior to 0.5.0. This means anyone can call the function externally, even if the developer intended it to be private or internal.	Default Function Visibility , Wrong Function Modifier
6B6.2	State variable default visibility	In Solidity, state variables must have an explicitly declared visibility (public, internal, or private). If none is specified, it defaults to internal.	Missing Visibility Modifier in Variable Declaration
6B7	Bypassable Modifier	Invalid or modifiers that could be bypassed from a error more unprotected smart contract functions.	
6C	Signature based vulnerabilities	This vulnerability includes issues like missing protection against signature replay attacks, lack of proper signature verification, hash collisions with multiple variable length arguments, signature malleability, signatures with zero-ing parameter, and any related vulnerabilities.	Insufficient sign ature information
6C1	Missing protection against signature replay attack	Signature verification may occasionally be required in smart contracts to improve usability or reduce gas costs. By maintaining a list of all digested message hashes and only allowing fresh message hashes to be processed, developers can prevent multiple processing of the same message hash supplied by different users. The vulnerabilities 6C1 and 6C5 may seem the same, but in the 6C5, the attacker creates a valid signature with different information to bypass the verification. However, with 6C1, the attacker only uses an existing and already executed transaction to make a replay attack.	Improper Check against Signature Replay Attacks
6C2	Lack of proper signature verification	Because of the flexibility and improved transferability that this offers, smart contract systems frequently permit users to sign messages off-chain rather than requiring them to do an on-chain transaction. Smart contract systems that handle signed messages must build their logic to determine their authenticity before continuing to process them. Smart contracts cannot directly engage with such systems since they cannot sign messages, which is a constraint for them. Some signature verification systems try to resolve this issue by presuming the authenticity of a signed message based on the presence of a valid signature. However, the validity of such a technique is its dependent on the value of the msg.sender property and presume that if a signed message came from the sender address, then the sender address also produced it. In situations where process may be used to relay transactions, this may result in vulnerabilities.	Method permit() lack for Arbitrary from in transferFrom() , Improper authenticity check
6C3	Hash collision with multiple variable length arguments	Hash collisions is possible when the abi.encodePacked() function is used with several parameters with different lengths. The abi.encodePacked() packs all items in order whether or not they are part of an array, elements could be transferred across arrays and thus still return the same encoding as long as all the corresponding items are the same order. An attacker might take advantage of this in a signature verification scenario by changing the assignment of components in a prior function call to successfully evade permission .	Incorrect Argument Encoding
6C4	Function Clashing	To call a function in some libraries like Ethereum, the user sends a function hash in the data field of his transaction. Then, based on that hash, the EVM chooses what function to execute. As the hash used in this operation is very weak, collisions are more likely to happen. The existence of the function clashing vulnerability could result in executing functions unconsciously, which then may lead to financial losses	
6C5	Signature Malleability	Although it is frequently assumed that each signature is unique when a cryptographic signature scheme is implemented in Ethereum contracts, signatures can be changed without having access to the private key and still be considered legitimate. One of the so-called "precompiled" contracts defined by the Ethereum virtual machine (EVM) standard is ecrecover, which implements elliptic curve public key recovery. The three values v, r, and s can be slightly altered by a malicious user to produce different valid signatures. If the signature is a component in the hash of the signed message, a system that verifies signatures at the contract level may be vulnerable to attacks. A malevolent user might produce valid signatures to replay previously signed communications.	Incorrect Verification of Cryptographic Signature
6C6	Map overlap	Solidity's map data structure (similar to a hash table or a dictionary) uses a hash function to compute an index into an array of slots. If two maps are used in a contract, the hash function may produce the same hash value for two different memory locations, one in each map, creating a collision across maps. We propose a new terminology map overlap for this vulnerability since hash collision is extended term for a collision of hash values within one hash table. It is part of the standard operation of a hash table and is, by itself, not a vulnerability. This vulnerability has occurred in a contract that used two maps, one for admins of the contract and the other for registered users, as arguments of a function that checked whether a given address belonged to admins or regularly registered users. In this case, an attacker can manipulate the position of elements while registering as a regular user so that the hash function returns the same hash results for certain addresses in the admins and regular users, thus bypassing authentication. This vulnerability can be addressed by establishing map disjointness, which asserts that the contract does not use more than one map data structure.	
