---

title: Smart contract security description: An overview of guidelines for building secure Ethereum smart contracts lang: en

---

Smart contracts are extremely flexible, and capable of controlling large amounts of value and data, while running immutable logic based on code deployed on the blockchain. This has created a vibrant ecosystem of trustless and decentralized applications that provide many advantages over legacy systems. They also represent opportunities for attackers looking to profit by exploiting vulnerabilities in smart contracts.

Public blockchains, like Ethereum, further complicate the issue of securing smart contracts. Deployed contract code *usually* cannot be changed to patch security flaws, while assets stolen from smart contracts are extremely difficult to track and mostly irrecoverable due to immutability.

Although figures vary, it is estimated that the total amount of value stolen or lost due to security defects in smart contracts is easily over $1 billion. This includes high-profile incidents, such as the DAO hack (3.6M ETH stolen, worth over $1B in today's prices), Parity multi-sig wallet hack ($30M lost to hackers), and the Parity frozen wallet issue (over $300M in ETH locked forever).

The aforementioned issues make it imperative for developers to invest effort in building secure, robust, and resilient smart contracts. Smart contract security is serious business, and one that every developer will do well to learn. This guide will cover security considerations for Ethereum developers and explore resources for improving smart contract security.

Prerequisites {#prerequisites}

Make sure you're familiar with the fundamentals of smart contract development before tackling security.

Guidelines for building secure Ethereum smart contracts {#smart-contract-security-guidelines}

1. Design proper access controls {#design-proper-access-controls}

In smart contracts, functions marked `public` or `external` can be called by any externally owned accounts (EOAs) or contract accounts. Specifying public visibility for functions is necessary if you want others to interact with your contract. Functions marked `private` however can only be called by functions within the smart contract, and not external accounts. Giving every network participant access to contract functions can cause problems, especially if it means anyone can perform sensitive operations (e.g., minting new tokens).

To prevent unauthorized use of smart contract functions, it is necessary to implement secure access controls. Access control mechanisms restrict the ability to use certain functions in a smart contract to approved entities, such as accounts responsible for managing the contract. The **Ownable pattern** and **role-based control** are two patterns useful for implementing access control in smart contracts:

Ownable pattern {#ownable-pattern}

In the Ownable pattern, an address is set as the "owner" of the contract during the contract-creation process. Protected functions are assigned an `OnlyOwner` modifier, which ensures the contract authenticates the identity of the calling address before executing the function. Calls to protected functions from other addresses aside from the contract owner always revert, preventing unwanted access.

Role-based access control {#role-based-access-control}

Registering a single address as `Owner` in a smart contract introduces the risk of centralization and represents a single point-of-failure. If the owner's account keys are compromised, attackers can attack the owned contract. This is why using a role-based access control pattern with multiple administrative accounts may be a better option.

In role-based access control, access to sensitive functions is distributed between a set of trusted participants. For instance, one account may be responsible for minting tokens, while another account performs upgrades or pauses the contract. Decentralizing access control this way eliminates single points of failure and reduces trust assumptions for users.

Using multi-signature wallets

Another approach for implementing secure access control is using a multi-signature account to manage a contract. Unlike a regular EOA, multi-signature accounts are owned by multiple entities and require signatures from a minimum number of accounts—say 3-of-5—to execute transactions.

Using a multisig for access control introduces an extra layer of security since actions on the target contract require consent from multiple parties. This is particularly useful if using the Ownable pattern is necessary, as it makes it more difficult for an attacker or rogue insider to manipulate sensitive contract functions for malicious purposes.

2. Use require(), assert(), and revert() statements to guard contract operations {#use-require-assert-revert}

As mentioned, anyone can call public functions in your smart contract once it is deployed on the blockchain. Since you cannot know in advance how external accounts will interact with a contract, it is ideal to implement internal safeguards against problematic operations before deploying. You can enforce correct behavior in smart contracts by using the `require()`, `assert()`, and `revert()` statements to trigger exceptions and revert state changes if execution fails to satisfy certain requirements.

`require()`: `require` are defined at the start of functions and ensures predefined conditions are met before the called function is executed. A `require` statement can be used to validate user inputs, check state variables, or authenticate the identity of the calling account before progressing with a function.

