LTIMindtree

Whitepaper

What are Smart Contracts and How to Secure Them?

Authored by:

Santhosh Anumolu

Table of Contents

Abstract

This paper briefly introduces smart contracts and covers various security aspects across the smart contract lifecycle. Identification of security issues with smart contracts is difficult since the ecosystem is still in its infancy, lacks standards, has multiple underlying platforms, and they bring along related security issues. In addition, the immutable nature of smart contracts adds more complexity, making the security-first approach the top priority for smart contract development.

Introduction

What are Smart Contracts?

Smart contracts are digital contracts or programs stored on the blockchain that automatically execute transactions once predetermined conditions are met. They do not require any intermediaries. The code and conditions in the contract are publicly available in the ledger and are immutable.

Smart contracts are like user accounts of Ethereum, which can hold ETH, the native cryptocurrency of Ethereum, and have an address and can transfer/ receive ETH but are managed by code and rules instead of a user. User accounts can then communicate with a smart contract by sending commands that carry out a function defined on the smart contract. Most common smart contracts are a series of instructions written using the programming languages like Solidity, Vyper, and Rust.

Few nuggets to get you familiarized with smart contracts:

- Smart contracts use either Proof of Work (PoW) or Proof of Stake (PoS) consensus mechanisms. PoS is gaining popularity as it is less resource-intensive. Ethereum moved from PoW- to PoS-based consensus on 15th Sept 2022.

- Ethereum Virtual Machine (EVM) is a runtime environment for Ethereum-based smart contracts. Think of it as a global supercomputer that carries out all the smart contracts.

- In the EVM, gas is a measurement unit used to determine each transaction's cost with a smart contract. A certain quantity of gas is required for each computation in the EVM. More complex computations require

more gas to run. Transaction Fee (TF) equals the Total Gas (TG) used multiplied by the Gas Price (GP) or TF=TG*GP.

- Decentralized Applications (DApps) are at the core of Web 3.0, which can operate autonomously, typically using smart contracts that run on the blockchain, decentralized computing, or other distributed ledger system. Like traditional applications, DApps provide some function or utility to their users. A DApp combines a front-end interface and smart contracts.

Types of Smart Contracts

Smart contracts can be fully on-chain standard compliant contracts, with or without ETH/any compliant tokens, or can have off-chain interactions like Oracles, Channels, etc. Some common types are:

1. **Smart Legal Contracts**
   Legally enforceable and require the parties to fulfill their contractual obligations.

2. **Decentralized Autonomous Organizations (DAOs)**
   These are blockchain communities bound to specific rules coded into blockchain contracts combined with governance mechanisms.

3. **Application Logics Contracts (ALCs)**
   Contains an application-based code that remains in sync with other blockchain contracts. It enables communication between various devices, such as when combined by blockchain technology and the Internet of Things (IoT).

How Do Smart Contracts Work?

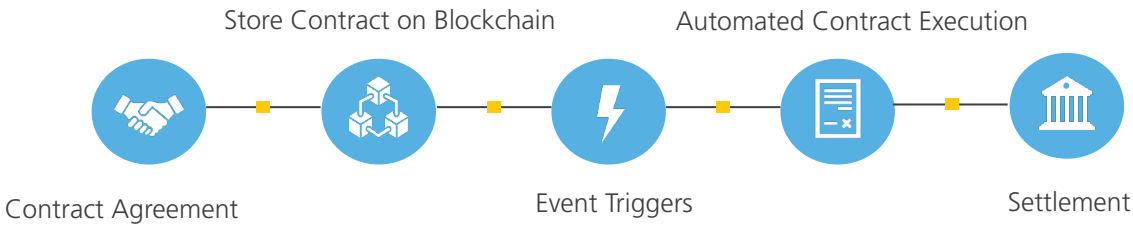

Fig 1: Smart Contract Process Flow

- Terms and conditions are agreed to by all parties involved
- The contract goes live and is stored on the ledger (blockchain network)
- Execution of the contract is triggered by an event such as a pre-defined date or a transaction initiation
- It automatically executes when conditions are met
- It may also involve the settlement of digital assets such as cryptocurrency or off-the-chain assets like shares