6D	Identity verification	This vulnerability happens when the smart contract tries to verify if the caller is a validator in a smart contract based on its code size. Unfortunately, this can be bypassed by calling the vulnerable contract in the constructor of the smart contract. Attackers, as the source code of the smart contract while executing the constructor is still 0.	
6E	Usage of public mint or burn	If a token contract exposes mint or burn functions as public or external without access restrictions, any external user can call them. Public mint allows anyone to create unlimited tokens, potentially inflating the supply and devaluing the token. Public burn would allow attackers to destroy other users' tokens or manipulate token balances logic, especially if combined with bugs that doesn't verify ownership .	
		7 - Arithmetic bugs	
7A	Integer over- or underflow	Integer over- and underflows occur when arithmetic operations exceed the maximum or minimum of integer types. Unless this exception is caught, a wrap-around occurs.	Arithmetic Underflow/Overflow, Integer bug, Value assignment with too many digits
7A1	Integer Overflow	Integer overflow occurs when arithmetic operations exceed the maximum value representable by an integer type in Solidity.	
7A2	Integer Underflow	Integer underflow occurs when arithmetic operations exceed the minimum value representable by an integer type in Solidity.	
7B	Unchecked division	Handling floating point numbers, represented as integers, incorrectly may lead to inaccuracies, errors, and vulnerabilities.	
7B1	Integer division	Solidity does not admit floating point or decimal types and integer division is always rounded toward zero. This choice of rounding method can produce results other than what some users might expect.	Money leak, Floating point & Precision
7B2	Integer Division before Multiplication	Solidity supports integers and (partially) fixed-point numbers. So, to prepare for floating points, integers must be used. The 256-bit Ethereum virtual machine assembles type shorter than 32 bytes together in the 32-byte slot. This process affects the accuracy of any operation. Nonetheless, rounding is incorrect if a division is performed before multiplication. Therefore, performing multiplication before division mitigates round-ing off errors in a smart contract.	Arithmetic Precision
7B3	Divide by Zero	This code is vulnerable to error-checked division, like dividing by zero.	
7C	Integer bugs or Arithmetic Issues	Operations between signed and unsigned integers or between integers of different lengths may lead to incorrect results, as can the use of unsupported literals.	Check on arithmetic operation , Unchecked Math, Wrong operator
7C1	Incorrect calculation	CVE-2023-39693 is a vulnerability identified in Ethereum Blockchain version 0.1.1-commit.ffffcb6. This issue arises when the combined value of a bet size and the casino's balance exceeds a certain threshold, causing the balances to be zeroed out. This flaw could potentially lead to financial losses for users interacting with affected Ethereum-based contracts.	
7C2	Type cast bugs	This vulnerability refers to the scenarios a variable declared in a certain type is converted to a smaller type, which means that data is lost during the conversion process	Integer Truncation
7C3	Signedness bugs	The conversion of a signed integer type to an unsigned type of the same width may change a negative value to a positive one (the opposite may also happen)	Integer sign
		8 - Bad Coding Quality and Programming Language Specifics	
8A	Type cast	To interact with another contract, the programmer specifies its interface and uses its typecast addresses. This allows the Solidity compiler to construct appropriate low-level calls. However, the compiler does not (cannot) check whether a interface actually matches the other contract, even though the programmer might have the false impression that the contract is correctly typed. Calling a contract with a wrong interface specification may invoke unintended entry points, with arguments interpreted differently than expected.	Unsafe type declaration, Type conversion, Wrong Function Call
8B	Coding error	Coding errors, including typos and other developer errors.	Immutable bugs, typographical error , coding bug
8C	Bad coding pattern	Use of various coding patterns that are regarded as bad style.	
