Secure Development Recommendations

This page demonstrates a number of patterns which should generally be followed when writing smart contracts.

Protocol specific recommendations

The following recommendations apply to the development of any contract system on Ethereum.

External Calls

Use caution when making external calls

Calls to untrusted contracts can introduce several unexpected risks or errors. External calls may execute malicious code in that contract *or* any other contract that it depends upon. As such, every external call should be treated as a potential security risk. When it is not possible, or undesirable to remove external calls, use the recommendations in the rest of this section to minimize the danger.

Mark untrusted contracts

When interacting with external contracts, name your variables, methods, and contract interfaces in a way that makes it clear that interacting with them is potentially unsafe. This applies to your own functions that call external contracts.

```
// bad
Bank.withdraw(100); // Unclear whether trusted or untrusted

function makeWithdrawal(uint amount) { // Isn't clear that this
function is potentially unsafe
```

```
    Bank.withdraw(amount);
}

// good
UntrustedBank.withdraw(100); // untrusted external call
TrustedBank.withdraw(100); // external but trusted bank contract
maintained by XYZ Corp

function makeUntrustedWithdrawal(uint amount) {
    UntrustedBank.withdraw(amount);
}
```

Avoid state changes after external calls

Whether using *raw calls* (of the form `someAddress.call()`) or *contract calls* (of the form `ExternalContract.someMethod()`), assume that malicious code might execute. Even if `ExternalContract` is not malicious, malicious code can be executed by any contracts *it* calls.

One particular danger is malicious code may hijack the control flow, leading to vulnerabilities due to reentrancy. (See Reentrancy [../known_attacks#reentrancy] for a fuller discussion of this problem).

If you are making a call to an untrusted external contract, *avoid state changes after the call*. This pattern is also sometimes known as the checks-effects-interactions pattern [http://solidity.readthedocs.io/en/develop/security-considerations.html?highlight=check%20effects#use-the-checks-effects-interactions-pattern].

See SWC-107 [https://swcregistry.io/docs/SWC-107]

Avoid `transfer()` and `send()`.

`.transfer()` and `.send()` forward exactly 2,300 gas to the recipient. The goal of this hardcoded gas stipend was to prevent reentrancy vulnerabilities [../known_attacks#reentrancy], but this only makes sense under the assumption that gas costs are constant. Recently EIP 1283 [https://eips.ethereum.org/EIPS/eip-1283] (backed out of the Constantinople hard fork at the last minute) and EIP 1884 [https://eips.ethereum.org/EIPS/eip-

1884] (expected to arrive in the Istanbul hard fork) have shown this assumption to be invalid.

To avoid things breaking when gas costs change in the future, it's best to use `.call.value(amount)("")` instead. Note that this does nothing to mitigate reentrancy attacks, so other precautions must be taken.

Handle errors in external calls

Solidity offers low-level call methods that work on raw addresses: `address.call()`, `address.callcode()`, `address.delegatecall()`, and `address.send()`. These low-level methods never throw an exception, but will return `false` if the call encounters an exception. On the other hand, *contract calls* (e.g., `ExternalContract.doSomething()`) will automatically propagate a throw (for example, `ExternalContract.doSomething()` will also `throw` if `doSomething()` throws).

If you choose to use the low-level call methods, make sure to handle the possibility that the call will fail, by checking the return value.

```
// bad
someAddress.send(55);
someAddress.call.value(55)(""); // this is doubly dangerous, as it will forward all remaining gas and doesn't check for result
someAddress.call.value(100)(bytes4(sha3("deposit()"))); // if deposit throws an exception, the raw call() will only return false and transaction will NOT be reverted

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

ExternalContract(someAddress).deposit.value(100)();
```

See SWC-104 [https://swcregistry.io/docs/SWC-104]

Favor *pull* over *push* for external calls

External calls can fail accidentally or deliberately. To minimize the damage caused by such failures, it is often better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically. (This also reduces the chance of problems with the gas limit [../known_attacks#dos-with-block-gas-limit].) Avoid combining multiple ether transfers in a single transaction.

```
// bad
contract auction {
    address highestBidder;
    uint highestBid;

    function bid() payable {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            (bool success, ) = highestBidder.call.value(highestBid)
("");
            require(success); // if this call consistently fails,
no one else can bid
        }

       highestBidder = msg.sender;
       highestBid = msg.value;
    }
}

// good
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            refunds[highestBidder] += highestBid; // record the
refund that this user can claim
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        (bool success, ) = msg.sender.call.value(refund)("");
        require(success);
    }
}
```

