ERC20 transfer and transferFrom: Should return a boolean. Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail. (See here)

ERC20 name, decimals, and symbol functions: Are present if used. These functions are optional in the ERC20 standard and might not be present. (See here)

ERC20 decimals returns a uint8: Several tokens incorrectly return a uint256. If this is the case, ensure the value returned is below 255. (See here)

ERC20 *approve* race-condition: The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens. (See here)

ERC777 hooks: ERC777 tokens have the concept of a hook function that is called before any calls to send, transfer, operatorSend, minting and burning. While these hooks enable a lot of interesting use cases, care should be taken to make sure they do not make external calls because that can lead to reentrancies. (See here)

Token Deflation via fees: Transfer and transferFrom should not take a fee. Deflationary tokens can lead to unexpected behavior. (See here)

Token Inflation via interest: Potential interest earned from the token should be taken into account. Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account. (See here)

Token contract avoids unneeded complexity: The token should be a simple contract; a token with complex code requires a higher standard of review. (See here)

Token contract has only a few non–token-related functions: Non–token-related functions increase the likelihood of an issue in the contract. (See here)

Token only has one address: Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g. *balances[token_address][msg.sender]* might not reflect the actual balance). (See here)

Token is not upgradeable: Upgradeable contracts might change their rules over time. (See here)

Token owner has limited minting capabilities: Malicious or compromised owners can abuse minting capabilities. (See here)

Token is not pausable: Malicious or compromised owners can trap contracts relying on pausable tokens. (See here)

Token owner cannot blacklist the contract: Malicious or compromised owners can trap contracts relying on tokens with a blacklist. (See here)

Token development team is known and can be held responsible for abuse: Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review. (See here)

No token user owns most of the supply: If a few users own most of the tokens, they can influence operations based on the token's repartition. (See here)

Token total supply is sufficient: Tokens with a low total supply can be easily manipulated. (See here)

Tokens are located in more than a few exchanges: If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token. (See here)

Token balance and Flash loans: Users understand the associated risks of large funds or flash loans. Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans. (See here)

Token does not allow flash minting: Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token. (See here)

ERC1400 permissioned addresses: Can block transfers from/to specific addresses. (See here)

ERC1400 forced transfers: Trusted actors have the ability to transfer funds however they choose. (See here)

ERC1644 forced transfers: Controller has the ability to steal funds. (See here)

ERC621 control of totalSupply: totalSupply can be changed by trusted actors (See here)

ERC884 cancel and reissue: Token implementers have the ability to cancel an address and move its tokens to a new address (See here)

ERC884 whitelisting: Tokens can only be sent to whitelisted addresses (See here)

Guarded launch via asset limits: Limiting the total asset value managed by a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via asset types: Limiting types of assets that can be used in the protocol initially upon launch and gradually expanding to other assets over time may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via user limits: Limiting the total number of users that can interact with a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. Initial users may also be whitelisted to limit to trusted actors before opening the system to everyone. (See here)

Guarded launch via usage limits: Enforcing transaction size limits, daily volume limits, per-account limits, or rate-limiting transactions may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via composability limits: Restricting the composability of the system to interface only with whitelisted trusted contracts before expanding to arbitrary external contracts may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via escrows: Escrowing high value transactions/operations with time locks and a governance capability to nullify or revert transactions may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via circuit breakers: Implementing capabilities to pause/unpause a system in extreme scenarios may reduce impact due to initial vulnerabilities or exploits. (See here)

Guarded launch via emergency shutdown: Implement capabilities that allow governance to shutdown new activity in the system and allow users to reclaim assets may reduce impact due to initial vulnerabilities or exploits. (See here)

System specification: Ensure that the specification of the entire system is considered, written and evaluated to the greatest detail possible. Specification describes how (and why) the different components of the system behave to achieve the design requirements. Without specification, a system implementation cannot be evaluated against the requirements for correctness.

System documentation: Ensure that roles, functionalities and interactions of the entire system are well documented to the greatest detail possible. Documentation describes what (and how) the implementation of different components of the system does to achieve the specification goals. Without documentation, a system implementation cannot be evaluated against the specification for correctness and one will have to rely on analyzing the implementation itself.

Function parameters: Ensure input validation for all function parameters especially if the visibility is external/public where (untrusted) users can control values. This is especially required for address parameters where maliciously/accidentally used incorrect/zero addresses can cause vulnerabilities or unexpected behavior.