`assert()`: `assert()` is used to detect internal errors and check for violations of “invariants” in your code. An invariant is a logical assertion about a contract’s state that should hold true for all function executions. An example invariant is the maximum total supply or balance of a token contract. Using `assert()` ensures that your contract never reaches a vulnerable state, and if it does, all changes to state variables are rolled back.

`revert()`: `revert()` can be used in an if-else statement that triggers an exception if the required condition is not satisfied. The sample contract below uses `revert()` to guard the execution of functions:

``` pragma solidity ^0.8.4;

contract VendingMachine { address owner; error Unauthorized(); function buy(uint amount) public payable { if (amount > msg.value / 2 ether) revert("Not enough Ether provided."); // Perform the purchase. } function withdraw() public { if (msg.sender != owner) revert Unauthorized();

```
    payable(msg.sender).transfer(address(this).balance);
}
```

} ```

## 3. Test smart contracts and verify code correctness {#test-smart-contracts-and-verify-code-correctness}

The immutability of code running in the Ethereum Virtual Machine means smart contracts demand a higher level of quality assessment during the development phase. Testing your contract extensively and observing it for any unexpected results will improve security a great deal and protect your users in the long run.

The usual method is to write small unit tests using mock data that the contract is expected to receive from users. Unit testing is good for testing the functionality of certain functions and ensuring a smart contract works as expected.

Unfortunately, unit testing is minimally effective for improving smart contract security when used in isolation. A unit test might prove a function executes properly for mock data, but unit tests are only as effective as the tests that are written. This makes it difficult to detect missed edge cases and vulnerabilities that could break the safety of your smart contract.

A better approach is to combine unit testing with property-based testing performed using static and dynamic analysis. Static analysis relies on low-level representations, such as control flow graphs and abstract syntax trees to analyze reachable program states and execution paths. Meanwhile, dynamic analysis techniques, such as fuzzing, execute contract code with random input values to detect operations that violate security properties.

Formal verification is another technique for verifying security properties in smart contracts. Unlike regular testing, formal verification can conclusively prove the absence of errors in a smart contract. This is achieved by creating a formal specification that captures desired security properties and proving that a formal model of the contracts adheres to this specification.
```

4. Ask for an independent review of your code {#get-independent-code-reviews}

After testing your contract, it is good to ask others to check the source code for any security issues. Testing will not uncover every flaw in a smart contract, but getting an independent review increases the possibility of spotting vulnerabilities.

Audits {#audits}

Commissioning a smart contract audit is one way of conducting an independent code review. Auditors play an important role in ensuring that smart contracts are secure and free from quality defects and design errors.

That said, you should avoid treating audits as a silver bullet. Smart contract audits won't catch every bug and are mostly designed to provide an additional round of reviews, which can help detect issues missed by developers during initial development and testing. You should also follow best practices for working with auditors, such as documenting code properly and adding inline comments, to maximize the benefit of a smart contract audit.

Bug bounties {#bug-bounties}

Setting up a bug bounty program is another approach for implementing external code reviews. A bug bounty is a financial reward given to individuals (usually whitehat hackers) that discover vulnerabilities in an application.

When used properly, bug bounties give members of the hacker community incentive to inspect your code for critical flaws. A real-life example is the “infinite money bug” that would have let an attacker create an unlimited amount of Ether on Optimism, a Layer 2 protocol running on Ethereum. Fortunately, a whitehat hacker discovered the flaw and notified the team, earning a large payout in the process.

A useful strategy is to set the payout of a bug bounty program in proportion to the amount of funds at stake. Described as the “scaling bug bounty”, this approach provides financial incentives for individuals to responsibly disclose vulnerabilities instead of exploiting them.