See SWC-128 [https://swcregistry.io/docs/SWC-128]

Don't delegatecall to untrusted code

The `delegatecall` function is used to call functions from other contracts as if they belong to the caller contract. Thus the callee may change the state of the calling address. This may be insecure. An example below shows how using `delegatecall` can lead to the destruction of the contract and loss of its balance.

```
contract Destructor
{
    function doWork() external
    {
        selfdestruct(0);
    }
}

contract Worker
{
    function doWork(address _internalWorker) public
    {
        // unsafe
_internalWorker.delegatecall(bytes4(keccak256("doWork()")));
    }
}
```

If `Worker.doWork()` is called with the address of the deployed `Destructor` contract as an argument, the `Worker` contract will self-destruct. Delegate execution only to trusted contracts, and **never to a user supplied address**.

Warning

> Don't assume contracts are created with zero balance An attacker can send ether to the address of a contract before it is created. Contracts should not assume that its initial state contains a zero balance. See issue 61 [https://github.com/ConsenSys/smart-contract-best-practices/issues/61] for more details.

See SWC-112 [https://swcregistry.io/docs/SWC-112]

Remember that Ether can be forcibly sent to an account

Beware of coding an invariant that strictly checks the balance of a contract.

An attacker can forcibly send ether to any account and this cannot be prevented (not even with a fallback function that does a `revert()`).

The attacker can do this by creating a contract, funding it with 1 wei, and invoking `selfdestruct(victimAddress)` . No code is invoked in `victimAddress` , so it cannot be prevented. This is also true for block reward which is sent to the address of the miner, which can be any arbitrary address.

Also, since contract addresses can be precomputed, ether can be sent to an address before the contract is deployed.

See SWC-132 [https://swcregistry.io/docs/SWC-132]

Remember that on-chain data is public

Many applications require submitted data to be private up until some point in time in order to work. Games (eg. on-chain rock-paper-scissors) and auction mechanisms (eg. sealed-bid Vickrey auctions [https://en.wikipedia.org/wiki/Vickrey_auction]) are two major categories of examples. If you are building an application where privacy is an issue, make sure you avoid requiring users to publish information too early. The best strategy is to use commitment schemes [https://en.wikipedia.org/wiki/Commitment_scheme] with separate phases: first commit using the hash of the values and in a later phase revealing the values.

Examples:

- In rock paper scissors, require both players to submit a hash of their intended move first, then require both players to submit their move; if the submitted move does not match the hash throw it out.
- In an auction, require players to submit a hash of their bid value in an initial phase (along with a deposit greater than their bid value), and then submit their auction bid value in the second phase.
- When developing an application that depends on a random number generator, the order should always be *(1)* players submit moves, *(2)* random number generated, *(3)* players paid out. The method by which random numbers are generated is itself an area of active research; current best-in-class solutions include Bitcoin block headers (verified through http://btcrelay.org [http://btcrelay.org]), hash-commit-reveal schemes (ie. one party generates a number, publishes its hash to "commit" to the value, and then reveals the value later) and RANDAO [http://github.com/randao/randao]. As Ethereum is a deterministic protocol, no variable within the protocol could be used as an unpredictable random number. Also be aware that miners are in some extent in control of the `block.blockhash()` value* [https://ethereum.stackexchange.com/questions/419/when-can-blockhash-be-safely-used-for-a-random-number-when-would-it-be-unsafe].

Beware of the possibility that some participants may "drop offline" and not return

Do not make refund or claim processes dependent on a specific party performing a particular action with no other way of getting the funds out. For example, in a rock-paper-scissors game, one common mistake is to not make a payout until both players submit their moves; however, a malicious player can "grief" the other by simply never submitting their move - in fact, if a player sees the other player's revealed move and determines that they lost, they have no reason to submit their own move at all. This issue may also arise in the context of state channel settlement. When such situations are an issue, (1) provide a way of circumventing non-participating participants, perhaps through a time limit, and (2) consider adding an additional economic incentive for participants to submit information in all of the situations in which they are supposed to do so.

