

# Safety

[Edit](#)[New Page](#)[Jump to bottom](#)

noroot edited this page 7 days ago · 17 revisions

---

## Contents

- [Ethereum Contract Security Techniques and Tips - Additional Requested Content](#)
  - [General Philosophy](#)
  - [Security Notifications](#)
  - [Recommendations for Smart Contract Security in Solidity](#)
    - [External Calls](#)
      - [Avoid external calls when possible](#)
      - [Use `send\(\)` , avoid `call.value\(\)`](#)
      - [Handle errors in external calls](#)
      - [Don't make control flow assumptions after external calls](#)
      - [Favor \*pull over push\* for external calls](#)
      - [Mark untrusted contracts](#)
    - [Beware rounding with integer division](#)
    - [Remember that on-chain data is public](#)
    - [In 2-party or N-party contracts, beware of the possibility that some participants may "drop offline" and not return](#)
    - [Keep fallback functions simple](#)
    - [Explicitly mark visibility in functions and state variables](#)
    - [Beware division by zero](#)
    - [Differentiate functions and events](#)
  - [Known Attacks](#)
    - [Call Depth Attack \(Deprecated\)](#)
    - [Race Conditions\\*](#)
      - [Reentrancy](#)
      - [Cross-function Race Conditions](#)
      - [Pitfalls in Race Condition Solutions](#)
    - [DoS with \(Unexpected\) Throw](#)
    - [DoS with Block Gas Limit](#)
    - [Timestamp Dependence](#)
    - [Transaction-Ordering Dependence \(TOD\)](#)
  - [Software Engineering Techniques](#)
    - [Upgrading Broken Contracts](#)

- [Circuit Breakers \(Pause contract functionality\)](#)
- [Speed Bumps \(Delay contract actions\)](#)
- [Rate Limiting](#)
- [Assert Guards](#)
- [Contract Rollout](#)
  - [Automatic Deprecation](#)
  - [Restrict amount of Ether per user/contract](#)
- [Security-related Documentation and Procedures](#)
- [Security Tools](#)
- [Future improvements](#)
- [Smart Contract Security Bibliography - By Ethereum core developers - By Community](#)
- [Attribution](#)
- [License](#)

## Ethereum Contract Security Techniques and Tips

---

The community is encouraged to keep this wiki updated: it becomes more complete as more contributions are added.

If you'd like to submit your Dapp and get early traffic, check out [Dapp Insight](#) for more information.

Currently this wiki is out-of-date and see its source "[Smart Contract Best Practices](#)" for more recent updates and corrections.

### Additional Requested Content

We especially welcome content in the following areas:

- Testing Solidity code (structure, frameworks, common test idioms)
- Software engineering practices for smart contracts and/or blockchain-based programming

### General Philosophy

---

Ethereum and complex blockchain programs are new and highly experimental. Therefore, you should expect constant changes in the security landscape, as new bugs and security risks are discovered, and new best practices are developed. Following the security practices in this document is therefore only the beginning of the security work you will need to do as a smart contract developer.

Smart contract programming requires a different engineering mindset than you may be used to. The cost of failure can be high, and change can be difficult, making it in some ways more similar to hardware programming or financial services programming than web or mobile development. It is therefore not enough to defend against known vulnerabilities. Instead, you will need to learn a new philosophy of development:

- **Prepare for failure.** Any non-trivial contract will have errors in it. Your code must therefore be able to respond to bugs and vulnerabilities gracefully.
  - Pause the contract when things are going wrong ('circuit breaker')
  - Manage the amount of money at risk (rate limiting, maximum usage)
  - Have an effective upgrade path for bugfixes and improvements
- **Roll out carefully.** It is always better to catch bugs before a full production release.
  - Test contracts thoroughly, and add tests whenever new attack vectors are discovered
  - Provide bug bounties starting from alpha testnet releases
  - Rollout in phases, with increasing usage and testing in each phase
- **Keep contracts simple.** Complexity increases the likelihood of errors.
  - Ensure the contract logic is simple
  - Modularize code to keep contracts and functions small
  - Use already-written tools or code where possible (eg. don't roll your own random number generator)
  - Prefer clarity to performance whenever possible
  - Only use the blockchain for the parts of your system that require decentralization
- **Stay up to date.** Use the resources listed in the next section to keep track of new security developments.
  - Check your contracts for any new bug that's discovered
  - Upgrade to the latest version of any tool or library as soon as possible
  - Adopt new security techniques that appear useful
- **Be aware of blockchain properties.** While much of your programming experience will be relevant to Ethereum programming, there are some pitfalls to be aware of.
  - Be extremely careful about external contract calls, which may execute malicious code and change control flow.
  - Understand that your public functions are public, and may be called maliciously. Your private data is also viewable by anyone.
  - Keep gas costs and the block gas limit in mind.

