

Blockchain and Crypto Security

Introduction

v. 1.1

Marin Ivezic (ciso.eth)

1 | Blockchain and Crypto Security Training | © Marin Ivezic 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Blockchain and Crypto Security by Marin Ivezic

Copyright © Marin Ivezic 2022

*This work is licensed under a
Creative Commons Attribution – NonCommercial – International License*

*The text of the license is available at
<https://creativecommons.org/licenses/by-nc/4.0/>*

Visit the author's website at <https://crypto.security>



Table of Contents

1. Introduction to Blockchain	00
2. Blockchain Cryptography	018
2.1. Hash Functions	020
2.2. Public Key Cryptography	034
2.3. Advanced Cryptographic Applications	051
3. Blockchain Consensus Security	059
3.1 Introduction to Consensus	060
3.2 Securing Proof of Work	067
(The 51% Attack, Denial of Service Attacks, Selfish Mining, SPV Mining)	
3.3 Securing Proof of Stake	086
(XX% Attack, The Proof of Stake “Timebomb”, Long-Range Attacks, The Nothing at Stake Problem, Resource Exhaustion Attacks)	



Table of Contents

4. Blockchain User, Node, and Network Security	105
4.1. User Security	111
(Non-Random Private Keys, Exposed Mnemonic Seeds, Nonexistent/Insecure Backups, Third-Party Key Management, Phishing Attacks, Compromised Hardware Wallets Unverified Transactions, DeFi Spend Approvals)	
4.2. Node Security	123
(Blockchain Breakouts, Denial of Service Attacks, Malware, Man-in-the-Middle Attacks, Software Misconfigurations)	
4.3. Network Security	134
(Denial of Service Attacks, Eclipse/Routing Attacks, Sybil Attacks)	



Table of Contents

5. Smart Contract Security	148
5.1. Introduction to Smart Contract Security	149
5.2. General Programming Vulnerabilities	156
(Arithmetic Vulnerabilities, Decimal Precision, Digital Signature Vulnerabilities External Dependencies, Text Direction, Unsafe Serialization)	
5.3. Blockchain-Specific Vulnerabilities	187
(Access Control, Denial of Service, Frontrunning, Rollback Attacks Timestamp Dependence, Weak Randomness)	
5.4. Platform-Specific Vulnerabilities	219
(Denial of Service: Block Gas Limits, Denial of Service: Unexpected Revert Forced Send of Ether, Missing Zero Address Checks, Reentrancy, Short Addresses Token Standards Compatibility, Unchecked Return Values, Unsafe External Calls)	



Table of Contents

5.5. Decentralized Finance (DeFi) Vulnerabilities (Access Control, Centralized Control and Governance, Cross-Chain Bridge Vulnerabilities, Frontend Vulnerabilities, Price Manipulation)	267
5.6. Non-Fungible Token (NFT) Vulnerabilities (Forged NFTs, Off-Chain Assets, Malicious NFTs, Unlimited Token Supplies)	295
5.7. Securing Smart Contracts Secure Smart Contract Development Resources Smart Contract Security Audit Tools	320
6. Developing Secure Blockchain Systems	341
6.1. Blockchain Architecture	343
6.2. Balancing Blockchain Benefits and Risks	351
6.3. Regulatory Considerations for Blockchain Systems	361



Module 1

Introduction to Blockchain

7 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Module 1. Introduction to the Blockchain

Currently, there are at least 1,000 blockchains with at least four types of blockchain networks. Bitcoin cryptocurrency was the first implementation of blockchain technology. Since then, the term has been used to describe a wide range of distributed ledger technologies for numerous uses beyond cryptocurrencies.

Overview

- What is Blockchain?
- Benefits of Blockchain
- Architecture of Blockchain
 - Blockchain Transactions and Data Storage
 - Blockchain's Blocks and Chains
 - Inside a Blockchain Node
 - The Blockchain's Peer-to-Peer Network
- Blockchain Security vs. "Traditional" Cybersecurity
- Why Blockchain Security is "Hard"



What is a Blockchain?

- The blockchain is designed to implement a distributed, decentralized immutable digital ledger
 - Each node in the network maintains a copy of the ledger
 - Complex protocols maintain consensus
- Very different from traditional systems
 - Centralized digital ledger
 - Based on trust in the centralized authority



The original blockchain technology was defined in the Bitcoin whitepaper (<https://bitcoin.org/bitcoin.pdf>) in October 2008 and first implemented in the Bitcoin network in January 2009. The goal of blockchain technology is to create a distributed, decentralized, and immutable digital ledger. Let's unpack this:

- **Distributed:** Instead of keeping a single, centralized copy of the ledger, copies are distributed across the blockchain network. All nodes in the blockchain network have the option to maintain a full copy of the distributed ledger.
- **Decentralized:** Blockchain systems have no centralized authority responsible for defining the "official" version of the blockchain's digital ledger. Instead, blockchain networks use consensus algorithms to determine which node in the network should create the next block added to the blockchain.
- **Immutable:** Blockchains' digital ledgers are designed to be immutable. This immutability is backed up by cryptographic algorithms and protocols making it infeasible for a node to rewrite the history of the network by creating an alternative version of an accepted block and getting the rest of the network to accept it.