5. Follow best practices during smart contract development {#follow-smart-contract-development-best-practices}

The existence of audits and bug bounties doesn’t excuse your responsibility to write high-quality code. Good smart contract security starts with following proper design and development processes:

- Store all code in a version control system, such as git
- Make all code modifications via pull requests
- Ensure pull requests have at least one independent reviewer—if you are working solo on a project, consider finding other developers and trade code reviews
- Use a development environment for testing, compiling, deploying smart contracts
- Run your code through basic code analysis tools, such as Mythril and Slither. Ideally, you should do this before each pull request is merged and compare differences in output
- Ensure your code compiles without errors, and the Solidity compiler emits no warnings
- Properly document your code (using NatSpec) and describe details about the contract architecture in easy-to-understand language. This will make it easier for others to audit and review your code.

6. Implement robust disaster recovery plans {#implement-disaster-recovery-plans}

Designing secure access controls, implementing function modifiers, and other suggestions can improve smart contract security, but they cannot rule out the possibility of malicious exploits. Building secure smart contracts requires “preparing for failure” and having a fallback plan for responding effectively to attacks. A proper disaster recovery plan will incorporate some or all of the following components:

Contract upgrades {#contract-upgrades}

While Ethereum smart contracts are immutable by default, it is possible to achieve some degree of mutability by using upgrade

patterns. Upgrading contracts is necessary in cases where a critical flaw renders your old contract unusable and deploying new logic is the most feasible option.

Contract upgrade mechanisms work differently, but the “proxy pattern” is one of the more popular approaches for upgrading smart contracts. Proxy patterns split an application’s state and logic between *two* contracts. The first contract (called a ‘proxy contract’) stores state variables (e.g., user balances), while the second contract (called a ‘logic contract’) holds the code for executing contract functions.

Accounts interact with the proxy contract, which dispatches all function calls to the logic contract using the `delegatecall()` low-level call. Unlike a regular message call, `delegatecall()` ensures the code running at the logic contract’s address is executed in the context of the calling contract. This means the logic contract will always write to the proxy’s storage (instead of its own storage) and the original values of `msg.sender` and `msg.value` are preserved.

Delegating calls to the logic contract requires storing its address in the proxy contract's storage. Hence, upgrading the contract's logic is only a matter of deploying another logic contract and storing the new address in the proxy contract. As subsequent calls to the proxy contract are automatically routed to the new logic contract, you would have “upgraded” the contract without actually modifying the code.

More on upgrading contracts.

Emergency stops {#emergency-stops}

As mentioned, extensive auditing and testing cannot possibly discover all bugs in a smart contract. If a vulnerability appears in your code after deployment, patching it is impossible since you cannot change the code running at the contract address. Also, upgrade mechanisms (e.g., proxy patterns) may take time to implement (they often require approval from different parties), which only gives attackers more time to cause more damage.

The nuclear option is to implement an “emergency stop” function that blocks calls to vulnerable functions in a contract. Emergency stops typically comprise the following components:

1. A global Boolean variable indicating if the smart contract is in a stopped state or not. This variable is set to `false` when setting up the contract, but will revert to `true` once the contract is stopped.
2. Functions that reference the Boolean variable in their execution. Such functions are accessible when the smart contract is not stopped, and become inaccessible when the emergency stop feature is triggered.
3. An entity that has access to the emergency stop function, which sets the Boolean variable to `true`. To prevent malicious actions, calls to this function can be restricted to a trusted address (e.g., the contract owner).

Once the contract activates the emergency stop, certain functions will not be callable. This is achieved by wrapping select functions in a modifier that references the global variable. Below is an example describing an implementation of this pattern in contracts:

```solidity // This code has not been professionally audited and makes no promises about safety or correctness. Use at your own risk.

contract EmergencyStop {

```
bool isStopped = false;

modifier stoppedInEmergency {
    require(!isStopped);
    _;
}

modifier onlyWhenStopped {
    require(isStopped);
    _;
}

modifier onlyAuthorized {
    // Check for authorization of msg.sender here
    _;
}

function stopContract() public onlyAuthorized {
    isStopped = true;
}

function resumeContract() public onlyAuthorized {
```
```

```
    isStopped = false;
}

function deposit() public payable stoppedInEmergency {
    // Deposit logic happening here
}

function emergencyWithdraw() public onlyWhenStopped {
    // Emergency withdraw happening here
}

} ```
```

This example shows the basic features of emergency stops:

- `isStopped` is a Boolean that evaluates to `false` at the beginning and `true` when the contract enters emergency mode.
- The function modifiers `onlyWhenStopped` and `stoppedInEmergency` check the `isStopped` variable. `stoppedInEmergency` is used to control functions that should be inaccessible when the contract is vulnerable (e.g., `deposit()`). Calls to these functions will simply revert.