# Security Notifications

---

This is a list of resources that will often highlight discovered exploits in Ethereum or Solidity. The official source of security notifications is the Ethereum Blog, but in many cases vulnerabilities will be disclosed and discussed earlier in other locations.

- [Ethereum Blog](#): The official Ethereum blog
  - [Ethereum Blog - Security only](#): All blog posts that are tagged *Security*
- [Ethereum Gitter](#) chat rooms
  - [Solidity](#)
  - [Go-Ethereum](#)
  - [CPP-Ethereum](#)
  - [Research](#)
- [Reddit](#)
- [Network Stats](#)

It's highly recommended that you *regularly* read all these sources, as exploits they note may impact your contracts.

Additionally, here is a list of Ethereum core developers who may write about security, and see the [bibliography](#) for more from the community.

- **Vitalik Buterin**: [Twitter](#), [Github](#), [Reddit](#), [Ethereum Blog](#)
- **Dr. Christian Reitwiessner**: [Twitter](#), [Github](#), [Ethereum Blog](#)
- **Dr. Gavin Wood**: [Twitter](#), [Blog](#), [Github](#)
- **Vlad Zamfir**: [Twitter](#), [Github](#), [Ethereum Blog](#)

Beyond following core developers, it is critical to participate in the wider blockchain-related security community - as security disclosures or observations will come through a variety of parties.

## Recommendations for Smart Contract Security in Solidity

---

### External Calls

#### Avoid external calls when possible

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, and removed if possible. When it is not possible to remove external calls, use the recommendations in the rest of this section to minimize the danger.

## Use `send()` , avoid `call.value()`

When sending Ether, use `someAddress.send()` and avoid `someAddress.call.value()` .

External calls such as `someAddress.call.value()` can trigger malicious code. While `send()` also triggers code, it is safe because it only has access to gas stipend of 2,300 gas. Currently, this is only enough to log an event, not enough to launch an attack.

```
// bad
if(!someAddress.call.value(100)()) {
    // Some failure code
}

// good
if(!someAddress.send(100)) {
    // Some failure code
}
```

## 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);
```

## Don't make control flow assumptions after external calls

Whether using *raw calls* or *contract calls*, assume that malicious code will execute if `ExternalContract` is untrusted. 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](#) for a fuller discussion of this problem).

### Favor *pull over push* for external calls

As we've seen, external calls can fail for a number of reasons, including external errors. 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](#).)

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

    function bid() {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            if (!highestBidder.send(highestBid)) { // if this call consistently
fails, no one else can bid
                throw;
            }
        }

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

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

    function bid() external {
        if (msg.value < highestBid) throw;

        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;
    if (!msg.sender.send(refund)) {
        refunds[msg.sender] = refund; // reverting state because send failed
    }
}
}

```

## 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
    UntrustedBank.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);
}

```

## 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 division rounds DOWN to the nearest
integer

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

uint numerator = 5;
uint denominator = 2;

```

## 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>), 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](#).
- 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.

## Keep fallback functions simple

[Fallback functions](#) 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()`. If you wish to be able to receive Ether from a `.send()`, 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() { balances[msg.sender] += msg.value; }

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

function() { 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`. 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 private for state variables, but it should be made
explicit
function transfer() { // the default is public
    // public code
}

// good
uint private y;
function transfer() public {
    // public code
}

function internalAction() internal {
    // internal code
}

```

## Beware division by zero

Currently, Solidity [returns zero](#) and does not throw an exception when a number is divided by zero. You therefore need to check for division by zero manually.