Use Cases

- Cryptocurrencies
- Non-Fungible Tokens (NFTs)
- Digital Identity Management
- Decentralized Finance (DeFi) Apps
- Supply Chain Traceability
- Real Estate – property ownership records, loans, and mortgages
- Electoral Records

For more use cases and details, refer to the guide from **block geeks.**

Smart Contracts Security

Smart contract security depends on the following:

- Blockchain itself (refer to layer-1 **blockchain security checklist**)
- Issues related to smart contracts
- Issues related to the application system (server and client side of DApp), including UI
- Blockchain networks like Ethereum are relatively more mature and offer better security, but the applications running on them might not be as secure.

Smart contract SDLC security best practices

Depending solely on reviews and audits for security is not smart. Shift the security left and follow best practices to build secure smart contracts. First and foremost, train the developers. Train the teams on secure design patterns, coding best practices, security best practices, common vulnerabilities and remediations, and development and test tools.

Following are some of the Software Development Life Cycle (SDLC) key best practices to be followed:

1 Scope and Design Patterns

1. Define and document the scope and objectives clearly
2. Learn and implement secure design patterns specific to Smart Contracts (SC). There are multiple design patterns like:

- Check affect interaction (when performing functions in SC, first check all pre-conditions, then apply effects to the contract's state, and finally interact with other contracts. Never alter this sequence).
- Authorization
- Ownership
- Rate Limit
- Balance Limit
- Speed Bump Patterns

2 Design

1. Follow coding best practices, code versioning, and exception handling
2. Use the latest safe component versions and update regularly
3. Write good documentation by creating Runbooks
4. Avoid common vulnerabilities and weaknesses (Speedy Web Compiler (SWC) registry)
5. Choose the right development and test tools
6. Provide fallback functions and ensure their safe execution
7. Use hash secrets to grant permissions
8. Encrypt on-chain data by implementing wallet security
9. Hash objects for tracking off-chain data

3 Coding and Development

1. Clearly understand what your SC needs to do
2. Select a relevant and clear name
3. Include fail-safe mechanisms
4. Use trustworthy dependencies/ libraries
5. Let the owner perform the withdrawal of funds using pull instead of push

4 Testing and Deployment

1. Fix compiler warnings
2. Lock programs to specific compiler version
3. Create and run unit tests by using both positive and negative tests
4. Use testnet environments like Rikeby or Ropsten for development, but test on fork of mainnet to ensure the functionality works as expected
5. Run linters and code analysis tools for each commit
   Use linters like solhint and ethlint, and code analysis tools like Mythril and Slither
6. Audit/review code
   - Conduct internal code reviews for functional coverage, coding best practices, business logic, misuse cases, and security issues
   - Have an external audit to ensure a comprehensive review
7. Regularly monitor your contracts after deployment

Fig 2: Software Development Life Cycle (SDLC) key best practices

Common Smart Contract Vulnerabilities/ Security Attacks

There are quite a few known vulnerabilities that are observed in smart contract audits. Some of the frequently identified vulnerabilities are:

Over/underflow If a balance reaches the maximum unit value, i.e., overflow, it will circle back to zero. The same is true for underflow. If a unit is made to be less than zero, it will cause an underflow and get set to its maximum value.

Re-entrancy DAO hack was caused by a re-entrancy vulnerability. This occurred when involved functions that could be called repeatedly were finished before the function's first invocation. An attacker may also be able to do a similar attack using two different functions that share the same state, called cross-function re-entrancy.

Denial of service Occurs due to unexpected reverts and the block gas limit.

Timestamp dependency The smart contract's execution environment is on the miner's side. In the case of a contract's logic being dependent on the current time, the miner with significant computational power can influence the execution result and meet a specific result by manipulating the current time. e.g., betting.

Front-running opportunities By examining poorly written code, one can get a head start on the market. Others, in turn, are then able to profit from the knowledge.

