Owasp Smart Contract Top 10

At The Hacker's Meetup & Owasp Pune Chapter By Harsh Tandel

Who Am I

- **Certified Blockchain Examiner**
- **Web3 Security Researcher**
- **Smart Contract Auditor**
- **Integrity Global Top 1000**
- **P1 Warrior Bugcrowd**
- **CVE 2022-334**

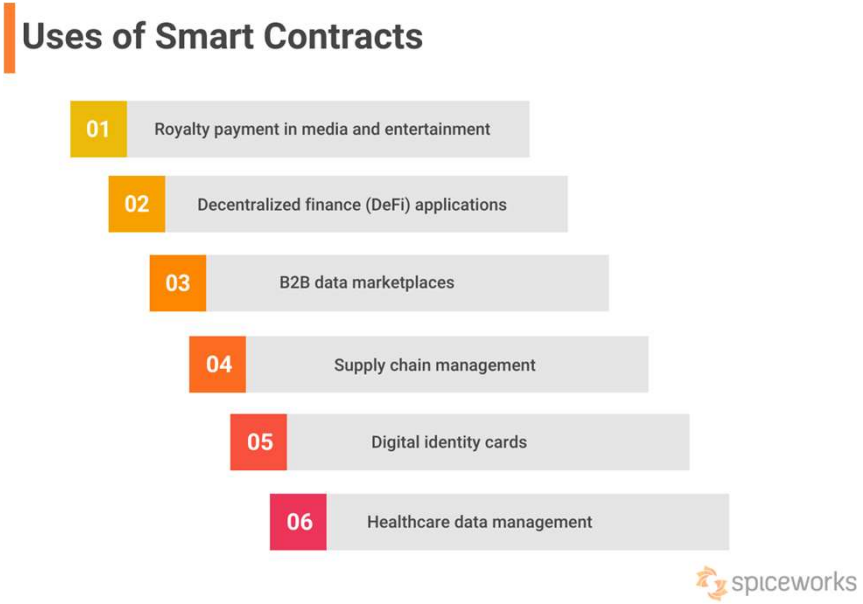
Harsh Tandel

H4r5h_T4nd37

What is Smart Contracts ?

- Smart Contracts are the well written contracts define to execute certain actions when certain predefined conditions Mets. They are mostly written in solidity and rust. They are immutable and public so once deployed no once can make any changes once it deployed and they are transparent.

- Smart Contracts are used in below mentioned way
  - Di-Fi,
  - DAO,
  - Legal,
  - NFT &Gaming



- More than $12.3 billion in financial losses have been caused by smart contract flaws so far in the Web3 industry.148+ projects on BNB networks, 75+ projects on the Ethereum chain, 18 hacks on the Polygon chain, and 80+ hacks on Abritrum, Solana, ZKSync, Algorand, etc. were all compromised.

OWASP Smart Contract Top 10

1.Reentrancy Attacks : Reentrancy vulnerabilities occur when a contract allows external calls to be made to it during its execution without properly managing the state changes and flow of execution. Reentrancy allows an attacker to repeatedly call a vulnerable contract before the previous call completes, leading to unexpected state changes and unauthorized fund transfers.

Eg. Balance is not updated before handing flow to other contract. Hence attacker can withdraw all funds from smart contracts.

To prevent reentrancy vulnerabilities, developers should follow secure state management patterns such as the “Checks-Effects-Interactions” (CEI) pattern. This pattern ensures that all state changes are made before any external calls are executed, preventing reentrancy attacks. Additionally, implementing mutex locks or using the “ReentrancyGuard” pattern can further safeguard against reentrancy by blocking reentrant calls.

2.Integer Overflow and Underflow : In Solidity, there are 2 types of integers:

unsigned integers (uint): These are the positive numbers ranging from 0 to (2256 – 1).

signed integers (int): This includes both positive and negative numbers ranging from -2255 to (2255 – 1).

Overflow : Let us assume an example of a smart contract that stores balance by using uint8 values. Upon executing a function with input that increases the balance beyond the maximum value, i.e., 255, the generated number would wrap around. The balance would change to the next lowest possible value, i.e., 0 in Solidity smart contracts prior to the 8.0 version.

Eg. Suppose there is contract which restrict Withdrawal to certain days but overflow can make that restriction day to 0.

uint8 a = 255;

a++;

a =o;

Underflow : When you call the decrease function after the balance rounds up to zero,

it will cause the contract function to generate the maximum value of 255 as the outcome.

Eg. Suppose there is contract which restrict Withdrawal to certain days but ,

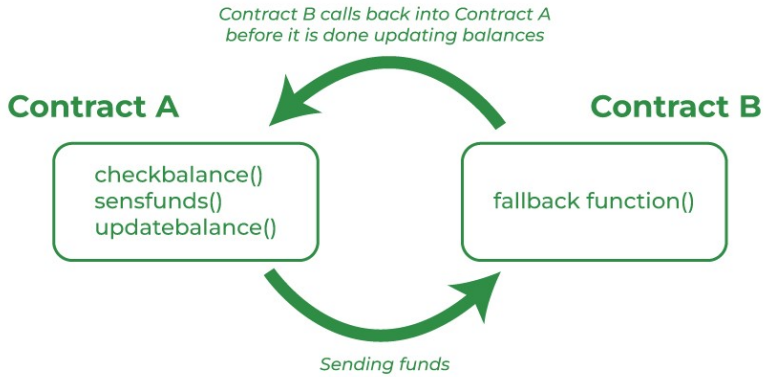
underflow can make that restriction day to maximum and fund unavailable to user.

uint8 a = 0;

a–;

a= 256

To prevent this vulnerability use safe math library from OpenZeppelin or upgrade to solidity version > 0.8.