Beware of negation of the most negative signed integer

Solidity provides several types to work with signed integers. Like in most programming languages, in Solidity a signed integer with `N` bits can represent values from `-2^(N-1)` to `2^(N-1)-1` . This means that there is no positive equivalent for the `MIN_INT` . Negation is implemented as finding the two's complement of a number, so the negation of the most negative number will result in the same number [https://en.wikipedia.org/wiki/Two%27s_complement#Most_negative_number].

This is true for all signed integer types in Solidity (`int8` , `int16` , ..., `int256`).

```
contract Negation {
    function negate8(int8 _i) public pure returns(int8) {
        return -_i;
    }

    function negate16(int16 _i) public pure returns(int16) {
        return -_i;
    }

    int8 public a = negate8(-128); // -128
    int16 public b = negate16(-128); // 128
    int16 public c = negate16(-32768); // -32768
}
```

One way to handle this is to check the value of a variable before negation and throw if it's equal to the `MIN_INT` . Another option is to make sure that the most negative number will never be achieved by using a type with a higher capacity (e.g. `int32` instead of `int16`).

A similar issue with `int` types occurs when `MIN_INT` is multiplied or divided by `-1` .

Solidity specific recommendations

The following recommendations are specific to Solidity, but may also be instructive for developing smart contracts in other languages.

Enforce invariants with `assert()`

An assert guard triggers when an assertion fails - such as an invariant property changing. For example, the token to ether issuance ratio, in a token issuance contract, may be fixed. You can verify that this is the case at all times with an `assert()` . Assert guards should often be combined with other techniques, such as pausing the contract and allowing upgrades. (Otherwise, you may end up stuck, with an assertion that is always failing.)

Example:

```
contract Token {
    mapping(address => uint) public balanceOf;
    uint public totalSupply;

    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        totalSupply += msg.value;
        assert(address(this).balance >= totalSupply);
    }
}
```

Note that the assertion is *not* a strict equality of the balance because the contract can be forcibly sent ether [#remember-that-ether-can-be-forcibly-sent-to-an-account] without going through the `deposit()` function!

Use `assert()`, `require()`, `revert()` properly

Info

The convenience functions **assert** and **require** can be used to check for conditions and throw an exception if the condition is not met.

The **assert** function should only be used to test for internal errors, and to check invariants.

The **require** function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts. * [https://solidity.readthedocs.io/en/latest/control-structures.html#error-handling-assert-require-revert-and-exceptions]

Following this paradigm allows formal analysis tools to verify that the invalid opcode can never be reached: meaning no invariants in the code are violated and that the code is formally verified.

```
pragma solidity ^0.5.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns
(uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
//Require() can have an optional message string
        uint balanceBeforeTransfer = address(this).balance;
        (bool success, ) = addr.call.value(msg.value / 2)("");
        require(success);
        // Since we reverted if the transfer failed, there should
be
        // no way for us to still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer -
msg.value / 2); // used for internal error checking
        return address(this).balance;
    }
}
```

See SWC-110 [https://swcregistry.io/docs/SWC-110] & SWC-123 [https://swcregistry.io/docs/SWC-123]

Use modifiers only for checks

The code inside a modifier is usually executed before the function body, so any state changes or external calls will violate the Checks-Effects-Interactions [https://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern] pattern. Moreover, these statements may also remain unnoticed by the developer, as the code for modifier may be far from the function declaration. For example, an external call in modifier can lead to the reentrancy attack:

```
contract Registry {
    address owner;

    function isVoter(address _addr) external returns(bool) {
        // Code
    }
}

contract Election {
    Registry registry;

    modifier isEligible(address _addr) {
        require(registry.isVoter(_addr));
        _;
    }

    function vote() isEligible(msg.sender) public {
        // Code
    }
}
```

In this case, the `Registry` contract can make a reentracy attack by calling `Election.vote()` inside `isVoter()` .

