# Solidity Smart Contract Security

**Learn how to implement secure code, and avoid known attackable issues**

Daniel (Sơn) PHAM

Solidity Developer Bootcamp

Ngày 31 tháng 7 năm 2023

## Outline

1. Detecting Various Vulnerability
2. Best Practices for Smart Contract Security
3. Tools for writing secure smart contracts
4. QnA

## Outline

1. Detecting Various Vulnerability
   1.1 Rentrancy Attack
   1.2 Costly operations
   1.3 Divide Before Multiplying
   1.4 Frontrunning
   1.5 Block.timestamp
   1.6 Tx.origin
   1.7 Race Condition
   1.8 Denial of Service (DoS)
   1.9 Unchecked send
   1.10 Overflow and Underflow
   1.11 Poor Visibility Specifiers
   1.12 Emergency stops

2. Best Practices for Smart Contract Security

3. Tools for writing secure smart contracts

4. QnA

## Rentrancy Attack

It occurs when a contract's function can be called multiple times before the previous call completes, potentially leading to unintended consequences and loss of funds.
The reentrancy attack became widely known after the infamous DAO (Decentralized Autonomous Organization) hack in 2016.

## Costly operations

Costly operations, also known as gas-guzzlers, refer to computationally expensive operations that consume a significant amount of gas on the Ethereum blockchain. High gas consumption can lead to several issues, including:

- Increased transaction fees
- Block gas limit
- Denial of Service (DoS) attacks

## Divide Before Multiplying

A specific coding mistake that can lead to unexpected and incorrect results in your smart contract. This vulnerability arises when division is performed before multiplication in an arithmetic expression, leading to incorrect computations and potential security risks.

## Frontrunning

Frontrunning is a vulnerability that can occur in Solidity smart contracts when a malicious actor exploits the time delay between a transaction being publicly announced and being confirmed on the blockchain. This can happen in scenarios where multiple users are trying to interact with the same contract simultaneously, and the malicious actor tries to "front-run" the other users' transactions to gain an advantage or manipulate the contract's state.

## Block.timestamp

Exploitation of the miner's ability to manipulate the timestamp to some extent. Since miners have some control over the timestamp of the block they are mining, malicious miners can potentially set the timestamp to a future date or a date in the past to their advantage, leading to various security issues.

## Tx.origin

The "Tx.origin" vulnerability arises when a smart contract relies on tx.origin for authentication or access control. Malicious actors can exploit this vulnerability by employing a technique called "phishing" or "proxy" attacks.

## Race Condition

In Solidity, race conditions can occur when multiple transactions try to modify the same state variables within a smart contract. These transactions may be processed in an unpredictable order by the miners, leading to unexpected and potentially undesirable outcomes.

## Denial of Service

Denial of Service (DoS) is a vulnerability in smart contracts that allows attackers to disrupt the normal functioning of the contract, making it unresponsive or causing excessive gas consumption.

## Unchecked send

The send() function is a low-level method for transferring Ether, and if it fails (e.g., due to out-of-gas issues or contract execution failures), it returns false. If the return value is not explicitly checked, it can lead to unexpected behavior in the contract, and the Ether might get locked in the contract forever.

## Overflow and Underflow

In Solidity, overflow and underflow vulnerabilities can occur when dealing with numeric data types such as uint and int. These vulnerabilities arise when arithmetic operations result in a value that exceeds the maximum or minimum representable value for the data type, respectively.

## Poor Visibility Specifiers

There are four visibility specifiers in Solidity: public, external, internal, and private.
The vulnerability related to poor visibility specifiers arises when the visibility specifiers are not properly set, leading to potential security issues. The incorrect use of visibility specifiers can expose sensitive data or allow unauthorized access to contract functions, which can be exploited by attackers.

## Emergency stops

In Solidity, an emergency stop is a mechanism used to pause certain critical functionalities of a smart contract in the event of a security breach or unexpected behavior. This allows developers to mitigate potential risks and protect users' funds until the issue is resolved. The emergency stop pattern is commonly used to prevent further damage and loss of assets when vulnerabilities are detected.

## Outline

1. Detecting Various Vulnerability

2. Best Practices for Smart Contract Security
   - 2.1 Prepare for Failure
   - 2.2 Stay Up-to-date
   - 2.3 Keep it Simple
   - 2.4 Rolling out
   - 2.5 Be Aware of the EVM's Idiosyncrasies

3. Tools for writing secure smart contracts

4. QnA

## 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

## Stay Up-to-date

Keep track of new security developments.

- Check your contracts for any new bug as soon as it is discovered
- Upgrade to the latest version of any tool or library as soon as possible
- Adopt new security techniques that appear useful

## Keep it 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

## Rolling out

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

## Be Aware of the EVM's Idiosyncrasies

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 and in any order. The private data in smart contracts is also viewable by anyone
- Keep gas costs and the block gas limit in mind
- Be aware that timestamps are imprecise on a blockchain, miners can influence the time of execution of a transaction within a margin of several seconds
- Randomness is non-trivial on blockchain, most approaches to random number generation are gameable on a blockchain

## Outline

1. Detecting Various Vulnerability

2. Best Practices for Smart Contract Security

3. Tools for writing secure smart contracts

    3.1 Tools for writing secure smart contracts

4. QnA

## Tools

- Slither
- echidna
- Solgraph
- Surya
- Solhint
- Mythril

## Outline

1. Detecting Various Vulnerability

2. Best Practices for Smart Contract Security

3. Tools for writing secure smart contracts

4. **QnA**

## QnA