```

// bad
function divide(uint x, uint y) returns(uint) {
    return x / y;
}

// good
function divide(uint x, uint y) returns(uint) {
    if (y == 0) { throw; }
}

```

```
    return x / y;
}
```

## 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 {}
```

## Known Attacks

---

### Call Depth Attack (Deprecated)

With the Call Depth Attack, *any* call (even a fully trusted and correct one) can fail. This is because there is a limit on how deep the "call stack" can go. If the attacker does a bunch of recursive calls and brings the stack depth to 1023, then they can call your function and automatically cause all of its subcalls to fail (subcalls include `send()`).

An example based on the previous auction code:

```
// INSECURE
contract auction {
    mapping(address => uint) refunds;

    // [...]

    function withdrawRefund(address recipient) {
        uint refund = refunds[recipient];
        refunds[recipient] = 0;
        recipient.send(refund); // this line is vulnerable to a call depth attack
    }
}
```

The `send()` can fail if the call depth is too large, causing ether to not be sent. However, the rest of the function would succeed, including the previous line which set the victim's refund balance to 0. The solution is to explicitly check for errors, as discussed previously:

```

contract auction {
    mapping(address => uint) refunds;

    // [...]

    function withdrawRefund(address recipient) {
        uint refund = refunds[recipient];
        refunds[recipient] = 0;
        if (!recipient.send(refund)) { throw; } // the transaction will be
reverted in case of call depth attack
    }
}

```

## Race Conditions\*

One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. This class of bug can take many forms, and both of the major bugs that led to the DAO's collapse were bugs of this sort.

### Reentrancy

The first version of this bug to be noticed involved functions that could be called repeatedly, before the first invocation of the function was finished. This may cause the different invocations of the function to interact in destructive ways.

```

// INSECURE
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; } // At this
point, the caller's code is executed, and can call withdrawBalance again
    userBalances[msg.sender] = 0;
}

```

Since the user's balance is not set to 0 until the very end of the function, the second (and later) invocations will still succeed, and will withdraw the balance over and over again. A very similar bug was one of the vulnerabilities in the DAO attack.

In the example given, the best way to avoid the problem is to [use `send\(\)` instead of `call.value\(\)`](#) . This will prevent any external code from being executed.

However, if you can't remove the external call, the next simplest way to prevent this attack is to make sure you don't call an external function until you've done all the internal work you need to do:

```

mapping (address => uint) private userBalances;

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

```

Note that if you had another function which called `withdrawBalance()`, it would be potentially subject to the same attack, so you must treat any function which calls an untrusted contract as itself untrusted. See below for further discussion of potential solutions.

### Cross-function Race Conditions

An attacker may also be able to do a similar attack using two different functions that share the same state.

```

// INSECURE
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
        userBalances[to] += amount;
        userBalances[msg.sender] -= amount;
    }
}

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; } // At this
    point, the caller's code is executed, and can call transfer()
    userBalances[msg.sender] = 0;
}

```

In this case, the attacker calls `transfer()` when their code is executed on the external call in `withdrawBalance`. Since their balance has not yet been set to 0, they are able to transfer the tokens even though they already received the withdrawal. This vulnerability was also used in the DAO attack.

The same solutions will work, with the same caveats. Also note that in this example, both functions were part of the same contract. However, the same bug can occur across multiple contracts, if those contracts share state.

### Pitfalls in Race Condition Solutions

Since race conditions can occur across multiple functions, and even multiple contracts, any solution aimed at preventing reentry will not be sufficient.

Instead, we have recommended finishing all internal work first, and only then calling the external function. This rule, if followed carefully, will allow you to avoid race conditions. However, you need to not only avoid calling external functions too soon, but also avoid calling functions which call external functions. For example, the following is insecure:

```
// INSECURE
mapping (address => uint) private userBalances;
mapping (address => bool) private claimedBonus;
mapping (address => uint) private rewardsForA;

function withdraw(address recipient) public {
    uint amountToWithdraw = userBalances[recipient];
    rewardsForA[recipient] = 0;
    if (!(recipient.call.value(amountToWithdraw)())) { throw; }
}

function getFirstWithdrawalBonus(address recipient) public {
    if (claimedBonus[recipient]) { throw; } // Each recipient should only be
able to claim the bonus once

    rewardsForA[recipient] += 100;
    withdraw(recipient); // At this point, the caller will be able to execute
getFirstWithdrawalBonus again.
    claimedBonus[recipient] = true;
}
```