`onlyWhenStopped` is used for functions that should be callable during an emergency (e.g.,`emergencyWithdraw()`). Such functions can help resolve the situation, hence their exclusion from the “restricted functions” list.

Using an emergency stop functionality provides an effective stopgap for dealing with serious vulnerabilities in your smart contract. However, it increases the need for users to trust developers not to activate it for self-serving reasons. To this end, decentralizing control of the emergency stop either by subjecting it to an on-chain voting mechanism, timelock, or approval from a multisig wallet are possible solutions.

Event monitoring {#event-monitoring}

Events allow you to track calls to smart contract functions and monitor changes to state variables. It is ideal to program your smart contract to emit an event whenever some party takes a safety-critical action (e.g., withdrawing funds).

Logging events and monitoring them off-chain provides insights on contract operations and aids faster discovery of malicious actions. This means your team can respond faster to hacks and take action to mitigate impact on users, such as pausing functions or performing an upgrade.

You can also opt for an off-the-shelf monitoring tool that automatically forwards alerts whenever someone interacts with your contracts. These tools will allow you to create custom alerts based on different triggers, such as transaction volume, frequency of function calls, or the specific functions involved. For example, you could program an alert that comes in when the amount withdrawn in a single transaction crosses a particular threshold.

7. Design secure governance systems {#design-secure-governance-systems}

You may want to decentralize your application by turning over control of core smart contracts to community members. In this case, the smart contract system will include a governance module—a mechanism that allows community members to approve administrative actions via an on-chain governance system. For example, a proposal to upgrade a proxy contract to a new implementation may be voted upon by token-holders.

Decentralized governance can be beneficial, especially because it aligns the interests of developers and end-users. Nevertheless, smart contract governance mechanisms may introduce new risks if implemented incorrectly. A plausible scenario is if an attacker acquires enormous voting power (measured in number of tokens held) by taking out a flash loan and pushes through a malicious proposal.

One way of preventing problems related to on-chain governance is touse a timelock. A timelock prevents a smart contract from executing certain actions until a specific amount of time passes. Other strategies include assigning a “voting weight” to each token based on how long it has been locked up for, or measuring the voting power of an address at a historical period (for example, 2-3 blocks in the past) instead of the current block. Both methods reduce the possibility of quickly amassing voting power to swing on-chain votes.

More on designing secure governance systems and different voting mechanisms in DAOs.

8. Reduce complexity in code to a minimum {#reduce-code-complexity}

Traditional software developers are familiar with the KISS (“keep it simple, stupid”) principle, which advises against introducing unnecessary complexity into software design. This follows the long-held thinking that “complex systems fail in complex ways” and are more susceptible to costly errors.

Keeping things simple is of particular importance when writing smart contracts, given that smart contracts are potentially controlling large amounts of value. A tip for achieving simplicity when writing smart contracts is to reuse existing libraries, such as OpenZeppelin Contracts, where possible. Because these libraries have been extensively audited and tested by developers, using them reduces the chances of introducing bugs by writing new functionality from scratch.

Another common advice is to write small functions and keep contracts modular by splitting business logic across multiple contracts. Not only does writing simpler code reduce the attack surface in a smart contract, it also makes it easier to reason about the correctness of the overall system and detect possible design errors early.

9. Defend against common smart contract vulnerabilities {#mitigate-common-smart-contract-vulnerabilities}

Reentrancy {#reentrancy}

The EVM doesn’t permit concurrency, meaning two contracts involved in a message call cannot run simultaneously. An external call pauses the calling contract's execution and memory until the call returns, at which point execution proceeds normally. This process can be formally described as transferring control flow to another contract.

Although mostly harmless, transferring control flow to untrusted contracts can cause problems, such as reentrancy. A reentrancy attack occurs when a malicious contract calls back into a vulnerable contract before the original function invocation is complete. This type of attack is best explained with an example.

