Recommendations for Smart Contract Security in Solidity

This page demonstrates a number of solidity 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 race conditions. (See Race Conditions [../known_attacks#race-conditions] 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].

Be aware of the tradeoffs between `send()`, `transfer()`, and `call.value()()`

When sending ether be aware of the relative tradeoffs between the use of `someAddress.send()`, `someAddress.transfer()`, and `someAddress.call.value()()`.

- `someAddress.send()` and `someAddress.transfer()` are considered *safe* against reentrancy [../known_attacks#reentrancy]. While these methods still trigger code execution, the called contract is only given a stipend of 2,300 gas which is currently only enough to log an event.

- `x.transfer(y)` is equivalent to `require(x.send(y));`, it will automatically revert if the send fails.
- `someAddress.call.value(y)()` will send the provided ether and trigger code execution. The executed code is given all available gas for execution making this type of value transfer *unsafe* against reentrancy.

Using `send()` or `transfer()` will prevent reentrancy but it does so at the cost of being incompatible with any contract whose fallback function requires more than 2,300 gas. It is also possible to use `someAddress.call.value(ethAmount).gas(gasAmount)()` to forward a custom amount of gas.

One pattern that attempts to balance this trade-off is to implement both a *push* and *pull* [#favor-pull-over-push-for-external-calls] mechanism, using `send()` or `transfer()` for the *push* component and `call.value()()` for the *pull* component.

It is worth pointing out that exclusive use of `send()` or `transfer()` for value transfers does not itself make a contract safe against reentrancy but only makes those specific value transfers safe against reentrancy.

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
if(!someAddress.send(55)) {
    // Some failure code
}

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

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 `send()` calls in a single transaction.

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

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

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid); // 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 != 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;
        msg.sender.transfer(refund);
    }
}
```

Don't assume contracts are created with zero balance

An attacker can send wei 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.

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 second-price auctions) are two major categories of examples. If you are building an application where privacy is an issue, take care to avoid requiring users to publish information too early.

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 action 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].
- If you are implementing a frequent batch auction, a hash-commit scheme is also desirable.

In 2-party or N-party contracts, 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.

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(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()` and `require()` properly

In Solidity 0.4.10 `assert()` and `require()` were introduced. `require(condition)` is meant to be used for input validation, which should be done on any user input, and reverts if the condition is false. `assert(condition)` also reverts if the condition is false but should be used only for invariants: internal errors or to check if your contract has reached an invalid state. 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.

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 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;
```

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 wei 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.

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.

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 or more gas is required.

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

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

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

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.

```
// 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
}

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

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

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;
```

Exception

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.

Differentiate functions and events

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.

```
// bad
event Transfer() {}
function transfer() {}

// good
event LogTransfer() {}
function transfer() external {}
```

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)`.

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`.

```
pragma solidity 0.4.18;

contract MyContract {

    address owner;

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

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.transfer(amount);
    }
}

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]

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`.

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.

Gameability

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 30 seconds of the block being validating, 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.

30-second Rule

A general rule of thumb in evaluating timestamp usage is:

If the contract function can tolerate a 30-second [https://ethereum.stackexchange.com/questions/5924/how-do-ethereum-mining-nodes-maintain-a-time-consistent-with-the-network/5931#5931] drift in time, it is safe to use `block.timestamp`

If the scale of your time-dependent event can vary by 30-seconds and maintain integrity, it is safe to use a timestamp. This includes things like ending of auctions, registration periods, etc.

Caution using `block.number` as a timestamp

When a contract creates an `auction_complete` modifier to signify the end of a token sale such as so [../(https://github.com/SpankChain/old-sc_auction/blob/master/contracts/Auction.sol)]

```
modifier auction_complete {
    require(auctionEndBlock <= block.number     ||
          currentAuctionState == AuctionState.success ||
          currentAuctionState == AuctionState.cancel)
        _;}
```

`block.number` and *average block time [https://etherscan.io/chart/blocktime]* can be used to estimate time as well, but 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 12-minute rule allows one to construct a more reliable estimate of time.

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 is deployed, the compiler will *linearize* the inheritance from left to right, as:

C -> B -> 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.

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.