Even though `getFirstWithdrawalBonus()` doesn't directly call an external contract, the call in `withdraw()` is enough to make it vulnerable to a race condition. you therefore need to treat `withdraw()` as if it were also untrusted.

```
mapping (address => uint) private userBalances;
mapping (address => bool) private claimedBonus;
mapping (address => uint) private rewardsForA;

function untrustedWithdraw(address recipient) public {
    uint amountToWithdraw = userBalances[recipient];
    rewardsForA[recipient] = 0;
    if (!(recipient.call.value(amountToWithdraw)())) { throw; }
}

function untrustedGetFirstWithdrawalBonus(address recipient) public {
    if (claimedBonus[recipient]) { throw; } // Each recipient should only be
able to claim the bonus once

    claimedBonus[recipient] = true;
    rewardsForA[recipient] += 100;
    untrustedWithdraw(recipient); // claimedBonus has been set to true, so
reentry is impossible
}
```

In addition to the fix making reentry impossible, [untrusted functions have been marked](#). This same pattern repeats at every level: since `untrustedGetFirstWithdrawalBonus()` calls `untrustedWithdraw()`, which calls an external contract, you must also treat `untrustedGetFirstWithdrawalBonus()` as insecure.

Another solution often suggested is a [mutex](#). This allows you to "lock" some state so it can only be changed by the owner of the lock. A simple example might look like this:

```
// Note: This is a rudimentary example, and mutexes are particularly useful
where there is substantial logic and/or shared state
mapping (address => uint) private balances;
bool private lockBalances;

function deposit() public returns (bool) {
    if (!lockBalances) {
        lockBalances = true;
        balances[msg.sender] += msg.value;
        lockBalances = false;
        return true;
    }
    throw;
}

function withdraw(uint amount) public returns (bool) {
    if (!lockBalances && amount > 0 && balances[msg.sender] >= amount) {
        lockBalances = true;

        if (msg.sender.call(amount)()) { // Normally insecure, but the mutex
saves it
            balances[msg.sender] -= amount;
        }

        lockBalances = false;
        return true;
    }

    throw;
}
```

If the user tries to call `withdraw()` again before the first call finishes, the lock will prevent it from having any effect. This can be an effective pattern, but it gets tricky when you have multiple contracts that need to cooperate. The following is insecure:

```
// INSECURE
contract StateHolder {
    uint private n;
    address private lockHolder;

    function getLock() {
        if (lockHolder != 0) { throw; }
    }
}
```

```

        lockHolder = msg.sender;
    }

    function releaseLock() {
        lockHolder = 0;
    }

    function set(uint newState) {
        if (msg.sender != lockHolder) { throw; }
        n = newState;
    }
}

```

An attacker can call `getLock()`, and then never call `releaseLock()`. If they do this, then the contract will be locked forever, and no further changes will be able to be made. If you use mutexes to protect against race conditions, you will need to carefully ensure that there are no ways for a lock to be claimed and never released. (There are other potential dangers when programming with mutexes, such as deadlocks and livelocks. You should consult the large amount of literature already written on mutexes, if you decide to go this route.)

\* Some may object to the use of the term *race condition*, since Ethereum does not currently have true parallelism. However, there is still the fundamental feature of logically distinct processes contending for resources, and the same sorts of pitfalls and potential solutions apply.

## DoS with (Unexpected) Throw

Consider a simple auction contract:

```

// INSECURE
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() {
        if (msg.value <= highestBid) { throw; }

        if (!currentLeader.send(highestBid)) { throw; } // Refund the old
        leader, and throw if it fails

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

```

When it tries to refund the old leader, it throws if the refund fails. This means that a malicious bidder can become the leader, while making sure that any refunds to their address will *always* fail. In this way, they can prevent anyone else from calling the `bid()` function, and stay the leader forever. A natural solution might be to continue even if the refund fails, under the theory that it's their own fault if they can't accept the refund.