Function arguments: Ensure that the arguments to function calls at the caller sites are the correct ones and in the right order as expected by the function definition.

Function visibility: Ensure that the strictest visibility is used for the required functionality. An accidental external/public visibility will allow (untrusted) users to invoke functionality that is supposed to be restricted internally.

Function modifiers: Ensure that the right set of function modifiers are used (in the correct order) for the specific functions so that the expected access control or validation is correctly enforced.

Function return values: Ensure that the appropriate return value(s) are returned from functions and checked without ignoring at function call sites, so that error conditions are caught and handled appropriately.

Function invocation timeliness: Externally accessible functions (*external*/*public* visibility) may be called at any time (or never). It is not safe to assume they will only be called at specific system phases (e.g. after initialization, when unpaused, during liquidation) that is meaningful to the system design. The reason for this can be accidental or malicious. Function implementation should be robust enough to track system state transitions, determine meaningful states for invocations and withstand arbitrary calls. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called atomically along with contract deployment to prevent anyone else from initializing with arbitrary values.

Function invocation repetitiveness: Externally accessible functions (*external*/*public* visibility) may be called any number of times. It is not safe to assume they will only be called only once or a specific number of times that is meaningful to the system design. Function implementation should be robust enough to track, prevent, ignore or account for arbitrarily repetitive invocations. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called only once.

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.

Function invocation arguments: Externally accessible functions (*external*/*public* visibility) may be called with any possible arguments. Without complete and proper validation (e.g. zero address checks, bound checks, threshold checks etc.), they cannot be assumed to comply with any assumptions made about them in the code.

Conditionals: Ensure that in conditional expressions (e.g. if statements), the correct variables are being checked and the correct operators, if any, are being used to combine them. For e.g. using || instead of && is a common error.

Access control specification: Ensure that the various system actors, their access control privileges and trust assumptions are accurately specified in great detail so that they are correctly implemented and enforced across different contracts, functions and system transitions/flows.

Access control implementation: Ensure that the specified access control is implemented uniformly across all the subjects (actors) seeking access and objects (variables, functions) being accessed so that there are no paths/flows where the access control is missing or may be side-stepped.

Missing modifiers: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. Ensure that such modifiers are present on all relevant functions which require that specific access control.

Incorrectly implemented modifiers: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that modifiers are implementing the expected checks on the correct roles/addresses with the right composition e.g. incorrect use of || instead of && is a common vulnerability while composing access checks.

Incorrectly used modifiers: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that correct modifiers are used on functions requiring specific access control enforced by that modifier.

Access control changes: Ensure that changes to access control (e.g. change of ownership to new addresses) are handled with extra security so that such transitions happen smoothly without contracts getting locked out or compromised due to use of incorrect credentials.

Comments: Ensure that the code is well commented both with NatSpec and inline comments for better readability and maintainability. The comments should accurately reflect what the corresponding code does. Stale comments should be removed. Discrepancies between code and comments should be addressed. Any TODO's indicated by comments should be addressed. Commented code should be removed.

Tests: Tests indicate that the system implementation has been validated against the specification. Unit tests, functional tests and integration tests should have been performed to achieve good test coverage across the entire codebase. Any code or parameterisation used specifically for testing should be removed from production code.

Unused constructs: Any unused imports, inherited contracts, functions, parameters, variables, modifiers, events or return values should be removed (or used appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

Redundant constructs: Redundant code and comments can be confusing and should be removed (or changed appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

ETH Handling: Contracts that accept/manage/transfer ETH should ensure that functions handling ETH are using *msg.value* appropriately, logic that depends on ETH value accounts for less/more ETH sent, logic that depends on contract ETH balance accounts for the different direct/indirect (e.g. *coinbase* transaction, *selfdestruct* recipient) ways of receiving ETH and transfers are reentrancy safe. Functions handling ETH should be checked extra carefully for access control, input validation and error handling.

Token Handling: Contracts that accept/manage/transfer ERC tokens should ensure that functions handling tokens account for different types of ERC tokens (e.g. ERC20 vs ERC777), deflationary/inflationary tokens, rebasing tokens and trusted/external tokens. Functions handling tokens should be checked extra carefully for access control, input validation and error handling.

