Security in Smart Contracts

Master of Cybersecurity

Marc Alba Cerveró, Léo Gabaix, Jean-Paul Morcos Doueihy, Nina Raganová

1. Introduction to Smart Contracts

Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They exist across a decentralized blockchain network, which means they operate without the need for a central authority, legal system, or external enforcement mechanism. The code controls the execution, and transactions are trackable and irreversible.

Smart contracts allow for the automation of complex processes, ensuring a high level of security and reducing reliance on trust between parties. They're integral to blockchain because they extend its use beyond simple transactions, enabling a wide range of decentralized applications (DApps) and business logic to be implemented on the blockchain.

1.1. Historical Context

The concept of smart contracts predates the blockchain. It was first proposed by cryptographer Nick Szabo in the 1990s as a way to use computerized transaction protocols. However, smart contracts as we know them today became feasible with the advent of blockchain technology, especially after the introduction of Ethereum in 2015. Ethereum introduced a Turing-complete programming language, allowing developers to create more complex contracts and decentralized applications beyond simple transaction scripts.

Since then, smart contracts have evolved rapidly. From basic coin-flipping games and escrow services in the early days of Ethereum to complex decentralized finance (DeFi) applications, Non-Fungible Tokens (NFTs), decentralized autonomous organizations (DAOs), and more, the evolution of smart contracts mirrors the growth and diversification of the blockchain ecosystem itself.

1.2. Use Cases

- **Decentralized Finance (DeFi)**

Smart contracts are the backbone of DeFi applications, allowing for the creation of decentralized exchanges, lending platforms, and yield farming applications. They enable users to lend, borrow, or trade assets without the need for traditional financial intermediaries.

- **NFTs and Digital Ownership**

The rise of NFTs (Non-Fungible Tokens) has been facilitated by smart contracts, allowing for the verification of unique digital ownership and the transfer of digital assets in a secure and transparent manner.

- **Voting Systems**

Blockchain-based voting systems can use smart contracts to ensure transparency and tamper-proof recording of votes, potentially revolutionizing how democratic processes are conducted.

2. Technical Foundations of Smart Contracts

2.1. Blockchain Basics

At its core, a blockchain is a distributed ledger that records transactions across many computers in such a way that the registered transactions cannot be altered retroactively. This technology provides a high level of security and transparency, making it ideal for smart contracts. The transparent nature of blockchain allows all parties to view the terms of a contract and its execution. This builds trust, as every action taken by a smart contract is visible and verifiable by all network participants. Key features of blockchain that are pertinent to smart contracts include:

- **Decentralization**

Unlike traditional centralized systems, a blockchain operates over a network of distributed nodes. This ensures that no single entity has control over the entire network, enhancing security and resilience against attacks or failures.

- **Immutability**

Once a transaction is recorded on the blockchain, it cannot be altered or deleted. This immutability is crucial for smart contracts, as it ensures that once a contract is executed, its outcome cannot be tampered with.

- **Consensus Mechanism**

Blockchains use consensus mechanisms like Proof of Work (PoW) or Proof of Stake (PoS) to agree on the state of the ledger. This consensus is vital for maintaining the integrity of smart contracts, ensuring that all parties agree on the contract terms and outcomes.

2.2. Programming Languages

- **Solidity**

The most popular language for writing smart contracts, especially on the Ethereum blockchain. It is a contract-oriented, high-level language influenced by C++, Python, and JavaScript. Solidity is designed to target the Ethereum Virtual Machine (EVM) directly, making it a powerful tool for creating complex smart contracts.

- **Vyper**

Another language used for Ethereum smart contracts, Vyper is designed to be more secure and simpler than Solidity. It aims to provide a Python-like syntax with fewer features to reduce potential security risks. Vyper focuses on readability and simplicity, making it easier to conduct smart contract audits.

- **Other Languages**

There are other languages too, like Chaincode (for Hyperledger Fabric) and Rust or Ink! for the Polkadot network, which are used for specific blockchain ecosystems.

2.3. Execution Environment

The EVM is the runtime environment for smart contracts in Ethereum. It's a quasi-Turing complete machine; the "quasi" comes from the fact that computation is intrinsically limited by gas. Every operation in the EVM consumes gas, which is paid in Ether, Ethereum's native cryptocurrency. This mechanism prevents infinite loops and encourages efficiency in smart contract code. When a smart contract is deployed on Ethereum, it's compiled into bytecode and executed by the EVM. The contract remains on the blockchain, where it can be interacted with through transactions. These transactions can trigger contract functions, and the resulting state changes are recorded on the blockchain. The execution of smart contracts requires computational resources. In Ethereum, these resources are quantified in units of "gas." Each operation in a smart contract costs a certain amount of gas, and users must pay this gas (in Ether) to execute the contract. This mechanism prevents network abuse and ensures that resources are efficiently used.

