Solidity Best Security Practices

“Your ugly mistakes will live there forever” - Konstantinos Karagiannis

General Recommendations from Consensys and Open Zeppelin

See https://consensys.github.io/smart-contract-best-practices/ (https://consensys.github.io/smart-contract-best-practices/)

- Take Warnings Seriously
- Restrict the Amount of Ether / Tokens
  - Remove ether / tokens from contracts to cold wallets
  - Avoid allowing unlimited approval of tokens
- Keep it Small and Modular
  - Modular code is easier to test and audit
- Use the Checks-Effects-Interactions Pattern
  - This pattern prevents re entrancy problems
- Include a Fail-Safe Mode
  - In case your contract is attacked include the abilityy to pause all user transactions to give yourself time to assess and fix problems. Open Zeppelin have a pausable contract for this purpose.
- Test Contract Upgrades
  In addition to testing an upgraded contract, plan, document and test the upgrade process itself.
- Be aware of current vulnerabilities
  See Smart Contract Weakness Registry (https://swcregistry.io/)
  For news and analysis of recent attacks in DeFi see Rekt (https://rekt.news/)

Solidity Specific

The Checks-Effects-Interactions Pattern

- Check for all preconditions
- Modify state variables
- Make external calls

```
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    // The user's balance is already 0,
    // so future invocations won't withdraw anything
    require(success);
}
```

The purpose of this pattern is to ensure that contract state is updated before external calls are made. If this is not possible you can use re entrancy guards.
An Open Zeppelin library (https://docs.openzeppelin.com/contracts/2.x/api/utils#ReentrancyGuard) exists for this

Reentrancy and ERC777 contracts

A re entrancy vulnerability was found in Uniswap V1 due to the before and after transfer hooks in an ERC777 contract.
See Consensys audit (https://github.com/ConsenSys/Uniswap-audit-report-2018-12#31-liquidity-pool-can-be-stolen-in-some-tokens-eg-erc-777-29)

Enforce access control on all public / external functions

Modifiers or Open Zeppelins Access Control contract can be used to restrict access to functions. Restrict the access as much as possible by default, it can be useful to document the roles in your contract and use those as a basis for unit tests. As a final checkthe Solidity Metrics plugin will show the visibility of all functions.

Avoid the use of tx.origin

See SWC-115 (https://swcregistry.io/docs/SWC-115)
This is a simple vulnerability to exploit, allowing an attacker to impersonate for example an administrator address.
Always use msg.sender instead.

Explicitly mark the visibility of functions and variables

See SWC-108 (https://swcregistry.io/docs/SWC-108)
For clarity, and to show assumptions, it is best to be explicit about visibility.
Note that marking variables as private does not imply privacy.

Shadowing variables is possible

See SWC-123 (https://swcregistry.io/docs/SWC-119)
The compiler will warn you when you shadow an existing variable, make sure that is what you intended.

Use interface types rather than address

When storing a contract address in a calling contract, use the interface type, or contract type if possible to allow compile time checks.

```
pragma solidity ^0.8.0;

contract MyContract {

  Foo otherContract;

  function callOtherContract() public view returns (bool){
      bool answer = otherContract.baz(69,true);
      return answer;
  }
}


contract Foo {
    function baz(uint32 x, bool y) public pure returns (bool r) {
    r = x > 32 || y;
    }
}
```

Casting

Take care when casting to smaller sized types for loss of information.
Use the Open Zeppelin SafeCast library for this.

```
import "@openzeppelin/contracts/utils/SafeCast.sol";

contract SomeContract {
    function downcast(uint256 input) public pure returns (uint64) {
        return SafeCast.toUint64(input);
    }
}
```

Compiler version

See SWC-103 (https://swcregistry.io/docs/SWC-103)
Avoid using a floating pragma, contracts should be tested against a specific compiler version and deployed with the same version.

Privacy

See SWC-136 (https://swcregistry.io/docs/SWC-136)
Assume all the data in contracts or transactions can be read by malicious actors. There may be GDPR implications if Personally Identifiable Information is added to the blockchain.

Unexpected Ether

See SWC-132 (https://swcregistry.io/docs/SWC-132)
It is possible to forcibly send Ether to a contract by using selfdestruct.
Logic that relies on exact balances in contracts can be upset with forced sending of ether to the contract.

