General Review Approach:

All functions are `internal` except where explictly required to be `public`/`external`. [?]
There are no arithmetic overflows/underflows in math operations.
Using the OpenZeppelin safe math library [?].
Ether or tokens cannot be accidentally sent to the address `0x0`.
Conditions are checked using `require` before operations and state changes.
State is being set before and performing actions.
Protected from reentry attacks (A calling B calling A). [?]
Properly implements the ERC20 interface [?].
Only using modifier if necessary in more than one place.
All types are being explicitly set (e.g. using `uint256` instead of `uint`).
All methods and loops are within the maximum allowed gas limt.
There are no unnecessary initalizations in the constructor (remember, default values are set).
There is complete test coverage; every smart contract method and every possible type of input is being tested.
Performed fuzz testing by using random inputs.
Tested all the possible different states that the contract can be in.
Ether and token amounts are dealt in wei units.
The crowdsale end block/timestamp comes after start block/timestamp.
The crowdsale token exchange/conversion rate is properly set.
The crowdsale soft/hard cap is set.
The crowdsale min/max contribution allowed is set and tested.
The crowdsale whitelisting functionality is tested.
The crowdsale refund logic is tested.
Crowdsale participants are given their proportional token amounts or are allowed to claim their contribution.
The length of each stage of the crowdsale is properly configured (e.g. presale, public sale).
Specified which functions are intented to be controlled by the owner only (e.g. pausing crowdsale, progressing crowdsale stage, enabling distribution of tokens, etc..).
The crowdsale vesting logic is tested.
The crowdsale has a fail-safe mode that when enabled by owner, restricts calls to function and enables refund functionality.
The crowdsale has a fallback function in place if it makes reasonable sense.

The fallback function does not accept call data or only accepts prefixed data to avoid function signature collisions.
Imported libraries have been previously audited and don't contain dyanmic parts that can be swapped out in future versions which can be be used maliciously. [?]
Token transfer statements are wrapped in a `require`.
Using `require` and `assert` properly. Only use `assert` for things that should never happen, typically used to validate state after making changes.
Using `keccak256` instead of the alias `sha3`.
Protected from ERC20 short address attack. [?].
Protected from recursive call attacks.
Arbitrary string inputs have length limits.
No secret data is exposed (all data on the blockchain is public).
Avoided using array where possible and using mappings instead.
Does not rely on block hashes for randomness (miners have influence on this).
Does not use `tx.origin` anywhere. [?]
Array items are shifted down when an item is deleted to avoid leaving a gap.
Use `revert` instead of `throw`.
Functions exit immediately when conditions aren't meant.
Using the latest stable version of Solidity.
Prefer pattern where receipient withdrawals funds instead of contract sending funds, however not always applicable.
Resolved warnings from compiler.

Variables

`V1` - Can it be `internal`?
`V2` - Can it be `constant`?
`V3` - Can it be `immutable`?
`V4` - Is its visibility set? (SWC-108)
`V5` - Is the purpose of the variable and other important information documented using natspec?
`V6` - Can it be packed with an adjacent storage variable?
[ ] `V7` - Can it be packed in a struct with more than 1 other variable?
`V8` - Use full 256 bit types unless packing with other variables.
`V9` - If it's a public array, is a separate function provided to return the full array?
`V10` - Only use `private` to intentionally prevent child contracts from accessing the variable, prefer `internal` for flexibility.

Structs

`S1` - Is a struct necessary? Can the variable be packed raw in storage?
`S2` - Are its fields packed together (if possible)?
`S3` - Is the purpose of the struct and all fields documented using natspec?

Functions

`F1` - Can it be `external`?
`F2` - Should it be `internal`?
`F3` - Should it be `payable`?
`F4` - Can it be combined with another similar function?
`F5` - Validate all parameters are within safe bounds, even if the function can only be called by a trusted users.
`F6` - Is the checks before effects pattern followed? (SWC-107)
[ ]- `F7` - Check for front-running possibilities, such as the approve function. (SWC-114)
`F8` - Is insufficient gas griefing possible? (SWC-126)
`F9` - Are the correct modifiers applied, such as `onlyOwner`/`requiresAuth`?
`F10` - Are return values always assigned?
`F11` - Write down and test invariants about state before a function can run correctly.
`F12` - Write down and test invariants about the return or any changes to state after a function has run.
`F13` - Take care when naming functions, because people will assume behavior based on the name.
`F14` - If a function is intentionally unsafe (to save gas, etc), use an unwieldy name to draw attention to its risk.
`F15` - Are all arguments, return values, side effects and other information documented using natspec?
`F16` - If the function allows operating on another user in the system, do not assume `msg.sender` is the user being operated on.
`F17` - If the function requires the contract be in an uninitialized state, check an explicit `initialized` variable. Do not use `owner == address(0)` or other similar checks as substitutes.
`F18` - Only use `private` to intentionally prevent child contracts from calling the function, prefer `internal` for flexibility.
`F19` - Use `virtual` if there are legitimate (and safe) instances where a child contract may wish to override the function's behavior.