Blockchain technology provides an alternative to traditional methods of maintaining digital ledgers and hosting applications. These traditional systems rely upon:

- **Centralized Ledgers:** Traditional systems rely on a centralized, “official” version of a ledger. While backups and other copies may exist, there is a centralized process or official version that makes all updates and distributes them to these other systems.
- **Trust in Centralized Authority:** Traditional systems rely on trust in a centralized authority to maintain the integrity and authenticity of the digital ledger. For example, financial institutions have the ability to reverse transactions that are determined to be fraudulent. Their customers rely on the financial institution to properly identify fraud and not reverse valid transactions.

Fundamentally, blockchain is a governance technology. Not a financial one. The power of blockchain is in creating trust without reliance on trusted third parties.

Benefits of the Blockchain

- Anonymity
- Decentralization
- Fault Tolerance
- Immutability
- Transparency
- Trustless



Blockchain technology differs significantly from traditional methods of maintaining a digital ledger, hosting applications, etc. The design of blockchain enables it to provide certain benefits or “promises”, including:

- **Anonymity:** In the blockchain, identity is managed at the account level, where actions on the blockchain are associated with a given blockchain account and address. Most blockchains have no tie back from this account to a person’s real-world identity. This provides a level of anonymity on the blockchain. However, this anonymity is not perfect. By analyzing transactions on the blockchain and “patterns of life”, it may be possible to determine the owner of a particular blockchain account.
- **Decentralization:** Blockchain systems are designed to be decentralized. Each node in the network participates in the operation of the blockchain, including storing a complete copy of the digital ledger and potentially participating in the consensus

and block creation processes. This eliminates the need to trust a centralized authority and provides some benefits in terms of resiliency and redundancy.

- **Fault Tolerance:** Blockchain decentralization provides a high level of fault tolerance. Since no node in the network is essential to the operations of the network, no single points of failure exist within the network. If one or even several nodes go down in a blockchain network, it should be able to continue operations with slightly decreased efficiency and security. However, this is far more resilient than a centralized system, where a failure of a critical component can bring the entire system offline.
- **Immutability:** The blockchain's digital ledger is designed to be immutable with this immutability backed up by cryptographic algorithms. Blockchain immutability is essential to the operation of the blockchain because, with no authoritative version of the ledger, some protection must exist to prevent nodes from rewriting their copies of the ledger at will. Blockchain immutability also provides stronger protections against modification than traditional systems, which can often be modified at will by the centralized authority maintaining the ledger.
- **Transparency:** A certain level of transparency is vital to blockchain decentralization. Nodes in the blockchain network are expected to validate transactions before including them in their copies of the digital ledger, which requires a level of insight into these transactions. Blockchain transparency can also build trust in its users by showing exactly how the system works, but this comes at the cost of some privacy since everything is open and visible on a blockchain's distributed ledger.
- **Trustless:** Traditional systems commonly require users to trust in the centralized authority responsible for maintaining the system. Blockchain systems attempt to transfer this trust to cryptographic algorithms and other protocols with incentives designed to encourage malicious, greedy nodes to act in the best interests of the network because it is more profitable than attacking. Blockchains are designed to make it possible for nodes to collaborate without trusting one another as long as

a certain percentage of the nodes remain honest and do not collude to break the system.

Architecture of the Blockchain

- Blockchains are complex, multi-layered systems
- Some of the key components of a blockchain system include:
 - Transactions
 - Blocks and chains
 - Nodes
 - Peer-to-Peer Network



Blockchain systems are designed to provide an alternative to traditional, centralized methods of maintaining a digital ledger. Instead of relying on a centralized authority to maintain an “authoritative” version of the ledger, blockchain networks composed of mutually-distrusting nodes do so.

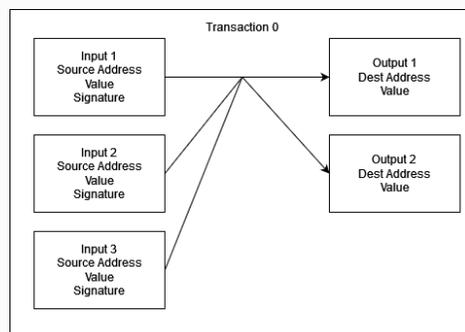
Blockchain systems accomplish this through complex, multi-layered protocols and ecosystems. In later modules, we’ll explore many of these layers in detail, but some core concepts are necessary to an understanding of how blockchains work and how they are secured, including:

- **Transactions:** Basic data storage structure of blockchain
- **Blocks and Chains:** Transactions are collected into blocks, which are linked together by “chains”
- **Nodes:** Computers that run blockchain software and store a copy of the distributed ledger
- **Peer-to-Peer Network:** Network over which blockchain nodes communicate

transactions, blocks, and any other information needed for the blockchain's operation

Blockchain Transactions and Data Storage

- Transactions are the basic unit of data storage in the blockchain
- Anyone can create a blockchain transaction
 - Digitally signed by an account's private key
- Transactions are then broadcast to the network



With a distributed, decentralized ledger, blockchain networks need a well-defined means of adding data to the digital ledger. Like a database has a particular format for records in a table, the blockchain uses transactions to organize the data being added to the blockchain.

Blockchain transactions get their name from the fact that – in original blockchains like Bitcoin – a transaction was literally designed to record one or more transfers of value. Bitcoin was designed to implement a decentralized financial system, and the blockchain was its account book, working near-identically to a bank's accounts ledger.

