Best Practices & Common Pitfalls

K B A

Best Practices

Using native sized types

Solidity is compiled to bytecode that is run on the EVM. When you are programming in Solidity you need to keep this in mind. Things that fit the EVM better use less gas.

- The EVM uses a 32 byte native word size
  - Bytes32 and uint256 are perfect fits for the EVM word size
  - Using smaller sizes might not actually save you gas.
- Use struct in solidity for gas optimization
  - struct will pack elements less than 32 bytes into units of 32 bytes
  - this will happen only if those elements are in order.

Prefer memory over storage

If you have a variable that is being used temporarily, put it in memory.

```
struct invoiceData {
     address invoicePayee;
     uint amount;
}
mapping(address => invoiceData) internal invoicesPaid;
function payInvoice(address payee, uint amountPaying) public {
     payingInvoice memory invoiceData;
     payingInvoice.invoicePayee = payee;
     payingInvoice.amount = amountPaying;
     invoicesPaid[msg.sender] = payingInvoice;
```

Use standard contracts and libraries

Look for standard contracts that you can derive from, that are tested, audited and reviewed by thousands of people. It is always better to have more eyes on software. Here are a few contracts and libraries that you can use:

- openzeppelin solidity contracts
- https://dapp.tools/dappsys/
- https://github.com/modular-network/ethereum-libraries

Saving gas on failures

- When a smart contract fails the logic, or an assert or require the gas used is not returned to the address that created the transaction.
- For that reason you should arrange your checks in order of least gas used first.
-
```
    function sendSmallAmountofMoney(uint amount, address sendTo) public
{
    require(amount < 100 ether);
    require(frozenAccounts[sendTo] == 0);
    require(checkOnWhitelist(sendTo) == true);
}
```

Keep it simple

- There is a gas expenditure to deploy a contract and for every call that changes the state of the contract.
- The simpler your contract, the less, the cost of usage (mostly).
- Smart contracts are meant to be small and simple. If you find you are using it to replace a database or an application, then you are using it wrong.

Avoid large array and iterating over them

- Arrays in Solidity are not meant to hold a lot of information.
- If you have designed your contract with an array that can grow over time then ***you may just break the contract.***
- When your array gets large enough, you will go over your gas limit and even if you increase that you may hit the block gas limit.
- If you find you are using arrays that way. Consider using an off chain database instead.

Keep contracts deterministic

You should be able to look at your smart contract functions and understand how many statements need to be executed to run it. If you can't make an estimate or it multiplies based on a control structure, you might have problems.
The below code is a bad practise for a smart contract.

```
function findCharacter(string paragraph, bytes1 char) public returns(bool) {
          for(i=0; i<paragraph.length; i++) {
               if(paragraph[i]==char) {
                    if(findCharacter(paragraph, "!")) {
                         return true;
                    }
               }
          }
}
```

K B A

Hybrid database/blockchain system

Until blockchain technology gets better, it does not make sense to use the blockchain as a database. You can store record indexes in a smart contract and use the database to keep all of the contextual information.

```
struct registration {
    uint studentID;
    uint courseID;
    uint sessionID;
};
```

More best practises (for gas optimization)

- Prefer fixed sized variables than dynamic sized
- Make fewer external calls
- Delete storage variables that you don’t require
- Don’t initialize variables with the default value of the type. For example default value of ‘uint’ is 0, so need to initialize with 0.
- Write reason strings with *require* but keep it short
- Use events to store data not required to persist
- Prefer mapping over arrays when appropriate.

K B A

Pitfalls

Things that can go wrong & things you need to understand going forward

Running out of gas

- If you have a *while* or *for loop* or a string or dynamic array you may surpass your gas limit. The more data you have to traverse through the more gas is used.
- We recommend that you use a mapping to store large amounts of data. You can’t iterate over a mapping so you wont have gas problems. If you need to iterate, then you should consider using a database hybrid.

Trapped Ether in contract

- If you are accepting payments to your contract you need to have a way to transfer that balance out of the contract.
- You can set your smart contract up to receive payments (payable) so you should have functions that allow the owner to transfer the balance of the contract to themselves or others.

Multi-dimensional arrays are tricky

Multi-dimensional arrays are accomplished in Solidity by having an array of arrays.

- uint[2][4] actually means four arrays with elements of two uints each.
- uint[3][ ] is a dynamic array with elements of fixed size of three.
- uint[ ][3] is a fixed size array that has three dynamic arrays.

Multi-dimensional arrays get very expensive as you start to fill them with data. A 10x10 array of uint8 can cost upwards of 800,000 gas to fill. So use with care, if you have to.

Random number generation

- Solidity is a very deterministic system so generating random numbers is difficult.
- The closest things you have to seed a random number generator are values from the transaction block.
- However, the miners have access to your code running and may be able to manipulate the values you are using to generate an random number, since they are the once creating the blocks.

The only current solution that is close to get a random number is by working with others in a DAO to generate them. [**https://github.com/randao/randao**](https://github.com/randao/randao)

Overflows/underflows

- Solidity stores uint as a 256 bit value. It does have an issue when operating near the 0 and 2**256. If you subtract 1 from 0 you can get 2**256 in some cases. This is a problem if you have the uint represent Ether.
- Use a combination of require checks to make sure that the user has enough in their balance to handle a transaction and the SafeMath contract from Open Zeppelin.

Private is not private

- You can set variables to be private or you can encrypt data in your smart contract. That does not mean that people cannot find out what values you entered. Anyone can look at the transaction records to see what data you passed to your smart contract.
- **If something needs to held as a secret, *don't put it up on a transparent distributed ledger.***

Solidity vulnerabilities (a select few ones that we know of)

As you get more experience with Solidity and you are ready to write smart contracts that represent large amounts of value, you should learn more about vulnerabilities in contracts. New ones are discovered all of the time. We recommend that you have your contracts audited before going for production release.
https://blog.bankex.org/nine-pitfalls-of-ethereum-smart-contracts-to-be-avoided-f7464761211c

Solidity Design Patterns

Adopting proven and open contract patterns would help with well known attack vectors and security considerations.

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

Keep yourself updated on the latest best practices and patterns, followed by the community, as much as you can.

A good document to read about best practises

THANK YOU