Beirao's Checklist

Summary

Author: @0xBeirao

Source: The ultimate security checklist

Info The list is more likely to be used as a reminder of the most common vulnerabilities and questions to ask yourself during an audit. It is not a complete list of all possible vulnerabilities.

> My aim with this article is to equip you with a comprehensive checklist of questions that auditors should be asking themselves during their smart contract audits. Each lines is voluntary short and can be used as a reminder. ~ Beirao

Checklist

`external` / `public` functions

[F-01] - Should it be `external`/`public`?

[F-02] - Does this function need to be restricted ? (`onlyOwner` ?)

[F-03] - Are the inputs checked ?

[F-04] - Is there any front run opportunities ?

- Beware of sandwich attack on Vaults and DEXes
- Tx must not be order dependent
- Auctions with a fixed endtime has the known vulnerability of being bid on at the last block
- Pausing mechanisme can be frontrun

[F-05] - Is this function making a `call()` or transfering ERC20 tokens ?

- Is the call address white listed ?
- Reentrance ?

[F-06] - Is that function payable or transfering funds ?

- Check if `msg.value == amount`

[F-07] - Is the code comments coherent with the implementation ?

[F-08] - Can edge case inputs (0, max) result of an unexpected behaviour ?

[F-09] - Requirement check for external call parameters can be to strong and not allowing all good possible inputs

[F-10] - When array of address/id in calldata: what happen if there are multiple time the same address in the array?

External call

[E-01] - Is an external contract call actually needed?

[E-02] - Is the address called whitelisted ?

[E-03] - Mistrust when there is a fixed gas amount in a `.call()` and check if the gas left is enough

[E-04] - Grief attack is possible when calling a unknown address by passing huge amount of data into the call ⇒ use inline assembly.

[E-05] - A call to an address that do not exist return true : is the existence of the address checked ?

[E-06] - Use the check, effect, interaction pattern. Only If necessary use a reentrancyGuard but beware of cross contract reentrancy

[E-07] - For sending ETH don't use `transfer()` or `send()` and instead use `call()`

[E-08] - `msg.value` not checked can have result in unexpected behaviour

[E-09] - Delegate calls that do not interact with stateless type of contract (library) should be triple check

[E-10] - Never delegate call to an untrusted contract

[E-11] - If the recipient of ETH had a fallback function that reverts, could it cause DoS?

[E-12] - Could it cause an out-of-gas in the calling contract if it returns a massive amount of data? Can the external call be manipulated to cause DoS?

[E-13] - Would it be harmful if the call reentered into the current function?

[E-14] - Would it be harmful if the call reentered into another function?

[E-15] - What if it uses all the gas provided?

[E-16] - Could it cause an out-of-gas in the calling contract if it returns a massive amount of data?

[E-17] - Care when using `msg.value` in a multi call.

Maths

[M-01] - Is the calculation even correct ?

[M-02] - Are the fees correctly calculated ?

[M-03] - Is there precision lost ? (especially for year/month/day calculation)

[M-04] - Regular expression like `1 day` is a `uint24` meaning that operation with these expression will be cast on `uint24` and potentially overflow

[M-05] - Always * before /

[M-06] - Is a library use to round results ?

[M-07] - Div by 0 ?

[M-08] - Even in `>0.8.0` take care that a variable can not find themselves in under or overflow that will cause revert

[M-09] - Assign a negative value to an uint reverts

[M-10] - `unchecked{}` need to be check

[M-11] - When < or > check if it should not be ≤ or ≥

[M-12] - Inline assembly math considerations

- div(x, 0) == 0
- Operations can overflow/underflow. You should add checks if necessary.

When for/while loop

[L-01] - Is the first iteration a problem ?

[L-02] - DOS

- Is there a call inside the loop ?
- Is the number of iteration limited ? ⇒ can an attacker add elements at no cost ?

[L-03] - Don't use `msg.value` in a loop.

Control access

[A-01] - Centralization risk

- Executors can perform token transfers on behalf of user ?
- Reclaiming / withdrawing any tokens ?
- Total upgradeability ?
- Instant parameters change (no timelock) ?
- Can pause freely ?
- Can rug user and steal assets ?
- Bugs that lead restricted function to steal all the assets is a centralization risk

[A-02] - Can corrupted owner destroy the protocol ?