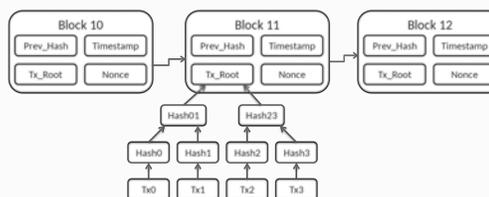
Now, blockchain transactions can contain different types of data. For example, smart contract platforms store executable code in transactions and take advantage of the structure and features of the blockchain to coordinate the execution of this code across the entire blockchain network.

Transactions can be created by any account on the blockchain network (not limited to blockchain nodes). If a transaction is valid and digitally signed using the private key associated with that account, then it will be accepted by the rest of the blockchain network. After a transaction is created, it is broadcast to the rest of the network to be

included in the digital ledger.

Blockchain's Blocks and Chains

- Blocks organize transactions and add them to the distributed ledger
- Blocks contain a header and a body
- The hash of the previous block header is included in the header of the next block



Source: https://en.wikipedia.org/wiki/Blockchain#/media/File:Bitcoin_Block_Data.svg

13 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Transactions record one or more transfers that need to be recorded on the blockchain's digital ledger. However, transactions are not immediately recorded on the blockchain's ledger. For all nodes in the network to agree on the history of the network, they need to be able to organize transactions in some way.

Blockchain consensus algorithms are designed to do so, but they operate on a block level rather than for each transaction. This is to improve the throughput, scalability, and predictability of the blockchain. Attempting to go through a full consensus process for each transaction is slow and unscalable and makes it difficult to predict when new data would be added to the blockchain (since transactions can be added at any time).

Blocks are produced at near-regular intervals by block producers and are organized into:

- **Header:** The block header contains metadata about the block. Common fields include the hash of the previous block header, a timestamp, the hash of the Merkle Tree organizing the block's transactions, and a nonce (used in Proof of Work).

- **Body:** The block body is a list of the transactions contained within a block. These transactions will be logically organized using a Merkle Tree (more on this in a later section) and summarized using the transaction root in the block header.

An individual block in the blockchain is like a single page in an accounts ledger. Creating a valid version of a block is not difficult; block producers do so at regular intervals. This means that an attacker could develop an alternative version of a block that replaces the transactions that it contains with a different collection of valid transactions.

Doing so enables a double-spend attack, where an attacker replaces one version of a transaction (sending cryptocurrency to Alice) with another (sending the same cryptocurrency to Bob). These transactions are mutually exclusive because they can't be included in the same version of the digital ledger. However, if an attacker can replace a version of the blockchain containing the transaction to Alice with one containing the transaction to Bob, this is a problem if Alice has already accepted the payment and acted on it (shipping a product, etc.).

The previous block header hash contained in each block header is the "chain" in the blockchain and makes it much harder to replace one version of a block with another. Changing one block's transactions changes its header, which changes the header of the next block, and so on. Blockchain immutability relies on the fact that replacing one block requires finding valid versions of every following block, which is much harder.

Inside a Blockchain Node

- A “node” is a computer that participates in the management of the blockchain
- Blockchain nodes can have various duties
 - Maintaining a copy of the distributed ledger
 - Creating and distributing blocks
 - Executing smart contract code



A blockchain node is a computer that participates in the running of the blockchain. This includes running the blockchain software, maintaining the ledger, etc. A blockchain node essentially does what a centralized authority would do in a traditional system; however, there are multiple nodes coordinating across the blockchain network.

Blockchain nodes can have various duties that help to maintain the blockchain network, including:

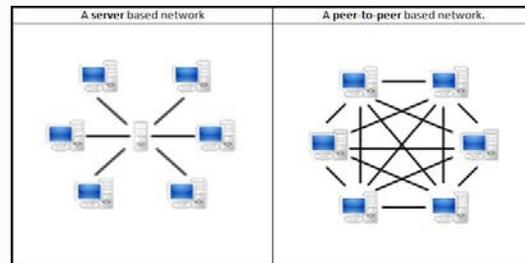
- **Ledger Storage:** Each node in the blockchain network can maintain a copy of the blockchain’s digital ledger. A full node will store copies of the headers and bodies of each block. Nodes will update their copies of the ledger by validating each block when it arrives and then adding it to its stored copy of the blockchain.
- **Block Production:** Nodes may elect to participate in the blockchain consensus and block production process. If this is the case, a node may occasionally be selected to create the next block in the blockchain. If so, they gather a set of transactions, verify that they are valid (no double-spends, etc.), and build a valid block. Once a block has been created, it is distributed via the blockchain’s peer-to-peer network.

- **Smart Contract Code Execution:** Smart contract platforms use the blockchain and its transactions to organize code to be executed on the distributed “world computer”. When a node receives and validates a new block, it is expected to run the code in its copy of the blockchain virtual machine (VM) and update the state of that VM accordingly. This allows programs to be run “on top of the blockchain” in a distributed and decentralized fashion.

Blockchain decentralization should mean that no node in the blockchain network is essential to its operations; however, all nodes play a part. Node security is essential to the functionality and security of the blockchain.