Transaction Ordering

See SWC-114 (https://swcregistry.io/docs/SWC-114)
Miners decide on the order of transactions, do not allow your code to be dependent on the order of transactions.

Timestamp dependence

See SWC-116 (https://swcregistry.io/docs/SWC-116)
Be aware that the block timestamp is under the control of miners and can be manipulated by them.
Furthermore its accuracy cannot be guaranteed, so do not use it for precise measurements.

Function Parameters

Check all parameters for valid values with require statements

Use require / assert / revert properly

These three statements have different semantics, use them appropriately. Situations needing Assert are less common than those needing require
See https://docs.soliditylang.org/en/v0.8.7/control-structures.html?highlight=assert require#error-handling-assert-require-revert-and-exceptions (https://docs.soliditylang.org/en/v0.8.7/control-structures.html?highlight=assert%20require#error-handling-assert-require-revert-and-exceptions)

Only use modifiers for checks

Do not allow modifiers to have ‘side effects’

Loops

See SWC-128 (https://swcregistry.io/docs/SWC-128)
Use pagination where looping through an entire array would cost too much gas.
For example rather than returning an entire array, make multiple calls returning part of it in each call. Be aware in this case that the transactions maybe in multiple blocks.
Pay particular attention where items can be added to arrays without restriction.
For example

```
pragma solidity ^0.6.0;

contract EnumerableStore {
    uint256[] private _entries;

    function entryAt(uint256 index) public view returns (uint256) {
        return _entries[index];
    }

    function totalEntries() public view returns (uint256) {
        return _entries.length;
    }

    ...
}

pragma solidity ^0.6.0;

contract PaginatedStoreQuerier {
    function entryAtPaginated(EnumerableStore store,
        uint256 startIndex, uint256 pageSize)
        public view returns(uint256[] memory) {
        uint256[] memory result;
        require(startIndex + pageSize
        <= store.totalEntries(), "Read index out of bounds");

        for (uint256 i = 0; i < pageSize; ++i) {
            result[i] = store.entryAt(i + startIndex);
        }

        return result;
    }
}
```

Admin Rights Transfer

When transfering admin rights, there is the possibility that an incorrect address could be used, rendering a contract unusable.
To prevent this, use a 2 transaction process

1. The currenct admin offers the admin right to a new specified address
2. The new address needs to claim the admin right, before it is transfered.

External Calls

See SWC-113 (https://swcregistry.io/docs/SWC-113)
See SWC-104 (https://swcregistry.io/docs/SWC-104)
See SWC-112 (https://swcregistry.io/docs/SWC-112)

1. Use low level calls sparingly
2. Always check the return values from external calls
3. Don’t use delegate call with potentially untrusted contracts.

```
(bool success, ) = someAddress.call.value(55)("");
if(!success) {
    // handle failure code
}
```

‘Safe’ calls

Open Zeppelins safe calls such as SafeTransfer guard against tokens being sent to an incorrect address.

Remove Dead Code

See SWC-135 (https://swcregistry.io/docs/SWC-135)

Withdraw Pattern

See Pull for external calls (https://consensys.github.io/smart-contract-best-practices/recommendations/#favor-pull-over-push-for-external-calls)
Use a ‘pull’ withdraw pattern to withdraw funds from a contract
This ensures that failure for one user will not result in a failure for all users.
A related attack is DoS with unexpected revert (https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-unexpected-revert)

Use Events to monitor state changes in a contract

Transaction receipts do not give information about calls to internal or private functions. It is therefore recommended to use events to record all relevant information in a state change.

Avoid division errors

Do divisions last in a formula to avaoid loss of accuracy

Revert statements

1. Always include a reason in a revert statement.
2. Keep the reson string under 32 bytes to reduce gas.

Multiple Inheritence

See SWC-125 (https://swcregistry.io/docs/SWC-128)
Solidity supports mutiple inheritence, which may produce different behaviour than that expected.

Example Audits

https://blog.openzeppelin.com/arrotoken-audit/ (https://blog.openzeppelin.com/arrotoken-audit/)
https://blog.openzeppelin.com/geb-protocol-audit/ (https://blog.openzeppelin.com/geb-protocol-audit/)