Modifiers

`M1` - Are no storage updates made (except in a reentrancy lock)?
`M2` - Are external calls avoided?
`M3` - Is the purpose of the modifier and other important information documented using natspec?

Code

`C1` - Using SafeMath or 0.8 checked math? (SWC-101)
`C2` - Are any storage slots read multiple times?
`C3` - Are any unbounded loops/arrays used that can cause DoS? (SWC-128)
`C4` - Use `block.timestamp` only for long intervals. (SWC-116)
`C5` - Don't use block.number for elapsed time. (SWC-116)
`C7` - Avoid delegatecall wherever possible, especially to external (even if trusted) contracts. (SWC-112)
`C8` - Do not update the length of an array while iterating over it.
`C9` - Don't use `blockhash()`, etc for randomness. (SWC-120)
`C10` - Are signatures protected against replay with a nonce and `block.chainid` (SWC-121)
`C11` - Ensure all signatures use EIP-712. (SWC-117 SWC-122)
`C12` - Output of `abi.encodePacked()` shouldn't be hashed if using >2 dynamic types. Prefer using `abi.encode()` in general. (SWC-133)
`C13` - Careful with assembly, don't use any arbitrary data. (SWC-127)
`C14` - Don't assume a specific ETH balance. (SWC-132)
`C15` - Avoid insufficient gas griefing. (SWC-126)
`C16` - Private data isn't private. (SWC-136)
`C17` - Updating a struct/array in memory won't modify it in storage.
`C18` - Never shadow state variables. (SWC-119)
`C19` - Do not mutate function parameters.
`C20` - Is calculating a value on the fly cheaper than storing it?
`C21` - Are all state variables read from the correct contract (master vs. clone)?
`C22` - Are comparison operators used correctly (`>`, `<`, `>=`, `<=`), especially to prevent off-by-one errors?
`C23` - Are logical operators used correctly (`==`, `!=`, `&&`, `||`, `!`), especially to prevent off-by-one errors?
`C24` - Always multiply before dividing, unless the multiplication could overflow.
`C25` - Are magic numbers replaced by a constant with an intuitive name?

`C26` - If the recipient of ETH had a fallback function that reverted, could it cause DoS? (SWC-113)
`C27` - Use SafeERC20 or check return values safely.
`C28` - Don't use `msg.value` in a loop.
`C29` - Don't use `msg.value` if recursive delegatecalls are possible (like if the contract inherits `Multicall`/`Batchable`).
`C30` - Don't assume `msg.sender` is always a relevant user.
`C31` - Don't use `assert()` unless for fuzzing or formal verification. (SWC-110)
`C32` - Don't use `tx.origin` for authorization. (SWC-115)
`C33` - Don't use `address.transfer()` or `address.send()`. Use `.call.value(...)("")` instead. (SWC-134)
`C34` - When using low-level calls, ensure the contract exists before calling.
`C35` - When calling a function with many parameters, use the named argument syntax.
`C36` - Do not use assembly for create2. Prefer the modern salted contract creation syntax.
`C37` - Do not use assembly to access chainid or contract code/size/hash. Prefer the modern Solidity syntax.
`C38` - Use the `delete` keyword when setting a variable to a zero value (`0`, `false`, `""`, etc).
`C39` - Comment the "why" as much as possible.
`C40` - Comment the "what" if using obscure syntax or writing unconventional code.
`C41` - Comment explanations + example inputs/outputs next to complex and fixed point math.
`C42` - Comment explanations wherever optimizations are done, along with an estimate of much gas they save.
`C43` - Comment explanations wherever certain optimizations are purposely avoided, along with an estimate of much gas they would/wouldn't save if implemented.
`C44` - Use `unchecked` blocks where overflow/underflow is impossible, or where an overflow/underflow is unrealistic on human timescales (counters, etc). Comment explanations wherever `unchecked` is used, along with an estimate of how much gas it saves (if relevant).
`C45` - Do not depend on Solidity's arithmetic operator precedence rules. In addition to the use of parentheses to override default operator precedence, parentheses should also be used to emphasise it.
`C46` - Expressions passed to logical/comparison operators (`&&`/`||`/`>=`/`==`/etc) should not have side-effects.