Note

Use modifiers [https://solidity.readthedocs.io/en/develop/contracts.html#function-modifiers] to replace duplicate condition checks in multiple functions, such as `isOwner()` , otherwise use `require` or `revert` inside the function. This makes your smart contract code more readable and easier to audit.

Beware rounding with integer division

All integer division rounds down to the nearest integer. If you need more precision, consider using a multiplier, or store both the numerator and denominator.

(In the future, Solidity will have a fixed-point [https://solidity.readthedocs.io/en/develop/types.html#fixed-point-numbers] type, which will make this easier.)

```
// bad
uint x = 5 / 2; // Result is 2, all integer divison rounds DOWN to
the nearest integer
```

Using a multiplier prevents rounding down, this multiplier needs to be accounted for when working with x in the future:

```
// good
uint multiplier = 10;
uint x = (5 * multiplier) / 2;
```

Storing the numerator and denominator means you can calculate the result of `numerator/denominator` off-chain:

```
// good
uint numerator = 5;
uint denominator = 2;
```

Be aware of the tradeoffs between **abstract contracts** and **interfaces**

Both interfaces and abstract contracts provide one with a customizable and re-usable approach for smart contracts. Interfaces, which were introduced in Solidity 0.4.11, are similar to abstract contracts but cannot have any functions implemented. Interfaces also have limitations such as not being able to access storage or inherit from other interfaces which generally makes abstract contracts more practical. Although, interfaces are certainly useful for designing contracts prior to implementation. Additionally, it is important to keep in mind that if a contract inherits from an abstract contract it must implement all non-implemented functions via overriding or it will be abstract as well.

Fallback Functions

Keep fallback functions simple

Fallback functions [http://solidity.readthedocs.io/en/latest/contracts.html#fallback-function] are called when a contract is sent a message with no arguments (or when no function matches), and only has access to 2,300 gas when called from a `.send()` or `.transfer()`. If you wish to be able to receive Ether from a

`.send()` or `.transfer()`, the most you can do in a fallback function is log an event. Use a proper function if a computation of more gas is required.

```
// bad
function() payable { balances[msg.sender] += msg.value; }

// good
function deposit() payable external { balances[msg.sender] +=
msg.value; }

function() payable { require(msg.data.length == 0); emit
LogDepositReceived(msg.sender); }
```

Check data length in fallback functions

Since the fallback functions [http://solidity.readthedocs.io/en/latest/contracts.html#fallback-function] is not only called for plain ether transfers (without data) but also when no other function matches, you should check that the data is empty if the fallback function is intended to be used only for the purpose of logging received Ether. Otherwise, callers will not notice if your contract is used incorrectly and functions that do not exist are called.

```
// bad
function() payable { emit LogDepositReceived(msg.sender); }

// good
function() payable { require(msg.data.length == 0); emit
LogDepositReceived(msg.sender); }
```

Explicitly mark payable functions and state variables

Starting from Solidity `0.4.0`, every function that is receiving ether must use `payable` modifier, otherwise if the transaction has `msg.value > 0` will revert (except when forced [../recommendations/#remember-that-ether-can-be-forcibly-sent-to-an-account]).

Note

> Something that might not be obvious: The `payable` modifier only applies to calls from *external* contracts. If I call a non-payable function in the payable function in the same contract, the non-payable function won't fail, though `msg.value` is still set

Explicitly mark visibility in functions and state variables

Explicitly label the visibility of functions and state variables. Functions can be specified as being `external`, `public`, `internal` or `private`. Please understand the differences between them, for example, `external` may be sufficient instead of `public`. For state variables, `external` is not possible. Labeling the visibility explicitly will make it easier to catch incorrect assumptions about who can call the function or access the variable.