[A-03] - Is a features lacking access controls ?

[A-04] - Some addresses need a whitelist ?

[A-05] - Is the owner change a two step process ?

[A-06] - Are critical functions accesible ? (like `mint()`)

Vault

[V-01] - Can transferring ERC20 or ETH directly break something ?

[V-02] - Is the vault balance tracked internally ?

[V-03] - Can the 1st deposit raise a problem ?

[V-04] - Is this vault taking into consideration that some ERC20 token are not 18 decimals ?

[V-05] - Is the fee calculation correct ?

[V-06] - What if only 1 wei remain in the pool ?

[V-07] - On vault with strategies implemented :

- flash deposit-harvest-withdraw attacks are possible?
- How the vault behave when locked fund are put in a strategy?
- Are losses handled ? (always should)
- What happen in case of a black swan event ? (protocol implemented in the strategy get hacked)
- Look at token-specific/protocol risks implemented in strategies :
    - For protocol
        - Pause ?
        - Emergency withdrawal ?
        - Depreciation ?
    - For tokens
        - All weird implementations

[V-08] - Can we manipulate the conversion rate between shares and underlying ? (here)

ERC20 (More edge cases here : weird-erc20)

[FT-01] - Using Safe functions ?

[FT-02] - Is the USDT approve race condition a problem ? (especially on DEXes)

[FT-03] - Is the decimals difference between ERC20 a problem ?

[FT-04] - The contract implement a white/blacklist ? or some kind of addresses check ?

[FT-05] - Multiple-address token can be a problem

[FT-06] - Are fees on transfer raising an issue ?

[FT-07] - When their is a `balanceOf(address(this))` check if manually sending tokens can break something ?

[FT-08] - ERC777 tokens can hook bad things on transfers (before and after transfers)

[FT-09] - Solmate `ERC20.safeTransferLib` do not check the contract existence

[FT-10] - `IERC20(address(0)).decimals()` will revert and cause DOS

[FT-11] - Flash mint increase the token supply

[FT-12] - Some tokens revert on 0 transfers : can cause DoS

[FT-13] - A contract is a target for token approvals ? Do not make arbitrary calls from user input

[FT-13] - `DOMAIN_SEPARATOR()` function in ERC2612 missing?

[FT-14] - Some ERC20 reverts when sending tokens to certain addresses (LUSD)

ERC721

[NFT-01] - Safe functions must be used

[NFT-02] - `SafeMint()` and `SafeTransfers()` from the ERC721 OpenZepplin contract as a callback that can reenter

[NFT-03] - OpenZeppelin implementation of ERC721 and ERC1155 vulnerable to reentrancy attacks, since safeTransferFrom functions perform an external call to the user address (onReceived)

[NFT-04] - Most of the time the 'from' parameter of `nft.transferFrom()` should be `msg.sender`. Otherwise hacker can take advantage of other user's appovals and rob them

Proxies

[P-01] - Is there a constructor ? (should not have one)

[P-02] - Is the modifier `initializer` added on "initialization()" function : deployer must always call the initialization function. (check deployment scripts if any)

[P-03] - if any contract inheritance has a constructor (erc20, reentrancyGuard, Pausable...) is used : use the upgreadable version for initialize

[P-04] - Check that `authorizeUpgrade()` is properly secured if UUPS

[P-05] - Can the new implementation cause storage collision with the old one ? :If children inherit from parents ⇒ set a gap

[P-06] - Was `disableInitializers()` called in the constructor ?

[P-07] - `selfdestruct` and `delegatecall` should not be used inside implementation contracts

[P-08] - The values in immutable variables are not preserved between upgrades

[P-09] - The order of storage variables declared or their type cannot be changed in-between upgrades

[P-10] - Rugs:

- Beware of function clashing (here)
- Beware of metamorphic Contract Rug Vulnerability (here)

Signature (ref)

[S-01] - Are signatures protected against replay with a `nonce` and `block.chainid` ? And ensure all signatures use EIP-712.