Another example is when a contract may iterate through an array to pay users (e.g., supporters in a crowdfunding contract). It's common to want to make sure that each payment succeeds. If not, one should throw. The issue is that if one call fails, you are reverting the whole payout system, meaning the loop will never complete. No one gets paid, because one address is forcing an error.

```
address[] private refundAddresses;
mapping (address => uint) public refunds;

// bad
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) { // arbitrary length iteration
based on how many addresses participated
        if(refundAddresses[x].send(refunds[refundAddresses[x]])) {
            throw; // doubly bad, now a single failure on send will hold up all
funds
        }
    }
}
```

Again, the recommended solution is to [favor pull over push payments](#).

## DoS with Block Gas Limit

You may have noticed another problem with the previous example: by paying out to everyone at once, you risk running into the block gas limit. Each Ethereum block can process a certain maximum amount of computation. If you try to go over that, your transaction will fail.

This can lead to problems even in the absence of an intentional attack. However, it's especially bad if an attacker can manipulate the amount of gas needed. In the case of the previous example, the attacker could add a bunch of addresses, each of which needs to get a very small refund. The gas cost of refunding each of the attacker's addresses could therefore end up being more than the gas limit, blocking the refund transaction from happening at all.

This is another reason to [favor pull over push payments](#).

If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions. You will need to keep track of how far you've gone, and be able to resume from that point, as in the following example:



```

struct Payee {
    address addr;
    uint256 value;
}
Payee payees[];
uint256 nextPayeeIndex;

function payOut() {
    uint256 i = nextPayeeIndex;
    while (i < payees.length && msg.gas > 200000) {
        payees[i].addr.send(payees[i].value);
        i++;
    }
    nextPayeeIndex = i;
}

```

You will need to make sure that nothing bad will happen if other transactions are processed while waiting for the next iteration of the `payOut()` function. So only use this pattern if absolutely necessary.

## Timestamp Dependence

The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. *Block numbers* and *average block time* can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper).

```

uint startTime = SOME_START_TIME;

if (now > startTime + 1 week) { // the now can be manipulated by the miner
}

```

## Transaction-Ordering Dependence (TOD)

Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. This can be troublesome for things like decentralized markets, where a transaction to buy some tokens can be seen, and a market order implemented before the other transaction gets included. Protecting against this is difficult, as it would come down to the specific contract itself. For example, in markets, it would be better to implement batch auctions (this also protects against high frequency trading concerns). Another way to use a pre-commit scheme ("I'm going to submit the details later").

## Software Engineering Techniques

---

As we discussed in the [General Philosophy](#) section, it is not enough to protect yourself against the known attacks. Since the cost of failure on a blockchain can be very high, you must also adapt the way you write software, to account for that risk.

The approach we advocate is to "prepare for failure". It is impossible to know in advance whether your code is secure. However, you can architect your contracts in a way that allows them to fail gracefully, and with minimal damage. This section presents a variety of techniques that will help you prepare for failure.

Note: There's always a risk when you add a new component to your system. A badly designed fail-safe could itself become a vulnerability - as can the interaction between a number of well designed fail-safes. Be thoughtful about each technique you use in your contracts, and consider carefully how they work together to create a robust system.

## Upgrading Broken Contracts

Code will need to be changed if errors are discovered or if improvements need to be made. It is no good to discover a bug, but have no way to deal with it.

Designing an effective upgrade system for smart contracts is an area of active research, and we won't be able to cover all of the complications in this document. However, there are two basic approaches that are most commonly used. The simpler of the two is to have a registry contract that holds the address of the latest version of the contract. A more seamless approach for contract users is to have a contract that forwards calls and data onto the latest version of the contract.

Whatever the technique, it's important to have modularization and good separation between components, so that code changes do not break functionality, orphan data, or require substantial costs to port. In particular, it is usually beneficial to separate complex logic from your data storage, so that you do not have to recreate all of the data in order to change the functionality.

It's also critical to have a secure way for parties to decide to upgrade the code. Depending on your contract, code changes may need to be approved by a single trusted party, a group of members, or a vote of the full set of stakeholders. If this process can take some time, you will want to consider if there are other ways to react more quickly in case of an attack, such as an [emergency stop or circuit-breaker](#).