- `External` functions are part of the contract interface. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). External functions are sometimes more efficient when they receive large arrays of data.
- `Public` functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function (see below) is generated.
- `Internal` functions and state variables can only be accessed internally, without using `this`.
- `Private` functions and state variables are only visible for the contract they are defined in and not in derived contracts. **Note**: Everything that is inside a contract is visible to all observers external to the blockchain, even `Private` variables.* [https://solidity.readthedocs.io/en/develop/contracts.html?#visibility-and-getters]

```
// bad
uint x; // the default is internal for state variables, but it should be made explicit
function buy() { // the default is public
    // public code
}

// good
uint private y;
function buy() external {
    // only callable externally or using this.buy()
}
```

```
function utility() public {
    // callable externally, as well as internally: changing this
code requires thinking about both cases.
}

function internalAction() internal {
    // internal code
}
```

See SWC-100 [https://swcregistry.io/docs/SWC-100] and SWC-108 [https://swcregistry.io/docs/SWC-108]

Lock pragmas to specific compiler version

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

```
// bad
pragma solidity ^0.4.4;


// good
pragma solidity 0.4.4;
```

Note: a floating pragma version (ie. `^0.4.25`) will compile fine with `0.4.26-nightly.2018.9.25`, however nightly builds should never be used to compile code for production.

Warning

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

See SWC-103 [https://swcregistry.io/docs/SWC-103]

Use events to monitor contract activity

It can be useful to have a way to monitor the contract's activity after it was deployed. One way to accomplish this is to look at all transactions of the contract, however that may be insufficient, as message calls between contracts are not recorded in the blockchain. Moreover, it shows only the input parameters, not the actual changes being made to the state. Also events could be used to trigger functions in the user interface.

```
contract Charity {
    mapping(address => uint) balances;

    function donate() payable public {
        balances[msg.sender] += msg.value;
    }
}

contract Game {
    function buyCoins() payable public {
        // 5% goes to charity
        charity.donate.value(msg.value / 20)();
    }
}
```

Here, `Game` contract will make an internal call to `Charity.donate()`. This transaction won't appear in the external transaction list of `Charity`, but only visible in the internal transactions.

An event is a convenient way to log something that happened in the contract. Events that were emitted stay in the blockchain along with the other contract data and they are available for future audit. Here is an improvement to the example above, using events to provide a history of the Charity's donations.

```
contract Charity {
    // define event
    event LogDonate(uint _amount);

    mapping(address => uint) balances;

    function donate() payable public {
        balances[msg.sender] += msg.value;
        // emit event
        emit LogDonate(msg.value);
    }
}

contract Game {
    function buyCoins() payable public {
        // 5% goes to charity
        charity.donate.value(msg.value / 20)();
    }
}
```

Here, all transactions that go through the `Charity` contract, either directly or not, will show up in the event list of that contract along with the amount of donated money.

Note

Prefer newer Solidity constructs

Prefer constructs/aliases such as `selfdestruct` (over `suicide`) and `keccak256` (over `sha3`). Patterns like `require(msg.sender.send(1 ether))` can also be simplified to using `transfer()`, as in `msg.sender.transfer(1 ether)`. Check out Solidity Change log [https://github.com/ethereum/solidity/blob/develop/Changelog.md] for more similar changes.

Be aware that 'Built-ins' can be shadowed

It is currently possible to shadow [https://en.wikipedia.org/wiki/Variable_shadowing] built-in globals in Solidity. This allows contracts to override the functionality of built-ins such as `msg` and `revert()`. Although this is intended [https://github.com/ethereum/solidity/issues/1249], it can mislead users of a contract as to the contract's true behavior.

```
contract PretendingToRevert {
    function revert() internal constant {}
}

contract ExampleContract is PretendingToRevert {
    function somethingBad() public {
        revert();
    }
}
```

Contract users (and auditors) should be aware of the full smart contract source code of any application they intend to use.

Avoid using `tx.origin`

Never use `tx.origin` for authorization, another contract can have a method which will call your contract (where the user has some funds for instance) and your contract will authorize that transaction as your address is in `tx.origin`.

```
contract MyContract {

    address owner;

    function MyContract() public {
        owner = msg.sender;
    }

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        (bool success, ) = receiver.call.value(amount)("");
        require(success);
    }

}

contract AttackingContract {

    MyContract myContract;
    address attacker;

    function AttackingContract(address myContractAddress) public {
        myContract = MyContract(myContractAddress);
        attacker = msg.sender;
    }

    function() public {
        myContract.sendTo(attacker, msg.sender.balance);
    }

}
```

You should use `msg.sender` for authorization (if another contract calls your contract `msg.sender` will be the address of the contract and not the address of the user who called the contract).