8D	Deprecated source language features	The use of deprecated language features is discouraged, as it may lead to known errors and vulnerabilities that were the cause for deprecation.	Use of Deprecated Solidity functions , Use of Deprecated Functions
8E	Use of assembly	The use of assembly instructions in high-level code is discouraged, as it can lead to various critical vulnerabilities.	Assembly based vulnerabilities, Usage of assembly , Inter-procedure Assembly Usage
8E1	Arbitrary jump with function type variable	Function types are supported by Solidity. In other words, a reference to a function with a matching signature can be given to a variable of a function type. It is possible to call the function stored in such a variable exactly like any other function. If the developer utilizes assembly instructions like "mstore" or the assign operator, an attacker might point a function type variable to any code in the code section, breaking necessary validations and state updates.	Arbitrary jump, Control-Flow Hijack, Function type variables assignment using arbitrary values , Specify function variable as any type, Arbitrary Function Jump via inline Assembly
8E2	Constant functions using assembly code	Functions declared as constant/pure/view using assembly code. Constant/Pure/View was not enforced before Solidity 0.4. Starting from Solidity 0.4, a call to a constant/pure/view function uses the STATICCALL opcode, which reverts in case of state modification.	Constant functions, Interoperability issues with other contracts
8E3	Returning results using assembly code in a constructor	The use of assembly within a Solidity constructor to manipulate the contract's deployment process can significantly alter the deployed bytecode. This method allows for a deployed contract that differs significantly from the source code. This approach exploits the flexibility of the EVM to execute low-level operation that bypass standard compilation and deployment procedures.	
8F	Incorrect inheritance order	Solidity supports multiple inheritance, which can lead to an ambiguity called the Diamond Problem if two or more base contracts define the same function. This is resolved using C3 Linearization, which leads to a deterministic method resolution order. Programmers unaware of this aspect may base their code on wrong assumptions, leading to unexpected behavior and vulnerabilities.	Wrong class inheritance order
8G	Variable shadowing	Solidity allows for ambiguous naming of variables when inheritance is used, leading to errors and vulnerabilities that are difficult to identify in complex contract systems.	Hidden state variables, Shadowing state variables, Variable name, Talked-through, Shadowing state, Shad owing abstract, Shadowing public , Shadowing local
8G1	Use of Same Variable or Function Name in Inherited Contract	When using the same name as a local variable, which was previously declared by an inherited contract, the program loses the reference of the inherited variable, causing the local variable to assume the role of the other variable	
8G2	Variables or Functions Named After Reserved Words	This bug occurs when creating variables named after keywords of the language itself. For example, in Solidity, creating a variable with the name now conflicts with the function that returns the date and time.	
8G3	Use of the Same Variable or Function Name in a Single Contract	This vulnerability refers to cases where the same name is used to more than one variable or function inside the contract. This makes the program lose the reference of the variable of the class, assuming the variable of the function as its size	
8H	Misleading source code	Malicious actors may apply various strategies to confuse or mislead auditors and contract code behavior and trick users.	API inconsistency
8H1	Right to Left override	Right-To-Left Override is a special Unicode character (U+202E) that allows the use of right-to-left (RTL) characters within the text so mostly read and left-to-right (LTR) text. While this feature is intended for supporting languages like Arabic or Hebrew, it can be misused in smart contracts to obscure the code's intent. For instance, an attacker might use RTL characters in a function name to make it appear benign while actually executing something entirely different. This deceptive practice can mislead developers and users, potentially leading to security issues or unintended behaviors in the contract. Proper vigilance and tools to detect and prevent such manipulations are crucial to maintain the integrity and security of smart contracts.	Function Call with Wrong Arguments
8H2	Misleading data location	Do not clarify the desired data types of local variables of struct, array, or mapping.	Modifying storage array by value
8H3	Hidden built-in symbols	This vulnerability happens when the developer uses a built-in symbol of the solidity language. This action could change the default behavior of the smart contract development language and cause business logic bugs	Built-in symbol shadowing, Use of expected built-in symbols
8I	Missing logic, incorrect errors or dead code	Missing logic, logical errors, or dead code can confounded lead to unwanted behavior and vulnerabilities.	Logical error in smart contract code, Business logic vulnerabilities, Wrong Logic
8I1	Code with no effects	In Solidity, a "code with no effects" vulnerability refers to a situation where a contract includes a function that performs computations or makes decisions but does not have any impact on the contract's state or the blockchain in general.	No effect call execution
8I2	Unused variable	This defect refers to the declaration of variables that are not used in the contract, which results directly in the allocation of unnecessary space in memory. As a consequence, the gas cost of executing the contract increases as well as the attack surface of the contract. Other effects are related to the readability or maintainability of the code.	Presence of unused variables
8I3	Unused return	In Solidity, some functions return values (e.g., call, send, delegatecall) indicating success or failure. If a contract ignores or does not use these return values, it may proceed under false assumptions — for instance, that a transfer succeeded when it didn't.	