[S-02] - Signature Malleability : do not use `escrevover()` but use the `openzepplin/ECDSA.sol` (The last version should be used here)

[S-03] - Check if the returned public key matches expected public key.

[S-04] - Check if the Signature is used from the right person (if not everyone should be able to use it)

[S-05] - Check if the deadline is not expired (if it is not needed that the signature is working forever)

Locktime for staking

[LS-01] - Check if a user can help other users to reduce the time lock by stacking their by stacking the tokens for them

[LS-02] - Check if a contract can wrap the collateral token to sell 100% liquid already stacked tokens

[LS-03] - Can rewards be delayed in payout, or claimed too early?

[LS-04] - Can deposited assets get stuck in the protocol (partially or fully) or be improperly delayed in withdrawal?

[LS-05] - If the payout is in a different asset or currency, can the value of it be manipulated within the scope of the smart contract in question? This is relevant if the protocol mints its own tokens to reward liquidity providers or stakers.

Account abstraction (ref)

[AA-01] - DoS possible when using paymaster: because tx are free

AMMs

[AMM-01] - Is there a slippage protection ?

[AMM-02] - Working for every token decimals ? type of tokens ?

[AMM-03] - The AMM should either support FoT or check if FoT (attack ex)

[AMM-04] - Rebasing tokens can break the AMM: build a blacklist

Lending (ref)

[LEN-01] - Can the position be liquidated if the loan is not paid back or the collateral does not drops below the threshold?

[LEN-02] - Can a user profit from self-liquidation?

[LEN-03] - If a token/transfer/add collateral is paused/bricked for a moment, can the user get liquidated even if he wants to add more money?

[LEN-04] - Will liquidation work in the event of a large price drop?

[LEN-05] - Can the liquidator receive less than expected?

[LEN-06] - Are liquidations suspended? What happen when unpause? (pause = solvency risk)

[LEN-07] - If a token/transfer/add collateral is paused/bricked for a moment, can the user get liquidated even if he wants to add more money?

[LEN-08] - Is griefing possible by front running by slightly increasing his collateral?

[LEN-09] - Are all positions properly incentivized for liquidation? even the small ones?

Multichains (ref)

[MC-01] - Block time is not the same on different chains : Look for hardcoded time values dependent on the `block.number` that may only be valid on Mainnet.

[MC-02] - Block production may not be constant

[MC-03] - `PUSH0` is not supported on all chains : `>=0.8.20` should be avoided on multichain app (here)

[MC-04] - Verify that the EVM opcodes and operations used by the protocol are compatible on all chains : Arbitrum, Optimism

[MC-05] - Verify that the expected behaviour of `tx.origin` and `msg.sender` holds on all deployed chains here)

[MC-06] - Analyze attack vectors that require low gas fees or where a considerable numbers of transactions have to be executed (here)

[MC-07] - ERC20 decimals can change across chains (here)

[MC-08] - Check the upgradability of contracts on different chains and evaluate their implications : ex USDT upgradable en Polygon but not on Ethereum

[MC-09] - Look for cross-chain messages implementations and verify the correct permissions and functionality considering all the actors involved

[MC-10] - When a protocol is cross chain make sure all compatible chains are whitelisted : be able to send a message from a unsupported chain can lead to unexpected result

[MC-11] - Check the compatibility of the contracts when being deployed to zkSync Era (here)

[MC-12] - Use evm-diff.com and check here for chains comparaison

Merkle trees

[MT-01] - Merkle trees are front-runnable

[MT-02] - Are leafs hashed with the claimable address inside?

[MT-03] - For airdrops: Does the `claim()` function rely on `msg.sender` to validate the mint?

[MT-04] - What happen if we pass the zero hash?

[MT-05] - What happen if the exact same proof exist twice in the tree?

General tips

[G-01] - For logic implemented several times, see if there are any differences and then standardize the logic.

[G-02] - Timelocks should be implemented for every protocol upgrade/change. (to let users exit if they don't agree)

[G-03] - Force-feeding a Smart Contract. Beware of miss use of `.balanceOf(this)`

- Self-destruct
- Deterministic address can be force feed before deployment

- (Coinbase Transactions : The attacker can start proof-of-work mining and set the target address to receive the reward)

[G-04] - Beware of Dos

- Never `call()` into for loop ⇒ pull payments
- Sensible withdraw logic should be done externally by the user
- Beware of Block Stuffing attack when the contract need to do an action within a certain time period.
- Maybe if there is a timelock a attacker can brick the logic at no cost

[G-05] - Beware of Oracle manipulation: Don't use spot price from an AMM as an oracle.