You can read more about it here: Solidity docs [https://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin]

Warning

Besides the issue with authorization, there is a chance that `tx.origin` will be removed from the Ethereum protocol in the future, so code that uses `tx.origin` won't be compatible with future releases Vitalik: 'Do NOT assume that tx.origin will continue to be usable or meaningful.' [https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof/200#200]

It's also worth mentioning that by using `tx.origin` you're limiting interoperability between contracts because the contract that uses tx.origin cannot be used by another contract as a contract can't be the `tx.origin`.

See SWC-115 [https://swcregistry.io/docs/SWC-115]

Timestamp Dependence

There are three main considerations when using a timestamp to execute a critical function in a contract, especially when actions involve fund transfer.

Timestamp Manipulation

Be aware that the timestamp of the block can be manipulated by a miner. Consider this contract [https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18#code]:

```
uint256 constant private salt =  block.timestamp;

function random(uint Max) constant private returns (uint256 result)
{
    //get the best seed for randomness
    uint256 x = salt * 100/Max;
    uint256 y = salt * block.number/(salt % 5) ;
    uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
    uint256 h = uint256(block.blockhash(seed));

    return uint256((h / x)) % Max + 1; //random number between 1
and Max
}
```

When the contract uses the timestamp to seed a random number, the miner can actually post a timestamp within 15 seconds of the block being validated, effectively allowing the miner to precompute an option more favorable to their chances in the lottery. Timestamps are not random and should not be used in that context.

The 15-second Rule

The Yellow Paper [https://ethereum.github.io/yellowpaper/paper.pdf] (Ethereum's reference specification) does not specify a constraint on how much blocks can drift in time, but it does specify [https://ethereum.stackexchange.com/a/5926/46821] that each timestamp should be bigger than the timestamp of its parent. Popular Ethereum protocol implementations Geth [https://github.com/ethereum/go-ethereum/blob/4e474c74dc2ac1d26b339c32064d0bac98775e77/consensus/ethash/consensus.go#L45] and Parity [https://github.com/paritytech/parity-ethereum/blob/73db5dda8c0109bb6bc1392624875078f973be14/ethcore/src/verification/verification.rs#L296-L307] both reject blocks with timestamp more than 15 seconds in future. Therefore, a good rule of thumb in evaluating timestamp usage is:

Note

If the scale of your time-dependent event can vary by 15 seconds and maintain integrity, it is safe to use a `block.timestamp` .

Avoid using `block.number` as a timestamp

It is possible to estimate a time delta using the `block.number` property and average block time [https://etherscan.io/chart/blocktime], however this is not future proof as block times may change (such as fork reorganisations [https://blog.ethereum.org/2015/08/08/chain-reorganisation-depth-expectations/] and the difficulty bomb [https://github.com/ethereum/EIPs/issues/649]). In a sale spanning days, the 15-second rule allows one to achieve a more reliable estimate of time.

See SWC-116 [https://swcregistry.io/docs/SWC-116]

Multiple Inheritance Caution

When utilizing multiple inheritance in Solidity, it is important to understand how the compiler composes the inheritance graph.

```
contract Final {
    uint public a;
    function Final(uint f) public {
        a = f;
    }
}

contract B is Final {
    int public fee;

    function B(uint f) Final(f) public {
    }
    function setFee() public {
        fee = 3;
    }
}

contract C is Final {
    int public fee;

    function C(uint f) Final(f) public {
    }
    function setFee() public {
        fee = 5;
    }
}

contract A is B, C {
  function A() public B(3) C(5) {
      setFee();
  }
}
```

When a contract is deployed, the compiler will *linearize* the inheritance from right to left (after the keyword *is* the parents are listed from the most base-like to the most derived). Here is contract A's linearization:

Final <- B <- C <- A

The consequence of the linearization will yield a `fee` value of 5, since C is the most derived contract. This may seem obvious, but imagine scenarios where C is able to shadow crucial functions, reorder boolean clauses, and cause the developer to write exploitable contracts. Static analysis currently does not raise issue with overshadowed functions, so it must be manually inspected.