Consider a simple smart contract (‘Victim’) that allows anyone to deposit and withdraw Ether:

```solidity // This contract is vulnerable. Do not use in production

contract Victim { mapping (address => uint256) public balances;

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

function withdraw() external {
    uint256 amount = balances[msg.sender];
    (bool success, ) = msg.sender.call.value(amount)("");
    require(success);
    balances[msg.sender] = 0;
}

} ```

This contract exposes a `withdraw()` function to allow users to withdraw ETH previously deposited in the contract. When processing a withdrawal, the contract performs the following operations:

1. Checks the user’s ETH balance
2. Sends funds to the calling address
3. Resets their balance to 0, preventing additional withdrawals from the user

The `withdraw()` function in `Victim` contract follows a “checks-interactions-effects” pattern. It *checks* if conditions necessary for execution are satisfied (i.e., the user has a positive ETH balance) and performs the *interaction* by sending ETH to the caller’s address, before applying the *effects* of the transaction (i.e., reducing the user’s balance).

If `withdraw()` is called from an externally owned account (EOA), the function executes as expected: `msg.sender.call.value()` sends ETH to the caller. However, if `msg.sender` is a smart contract account calls `withdraw()`, sending funds using `msg.sender.call.value()` will also trigger code stored at that address to run.

Imagine this is the code deployed at the contract address:

```solidity contract Attacker { function beginAttack() external payable { Victim(victim_address).deposit.value(1 ether)(); Victim(victim_address).withdraw(); }

function() external payable {

```
    if (gasleft() > 40000) {
        Victim(victim_address).withdraw();
    }
}

}```

This contract is designed to do three things:

1. Accept a deposit from another account (likely the attacker's EOA)
2. Deposit 1 ETH into the Victim contract
3. Withdraw the 1 ETH stored in the smart contract

There's nothing wrong here, except that `Attacker` has another function that calls `withdraw()` in `Victim` again if the gas left over from the incoming `msg.sender.call.value` is more than 40,000. This gives `Attacker` the ability to reenter `Victim` and withdraw more funds *before* the first invocation of `withdraw` completes. The cycle looks like this:

```
solidity - Attacker's EOA calls `Attacker.beginAttack()` with 1 ETH - `Attacker.beginAttack()` deposits 1 ETH into `Victim` - `Attacker` calls `withdraw() in `Victim` - `Victim` checks `Attacker`'s balance (1 ETH) - `Victim` sends 1 ETH to `Attacker` (which triggers the default function) - `Attacker` calls `Victim.withdraw()` again (note that `Victim` hasn't reduced `Attacker`'s balance from the first withdrawal) - `Victim` checks `Attacker`'s balance (which is still 1 ETH because it hasn't applied the effects of the first call) - `Victim` sends 1 ETH to `Attacker` (which triggers the default function and allows `Attacker` to reenter the `withdraw` function) - The process repeats until `Attacker` runs out of gas, at which point `msg.sender.call.value` returns without triggering additional withdrawals - `Victim` finally applies the results of the first transaction (and subsequent ones) to its state, so `Attacker`'s balance is set to 0
```

The summary is that because the caller's balance isn't set to 0 until the function execution completes, subsequent invocations will succeed and allow the caller to withdraw their balance multiple times. This kind of attack can be used to drain a smart contract of its funds, like what happened in the 2016 DAO hack. Reentrancy attacks are still a critical issue for smart contracts today as public listings of reentrancy exploits show.

How to prevent reentrancy attacks

An approach to dealing with reentrancy is following the checks-effects-interactions pattern. This pattern orders the execution of functions in a way that code that performs necessary checks before progressing with execution comes first, followed by code that manipulates contract state, with code that interacts with other contracts or EOAs arriving last.

The checks-effect-interaction pattern is used in a revised version of the `Victim` contract shown below:

```
solidity contract NoLongerAVictim { function withdraw() external { uint256 amount = balances[msg.sender]; balances[msg.sender] = 0; (bool success, ) = msg.sender.call.value(amount)(""); require(success); } }
```

This contract performs a *check* on the user's balance, applies the *effects* of the `withdraw()` function (by resetting the user's balance to 0), and proceeds to perform the *interaction* (sending ETH to the user's address). This ensures the contract updates its storage before the external call, eliminating the re-entrancy condition that enabled the first attack. The `Attacker` contract could still call back into `NoLongerAVictim`, but since `balances[msg.sender]` has been set to 0, additional withdrawals will throw an error.

Another option is to use a mutual exclusion lock (commonly described as a "mutex") that locks a portion of a contract's state until a function invocation completes. This is implemented using a Boolean variable that is set to `true` before the function executes and reverts to `false` after the invocation is done. As seen in the example below, using a mutex protects a function against recursive calls while the original invocation is still processing, effectively stopping reentrancy.

```solidity pragma solidity ^0.7.0;

contract MutexPattern { bool locked = false; mapping(address => uint256) public balances;

modifier noReentrancy() {
    require(!locked, "Blocked from reentrancy.");
    locked = true;
    _;
    locked = false;
}
// This function is protected by a mutex, so reentrant calls from within `msg.sender.call` cannot call `withdraw` again.
//  The `return` statement evaluates to `true` but still evaluates the `locked = false` statement in the modifier
function withdraw(uint _amount) public payable noReentrancy returns(bool) {
    require(balances[msg.sender] >= _amount, "No balance to withdraw.");

    balances[msg.sender] -= _amount;
    bool (success, ) = msg.sender.call{value: _amount}("");
    require(success);
```
```

```
    return true;
}

} ```
```

You can also use a pull payments system that requires users to withdraw funds from the smart contracts, instead of a "push payments" system that sends funds to accounts. This removes the possibility of inadvertently triggering code at unknown addresses (and can also prevent certain denial-of-service attacks).

Integer underflows and overflows {#integer-underflows-and-overflows}

An integer overflow occurs when the results of an arithmetic operation falls outside the acceptable range of values, causing it to "roll over" to the lowest representable value. For example, a `uint8` can only store values up to 2^8-1=255. Arithmetic operations that result in values higher than `255` will overflow and reset `uint` to `0`, similar to how the odometer on a car resets to 0 once it reaches the maximum mileage (999999).

Integer underflows happen for similar reasons: the the results of an arithmetic operation falls below the acceptable range. Say you tried decrementing `0` in a `uint8`, the result would simply roll over to the maximum representable value (`255`).

Both integer overflows and underflows can lead to unexpected changes to a contract's state variables and result in unplanned execution. Below is an example showing how an attacker can exploit arithmetic overflow in a smart contract to perform an invalid operation:

``` pragma solidity ^0.7.6;

// This contract is designed to act as a time vault. // User can deposit into this contract but cannot withdraw for at least a week. // User can also extend the wait time beyond the 1 week waiting period.

/* 1. Deploy TimeLock 2. Deploy Attack with address of TimeLock 3. Call Attack.attack sending 1 ether. You will immediately be able to withdraw your ether.

What happened? Attack caused the TimeLock.lockTime to overflow and was able to withdraw before the 1 week waiting period. */

contract TimeLock { mapping(address => uint) public balances; mapping(address => uint) public lockTime;

```
function deposit() external payable {
    balances[msg.sender] += msg.value;
    lockTime[msg.sender] = block.timestamp + 1 weeks;
}

function increaseLockTime(uint _secondsToIncrease) public {
    lockTime[msg.sender] += _secondsToIncrease;
}

function withdraw() public {
    require(balances[msg.sender] > 0, "Insufficient funds");
    require(block.timestamp > lockTime[msg.sender], "Lock time not expired");

    uint amount = balances[msg.sender];
    balances[msg.sender] = 0;

    (bool sent,) = msg.sender.call{value: amount}("");
    require(sent, "Failed to send Ether");
}
```

}

contract Attack { TimeLock timeLock;