The Blockchain's Peer-to-Peer Network

- Blockchain decentralization demands a peer-to-peer network
- Each node in the network is connected to a few neighbors
- Data moves through the network via multiple hops



<https://en.bitcoinwiki.org/wiki/Peer-to-peer>

15 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Blockchain systems are heavily reliant on node-to-node communications. The network needs to communicate to share new transactions and blocks regularly to make updates to the distributed ledger. The decentralization focus of blockchain technology means that it cannot use a client-server model, where all data is uploaded and downloaded from a centralized system.

Instead, blockchain networks are implemented as peer-to-peer networks, where each node is directly connected to other nodes. Most blockchain networks do not use fully-connected peer-to-peer networks due to their scalability limitations. In a fully-connected network, the number of links scales with the square of the number of nodes, calculated as $n(n-1)/2$. For large networks like the Bitcoin blockchain, this quickly becomes unscalable. Also, nodes would have to deal with the complexity of tracking the status of links to nodes that go up and down unexpectedly.

Blockchain nodes typically are directly connected to only a few other nodes. In this model, information moves through the blockchain network through a series of hops from one node to another. With multiple different routes from a source to a destination, peer-to-peer networks are highly redundant and resilient.

Blockchain Security vs. Cybersecurity

- Traditional cybersecurity is a vital component of blockchain security
 - Application Security
 - Endpoint Security
 - Network Security
- However, blockchain builds an entire ecosystem using the blockchain software
- This creates blockchain-specific security concerns
 - Consensus
 - Ledger correctness
 - Smart contract security



Blockchain systems are defined as theoretical protocols, but they need to be implemented to be usable. While it is possible to implement protocols like Bitcoin using pen, paper, and carrier pigeon, modern blockchain systems are implemented using modern IT systems.

This means that traditional cybersecurity is a core component of blockchain security, including:

- **Application Security:** Blockchain systems are implemented as software that performs the functions outlined by the blockchain protocol just like a web browser performs functions related to the HTTP(S) protocol. This blockchain software can contain vulnerabilities and should be designed, implemented, and tested in accordance with DevSecOps best practices.
- **Endpoint Security:** The blockchain software is run on blockchain nodes, and the security of these blockchain nodes is vital to the performance and security of the blockchain system. Endpoint security best practices, such as the use of antivirus and deployment of anti-DDoS solutions, are equally applicable to blockchain nodes as other computer systems.

- **Network Security:** Blockchain networks are heavily dependent on the performance and security of their peer-to-peer networks to carry block and transaction data and support the consensus. The design and security of these networks are essential to ensuring the performance and security of the blockchain.

While all of these are important components of blockchain security, the security needs of the blockchain go beyond these. Within the blockchain software and protocols is implemented a complete ecosystem that creates, maintains, and secures a shared digital ledger and enables decentralized execution of smart contract code. As a result, the complexity of this blockchain ecosystem creates blockchain-specific security challenges and concerns. Some examples include:

- **Consensus:** The blockchain network needs to be in an agreement regarding the current state of the digital ledger.
- **Ledger Correctness:** The digital ledger needs to be free from invalid transactions.
- **Smart Contract Security:** Programs that run on top of the blockchain must be protected against exploitation.

Why Blockchain Security is “Hard”

- Blockchain systems have undergone numerous hacks and audited code is still vulnerable
- Blockchain security faces numerous challenges
 - Immature Technology
 - Fragmented Ecosystem
 - Decentralized Governance
 - Open-Source Code
 - Publicly-Accessible Platforms
 - Immutable Digital Ledger



Blockchain technology has been around for several years and has received significant investment and research effort. However, blockchain security still lags far behind blockchain development. Major blockchain hacks occur on a regular basis, and the amount of money stolen from Decentralized Finance (DeFi) projects alone is in the tens of billions of dollars.

Blockchain security face significant challenges, including:

- **Immature Technology:** While Bitcoin has existed for over a decade, smart contract platforms are much younger, and new ones are still emerging. This means that the blockchain environment is still evolving rapidly and platforms are not yet stable, making security research more complex.
- **Fragmented Ecosystem:** Numerous blockchain platforms exist, and it is still uncertain which (if any) of them will end up winning in the long term. With many different platforms, all with their own unique take on blockchain and smart contract technology, security research efforts are diluted. Additionally, the existence of cross-chain bridges and other links between platforms increases the complexity of securing these systems since multiple independent systems are

interrelated.

- **Decentralized Update Processes:** Decentralization is a core tenet of blockchain and this extends to the management of the nodes that run the blockchain software. Without the ability to compel updates to blockchain software, it can be difficult to ensure that nodes install vital security updates before vulnerabilities are discovered and exploited by an attacker. This differs significantly from traditional cybersecurity, where patching is a core part of the vulnerability management process.
- **Open-Source Code:** Blockchain systems operate under an open-source ethos, which is both good and bad for security. On the positive side, it allows third-party audits of critical blockchain code; however, it also makes it easier for attackers to identify and exploit vulnerabilities in blockchain systems.
- **Publicly Accessible Platforms:** Most blockchains are designed to be publicly accessible, and creating an account is as simple as generating the private key used to control access to that account. Public access to blockchain systems makes it easy for attackers to exploit identified vulnerabilities since they have full, anonymous access.
- **Immutable Digital Ledger:** All blockchain transactions are recorded on an immutable ledger. This makes it infeasible to reverse successful attacks as rewriting the history of the ledger requires either significant resources or breaking the rules of the protocol.