### Example 1: Use a registry contract to store latest version of a contract

In this example, the calls aren't forwarded, so users should fetch the current address each time before interacting with it.

```
contract SomeRegister {
    address backendContract;
    address[] previousBackends;
```

```

address owner;

function SomeRegister() {
    owner = msg.sender;
}

modifier onlyOwner() {
    if (msg.sender != owner) {
        throw;
    }
    -
}

function changeBackend(address newBackend) public
onlyOwner()
returns (bool)
{
    if(newBackend != backendContract) {
        previousBackends.push(backendContract);
        backendContract = newBackend;
        return true;
    }

    return false;
}
}

```

There are two main disadvantages to this approach:

1. Users must always look up the current address, and anyone who fails to do so risks using an old version of the contract
2. You will need to think carefully about how to deal with the contract data, when you replace the contract

The alternate approach is to have a contract forward calls and data to the latest version of the contract:

### Example 2: Use a **DELEGATECALL** to forward data and calls

```

contract Relay {
    address public currentVersion;
    address public owner;

    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
        -
    }

    function Relay(address initAddr) {

```

```

        currentVersion = initAddr;
        owner = msg.sender; // this owner may be another contract with multisig,
not a single contract owner
    }

    function changeContract(address newVersion) public
onlyOwner()
    {
        currentVersion = newVersion;
    }

    function() {
        if(!currentVersion.delegatecall(msg.data)) throw;
    }
}

```

This approach avoids the previous problems, but has problems of its own. You must be extremely careful with how you store data in this contract. If your new contract has a different storage layout than the first, your data may end up corrupted. Additionally, this simple version of the pattern cannot return values from functions, only forward them, which limits its applicability. ([More complex implementations](#) attempt to solve this with in-line assembly code and a registry of return sizes.)

Regardless of your approach, it is important to have some way to upgrade your contracts, or they will become unusable when the inevitable bugs are discovered in them.

## Circuit Breakers (Pause contract functionality)

Circuit breakers stop execution if certain conditions are met, and can be useful when new errors are discovered. For example, most actions may be suspended in a contract if a bug is discovered, and the only action now active is a withdrawal. You can either give certain trusted parties the ability to trigger the circuit breaker, or else have programmatic rules that automatically trigger the certain breaker when certain conditions are met.

Example:

```

bool private stopped = false;
address private owner;

modifier isAdmin() {
    if(msg.sender != owner) {
        throw;
    }
}

function toggleContractActive() isAdmin public
{
    // You can add an additional modifier that restricts stopping a contract to
    be based on another action, such as a vote of users
}

```

```

    stopped = !stopped;
}

modifier stopInEmergency { if (!stopped) _ }
modifier onlyInEmergency { if (stopped) _ }

function deposit() stopInEmergency public
{
    // some code
}

function withdraw() onlyInEmergency public
{
    // some code
}

```

## Speed Bumps (Delay contract actions)

Speed bumps slow down actions, so that if malicious actions occur, there is time to recover. For example, [The DAO](#) required 27 days between a successful request to split the DAO and the ability to do so. This ensured the funds were kept within the contract, increasing the likelihood of recovery. In the case of the DAO, there was no effective action that could be taken during the time given by the speed bump, but in combination with our other techniques, they can be quite effective.

Example:

```

struct RequestedWithdrawal {
    uint amount;
    uint time;
}

mapping (address => uint) private balances;
mapping (address => RequestedWithdrawal) private requestedWithdrawals;
uint constant withdrawalWaitPeriod = 28 days; // 4 weeks

function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
        uint amountToWithdraw = balances[msg.sender];
        balances[msg.sender] = 0; // for simplicity, we withdraw everything;
        // presumably, the deposit function prevents new deposits when
        // withdrawals are in progress

        requestedWithdrawals[msg.sender] = RequestedWithdrawal({
            amount: amountToWithdraw,
            time: now
        });
    }
}

function withdraw() public {

```

```

    if(requestedWithdrawals[msg.sender].amount > 0 && now >
requestedWithdrawals[msg.sender].time + withdrawalWaitPeriod) {
        uint amountToWithdraw = requestedWithdrawals[msg.sender].amount;
        requestedWithdrawals[msg.sender].amount = 0;

        if(!msg.sender.send(amountToWithdraw)) {
            throw;
        }
    }
}

```

## Rate Limiting

Rate limiting halts or requires approval for substantial changes. For example, a depositor may only be allowed to withdraw a certain amount or percentage of total deposits over a certain time period (e.g., max 100 ether over 1 day) - additional withdrawals in that time period may fail or require some sort of special approval. Or the rate limit could be at the contract level, with only a certain amount of tokens issued by the contract over a time period.