```
constructor(TimeLock _timeLock) {
    timeLock = TimeLock(_timeLock);
}

fallback() external payable {}

function attack() public payable {
    timeLock.deposit{value: msg.value}();
    /*
    if t = current lock time then we need to find x such that
    x + t = 2**256 = 0
    so x = -t
    2**256 = type(uint).max + 1
    so x = type(uint).max + 1 - t
    */
```
```

```
    timeLock.increaseLockTime(
        type(uint).max + 1 - timeLock.lockTime(address(this))
    );
    timeLock.withdraw();
}

} ```
```

How to prevent integer underflows and overflows

As of version 0.8.0, the Solidity compiler rejects code that results in integer underflows and overflows. However, contracts compiled with a lower compiler version should either perform checks on functions involving arithmetic operations or use a library (e.g., SafeMath) that checks for underflow/overflow.

Oracle manipulation {#oracle-manipulation}

Oracles source off-chain information and send it on-chain for smart contracts to use. With oracles, you can design smart contracts that interoperate with off-chain systems, such as capital markets, greatly expanding their application.

But if the oracle is corrupted and sends incorrect information on-chain, smart contracts will execute based on erroneous inputs, which can cause problems. This is the basis of the “oracle problem”, which concerns the task of making sure information from a blockchain oracle is accurate, up-to-date, and timely.

A related security concern is using an on-chain oracle, such as a decentralized exchange, to get the spot price for an asset. Lending platforms in the decentralized finance (DeFi) industry often do this to determine the value of a user’s collateral to determine how much they can borrow.

DEX prices are often accurate, largely due to arbitrageurs restoring parity in markets. However, they are open to manipulation, particularly if the on-chain oracle calculates asset prices based on historical trading patterns (as is usually the case).

For instance, an attacker could artificially pump the spot price of an asset by taking out a flash loan right before interacting with your lending contract. Querying the DEX for the asset’s price would return a higher-than-normal value (due to the attacker’s large “buy order” skewing demand for the asset), allowing them to borrow more than they should. Such "flash loan attacks" have been used to exploit reliance on price oracles among DeFi applications, costing protocols millions in lost funds.

How to prevent oracle manipulation

The minimum requirement to avoid oracle manipulation is to use a decentralized oracle network that queries information from multiple sources to avoid single points of failure. In most cases, decentralized oracles have built-in cryptoeconomic incentives to encourage oracle nodes to report correct information, making them more secure than centralized oracles.

If you plan on querying an on-chain oracle for asset prices, consider using one that implements a time-weighted average price (TWAP) mechanism. A TWAP oracle queries the price of an asset at two different points in time (which you can modify) and calculates the spot price based on the average obtained. Choosing longer time periods protects your protocol against price manipulation since large orders executed recently cannot impact asset prices.

Smart contract security resources for developers {#smart-contract-security-resources-for-developers}

Tools for analyzing smart contracts and verifying code correctness {#code-analysis-tools}

- **Testing tools and libraries** - *Collection of industry-standard tools and libraries for performing unit tests, static analysis, and dynamic analysis on smart contracts.*
- **Formal verification tools** - *Tools for verifying functional correctness in smart contracts and checking invariants.*
- **Smart contract auditing services** - *Listing of organizations providing smart contract auditing services for Ethereum development projects.*
- **Bug bounty platforms** - *Platforms for coordinating bug bounties and rewarding responsible disclosure of critical vulnerabilities in smart contracts.*
- **Fork Checker** - *A free online tool for checking all available information regarding a forked contract.*

- **ABI Encoder** - *A free online service for encoding your Solidity contract functions and constructor arguments.*

Tools for monitoring smart contracts {#smart-contract-monitoring-tools}

- **OpenZeppelin Defender Sentinels** - *A tool for automatically monitoring and responding to events, functions, and transaction parameters on your smart contracts.*
- **Tenderly Real-Time Alerting** - *A tool for getting real-time notifications when unusual or unexpected events happen on your smart contracts or wallets.*

Tools for secure administration of smart contracts {#smart-contract-administration-tools}

- **OpenZeppelin Defender Admin** - *Interface for managing smart contract administration, including access controls, upgrades, and pausing.*
- **Safe** - *Smart contract wallet running on Ethereum that requires a minimum number of people to approve a transaction before it can occur (M-of-N).*
- **OpenZeppelin Contracts** - *Contract libraries for implementing administrative features, including contract ownership, upgrades, access controls, governance, pauseability, and more.*