Summary

- Blockchain is a distributed, decentralized, and immutable digital ledger.
- Blockchain benefits include Anonymity, Decentralization, Fault Tolerance, Immutability, Transparency, Trustless
- Blockchains are complex, multi-layered systems. Key components include Transactions, Blocks and Chains, Nodes, Peer-to-Peer Network
- “Traditional” Cybersecurity is a prerequisite for blockchain security. However, blockchain introduces blockchain-specific security challenges and concerns such as Consensus, Ledger Correctness and Smart Contract Security
- Blockchain security faces numerous challenges including Immature Technology, Fragmented Ecosystem, Decentralized Update Process, Open-Source Code, Publicly Accessible Platforms, Immutable Digital Ledger



In this module we covered:

Blockchain is a distributed, decentralized, and immutable digital ledger.

Blockchain benefits include Anonymity, Decentralization, Fault Tolerance, Immutability, Transparency, Trustless

Blockchains are complex, multi-layered systems. Key components include Transactions, Blocks and Chains, Nodes, Peer-to-Peer Network

“Traditional” Cybersecurity is a prerequisite for blockchain security. However, blockchain introduces blockchain-specific security challenges and concerns such as Consensus, Ledger Correctness and Smart Contract Security

Blockchain security faces numerous challenges including Immature Technology, Fragmented Ecosystem, Decentralized Update Process, Open-Source Code, Publicly Accessible Platforms, Immutable Digital Ledger

Module 2

Blockchain Cryptography

19 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Module 2. Blockchain Cryptography

Introduction to Blockchain Cryptography

- Blockchain technology is designed to provide a decentralized alternative to traditional systems
- Cryptographic algorithms and protocols fulfill their role
 - Provides stronger security guarantees
- Blockchain technology uses
 - Hash functions
 - Public-key cryptography
 - Advanced cryptography



Traditional ledger systems rely heavily on a centralized authority to perform certain roles. In addition to synchronizing the updates to the ledger, the centralized authority is responsible for authenticating and validating this data and ensuring that unauthorized changes cannot be made to the ledger.

Blockchain technology is designed to eliminate the need for these centralized authorities. To do so, they replace these key duties of a centralized authority with cryptographic algorithms and protocols. This provides stronger security guarantees than trust in a centralized authority.

For example, consider the problem of ledger immutability. As mentioned previously, the blockchain is structured so that changing one block has a cascading effect, making it necessary to find valid replacements for many blocks. The difficulty of doing so provides much stronger immutability protections than trusting in a centralized authority not to make modifications to its own digital ledgers.

Blockchain technology uses cryptographic algorithms in many ways. At its most fundamental level, blockchain technology is built using hash functions and public-key cryptography. Some blockchains extend the use of cryptography to incorporate

advanced cryptographic algorithms.

Module 2: Blockchain Cryptography

Section 2.1: Hash Functions

21 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

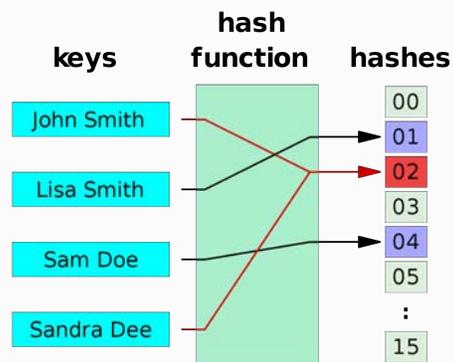
Overview

- Hash Functions
 - What is a Hash Function?
 - Security Assumptions of Hash Functions
 - Benefits of Hash Functions
 - Hash Functions and the Blockchain
 - Security Threats to Hash Functions



What is a Hash Function?

- Hash functions can take any input and map it to a fixed-size output
 - SHA-256 has 256 bit outputs
- Multiple inputs will map to the same hash output
 - This is called a hash “collision”



Source: https://en.wikipedia.org/wiki/Hash_function#/media/File:Hash_table_4_1_1_0_1_0_LL.svg

23 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Hash functions are mathematical functions that pseudo-randomly map any input to a fixed-size output. For example, SHA-256 is named for its 256-bit hash output.

By design, hash functions have a many-to-one relationship between inputs and outputs. By taking any value as an input, a hash function has an infinite input space. However, its output length is fixed, making it finite. Mapping any infinite space to a finite space means that multiple inputs will produce the same output. In fact, an infinite number of inputs will produce the same output.

The image to the right illustrates the operation of a hash function being used in a hash table. Hash tables take advantage of the pseudorandom mapping of hash function inputs to outputs to provide a way to organize and later query random data. In this case, the inputs are text (names) and the outputs are the values 0-15.

An application could use the hash function to determine that John Smith hashes to 2 and store the name in the corresponding location within an array. Since hash functions are deterministic, John Smith will always be mapped to 2, making it possible to retrieve values later.

As shown in the image, two inputs (John Smith and Sandra Dee) map to the same hash output (2). This is an example of a hash collision. Hash collisions are undesirable in any application of hash functions, and, in the case of the blockchain, can undermine the security of a blockchain protocol.