3. Security Concerns in Smart Contracts

To ensure and enhance the robustness and resilience of blockchain Smart Contracts, developers must address carefully any potential security threat.

Thereupon we address the following most common security topics impacting developing Smart Contracts:

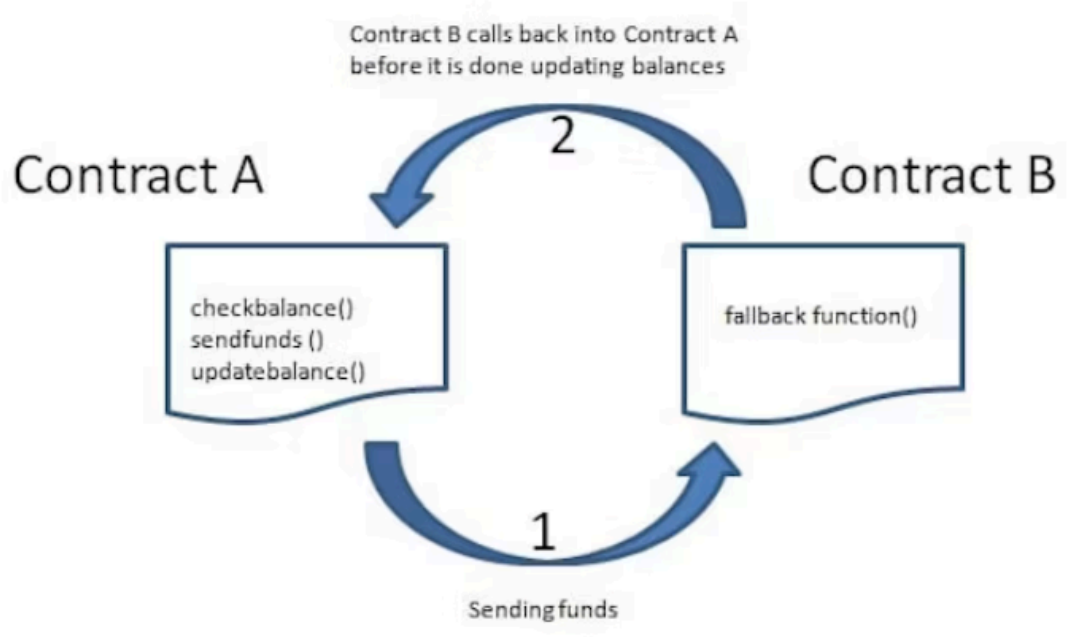
- Vulnerabilities by coding
- Vulnerabilities by design

Additionally, we will also look into some cases of high-profile breaches, due to or related to the exploitation of vulnerabilities.

3.1. Vulnerabilities

- **Re-entrancy Attacks**

This occurs when a smart contract function makes an external call to another untrusted contract before it resolves its own state. The external contract can then make recursive calls back to the original function, potentially draining funds. The most famous example is the DAO attack.



Reentrancy attack overview

Possible solutions:

- Guard Solution

The reentrancy guard stops the execution of multiple vital functions at the same time.

```
bool internal locked;

modifier noReentrant() {
    require(!locked, "cannot reenter");
    locked = true;
    _;
    locked = false;
}

function withdrawalAll() external noReentrant {
    uint balance = getUserBalance(msg.sender);
    require(balance > 1 ether, "your balance ould be 1 ether or more");
    (bool success, ) = msg.sender.call{value: balance} ("");
    require(success, "transaction failed");
    theBalances[msg.sender] = 0;
}
```

Guard Solution Example

By locking the function that has been called we can avoid reentrancy. In this code example, the function for withdrawal has an external modifier requirement.

- Check and Effects Convention

Immediately update your state after you have set the requirement check.

```
function withdrawalAll() external noReentrant {
    uint balance = getUserBalance(msg.sender);
    require(balance > 1 ether, "your balance ould be 1 ether or more");
    theBalances[msg.sender] = 0;
    (bool success, ) = msg.sender.call{value: balance} ("");
    require(success, "transaction failed");
}
```

Check and Effect Example

In this example the balance is updated immediately after the requirement has been checked.

- **Integer Overflow/Underflow**

Smart contracts often involve arithmetic operations. If these operations exceed the maximum or minimum limits of the data type (e.g., adding 1 to the maximum value of a uint256 in Solidity), it can result in overflow or underflow, leading to unintended behavior.