Short address attack The attack consists of finding an address with a hex representation ending in various 0s and leaving out these trailing 0s in a withdrawal request.

Unchecked external calls Calls to the external contract are expected to fail. Check for the return value and deal with the errors when sending ETH. The recommended way of doing ETH transfers is to call instead of send or transfer.

Function visibility errors In Solidity, a function's default visibility property is public. Anyone can access if a developer forgets to specify a private function's visibility. For example, anyone can immediately call the Destruct function to destroy the contract.

Gas limit and loops (costly loop) Each block has the maximum amount of gas that can be spent on a transaction. If the consumed gas exceeds the allowed maximum, it leads to transaction failure. The out-of-gas problems account for about 90% of all exceptions on Ethereum.

Transaction ordering The miners select which transactions to include in a block based on who has paid a high enough GP. A race condition vulnerability occurs when code depends on the order of the transactions submitted.

Poor randomness Avoid using block variables such as hashes, timestamps, block numbers, or gas limits as a source of entropy.

Refer to guides from **Solidity**, **Immune Bytes**, and **Consensys** for more details on vulnerabilities and remediations..

Smart Contract Audit

Deploying smart contracts without proper audits could result in data loss, inefficiencies, and security breaches.

- The famous DAO Hack of Ethereum resulted in a hard fork where hacker(s) stole almost 14% of the total ETH available as of that date.
- One of the biggest cryptocurrency heists happened in August 2021. Hackers stole digital currency valued at USD 613 million from Poly Network. They leveraged a vulnerability in the digital contracts.

The immutable nature of smart contracts makes the audit more important, as the contract cannot be edited once it is stored on the blockchain. To ensure overall security and optimize performance, you must thoroughly audit smart contracts. The audit helps in:

- Better optimization of the code
- Improved performance of smart contracts
- Security against hacking attacks
- Enhanced security of wallets

Smart Contract Audit Flow

Smart contract audit methodology may vary slightly depending on the third party doing the audit. But typically, it will have the following phases:

- Understanding the smart contract specifications, including architecture, build process, and design choices of the project, helps in understanding the goals of the project and determining the scope
- Manual analysis, including functional test coverage, detailed code review, and automated scan for vulnerabilities
- Preparing audit reports with recommendations and sharing them with developers
- Re-auditing after reported issues are addressed
- Providing final report/certification

What Should a Smart Contract Audit Cover?

The audit should ensure the smart contract's desired working, conditions, outcome, security, efficiency, and proper error handling. It should aim for complete code coverage by manually running positive and negative tests or using automated scanners.

Below are some of the critical aspects to be covered as part of the audit:

- Review architecture, design, and specifications
- Check if access controls are implemented properly
- Look for known vulnerabilities/security attacks of smart contracts
- Check the compiler version used and any compilation errors in the smart contract
- Follow strong encryption standards
- Look for any logic errors [locked ETH, unchecked math, re-entrancy, and more]w
- Check the processes involving off-chain processing - Oracles
- Ensure input sanitization is done correctly to eliminate malicious inputs
- Check all variables, constants, and their visibility
- Look for any costly loops
- Look for known vulnerabilities in the host platform (blockchain platform)
- Check the gas usage during contract processing and any conditions that could stop the processing due to gas limit
- Ensure error handling is done correctly and check the possible misuse cases
- Validate all external calls
- Ensure the security of on-chain data

Explore the sample audit reports from **Open Zeppelin**, **Immune Bytes**, **Hacken**, and **Halborn** to understand the contents of typical audit reports, type of tests conducted, audit scope definition, vulnerability presentation, and assigning issue criticality.