Security Assumptions of Hash Functions

- Blockchain security relies on hash function collision resistance
 - Collisions definitely exist within a hash function
 - Need to make them infeasible to find
- A collision-resistant cryptographic hash function must:
 - Be a one-way function
 - Have a large output space
 - Be a non-local function



Blockchain systems use hash functions for integrity protections. A strong, cryptographic hash function is collision-resistant, meaning that it is infeasible to find two inputs that produce the same output hash.

If this is true, then guaranteeing the integrity of the hash of a chunk of data guarantees the integrity of the data itself. Since hash functions are deterministic and require no secret data, anyone can verify that a chunk of data matches a stored hash output by computing the data themselves and comparing the two hashes. If the hash function is collision-resistant, then it is infeasible for an attacker to find another version of the data that would produce the same hash output, defeating this validation.

Due to the nature of hash functions, making collisions impossible is impossible. With an infinite input space and finite output space, there will always be an infinite number of collisions or inputs that map to the same output. A strong cryptographic hash function is designed simply to make finding a pair of these inputs computationally infeasible.

To do it, it needs to have three main properties:

- **One-Way Function:** Can't reverse engineer an input from the corresponding output
- **Large Output Space:** There are many different outputs that an input could be mapped to
- **Non-Locality:** Similar inputs produce very dissimilar outputs

Hash Functions: One-Way Functions

- One-way functions are vital to collision resistance
 - Can't calculate an input from an output
- Being many-to-one is an important requirement for one-way functions
 - Impossible to identify which of the potential inputs produced an output
- Also need to use a function that is “hard” to reverse
 - Polynomial complexity to perform, exponential to undo



The first requirement for hash function collision resistance is that the hash function is a one-way function. This means that it is possible and “easy” to calculate a hash output from an input but it is infeasible to reverse the process and compute an input from an output. If this was not true, finding collisions would be possible simply by reversing the calculations performed within the hash function.

Being a many-to-one function is not enough to be a one-way function. For example, the modulo function (the remainder after division) is an example of a one-way function. $15 \pmod{10} = 5 \pmod{10}$. However, it is trivial to find collisions for this function. Any value of the equation $y = 10 \cdot x + 5$ is a collision with an input of 5 in modulo 10.

One-way functions also need to use a mathematical function that is “easy” (polynomial complexity) to perform but “hard” (exponential complexity) to reverse. This asymmetry between legitimate and malicious operations makes it possible to design an algorithm that is usable but infeasible to attack. We’ll discuss this further in the context of public-key cryptography.

Hash Functions: Large State Space

- Hash function collisions are always possible
 - Many-to-one relationship
- According to the Pigeonhole Principle, a brute force search is always possible
- Need to make this search infeasible
 - Harder than expected due to Birthday Paradox



Source: <https://en.wikipedia.org/wiki/Pigeonholing#/media/File:TooManyPigeons.jpg>

26 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

By definition, hash function collisions will always be possible. Hash functions create a many-to-one relationship between inputs and outputs. As a result, it is always possible to find two inputs that produce the same output.

This means that a brute-force search for collisions is guaranteed to succeed. The Pigeonhole Principle states that an attacker would have to check one more input than the number of possible outputs to be guaranteed to find a collision. As shown in the image, it is impossible to find a way to fit 10 pigeons into 9 pigeonholes without placing two different pigeons in the same hole.

With a 256-bit hash function, an attacker must try $(2^{256})+1$ inputs to be guaranteed a hash collision. This is computationally infeasible on modern computers or any computer. For reference, a modern hard drive using every atom in the universe would be too small to store all of these options by several orders of magnitude. There are less than 2^{272} atoms in the universe and a modern hard drive requires 1 million ($>2^{19}$) atoms per bit, and $(2^{256} \cdot 2^{19}) = 2^{275} > 2^{272}$.

This only defines the worst-case scenario for a brute force search. An attacker may get lucky on the first try (with vanishingly low probability) and will need to search

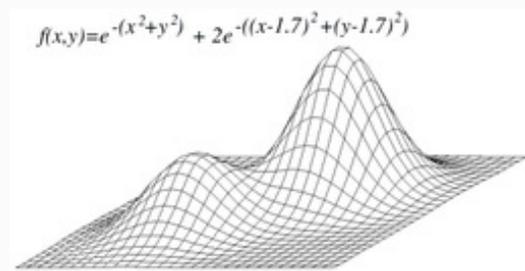
$(2^{256})/2=2^{255}$ possibilities in the average case, which is still infeasible.

However, the Birthday Paradox states that an attacker will have to search $2^{(N/2)}$ options to have a high probability of finding a collision for an N-bit hash function. Consider the question of whether two people in a room have the same birthday. To be guaranteed for this to be true you need to have at least 367 people in the room (accounting for leap days), but with only 23 people, there is more than a 50% chance a “collision” will occur. The reason for this is that the number of potential pairings between people is $23! = 253$. Since 253 is greater than half the number of days in the year, there is an over 50% chance of a match. With 60, the probability is 99.4%.

As a result of the Birthday Paradox, the number of guesses needed to have a high probability of finding a hash collision in a 256-bit hash function is 2^{128} , which is many orders of magnitude smaller than 2^{256} but still infeasible. Modern hash functions have output spaces deliberately selected with the Birthday Paradox in mind.