`C47` - Wherever arithmetic operations are performed that could result in precision loss, ensure it benefits the right actors in the system, and document it with comments.
`C48` - Document the reason why a reentrancy lock is necessary whenever it's used with an inline or `@dev` natspec comment.
`C49` - When fuzzing functions that only operate on specific numerical ranges use modulo to tighten the fuzzer's inputs (such as `x = x % 10000 + 1` to restrict from 1 to 10,000).
`C50` - Use ternary expressions to simplify branching logic wherever possible.
`C51` - When operating on more than one address, ask yourself what happens if they're the same.

External Calls

`X1` - Is an external contract call actually needed?
`X2` - If there is an error, could it cause DoS? Like `balanceOf()` reverting. (SWC-113)
`X3` - Would it be harmful if the call reentered into the current function?
`X4` - Would it be harmful if the call reentered into another function?
`X5` - Is the result checked and errors dealt with? (SWC-104)
`X6` - What if it uses all the gas provided?
`X7` - Could it cause an out-of-gas in the calling contract if it returns a massive amount of data?
`X8` - If you are calling a particular function, do not assume that `success` implies that the function exists (phantom functions).

Static Calls

`S1` - Is an external contract call actually needed?
`S2` - Is it actually marked as view in the interface?
`S3` - If there is an error, could it cause DoS? Like `balanceOf()` reverting. (SWC-113)
`S4` - If the call entered an infinite loop, could it cause DoS?

Events

`E1` - Should any fields be indexed?
`E2` - Is the creator of the relevant action included as an indexed field?

`E3` - Do not index dynamic types like strings or bytes.
`E4` - Is when the event emitted and all fields documented using natspec?
`E5` - Are all users/ids that are operated on in functions that emit the event stored as indexed fields?

Contract

`T1` - Use an SPDX license identifier.
`T2` - Are events emitted for every storage mutating function?
`T3` - Check for correct inheritance, keep it simple and linear. (SWC-125)
`T4` - Use a `receive() external payable` function if the contract should accept transferred ETH.
`T5` - Write down and test invariants about relationships between stored state.
`T6` - Is the purpose of the contract and how it interacts with others documented using natspec?
`T7` - The contract should be marked `abstract` if another contract must inherit it to unlock its full functionality.
`T8` - Emit an appropriate event for any non-immutable variable set in the constructor that emits an event when mutated elsewhere.
`T9` - Avoid over-inheritance as it masks complexity and encourages over-abstraction.
`T10` - Always use the named import syntax to explicitly declare which contracts are being imported from another file.
`T11` - Group imports by their folder/package. Separate groups with an empty line. Groups of external dependencies should come first, then mock/testing contracts (if relevant), and finally local imports.
`T12` - Summarize the purpose and functionality of the contract with a `@notice` natspec comment. Document how the contract interacts with other contracts inside/outside the project in a `@dev` natspec comment.

Project

`P1` - Use the right license (you must use GPL if you depend on GPL code, etc).
`P2` - Unit test everything.
`P3` - Fuzz test as much as possible.
`P4` - Use symbolic execution where possible.
`P5` - Run Slither/Solhint and review all findings.

DeFi

`D1` - Check your assumptions about what other contracts do and return.
`D2` - Don't mix internal accounting with actual balances.
`D3` - Don't use spot price from an AMM as an oracle.
`D4` - Do not trade on AMMs without receiving a price target off-chain or via an oracle.
`D5` - Use sanity checks to prevent oracle/price manipulation.
`D6` - Watch out for rebasing tokens. If they are unsupported, ensure that property is documented.
`D7` - Watch out for ERC-777 tokens. Even a token you trust could preform reentrancy if it's an ERC-777.
`D8` - Watch out for fee-on-transfer tokens. If they are unsupported, ensure that property is documented.
`D9` - Watch out for tokens that use too many or too few decimals. Ensure the max and min supported values are documented.
`D10` - Be careful of relying on the raw token balance of a contract to determine earnings. Contracts which provide a way to recover assets sent directly to them can mess up share price functions that rely on the raw Ether or token balances of an address.
`D11` - If your contract is a target for token approvals, do not make arbitrary calls from user input.