8I4	Unused statement	An unused statement is a line of code that executes but has no effect on the program — e.g., expressions that compute a value but do not store or return it.	
8I5	Missing return type on function	This vulnerability refers to a missing return type in the definition of a smart contract interface. At runtime, if a contract that implements that interface contains two functions with the same name and arguments but have different return types, there is a chance that the wrong function will be called. This may lead to unexpected results since the calling contract receives the wrong data type	
8I6	Missing type in variable declaration	In Solidity, whenever a variable is declared without an associated type, the compiler infers the data type based on the assigned value. This additional computation may lead to higher costs (i.e., in gas) and thereby usage and especially allows for overflow or underflow problems to occur. For instance, the compiler may infer that a signed integer is the right data type for a certain variable, where an unsigned integer should be used	Unsafe type inference, Insecure inference
8I7	Missing Variable Initialization	This defect refers to the lack of initialization of variables that are used throughout the contract. Obviously, the effects can largely vary, depending on the variable itself and on the context in which is being used	Uninitialized local/state variable, Uninitialized state variables, Uninitialized state, Uninitialized local, Initialization of local variables, Initialization of state variables, Missing initialization
8I8	Missing Input Validation	Missing input validation refers to not verifying user input or parameters passed to functions before using them in logic or state changes.	Improper Input Validation
8I9	Function return type mismatch	The develop er implemented a function (starting from an interface), but it selected the wrong data type for the returns (e.g., it offers from what is specified in the interface)	
8I10	Parameter type mismatch	This issue refers to a divergence regarding the types of arguments used in a function that implements an interface. In this situation, even if the call is do ne with the right function name and arguments, the EVM considers it to be a non-existent function error	
8I11	Wrong Type of Function	In Solidity, it is possible to specify a type for each function. Functions of type view can read data from state variables but cannot modify them, and no gas costs are involved, whereas functions of type pure neither can read nor modify state variables and similarly to view functions, no gas costs are associated with this type of function. This vulnerability occurs when a developer sets the wrong type for a function. For instance, there is an issue reported in Ethereum's GitHub (Ethereum's GitHub, 2022) in which a state variable conversion operation (from storage to memory) inside a pure function results in a problem (i.e., to avoid this problem, the function type should be view)	
8I12	Redundant Functionality	Contracts that are written with redundant functionality increase code size and make maintainability difficult. In example scenario, a programmer creates a function and later (by best practices) end up creating the same functionality again in a new function. Heine identifies a vulnerability in the new function and flags it, but the old function with the defect is used by the caller.	
8J	Insecure contract upgrading	Even though upgrading contracts allow for fixing existing vulnerabilities, it counters the goal of an inherently decentralized smart contracts. If a contract developer becomes malicious or is compromised, the upgraded contract can be as malicious. The updating methods themselves are also hard to implement securely and without flaws.	Upgradable contract, Patching
8K	Insecure or incorrect logging or documentation	Inadequate logging and documentation can confuse, lead to logical errors or unexpected behavior, and hamper auditing or testing the code for vulnerabilities.	
8L	Hardcoded addresses	Using a hard-coded address could make the smart contract useless in the future if one of the addresses did change	
8M	Sends to zero address	This vulnerability happens when the smart contract does not check if the receiver is a zero address.	
8N	Double constructor	The 0.4.22 version of the solidity compiler allows smart contracts to have a double constructor. This could lead to a expected behavior in the contract.	
8O	Missing Constructor	A smart contract constructor is a function that is executed exactly once during the lifetime of a contract. It executes at deployment time and initializes state variables, performs a few necessary tasks that the specific contract requires, and sets the contract owner. If there is no constructor, the developer will have to implement such tasks manually, which is prone to security issues (e.g., variables may be set with inconsistencies or forgotten, which may result in security problems).	
8P	Extraneous Field Declaration	This vulnerability occurs when a struct or contract contains unnecessary field declarations, and these fields are publicly accessible or can be accidentally accessed or manipulated. This can lead to inconsistent state across nodes or unexpected behavior during upgrades	
8Q	Insecure Data Representation	The numbers to represent the credits (e.g., Ether) can be very large (i.e., 18 digits with many digits are difficult to read and review). Thus it is recommended that the programmer use the native resources of the language to make this representation (e.g., Solidity 1000000000000000000 for 1 ether).	
		***9 - Environ