Hash Functions: Non-Locality

- Non-locality is vital for collision resistance
 - Similar inputs produce outputs that differ in half their bits
- Locality enables hill-climbing attacks
 - Flip one bit of the input
 - Keep change if it is “closer” to goal
 - Repeat until maxima is found



Source: https://upload.wikimedia.org/wikipedia/commons/thumb/7/7e/Local_maximum.png/260px-Local_maximum.png

27 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Brute force attacks are a worst-case approach to attacking a hash function. Another option is a hill-climbing attack, which takes incremental steps and uses an evaluation function to move slowly towards the goal.

A hill-climbing attack is composed of the following steps:

1. Select a random starting input and calculate its hash
2. Flip a single, random bit of that input and calculate the hash of the result
 1. If result is closer to the goal, keep the change
 2. Otherwise, discard the change
3. Repeat step 2 until a collision is found

The image on the slide is an example of a local function, which enables hill-climbing attacks. Note that similar inputs produce similar outputs, so it is possible to calculate how “close” a particular input is to the local maximum. By only taking steps “uphill,”

an attacker can find the highest point in their vicinity much more quickly than with a random search. While this local maximum may not be the goal (i.e. the highest point or a collision), performing a hill climbing attack multiple times from random starting points provides a high probability of success.

A non-local hash function is one where a single bit difference in inputs produces outputs that differ in about half of their bits. This provides the minimum possible amount of information to an attacker about the relationship between inputs. (Flipping all but one bit is as bad as only flipping one bit). By destroying this information, a hash function makes it impossible to complete steps 2.1 and 2.2 above, making a hill-climbing attack no more effective than a brute force search.

Benefits of Hash Functions

- The design of hash functions enables them to provide a few benefits
 - Data Summarization
 - Integrity Protection
- These benefits lead to their use in various areas, including the blockchain



A hash function converts any input into a fixed-size output. Additionally, it performs this process in such a way that it is highly unlikely that another input will be found (intentionally or accidentally) that will map to the same output.

This provides a couple of key benefits of hash functions, including:

- **Data Summarization:** Hash functions convert an arbitrary-sized input into a fixed-size output in a collision-resistant fashion. This is useful in hash tables and similar applications.
- **Integrity Protection:** Hash function collision resistance means that hash values can be used as a proxy for data for integrity protections. If a hash output is protected against modification, then collision resistance means that the associated data cannot be modified without detection. This hash function feature is commonly used to demonstrate that a program or software update has not had malicious code inserted into it.

Hash Functions in the Blockchain

- Hash functions are one of the fundamental cryptographic algorithms used in blockchain
- Applications of hash functions in blockchain include
 - The blocks' "chains"
 - Merkle trees
 - Digital signatures

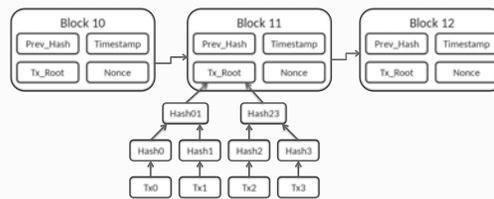


Hash functions have significant benefits, and they are one of the primary building blocks of blockchain technology. Some of the applications of hash functions in blockchain include:

- **Blocks' chains:** Hash functions link the blocks of the digital ledger into the blockchain
- **Merkle trees:** Merkle trees summarize the transactions contained within a block body into a single value in a block header
- **Digital signatures:** Hash functions are part of digital signature algorithms, providing data authentication and integrity protection

The Blocks' Chains

- Blockchain blocks each contain a set of transactions
 - A valid block is relatively easy to create
 - Block producers do so regularly
- “Chains” link blocks together
 - Previous block header’s hash in each block header
 - Changes to blocks cascade down the chain



Source: https://en.wikipedia.org/wiki/Blockchain#/media/File:Bitcoin_Block_Data.svg

30 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

One of the defining features of the blockchain is the immutability of its digital ledger. Once a transaction has been added to the ledger as part of a block, it quickly becomes infeasible to remove and replace it.

This is true because each block in the blockchain is “chained” together with hash functions. Each block header contains the hash of the previous block header. This creates a cascading effect if an attacker attempts to change the contents of a block in the blockchain. Changing one block substantially changes its hash due to hash function non-locality. This change will be reflected in the previous block hash of the next block, which changes its hash. This process repeats down the rest of the blockchain.

To change the contents of an existing block in the blockchain, an attacker has two options:

1. **Find a Collision:** The cascading effects of a change to a block only occur if the hash of that block’s header changes. If the attacker can find another version of a block with the same header hash, then the header of the next block (and every following block) does not change. However, hash function collision resistance

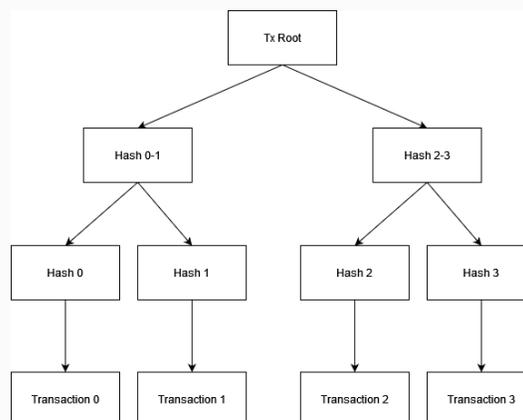
makes this infeasible.