Solution

Developers should make sure to use safe math libraries or appropriate types that provide overflow/underflow detection. In case of an error, the library fails the transaction and updates the status of the transaction to be reverted. Updated versions of the compiler like solidity >= 0.8 already incorporate this solution, libraries like SafeMath are embedded in the compiled code.

Additionally, using modifiers restrictions like ‘onlyOwner’, can also add checks in their code to ensure the values never exceed their expected range.

- **Gas Limit and Loops**

Contracts with loops that don't have a fixed number of iterations can run out of gas if the loop runs too many times, causing the transaction to fail. This can be exploited in some cases to make contracts unusable.

- **Timestamp Dependence**

Some contracts use block timestamps as a source of randomness or for timing conditions. However, block timestamps can be manipulated by miners to some extent, which can lead to vulnerabilities.

- **Front-Running**

Since all transactions are visible in the mempool before being mined, an attacker can see a transaction and quickly send another transaction with a higher gas fee, effectively "cutting in line" and influencing the outcome.

- **Access control**

Access control vulnerabilities exist when a contract fails to properly restrict who can call certain functions. This can result in unauthorized function calls.

If a contract function isn’t protected adequately, unauthorized actors can manipulate the contract state, steal funds, or take other damaging actions.

3.2. High-Profile Breaches

- **The DAO Attack (2016)**

The DAO (Decentralized Autonomous Organization) was a complex smart contract on the Ethereum blockchain that aimed to operate as a venture capital fund for the crypto and decentralization space. However, it suffered a major attack where an attacker exploited a reentrancy vulnerability, draining about 3.6 million Ether, which was about a third of The DAO's funds at the time. This attack had a massive impact, leading to a hard fork in the Ethereum network, resulting in Ethereum (ETH) and Ethereum Classic (ETC).

- **Parity Wallet Bugs**

Parity Technologies suffered two major vulnerabilities. The first, in 2017, involved a bug in a multi-signature wallet that led to about 150,000 ETH being frozen. The second, later in the same year, was a critical vulnerability in Parity's multi-sig wallet contract, resulting in about 513,000 ETH being permanently frozen. These incidents raised questions about code auditing and security practices in smart contract development.

These breaches and vulnerabilities highlight the critical need for rigorous testing, code review, and security audits in smart contract development. They also underscore the importance of considering security at every stage of smart contract design and

implementation. Due to the immutable nature of blockchain, once a smart contract is deployed with a vulnerability, it can be difficult or even impossible to rectify without significant consensus and, in some cases, drastic measures like hard forks.

4. Smart Contract Security Measures

4.1. Best Practices

Keeping the code simple and clear makes it easier to review and audit. Complex logic increases the likelihood of errors. If the logic must be complex, it's crucial to document it thoroughly. Breaking down the contract into smaller, reusable, and independent modules helps in isolating functionality and making the codebase more manageable. This approach also aids in testing and auditing individual components. Be aware of and actively avoid known vulnerabilities like reentrancy, integer overflows/underflows, and unchecked external calls. You should also implement well-known design patterns and best practices, like using the Checks-Effects-Interactions pattern to mitigate reentrancy attacks. Implement also proper access control mechanisms to restrict who can call sensitive functions in the contract. If your contract relies on external libraries or other contracts, ensure they are regularly updated and secure.

4.2. Testing and Audits

Before deploying a smart contract, it's crucial to conduct thorough testing, including unit tests, integration tests, and functional tests. These tests should cover all possible paths and edge cases in the contract. Automated testing frameworks can help in continuously testing the contract for vulnerabilities, especially as new changes are introduced. Also, having an external party audit the smart contract is critical. These auditors are experts in smart contract security and can identify vulnerabilities that the original developers might have missed. Another effective way to discover and fix security issues may be to offer rewards for identifying vulnerabilities (bug bounty).

4.3. Formal Verification

Formal verification is the process of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics. The process can be broken down into the following steps:

1. Translation of code into the formal representation
2. Computation of the state space
3. Specification of the desired behavior
4. Detection of bugs

By creating mathematical statements that can be formally verified, we rely on the logic rules and express the code in a way that we can make sure that it will work properly. To compute the state space, we try to execute every possible action and interaction with the contract to observe all the possible outcomes and by comparing it to our specification, we can find the deviations from our expected model. Ensuring that the contract will behave as expected is crucial because, unlike traditional software, updating a smart contract post-deployment can be very challenging or impossible. However, while powerful, formal verification is complex and can be time-consuming. Also, it can only prove properties that are formally specified; hence, the quality of verification depends heavily on the completeness and accuracy of the specifications.