Automated Tools

Though most of these tools are relatively new, they would identify many common issues and save a lot of time during the reviews/audits. And when combined with manual analysis, these tools provide better coverage. A few popular choices are

1. **Slither:** Static analysis framework with detectors for many common Solidity issues. It has taint and value-tracking capabilities and is written in Python.
2. **Mythril:** Swiss army knife for smart contract security that analyses EVM byte code.
3. **MythX:** High-quality cloud service that employs symbolic analysis and input fuzzing to find common security flaws and ensure that smart contract code is valid.
4. **Echidna:** Property-based (fuzz) testing of Ethereum smart contracts.
5. **Harvey:** Property-based (fuzz) testing.
6. **SmartCheck:** Static analysis of Solidity source code for security vulnerabilities and best practices.
7. **Manticore:** Dynamic binary analysis tool with EVM support ensures symbolic analysis by checking for program correctness.
8. **Oyente:** Static analysis tool for smart contract security.
9. **Solgraph:** Generates a dot graph that visualizes the function control flow of a Solidity contract and highlights potential security vulnerabilities.
10. **Securify2:** Security scanner supported by Ethereum foundation.

Conclusion

Security should be the critical building block and should be at the core of the requirements and design of smart contracts. Earlier cited examples of DAO and poly network hacks clearly articulate the impact of security vulnerabilities on the success of the projects.

In my view, security should not be limited to audits and should also include:

- User training on the end-to-end ecosystem, including the underlying platform, security best practices at each phase, and common vulnerabilities along with remediations
- Definition and implementation of SDLC best practices from design to deployment
- Usage of automated scanners early in the lifecycle with regular rescans
- Comprehensive internal and external audits

I hope this paper helps you gain a little more understanding of smart contract security.

References

1. https://blockgeeks.com/guides/smart-contracts/
2. https://www.immunebytes.com/blog/top-blockchain-security-issues-how-to-prevent-them
3. https://swcregistry.io/
4. https://github.com/sigp/solidity-security-blog
5. https://www.immunebytes.com/blog/smart-contract-vulnerabilities/
6. https://consensys.github.io/smart-contract-best-practices/attacks/
7. https://blog.openzeppelin.com/security-audits/
8. https://github.com/ImmuneBytes/Smart-Contract-Audit-Reports/
9. https://hacken.io/audits/
10. https://github.com/HalbornSecurity/PublicReports
11. https://www.wallstreetmojo.com/smart-contracts/

Smart contracts, audits, and tools

1. https://ethereum.org/en/developers/docs/smart-contracts/
2. https://www.devteam.space/blog/how-to-audit-a-smart-contract-a-guide/
3. https://101blockchains.com/smart-contract-audit/
4. https://4irelabs.com/articles/how-to-audit-smart-contracts/
5. https://www.getastra.com/blog/security-audit/smart-contract-security/
6. https://s-tikhomirov.github.io/assets/papers/smartcheck.pdf
7. https://consensys.github.io/smart-contract-best-practices/security-tools/

Best practices

1. https://arxiv.org/pdf/2008.04761.pdf
2. https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/guidelines.md
3. https://consensys.github.io/smart-contract-best-practices/development-recommendations/
4. https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/
5. https://yos.io/2019/11/10/smart-contract-development-best-practices/
6. https://blog.chain.link/defi-security-best-practices/
7. https://www.useweb3.xyz/guides/clean-contracts

More resources for learning

1. https://immunefi.com/learn/
2. https://www.youtube.com/channel/UCJWh7F3AFyQ_x01VKzr9eyA/videos
3. https://media.consensys.net/solidity-best-practices-for-smart-contract-security-54d309a622c2
4. https://dappsys.readthedocs.io/en/latest/
5. https://learn.openzeppelin.c=om/security-audits/readiness-guide
6. ZIION – Open-source VM image for development and testing smart contracts

Santhosh Anumolu

Associate Director

Santhosh Anumolu is working as a Security Architect at LTIMindtree. He is a technology enthusiast with over 18 years of experience in information security. He initiated and established processes for end-to-end application security in various projects. He worked on many application security tools covering Software Composition Analysis, Static Application Security Testing (SAST), and Dynamic Application Security Testing (DAST). He aspires to be a specialist in improving application security by training teams on security and establishing slick and repeatable security compliance checks in SDLC. He enjoys listening to music and traveling.