Owen's Checklist

Summary

Author: Owen

Source:

- High-level checklist
- Full Checklist

High-level checklist

17 Common attack vectors

- Frontrunning/backrunning
- Using very small amounts as inputs (e.g. 1 wei)
- Passing zero as an input
- Using contracts that cannot accept ether
- Gas griefing with external calls
- Weird ERC20 tokens (fees, 777, return values, etc...)
- Price manipulation
- Blacklisted ERC20 addresses
- Potential overflow/underflow
- Block re-orgs
- Reentrancy (721, inter-function, inter-contract, inter-system (read-only))
- Sybil attacks on incentives/tokenomics
- Flash loans (even flash mints e.g. Dai)
- Accepting any data from an arbitrary address (Malicious bytes)
- Inflating internal accounting by sending tokens to the system
- Forced precision loss when precision really matters (min balance checks etc...)
- Addresses that might be empty at one point, yet house contract code at another

5 Bonuses:

- Reverting (external calls I can make revert, inputs I can use to cause a revert)
- Unexpected addresses (provide a `receiver` address pointing to another contract in the system)
- Selector clashing
- Signatures (replay, malleability, recover to 0 address etc...)
- Hash collision (encodePacked)

Checklist

- Block re-orgs
- Block Timestamp Manipulation
- Delegate call to Untrusted Callee

- Denial Of Service (DOS) (revert or infinite gas consumption, Underflow / Overflow, Unexpected Ether)
- Events emission on critical functions
- Flash loans (even flash mints e.g. DAI, oracle manipulation)
- Floating Point Arithmetic
- Hash collision (encodePacked)
- Inflating internal accounting by sending tokens to the system
- Improper Array Deletion
- Lack of Access Controls
- Message call with hard-coded gas
- Misinitialization of contracts (ownership/proxy)
- Price manipulation
- Reverting (external calls I can make revert, inputs I can use to cause a revert)
- Selector clashing
- Unexpected addresses (provide a `receiver` address pointing to another contract in the system)
- Unsafe Ownership transfer
- Unsafe Typecast
- Use of deprecated solidity Functions
- Using contracts that cannot accept ether
- Using very small amounts as inputs (e.g. 1 wei)
- Code asymmetries

  In many projects, there should be some symmetries for different functions. For instance, a `withdraw` function should (usually) undo all the state changes of a `deposit` function and a `delete` function should undo all the state changes of the corresponding `add` function. Asymmetries in these function pairs (e.g., forgetting to unset a field or to subtract from a value) can often lead to undesired behavior.

- Calling a function X times with value Y == Calling it once with value XY

  For some functions (e.g., a `deposit` function), the resulting state should be (almost) the same when the function is called X times with a value Y or when it is only called once with a value XY. While some small differences can be normal (e.g., because of rounding), large differences can indicate an error in the implementation. Note that this can also be a good fuzzing target, as it is very easy to split up a provided test case and compare the resulting state.

- Rounding errors that can be amplified

While some small rounding errors are generally unavoidable and not a bad thing, they can become a large problem when it is possible to amplify them. One way to amplify can be to call a function repeatedly at predetermined moments in time (see here for an example). The significance of the rounding error may also depend on the values (or rather their order of magnitude), so thinking about different scales can also help.

- Off-by-one errors

  Off-by-one errors are very common (not only in smart contracts), so it always makes sense to think about the boundaries. Is `<=` correct in this context or should `<` be used? Should a variable be set to the length of a list or the length - 1? Should an iteration start at 1 or 0?

- Duplicates in lists

  Some programmers have the implicit invariant that a list will not contain duplicates, but do not check so explicitly. This can be especially harmful when addresses are passed by a user (and they are used for some queries, e.g. to query balances), because some values may be double-counted. Thinking about what happens when there are duplicates (and if it is possible) can be very helpful

- Not following EIPs / Standards

  Ethereum Improvement Proposals and in general standards (like RFCs) often have very detailed instructions on how different functions/implementations should behave in different scenarios. Not following these is very bad for composability (after all, the reason for these standards is that you can rely on a certain behavior) and it can also result in vulnerabilities. In my experience, there are often slight details (e.g., not reverting when one should revert or not following the specified rounding behavior) that are ignored by implementers.

- Behavior when src == dst

  In a lot of smart contracts, there are functions where you have to specify a source (or sender) and a destination (or recipient). Sometimes, the programmer did not think about what happens when these (user-specified) parameters are equal, which can result in undesired behavior. For instance, the balance of the source and destination may be cached in the beginning, in which case you can inflate your balance by specifying yourself as the source and the destination.

- Detection of uninitialized state

  It is very common to check if a variable is equal to the default value (0, `address(0)`, empty list, empty string, ...) when checking if it is already initialized. However, these values may also be valid values for this variable. This can result in situations where it is possible to trigger the initialization logic multiple times, which can have negative side effects.

- Assumption that the contract balance is equal to the deposits

  An implicit assumption that some developers make is that the contract balance (ERC20 or ETH balance) is equal to some state variable that is increased/decreased on deposits/withdrawals. This will not be true in practice (you can always manually transfer tokens or ETH to a contract, the latter even if it does not have a `receive` function), so it can be very dangerous to use this state variable and the balance interchangeably.

- Calling function multiple times with the same parameters

  Some functions that should only be callable once with the same set of parameters (or slightly modified parameters) and fail afterward. A very common example is signatures that should only be used once (and where malleability can be an issue), but it is in general a good idea to think about if this function should be callable multiple times with the same parameters and if not, validate that there are measures against it.

- Forgetting to update the global state

  For optimization purposes, contracts often work on a copy of some data (e.g., a struct that is declared as `memory`) to avoid accessing the global state repeatedly. Sometimes, they forget to update the global state after doing so and only update the copy. Although it is a simple mistake, it can be relatively hard to spot in a large codebase (as the only difference can be a `memory` instead of a `storage` keyword) and an extension that highlights `storage` variables (e.g., Solidity Visual Developer) can help a lot to catch such mistakes.

- Arbitrarily long loops

  When there is an arbitrarily long loop (where the length is for instance controllable by the user) and there is no way to specify the start/end index, there is the risk that the list grows too large at some point and looping over it will no longer be possible because of the gas limit. Thinking about the maximum length (or in general if the list is bounded and if there is a possibility to specify a subset) can help to catch such issues.

- Improper state update when deleting items from a list

  To delete an item from a list in Solidity, you usually replace it with the last item in the list and reduce the length of the list by one (using `pop`). However, this actually changes the state of two items: The item you want to delete (which is usually handled correctly) and the item that was previously the last. This item now has a different list index. Because the index is often referenced in other places, this can lead to wrong references.

- ETH / WETH handling

  Many contracts have some special logic to handle ETH. For instance, they may wrap it to WETH when `msg.value > 0`, but may also accept WETH directly. One problem that can occur there is that a developer implicitly assumes that these cases are mutually exclusive, but does not explicitly check so. When the user then specifies WETH as currency but also sends ETH with the call, some invariants may be broken, which can lead to wrong behavior.

- Liquidation

  - 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?
  - Is the eligibility for liquidation calculated consistently everywhere?
  - Does the liquidation clear all the debt? Is there bad debt left even if no collateral?
  - Is griefing possible by front running by slightly increasing his collateral?
  - Will the liquidation work if the price falls by 99%?
  - Is it possible to front-run the liquidation with small efforts but still keep the unhealthy state?

- If there's the possibility for partial liquidation, does the burning functionality account for that?
- Can the liquidator receive less than expected?
- Can the liquidator liquidate more than the liquidated amount?
- If repayments are paused, are liquidations paused too?