Tools like K Framework, CertiK, and others provide formal verification services for smart contracts. These tools help in translating contract code into a form that can be mathematically analyzed.

Implementing these security measures is essential in mitigating the risks associated with smart contracts. Given the potential financial and reputational implications of a security breach, investing in robust security practices is not just advisable but essential for any serious smart contract development project.

5. Tools and Technologies for Enhancing Security

5.1. Static Analysis

Static analysis can be either performed manually (e.g. code reviews) or using automatic tools. It is essential in the early stages of smart contract development, as it helps to identify vulnerabilities and code smells without executing the code. Static analysis focuses mostly on the code syntax and adherence to coding standards, but also tries to find potential vulnerabilities that may be present in the code.

- **Slither**

Developed by Trail of Bits, Slither analyzes Solidity code for vulnerabilities, runs a suite of vulnerability detectors, and provides an overview of the contract's structure. It's known for its ease of use and thoroughness in detecting common issues. It operates on the abstract syntax tree (AST), which is the formal representation of the code produced by the compiler.

- **Oyente**

Another popular tool, Oyente, focuses on analyzing Ethereum smart contracts for security issues, bugs, and inefficiencies. It's particularly good at identifying problems like transaction-ordering dependence, timestamp dependence, and reentrancy vulnerabilities. It performs symbolic execution directly with the EVM bytecode, firstly assigning symbolic values to the variables and then expressing their changes in time by creating mathematical expressions using the symbolic values.

5.2. Dynamic Analysis

Dynamic analysis involves checking the smart contract's behavior during execution to ensure it conforms to the expected behavior. Runtime verification tools analyze the contract's behavior on the blockchain and compare it against its expected behavior. This can include checking for compliance with certain conditions or detecting unusual patterns that might indicate a security breach. Monitoring services like Tenderly and Forta offer real-time monitoring and alerting for smart contract interactions. They can provide insights into how contracts are being used and alert developers to potential security incidents as they happen.

- **Mythril**

Mythril is a dynamic security analysis tool for Ethereum smart contracts. It performs symbolic analysis, taint analysis, and control flow checking to detect a range of security vulnerabilities. Mythril can be integrated into the development workflow to catch issues early. It simulates the execution of the smart contract to monitor its behavior on different inputs and actions.

5.3. Upgradable Contracts

Upgradable contracts are a way to address the immutability of smart contracts, which, while a strength, can also be a limitation, especially in the face of bugs or evolving requirements. An upgradable smart contract separates the contract's state and logic. The state remains in one contract (the storage contract), while the logic can be in another contract (the logic contract). If a change is needed, the logic contract can be swapped without losing the state. Commonly, a proxy pattern is used, where a proxy contract delegates calls to a logic

contract. The logic contract can be changed by updating the proxy's reference to a new version.

While upgradability adds flexibility, it also introduces additional complexity and potential security risks. The upgrade process itself must be secure, and the authority to upgrade must be controlled and transparent. If not properly managed, it can become a central point of failure. It's crucial to have robust governance mechanisms in place for upgrades, potentially involving multi-signature wallets or DAOs to approve changes. Also, any new version of the contract should undergo rigorous testing and auditing before deployment.

These tools and concepts are vital in the smart contract development ecosystem. They not only help in identifying and mitigating risks but also provide mechanisms to adapt and respond to issues in a landscape where the technology and associated threats are constantly evolving.

6. Regulatory and Legal Landscape

6.1. Regularity Considerations

The regulation of smart contracts varies significantly across different jurisdictions, reflecting the diverse legal and regulatory environments worldwide. Here's how some regions are approaching this issue:

- **United States**

The U.S. doesn't have federal laws specifically governing smart contracts. However, various states have passed laws recognizing the legal validity of blockchain transactions and smart contracts. The SEC (Securities and Exchange Commission) has been active in regulating smart contracts when they are used for ICOs (Initial Coin Offerings), treating them as securities in many cases.

- **European Union**

The EU is exploring the integration of blockchain technology into its digital market strategy. While there are no EU-wide regulations specific to smart contracts yet, the European Blockchain Observatory and Forum is actively working on research and recommendations.

- **Asia**

Singapore is known for its proactive and positive stance towards blockchain technology and smart contracts, providing a supportive legal framework. China, while restrictive towards cryptocurrencies, is actively exploring and implementing blockchain technology in various sectors, including legal recognition of smart contracts in certain contexts.

- **Global Variation**

Other countries have their own approaches, often reflecting their stance on digital assets and blockchain technology. It's a rapidly evolving field, with many jurisdictions working to develop regulations that balance innovation with consumer protection and legal certainty.