Smart contract auditing services {#smart-contract-auditing-services}

- **ConsenSys Diligence** - *Smart contract auditing service helping projects across the blockchain ecosystem ensure their protocols are ready for launch and built to protect users.*
- **CertiK** - *Blockchain security firm pioneering the use of cutting-edge formal Verification technology on smart contracts and blockchain networks.*
- **Trail of Bits** - *Cybersecurity company that combines security research with an attacker mentality to reduce risk and fortify code.*
- **PeckShield** - *Blockchain security company offering products and services for the security, privacy, and usability of the entire blockchain ecosystem.*
- **QuantStamp** - *Auditing service facilitating the mainstream adoption of blockchain technology through security and risk assessment services.*
- **OpenZeppelin** - *Smart contract security company providing security audits for distributed systems.*
- **Runtime Verification** - *Security company specializing in formal modeling and verification of smart contracts.*
- **Hacken** - *Web3 cybersecurity auditor bringing the 360-degree approach to blockchain security.*
- **Nethermind** - *Solidity and Cairo auditing services, ensuring the integrity of smart contracts and the safety of users across Ethereum and Starknet.*
- **HashEx** - *HashEx focuses on blockchain and smart contract auditing to ensure the security of cryptocurrencies, providing services such as smart contract development, penetration testing, blockchain consulting.*
- **Code4rena** - *Competitive audit platform that incentivizes smart contract security experts to find vulnerabilities and help make web3 more secure.*

Bug bounty platforms {#bug-bounty-platforms}

- **Immunefi** - *Bug bounty platform for smart contracts and DeFi projects, where security researchers review code, disclose vulnerabilities, get paid, and make crypto safer.*
- **HackerOne** - *Vulnerability coordination and bug bounty platform that connects businesses with penetration testers and cybersecurity researchers.*
- **HackenProof** - *Expert bug bounty platform for crypto projects (DeFi, Smart Contracts, Wallets, CEX and more), where security professionals provide triage services and researchers get paid for relevant, verified bug reports.*

Publications of known smart contract vulnerabilities and exploits {#common-smart-contract-vulnerabilities-and-exploits}

- **ConsenSys: Smart Contract Known Attacks** - *Beginner-friendly explanation of the most significant contract vulnerabilities, with sample code for most cases.*
- **SWC Registry** - *Curated list of Common Weakness Enumeration (CWE) items that apply to Ethereum smart contracts.*
- **Rekt** - *Regularly updated publication of high-profile crypto hacks and exploits, along with detailed post-mortem reports.*

Challenges for learning smart contract security {#challenges-for-learning-smart-contract-security}

- **Awesome BlockSec CTF** - *Curated list of blockchain security wargames, challenges, and Capture The Flag competitions and solution writeups.*
- **Damn Vulnerable DeFi** - *Wargame to learn offensive security of DeFi smart contracts and build skills in bug-hunting and security auditing.*
- **Ethernaut** - *Web3/Solidity-based wargame where each level is a smart contract that needs to be 'hacked'.*

Best practices for securing smart contracts {#smart-contract-security-best-practices}

- **ConsenSys: Ethereum Smart Contract Security Best Practices** - *Comprehensive list of guidelines for securing Ethereum smart contracts.*
- **Nascent: Simple Security Toolkit** - *Collection of practical security-focused guides and checklists for smart contract development.*
- **Solidity Patterns** - *Useful compilation of secure patterns and best practices for the smart contract programming language Solidity.*
- **Solidity Docs: Security Considerations** - *Guidelines for writing secure smart contracts with Solidity.*
- **Smart Contract Security Verification Standard** - *Fourteen-part checklist created to standardize the security of smart contracts for developers, architects, security reviewers and vendors.*

Tutorials on smart contract security {#tutorials-on-smart-contract-security}

- How to write secure smart contracts
- How to use Slither to find smart contract bugs
- How to use Manticore to find smart contract bugs
- Smart contract security guidelines
- How to safely integrate your token contract with arbitrary tokens