3.Timestamp Dependence : Timestamp dependence vulnerability is when a smart contract utilizes the block timestamp function when performing critical logic in a smart contract. This includes actions such as sending ETH, or using the function as the source of entropy in order to generate necessary random numbers. Contracts that depend on block timestamps for critical operations are susceptible to manipulation, as miners can slightly adjust the timestamps.

Miners can manipulate timestamp with some constraint.1st it must be after previous timestamp and it can too far in the future.

E.g Contract will reward if transaction is sent on time which is meeting certain condition then miner will manipulate and falsely get rewards.

To prevent Instead of relying solely on timestamps, consider utilizing block numbers for time-sensitive operations. Block numbers offer a deterministic and less manipulable alternative, providing a more secure foundation for your smart contracts.

4.Access Control Vulnerabilities : An access control vulnerability in a Solidity smart contract is a type of security flaw that lets unauthorized users access or modify the contract's data or functions.

Eg. Withdraw function without onlyOwner () or defined address restriction attacker can withdraw funds from smart contract .

To prevent it implement least-privilege roles using libraries like OpenZeppelin's Access Control. Add proper access control modifiers to sensitive functions, such as onlyOwner or custom roles.

5.Front-running Attacks : Front running occurs when an attacker gains an advantage by executing transactions ahead of others, particularly when they have knowledge of pending transactions that are about to be included in a block. In the context of smart contracts, front running can be detrimental in scenarios where transaction order impacts the outcome.

Some preventive measures include the use of secret or commit-reveal schemes, implementing schemes where sensitive information such as prices or bids is kept secret until the transaction is confirmed, off-chain order matching, use of flatboats which allow users to bundle transactions together and submit them directly to miners reducing the opportunity for front running.

6.Denial of Service (DoS) Attacks : Denial of Service (DoS) is an attack where an adversary manages to halt the normal operations of a smart contract, making it unavailable to users.

To prevent this vulnerability when using the 'send' and 'transfer' functions as they can potentially cause a DoS attack. Use 'call' instead, and handle the potential 'false' return value. Limit the number of iterations or actions that can be taken in a single transaction to avoid reaching the gas limit.

7.Logic Errors : Logic errors occur when the contract's code doesn't correctly implement the intended logic. Logic errors, often referred to as business logic vulnerabilities, are like hidden landmines within smart contracts. They occur when the contract’s code doesn’t align with its intended behavior.

To prevent use automated testing frameworks to write extensive unit tests covering all possible edge cases.

8.Insecure Randomness : Insecure randomness occurs when a smart contract uses a source of randomness that is predictable or can be manipulated. This can happen when a smart contract relies on a deterministic function, such as the block hash or timestamp, to generate random numbers. Since these values are publicly visible on the blockchain, an attacker can predict them and use this knowledge to exploit the smart contract.

To prevent use external oracles and Use commit-reveal schemes, where users submit hashed values and reveal them later, to generate randomness.

9.Gas Limit Vulnerabilities : if a transaction consumes too much gas, that transaction will never fit in a block and thus the transaction will never be executed.

To prevent Avoid loops that iterate over dynamic data structures. If possible, use mappings and keep track of keys separately. Break down complex computations into multiple transactions if needed.

Implement gas-efficient code and test functions with large inputs to ensure they won’t exceed the block gas limit.

10.Unchecked External Calls : Unchecked external call vulnerability occurs when a smart contract interacts with external contracts or addresses without adequately checking the consequences of these interactions.

To prevent Always check the return value of call, delegatecall and callcode . Use Solidity’s transfer or send functions instead of call.value()(), as they automatically reverts on failure

Bonus Vulnerabilities

- **Unprotected SelfDestruct**: SelfDestruct is a function in Solidity that can be used to “kill” a contract, making it unusable and sending all remaining Ether stored in it to a specified address. If a contract doesn’t properly protect this function, an attacker might be able to call it and essentially delete the contract, causing loss of data or even funds.

- **Oracle manipulation attacks** :Smart contracts often rely on external data sources, called oracles, to make informed decisions. If these oracles are compromised or manipulated, it may lead to incorrect pricing for swaps, improper reward calculation, the ability to borrow more assets than a collateralization ratio allows, or other general issues with financial transactions.

To prevent this use Di-Fi Oracle Network like chainlink datafeeds.

- **Default Visibility** : Default Visibility becomes a problem when developers ignore visibility specifiers on functions that should be private (or only callable within the contract itself). It is good practice to specify the visibility of all functions in a contract, even if they are designed to be public.

Thank You All