General Philosophy

Keeping contracts simple

* logic is simple
* Code are modularised into their own functions. Every potential scenario has it’s own corresponding function. No function will be able to do 2 things like update details and approve user.
* Use tools that are already written
* Clarity over performance (while still keeping gas costs in mind)
* Use blockchain only for parts that require decentralisation

Stay up to date

* using latest version of library / tool (did not do this cause web3js is still in beta, opted for a more stable version)

Blockchain properties

* becareful of external contract calls (none needed)
* Functions are public so other contracts can access it however I’ve added all the necessary modifiers to limit access.

Rigid vs upgradeable

Monolithic vs modular (small enough to be monolithic)

Duplication vs reuse

Recommendations

External calls

- mark untrusted contracts

- avoid state change after external call

- send, trf, call.value()()

- error handling for external calls

- favouring pull over push (e.g. bidding, someone bid higher and the previous bidder get refunded vs bidding to store just bidding amt and withdraw to store withdrawal amt)

Dont use contract balance for any calculation (ether can be forcibly sent into your contract even if you don’t have a payable fallback function)

On chain data is public even if it’s marked private

Multiple party contracts, N users might drop out and may not return. Need functions to be BFT (<https://en.wikipedia.org/wiki/Byzantine_fault_tolerance>)

Know the diff between require and assert. <https://ethereum.stackexchange.com/questions/49200/why-not-use-require-in-place-of-assert>

Be careful of rounding with int division

Keep fallback functions simple and check data length

Explicitly mark visibility for all functions and state variables (only functions that should be accessed by the public are marked public and with the corresponding modifiers. All variables are marked private as I do not want others implementing functions to manipulate my variables)

Lock pragma to specific compiler versions (unless it’s being used by other contracts eg libraries)

Differentiate between function and events.

Use newer solidity constructs and aliases e.g. transfer over send.

Avoid using tx origin (use msg.sender, and may be deprecated in the future)

Avoid being dependent on time stamps (they can be manipulated in blocks)

Be careful about multiple inheritance (know which is the most derived contract)

Known attacks

Race condition

* reentrancy
* Multiple functions sharing the same state

Transaction ordering dependence (<https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e>)

Integer over/underflow

DOS with unexpected revert (revert in the recipient’s fallback function)

DOS with block gas limit (when you have to iterate through an array to pay many people and you run into the block gas limit)

Forcibly sending ether to a contract (you can also precompute a contract’s address and send ether to that address before the contract is deployed, <https://github.com/Arachnid/uscc/tree/master/submissions-2017/ricmoo>)

Software Engineering techniques

Upgrading broken contracts

* using registry contract which holds the address of the latest contract (proxy)
* Or a contract that forwards calls and data onto the latest version of the contract (relay)
* <https://blog.zeppelinos.org/proxy-patterns/>
* <https://consensys.github.io/smart-contract-best-practices/software_engineering/>

Circuit breakers (pause contract functionality if certain conditions are met and useful when new errors are found)

Speed bumps (slow actions down)

Rate limiting (limit the number of actions a user can take over a period of time)

Preparation before contract roll out (locally then test nets)

Bounty programs

Design patterns

<https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html>

<https://medium.com/@i6mi6/solidty-smart-contracts-design-patterns-ecfa3b1e9784>

<https://github.com/fravoll/solidity-patterns>

Instead of using the factory contract to deploy child user contracts, I’ve opted to use a child struct as a data type in my contract. Easier to manage?