Trusted actors: Ideally there should be no trusted actors while interacting with smart contracts. However, in guarded launch scenarios, the goal is to start with trusted actors and then progressively decentralise towards automated governance by community/DAO. For the trusted phase, all the trusted actors, their roles and capabilities should be clearly specified, implemented accordingly and documented for user information and examination.

Privileged roles and EOAs: Trusted actors who have privileged roles with capabilities to deploy contracts, change critical parameters, pause/unpause system, trigger emergency shutdown, withdraw/transfer/drain funds and allow/deny other actors should be addresses controlled by multiple, independent, mutually distrusting entities. They should not be controlled by private keys of EOAs but with Multisigs with a high threshold (e.g. 5-of-7, 9-of-11) and eventually by a DAO of token holders. EOA has a single point of failure.

Two-step change of privileged roles: When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. For e.g., in a single-step change, if the current admin accidentally changes the new admin to a zero-address or an incorrect address (where the private keys are not available), the system is left without an operational admin and will have to be redeployed.

Time-delayed change of critical parameters: When critical parameters of systems need to be changed, it is required to broadcast the change via event emission and recommended to enforce the changes after a time-delay. This is to allow system users to be aware of such critical changes and give them an opportunity to exit or adjust their engagement with the system accordingly. For e.g. reducing the rewards or increasing the fees in a system might not be acceptable to some users who may wish to withdraw their funds and exit.

Explicit over Implicit: While Solidity has progressively adopted explicit declarations of intent for e.g. with function visibility and variable storage, it is recommended to do the same at the application level where all requirements should be explicitly validated (e.g. input parameters) and assumptions should be documented and checked. Implicit requirements and assumptions should be flagged as dangerous.

Configuration issues: Misconfiguration of system components such contracts, parameters, addresses and permissions may lead to security issues.

Initialization issues: Lack of initialization, initializing with incorrect values or allowing untrusted actors to initialize system parameters may lead to security issues.

Cleanup issues: Missing to clean up old state or cleaning up incorrectly/insufficiently will lead to reuse of stale state which may lead to security issues.

Data processing issues: Processing data incorrectly will cause unexpected behavior which may lead to security issues.

Data validation issues: Missing validation of data or incorrectly/insufficiently validating data, especially tainted data from untrusted users, will cause untrustworthy system behavior which may lead to security issues.

Numerical issues: Incorrect numerical computation will cause unexpected behavior which may lead to security issues.

Accounting issues: Incorrect or insufficient tracking or accounting of business logic related aspects such as states, phases, permissions, roles, funds (deposits/withdrawals) and tokens (mints/burns/transfers) may lead to security issues.

Access control issues: Incorrect or insufficient access control or authorization related to system actors, roles, assets and permissions may lead to security issues.

Auditing/logging issues: Incorrect or insufficient emission of events will impact off-chain monitoring and incident response capabilities which may lead to security issues.

Cryptography issues: Incorrect or insufficient cryptography especially related to on-chain signature verification or off-chain key management will impact access control and may lead to security issues.

Error-reporting issues: Incorrect or insufficient detecting, reporting and handling of error conditions will cause exceptional behavior to go unnoticed which may lead to security issues.

Denial-of-Service (DoS) issues: Preventing other users from successfully accessing system services by modifying system parameters or state causes denial-of-service issues which affects the availability of the system. This may also manifest as security issues if users are not able to access their funds locked in the system.

Timing issues: Incorrect assumptions on timing of user actions, system state transitions or blockchain state/blocks/transactions may lead to security issues.

Ordering issues: Incorrect assumptions on ordering of user actions or system state transitions may lead to security issues. For e.g., a user may accidentally/maliciously call a finalization function even before the initialization function if the system allows.

Undefined behavior issues: Any behavior that is in the specification but is allowed in the implementation will result in unexpected outcomes which may lead to security issues.

External interaction issues: Interacting with external components (e.g. tokens, contracts, oracles) forces system to trust or make assumptions on their correctness/availability requiring validation of their existence and outputs without which may lead to security issues.

Trust issues: Incorrect or Insufficient trust assumption about/among system actors and external entities will lead to privilege escalation/abuse which may lead to security issues.

Gas issues: Incorrect assumptions about gas requirements especially for loops or external calls will lead to out-of-gas exceptions which may lead to security issues such as failed transfers or locked funds.

Dependency issues: Dependencies on external actors or software (imports, contracts, libraries, tokens etc.) will lead to trust/availability/correctness assumptions which if/when broken may lead to security issues.