[G-06] - When deleting a structure you should delete mapping inside first

[G-07] - Be careful of relying on the raw token balance of a contract to determine earnings

[G-08] - Take care `if (receiver == caller)` can have unexpected behaviour

[G-09] - When pause mechanism :

- can pause something that should not be pause (e.g liquidation)
- Check can brick the contract
- Check if `whenNotPaused` is well implemented on every functions where it needs to be

[G-10] - When seftdestruct beware of`CREATE2`reinitialize trick (here)

[G-11] - Semantic Overload need to be avoided or checked intensively

[G-12] - Check the doc and the code searching for inconsistencies

[G-13] - If available deployment script should be checked

[G-14] - It's possible to bypass contract size check by implementing the logic in the constructor.

[G-15] - Hash collisions are possible with `abi.encodePacked` when using >2 dynamic types (here)

[G-16] - Check if there is problematic bug in the version used at this (here)

[G-17] - NoReentrance modifier should be placed before every other modifiers

[G-18] - `try/catch` block can always fail by not suppling enough gas (here)

[G-19] - Reorg can create a loss of fund (use create2 instead of create) (here)

[G-20] - Beware of cross contract reentrancy (here)

[G-21] - Beware of read-only contract reentrancy

[G-22] - When there is a multi agent system: what if all agents are the same person? (Ex : self liquidation)

[G-23] - DoS with (Unexpected) revert (here)

[G-24] - `msg.value` multicall can hide a vulnerability

[G-25] - Check for correct inheritance, keep it simple and linear.

[G-26] - Check for code asymmetries. (`withdraw` function should (usually) undo all the state changes of a `deposit` function ref)

[G-27] - When a EIP is implemented. Are all EIP recommendations being followed?

[G-28] - Is `block.timestamp` only use for long intervals?

[G-29] - Are comparison operators used correctly (`>`, `<`, `>=`, `<=`)?

[G-30] - Are logical operators used correctly (`==`, `!=`, `&&`, `||`, `!`)?

[G-31] - Unexpected addresses (provide a `receiver` address pointing to another contract in the system)

DeFi integrations+

Gnosis Safe integration

[GS-01] - make sure your modules execute Guard's hooks (`checkTransaction()`, `checkAfterExecution()`)

[GS-01] - `execTransactionFromModule()` does not increment the Safe nonce: if modules are used don't rely on it for signatures.

LSD integration (ref)

stETH

[LSD-01] - `stETH` is a rebasing token: using `wstETH` is simpler for DeFi integration

[LSD-02] - Care if your app convert tokens for `stETH` to `wstETH`: the rebasing have to be handled

[LSD-03] - Withdraw `stETH`/`wstETH` bring overhead (queue time, receives a NFT, withdrawal amount limits)

rETH

[LSD-04] - `burn()` function can revert is there is not enough ether in the `RocketDepositPool` contract

[LSD-05] - The rate `rETH`/`ETH` can decrease in case of a slashing event

[LSD-06] - Remain vigilant about the risk of consensus attacks on RPL nodes, where nodes may submit incorrect exchange rate data.

cbETH

[LSD-06] - Blacklisting feature on transfers, approvals, mints, and burns.

[LSD-07] - The `cbETH`/`ETH` rate can be changed by few addresses thanks to the `onlyOracle` modifier

[LSD-08] - The `cbETH`/`ETH` rate can decrease

sfrxETH

[LSD-07] - `sfrxETH` may detach from `frxETH` during reward transfers by the Frax team's multi-sig contract

[LSD-08] - For now the `sfrxETH`/`ETH` rate can't decrease but thsi may change in the future

LayerZero integration (Official LZ integration check list here)

[LZ-01] - What is used ? Blocking or none blocking transactions ? Blocking can result of a DoS (here)

[LZ-02] - Gas should be estimated correctly, otherwise the cross-chain message will fail

[LZ-03] - The *`_debitFrom`* function in ONFT must verify whether the specified owner is the owner of the tokenId passed in the parameters and whether the sender is allowed to transfer this token. (ref)

[LZ-04] - If *LzApp* inherited, remember to use *`_lzSend`* function instead of directly calling *`lzEndpoint.send`* function,

[LZ-05] - The User Application should implement the *`ILayerZeroUserApplicationConfig`* interface, including the *forceResumeReceive* function which, in the worst case can allow the owner/multisig to unblock the queue of messages if something unexpected happens.