### Example

## Assert Guards

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 assertion. 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.)

The following example reverts transactions if the ratio of ether to total number of tokens changes:

```

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

    modifier checkInvariants {
        if (this.balance < totalSupply) throw;
    }

    function deposit(uint amount) public checkInvariants {
        // intentionally vulnerable
        balanceOf[msg.sender] += amount;
        totalSupply += amount;
    }

    function transfer(address to, uint value) public checkInvariants {
        if (balanceOf[msg.sender] >= value) {

```

```

        balanceOf[to] += value;
        balanceOf[msg.sender] -= value;
    }
}

function withdraw() public checkInvariants {
    // intentionally vulnerable
    uint balance = balanceOf[msg.sender];
    if (msg.sender.call.value(balance)()) {
        totalSupply -= balance;
        balanceOf[msg.sender] = 0;
    }
}
}

```

## Contract Rollout

Contracts should have a substantial and prolonged testing period - before substantial money is put at risk.

At minimum, you should:

- Have a full test suite with 100% test coverage (or close to it)
- Deploy on your own testnet
- Deploy on the public testnet with substantial testing and bug bounties
- Exhaustive testing should allow various players to interact with the contract at volume
- Deploy on the mainnet in beta, with limits to the amount at risk

## Automatic Deprecation

During testing, you can force an automatic deprecation by preventing any actions, after a certain time period. For example, an alpha contract may work for several weeks and then automatically shut down all actions, except for the final withdrawal.

```

modifier isActive() {
    if (block.number > SOME_BLOCK_NUMBER) {
        throw;
    }
}

function deposit() public
isActive() {
    // some code
}

function withdraw() public {
    // some code
}

```

}

### **Restrict amount of Ether per user/contract**

In the early stages, you can restrict the amount of Ether for any user (or for the entire contract) - reducing the risk.

## **Security-related Documentation and Procedures**

---

When launching a contract that will have substantial funds or is required to be mission critical, it is important to include proper documentation. Some documentation related to security includes:

### **Status**

- Where current code is deployed
- Current status of deployed code (including outstanding issues, performance stats, etc.)

### **Known Issues**

- Key risks with contract
  - e.g., You can lose all your money, hacker can vote for certain outcomes
- All known bugs/limitations
- Potential attacks and mitigants
- Potential conflicts of interest (e.g., will be using yourself, like Slock.it did with the DAO)

### **History**

- Testing (including usage stats, discovered bugs, length of testing)
- People who have reviewed code (and their key feedback)

### **Procedures**

- Action plan in case a bug is discovered (e.g., emergency options, public notification process, etc.)
- Wind down process if something goes wrong (e.g., funders will get percentage of your balance before attack, from remaining funds)
- Responsible disclosure policy (e.g., where to report bugs found, the rules of any bug bounty program)
- Recourse in case of failure (e.g., insurance, penalty fund, no recourse)

### **Contact Information**

- Who to contact with issues



- Names of programmers and/or other important parties
- Chat room where questions can be asked

## Security Tools

---

- [Oyente](#) - An upcoming tool, will analyze Ethereum code to find common vulnerabilities (e.g., Transaction Order Dependence, no checking for exceptions)
- [Solgraph](#) - Generates a DOT graph that visualizes function control flow of a Solidity contract and highlights potential security vulnerabilities.
- [solint](#) - Another upcoming tool, will provide Solidity linting that helps you enforce consistent conventions and avoid errors in your Solidity smart-contracts.

## Future improvements

---

- **Editor Security Warnings:** Editors will soon alert for common security errors, not just compilation errors. Browser Solidity is getting these features soon.
- **New functional languages that compile to EVM bytecode:** Functional languages gives certain guarantees over procedural languages like Solidity, namely immutability within a function and strong compile time checking. This can reduce the risk of errors by providing deterministic behavior. (for more see [this](#), Curry-Howard correspondence, and linear logic)

## Smart Contract Security Bibliography

---

A lot of this document contains code, examples and insights gained from various parts already written by the community. Here are some of them. Feel free to add more.