6.2. Legal Challenges

Smart contracts face several legal challenges and uncertainties, primarily due to their novel nature and the complexities of applying traditional legal frameworks to them.

- **Enforceability**

One of the biggest challenges is whether and how traditional legal systems will recognize and enforce smart contracts. The enforceability of a smart contract as a legal contract depends on it meeting the criteria of a traditional contract, including offer, acceptance, intention to create legal relations, and consideration.

- **Jurisdiction and Conflict of Laws**

Given the decentralized nature of blockchain, determining jurisdiction can be complex. Which laws apply when parties to a smart contract are in different countries?

- **Liability Issues**

In cases where a smart contract error leads to financial loss, determining liability is challenging. Who is responsible - the contract developer, the user, or the platform?

- **Consumer Protection**

Smart contracts may pose risks to consumers, especially in cases where there is an imbalance of power or knowledge. Ensuring consumer protection while not stifling innovation is a significant challenge.

- **Regulatory Compliance**

Smart contracts must comply with existing laws and regulations, such as those related to anti-money laundering (AML) and Know Your Customer (KYC) requirements. Ensuring compliance, especially in a rapidly evolving regulatory landscape, can be complex.

The legal and regulatory landscape for smart contracts is still in its formative stages. As this technology becomes more widespread, we can expect more specific laws and regulations to emerge. For now, navigating this landscape requires a cautious and well-informed approach, considering both the technological capabilities and limitations of smart contracts, as well as the existing legal and regulatory frameworks.

7. Future Directions and Emerging Trends

7.1. Advancements in Technology

The future of smart contract technology is poised for significant advancements, which will have a considerable impact on their security and functionality:

- **Layer 2 Solutions and Scalability**

As blockchain networks like Ethereum evolve, layer 2 solutions like roll ups and side chains are being developed to enhance scalability and reduce transaction costs. These solutions will likely bring new security considerations, especially in how smart contracts interact with these layers.

- **Improved Programming Languages**

The development of more secure and efficient smart contract programming languages is ongoing. These languages aim to reduce common vulnerabilities and make it easier to write safe code.

- **Enhanced Formal Verification Tools**

Advances in formal verification can make it more feasible and cost-effective to mathematically prove the correctness of smart contracts, leading to higher security guarantees.

- **AI and Smart Contracts**

Integrating artificial intelligence with smart contracts could lead to more dynamic and intelligent contracts. However, this also introduces new security challenges, such as ensuring the reliability and transparency of AI-driven decisions.

7.2. Decentralized Finance (DeFi)

DeFi is one of the most dynamic areas of smart contract application, and it brings its own set of security challenges:

- **Complex Financial Products**

As DeFi platforms develop more complex financial products, the underlying smart contracts become more intricate, increasing the risk of vulnerabilities.

- **Flash Loans and Price Oracles**

The use of flash loans and reliance on external price oracles have exposed DeFi protocols to unique attack vectors, such as price manipulation and oracle attacks.

- **Regulatory Attention**

As DeFi grows, it's likely to attract more regulatory scrutiny, which could impact how smart contracts are developed and used in the DeFi space.

7.3. Cross-Chain Functionality

Interoperability and cross-chain functionality are becoming increasingly important as the blockchain ecosystem grows more fragmented:

- **Interoperability Protocols**

Protocols that enable interoperability between different blockchains are becoming more common. These protocols often rely on smart contracts to manage cross-chain transactions.

- **Security Implications**

Cross-chain functionality introduces new security challenges. For instance, if one chain in an interoperable pair is compromised, it could potentially impact the other. Ensuring the security of bridges and cross-chain protocols is critical.

- **Standardization Efforts**

Efforts to create standards for cross-chain interactions are underway. These standards will likely focus on ensuring security and compatibility between different blockchain systems.

The future of smart contract technology is closely tied to the evolution of the broader blockchain ecosystem. As this technology matures, we can expect more robust security features, more complex and varied applications, and an increasingly nuanced regulatory landscape. These advancements will open up new possibilities while presenting fresh challenges that will need to be addressed through innovation and collaboration within the blockchain community.

Conclusion

In conclusion, while smart contract technology has made significant strides in terms of innovation and application, security remains a critical and ongoing concern. The complexity of the technology, combined with the high stakes involved in many smart contract applications, necessitates a rigorous approach to security. This includes adopting best practices in development, conducting thorough testing and audits, and staying abreast of the latest advancements and regulatory developments.

The future of smart contract security will likely be characterized by continuous adaptation and improvement, as developers, auditors, and regulators collaborate to address emerging challenges and harness the full potential of this transformative technology.