[LZ-06] - If you don't want to outsource your security, it is recommended to configure the applications to not use the default configuration. The default contracts are upgradeable and can be changed by the LayerZero team.

[LZ-07] - Choose the correct confirmations number. Depending on the chain and past reorg.

Chainlink integration (deep dive here)

VRF (VRF Security Considerations)

[CL-01] - When chainlink VRF is call, make sure that all parameters passed are verified else fullfillRandomWord will not revert and return a bad value

[CL-01] - Chainlink vrf get in pending if there is not enough LINK in the subscription. Meaning that when the subscription operates again the tx can be frontrun

[CL-01] - Choose a high enough request confirmation number for chain re-orgs (here)

[CL-01] - VFR calls are front-runnable : the betting phase must be closed before the VRF call

Pricefeed

[CL-01] - Pricefeed may not be supported in the future. To prevent wrong price to be used you can check the last update timestamp : `require(block.timestamp - supplyUpdatedAt <= MAX_DELAY)` : Check the heartbeat of each price feed to adjust the `MAX_DELAY` variable.

[CL-02] - Rollup sequencer can go offline as a result of an not updated response (here)

[CL-03] - Check that the price feed for the desired pair is supported on all of the deployed chains.

[CL-04] - Is the pricefeed heartbeat adapted for the use case? too slow?

[CL-05] - Are different price feeds used? Does the code handle different decimal precision?

[CL-06] - The pricefeed address is hardcoded?

- Is this address even correct ?
- Beware that this pricefeed may become deprecated in the future? (especially for pricefeed not very common)

[CL-07] - Oracle price updates can be front-run (Angle' solution)

[CL-08] - Are oracle revert DoS handled ? (wrapping calls to Oracles in try/catch blocks and provide an alternative solution)

[CL-09] - Are ETH pricefeed use for stETH ? or BTC pricefeed used for WBTC ? The depeg risk must be addressed

[CL-10] - Oracle returns incorrect price during flash crashes : To help mitigate such an attack on-chain, smart contracts could check that minAnswer < receivedAnswer < maxAnswer.

Uniswap integration

[U-01] - Hard coded slippage is forbidden

[U-02] - Proper slippage strategy should be implemented

- No Expiration Deadline ?
- Incorrect Slippage Calculation ?
- Mismatched Slippage Precision ?
- Is the slippage calculated on-chain ? On-chain slippage calculation can be manipulated
- Never hardcode slippage

[U-03] - Is there a refunds after a swaps ?

[U-04] - AMM pools `token0` and `token1` order differ depending of the chain

[U-05] - Are pools called whitelisted? if not, check if the pool has the right factory address

[U-06] - Don't rely on pool reserve since they can be manipulated

[U-07] - Don't rely on `pool.swap()`: Always use the Router contract

AAVE/Compound integration (help to remember concepts)

[AC-01] - What happen if the utilisation rate is to high and collateral can't be retrieved?

[AC-02] - What happen if the protocol is paused?

[AC-03] - What happen if the pool become deprecated?

[AC-04] - What happen if assets you lend/borrow are within the same eMode category?

[AC-05] - Flashloans on Aave inflate the pool index (a maximum of 180 flashloans can be performed within a block).

[AC-06] - Does the protocol properly implement AAVE/COMP reward claims?

[AC-06] - The cETH token contract has no `underlying()`

[AC-07] - Borrowing a AAVE siloed asset will prohibits you from borrowing any other asset.
Use `getSiloedBorrowing(address asset)` to know.

[AC-06] - On AAVE, what happen if you reach to maximum debt on a isolated asset? (DOS?)