Constant issues: Incorrect assumptions about system actors, entities or parameters being constant may lead to security issues if/when such factors change unexpectedly.

Freshness issues: Incorrect assumptions about the status of or data from system actors or entities being fresh (i.e. not stale), because of lack of updation or availability, may lead to security issues if/when such factors have been updated. For e.g., getting a stale price from an Oracle.

Scarcity issues: Incorrect assumptions about the scarcity of tokens/funds available to any system actor will lead to unexpected outcomes which may lead to security issues. For e.g., flash loans.

Incentive issues: Incorrect assumptions about the incentives of system/external actors to perform or not perform certain actions will lead to unexpected behavior being triggered or expected behavior not being triggered, both of which may lead to security issues. For e.g., incentive to liquidate positions, lack of incentive to DoS or grief system.

Clarity issues: Lack of clarity in system specification, documentation, implementation or UI/UX will lead to incorrect expectations/outcome which may lead to security issues.

Privacy issues: Data and transactions on the Ethereum blockchain are not private. Anyone can observe contract state and track transactions (both included in a block and pending in the mempool). Incorrect assumptions about privacy aspects of data or transactions can be abused which may lead to security issues.

Cloning issues: Copy-pasting code from other libraries, contracts or even different parts of the same contract may result in incorrect code semantics for the context being copied to, copy over any vulnerabilities or miss any security fixes applied to the original code. All these may lead to security issues.

Business logic issues: Incorrect or insufficient assumptions about the higher-order business logic being implemented in the application will lead to differences in expected and actual behavior, which may result in security issues.

Principle of Least Privilege: “Every program and every user of the system should operate using the least set of privileges necessary to complete the job” — Ensure that various system actors have the least amount of privilege granted as required by their roles to execute their specified tasks. Granting excess privilege is prone to misuse/abuse when trusted actors misbehave or their access is hijacked by malicious entities. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Separation of Privilege: “Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key” — Ensure that critical privileges are separated across multiple actors so that there are no single points of failure/abuse. A good example of this is to require a multisig address (not EOA) for privileged actors (e.g. owner, admin, governor, deployer) who control key contract functionality such as pause/unpause/shutdown, emergency fund drain, upgradeability, allow/deny list and critical parameters. The multisig address should be composed of entities that are different and mutually

Principle of Least Common Mechanism: "Minimize the amount of mechanism common to more than one user and depended on by all users" — Ensure that only the least number of security-critical modules/paths as required are shared amongst the different actors/code so that impact from any vulnerability/compromise in shared components is limited and contained to the smallest possible subset. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Fail-safe Defaults: "Base access decisions on permission rather than exclusion" — Ensure that variables or permissions are initialized to fail-safe default values which can be made more inclusive later instead of opening up the system to everyone including untrusted actors. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Complete Mediation: "Every access to every object must be checked for authority." — Ensure that any required access control is enforced along all access paths to the object or function being protected. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Economy of Mechanism: "Keep the design as simple and small as possible" — Ensure that contracts and functions are not overly complex or large so as to reduce readability or maintainability. Complexity typically leads to insecurity. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Open Design: “The design should not be secret” — Smart contracts are expected to be open-sourced and accessible to everyone. Security by obscurity of code or underlying algorithms is not an option. Security should be derived from the strength of the design and implementation under the assumption that (byzantine) attackers will study their details and try to exploit them in arbitrary ways. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Psychological Acceptability: “It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly” — Ensure that security aspects of smart contract interfaces and system designs/flows are user-friendly and intuitive so that users can interact with minimal risk. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Work Factor: “Compare the cost of circumventing the mechanism with the resources of a potential attacker” — Given the magnitude of value managed by smart contracts, it is safe to assume that byzantine attackers will risk the greatest amounts of intellectual/financial/social capital possible to subvert such systems. Therefore, the mitigation mechanisms must factor in the highest levels of risk. (See Saltzer and Schroeder's Secure Design Principles)

Principle of Compromise Recording: “Mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss” — Ensure that smart contracts and their accompanying operational infrastructure can be monitored/analyzed at all times (development/deployment/runtime) for minimizing loss from any compromise due to vulnerabilities/exploits. For e.g., critical operations in contracts should necessarily emit events to facilitate monitoring at runtime. (See Saltzer and Schroeder's Secure Design Principles)