### By Ethereum core developers

- [How to Write Safe Smart Contracts](#) (Christian Reitwiessner)
- [Smart Contract Security](#) (Christian Reitwiessner)
- [Thinking about Smart Contract Security](#) (Vitalik Buterin)
- [Solidity](#)
- [Solidity Security Considerations](#)

### By Community

- <http://forum.ethereum.org/discussion/1317/reentrant-contracts>
- <http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/>
- <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>
- <http://hackingdistributed.com/2016/06/22/smart-contract-escape-hatches/>

- <http://martin.swende.se/blog/Devcon1-and-contract-security.html>
- <http://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf>
- <http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour>
- <http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful>
- <http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal>
- <https://blog.blockstack.org/simple-contracts-are-better-contracts-what-we-can-learn-from-the-dao-6293214bad3a>
- <https://blog.slock.it/deja-vu-dao-smart-contracts-audit-results-d26bc088e32e>
- <https://github.com/Bunjini/Rouleth/blob/master/Security.md>
- <https://github.com/LeastAuthority/ethereum-analyses>
- <https://medium.com/@ConsenSys/assert-guards-towards-automated-code-bounties-safe-smart-contract-coding-on-ethereum-8e74364b795c>
- <https://medium.com/@coriacetic/in-bits-we-trust-4e464b418f0b>
- <https://medium.com/@hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007>
- <https://medium.com/@peterborah/we-need-fault-tolerant-smart-contracts-ec1b56596dbc>
- <https://pdaian.com/blog/chasing-the-dao-attackers-wake>
- <http://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf>

## Attribution

---

This work, "Safety", is a derivative of "Smart Contract Best Practices", used under CC BY. "Safety" is licensed under CC BY by the Ethereum community.

## License

---

Licensed under [Apache 2.0](#)

Licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#)

White paper needs some Chinese translation. 

► Pages 206

Basics 

- [Home](#)
- [Ethereum Whitepaper](#)
- [Ethereum Introduction](#)

- [Uses: DAOs and dapps](#)
- [Getting Ether](#)
- [Releases](#)
- [FAQs](#)
- [Design Rationale](#)
- [EVM intro: Ethereum Yellow Paper, Beige Paper and Py-EVM.](#)
- [Wiki for \(old\) website](#) (still a good introduction)
- [Glossary](#)

## R&D

- [Sharding introduction & R&D Compendium, FAQs & roadmap](#)
- [Casper Proof-of-Stake compendium and FAQs.](#)
- [Alternative blockchains, randomness, economics, and other research topics](#)
- [Hard Problems of Cryptocurrency](#)
- [Governance](#)

## Ethereum Virtual Machine (EVM)

### Ethereum clients, tools, wallets, dapp browsers and other projects

## ÐApp Development

### Infrastructure

- [Chain Spec Format](#)
- [Inter-exchange Client Address Protocol](#)
- [URL Hint Protocol](#)
- [NatSpec Determination](#)
- [Network Status](#)
- [Raspberry Pi](#)
- [Mining](#)
- [Licensing](#)
- [Consortium Chain Development](#)

## ÐEV Technologies

- [RLP Encoding](#)
- [Node Discovery Protocol](#)
- [ÐEVp2p Wire Protocol](#)
- [ÐEVp2p Whitepaper \(WiP\)](#)
- [Web3 Secret Storage](#)
- [libp2p](#)

## Ethereum Technologies

- [Patricia Tree](#)
- [Wire protocol](#)
- [Light client protocol](#)
- [Subtleties](#)
- [Solidity Documentation](#)

- [NatSpec Format](#)
- [Contract ABI](#)
- [Bad Block Reporting](#)
- [Bad Chain Canary](#)

### Ethash/Dashimoto

- [Ethash](#)
- [Ethash Yellow Paper appendix](#)
- [Ethash C API](#)
- [Ethash DAG](#)


### Whisper

- [Whisper Proposal](#)
- [Whisper Overview](#)
- [PoC-1 Wire protocol](#)
- [PoC-2 Wire protocol](#)
- [PoC-2 Whitepaper](#)

### Clone this wiki locally

<https://github.com/ethereum/wiki.wiki.git>



 Clone in Desktop