For more on security and inheritance, check out this article [https://pdaian.com/blog/solidity-anti-patterns-fun-with-inheritance-dag-abuse/]

To help contribute, Solidity's Github has a project [https://github.com/ethereum/solidity/projects/9#card-8027020] with all inheritance-related issues.

See SWC-125 [https://swcregistry.io/docs/SWC-125]

Use interface type instead of the address for type safety

When a function takes a contract address as an argument, it is better to pass an interface or contract type rather than raw `address`. If the function is called elsewhere within the source code, the compiler it will provide additional type safety guarantees.

Here we see two alternatives:

```
contract Validator {
    function validate(uint) external returns(bool);
}

contract TypeSafeAuction {
    // good
    function validateBet(Validator _validator, uint _value)
internal returns(bool) {
        bool valid = _validator.validate(_value);
        return valid;
    }
}

contract TypeUnsafeAuction {
    // bad
    function validateBet(address _addr, uint _value) internal
returns(bool) {
        Validator validator = Validator(_addr);
        bool valid = validator.validate(_value);
        return valid;
    }
}
```

The benefits of using the `TypeSafeAuction` contract above can then be seen from the following example. If `validateBet()` is called with an `address` argument, or a contract type other than `Validator`, the compiler will throw this error:

```
contract NonValidator{}

contract Auction is TypeSafeAuction {
    NonValidator nonValidator;

    function bet(uint _value) {
        bool valid = validateBet(nonValidator, _value); //
TypeError: Invalid type for argument in function call.
                                                        // Invalid
implicit conversion from contract NonValidator
                                                        // to
contract Validator requested.
    }
}
```

Avoid using `extcodesize` to check for Externally Owned Accounts

The following modifier (or a similar check) is often used to verify whether a call was made from an externally owned account (EOA) or a contract account:

```
// bad
modifier isNotContract(address _a) {
  uint size;
  assembly {
    size := extcodesize(_a)
  }
    require(size == 0);
     _;
}
```

The idea is straight forward: if an address contains code, it's not an EOA but a contract account. However, **a contract does not have source code available during construction**. This means that while the constructor is running, it can make calls to other contracts, but `extcodesize` for its address returns zero. Below is a minimal example that shows how this check can be circumvented:

```
contract OnlyForEOA {
    uint public flag;

    // bad
    modifier isNotContract(address _a){
        uint len;
        assembly { len := extcodesize(_a) }
        require(len == 0);
        _;
    }

    function setFlag(uint i) public isNotContract(msg.sender){
        flag = i;
    }
}

contract FakeEOA {
    constructor(address _a) public {
        OnlyForEOA c = OnlyForEOA(_a);
        c.setFlag(1);
    }
}
```

Because contract addresses can be pre-computed, this check could also fail if it checks an address which is empty at block `n`, but which has a contract deployed to it at some block greater than `n`.

Warning

This issue is nuanced.

> If your goal is to prevent other contracts from being able to call your contract, the `extcodesize` check is probably sufficient. An alternative approach is to check the value of `(tx.origin == msg.sender)`, though this also has drawbacks [../recommendations/#avoid-using-txorigin].
>
> There may be other situations in which the `extcodesize` check serves your purpose. Describing all of them here is out of scope. Understand the underlying behaviors of the EVM and use your Judgement.

Deprecated/historical recommendations

These are recommendations which are no longer relevant due to changes in the protocol or improvements to solidity. They are recorded here for posterity and awareness.

Beware division by zero (Solidity < 0.4)

Prior to version 0.4, Solidity returns zero [https://github.com/ethereum/solidity/issues/670] and does not `throw` an exception when a number is divided by zero. Ensure you're running at least version 0.4.

Differentiate functions and events (Solidity < 0.4.21)

Favor capitalization and a prefix in front of events (we suggest *Log*), to prevent the risk of confusion between functions and events. For functions, always start with a lowercase letter, except for the constructor.

> **Note**
>
> In v0.4.21 [https://github.com/ethereum/solidity/blob/develop/Changelog.md#0421-2018-03-07] Solidity introduced the `emit` keyword to indicate an event `emit EventName();`. As of 0.5.0, it is required.