- 2. Replace All Modified Blocks:** Under the longest chain rule, an attacker can create a different version of the blockchain and get it accepted by the network if it is “longer” than the official chain. However, this requires the ability to create alternative versions of valid blocks faster than the rest of the blockchain network. Blockchain consensus algorithms are designed to make this infeasible as well.

Hash functions’ use as blocks’ chains is essential to the immutability and security of the blockchain. By making it much more difficult to modify the contents of the digital ledger, these chains protect against changes to the blockchain’s history.

Merkle Trees

- A block header stores metadata about a block
 - Not the transactions that it contains
- Transactions are organized into a Merkle tree
 - Root of the Merkle tree is included in the block header
 - Ensures transaction immutability



The blocks' chains protect the immutability of the headers of each block in the blockchain. However, the transactions contained within a block are not included in the block header or within this hash value.

Merkle trees are used to securely summarize transactions and extend the protection of the blocks' chains to them. In the Merkle tree shown to the right, the transactions contained within a block are located at the bottom of the tree. Each node above them (Hash 0, Hash 1, etc.) contains the hash of the transaction below it. All nodes above these nodes contain the hash of the nodes below them (the two values are concatenated together). The root hash of the Merkle tree is then included as one of the fields within the block header, which has the protection of the blocks' chains.

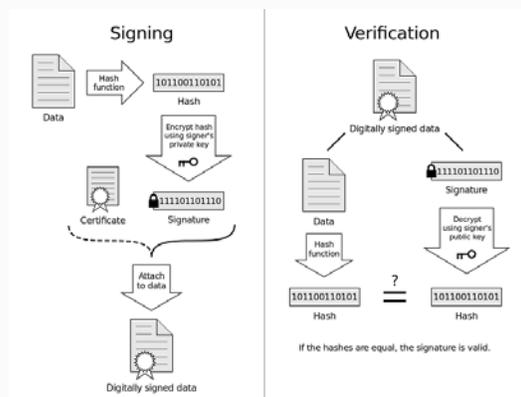
Due to hash function collision resistance, this scheme provides immutability protections to the transactions contained within the block's body. Since it is infeasible to find a collision for a strong hash function, it is infeasible to find two versions of a Merkle tree that produce the same root hash. As long as the root hash is protected by the blocks' chains, so is every transaction within the Merkle tree.

Merkle trees also provide the ability to verify the presence of a transaction within a

tree without knowledge of any of the other transactions. Given Transaction 0, Hash 1, Hash 2-3, and the root hash, it is possible to calculate Hash 0 from Transaction 0, Hash 0-1 from Hash 0 and Hash 1, and the root hash from Hash 0-1 and Hash 2-3. Since it is infeasible to find two versions of the tree with the same root hash, the fact that the calculated root hash matches the one in the block header proves the presence of Transaction 0 in that block without the need to know Transactions 1-3.

Digital Signatures

- Digital signatures provide authenticity and integrity
- All blockchain transactions are signed by their creators
- Hash functions are one part of the digital signature process



Source: <https://cheapsslsecurity.com/blog/digital-signature-vs-digital-certificate-the-difference-explained/>

32 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Digital signatures are a combination of hash functions and public-key cryptography. Hash functions securely summarize the data being signed, using its collision resistance to make it infeasible to find two versions of the data that will produce the same valid signatures. Public-key cryptography generates the actual signature, providing the authenticity of data and that it has not been modified since the signature was created.

Blockchain systems use digital signatures to protect the integrity and authenticity of transactions being distributed through the blockchain network and added to the digital ledger. Every transaction will pass through multiple different nodes as it moves through the peer-to-peer network, and each node has an incentive to modify the transaction to its benefit. Digital signatures make this infeasible, providing a basis for trust in the transactions contained within the digital ledger.

Security Threats to Hash Functions

- Blockchain uses hash functions for integrity protection
 - These integrity protections depend on collision resistance
- Hash function collision resistance can be threatened by
 - Flawed or insecure algorithms
 - Technological advancements
 - Quantum computing
 - Grover's Algorithm



Hash function security is critical to blockchain security. Hash functions are used to provide data integrity guarantees in the blocks' chains, Merkle trees, and digital signatures. These integrity protections depend on the collision resistance of hash functions. If an attacker can find a hash collision in a block's chain, a Merkle tree, or a digital signature, then they can break ledger immutability.

Modern hash functions are designed to be resistant to attack, but some threats do exist to hash function security. These include:

- **Insecure Algorithms:** Hash function collision resistance depends on the assumption that the best way to find a hash collision is via a brute force search. If weaknesses in a hash algorithm make a more efficient attack possible, this could break hash function collision resistance and security.
- **Technological Advancements:** A brute force search against a modern hash function is computationally infeasible on modern technology and likely to remain so in the future. However, if technological advancements make a brute force search possible for a hash function, this breaks the security of the algorithm.

- **Quantum Computing:** Quantum computing works differently than traditional computers, and these differences can have significant impacts on cryptographic security. Grover's algorithm is a quantum computing algorithm that reduces the complexity of a brute force search on an N-bit hash function from 2^N to $2^{(N/2)}$, effectively halving the output length of the algorithm. While this does not break modern hash algorithms (128 bits is still ample for security), it dramatically weakens them, and more effective algorithms may be discovered in the future.

Summary

- Hash Functions
 - What is a Hash Function?
 - Security Assumptions of Hash Functions
 - Benefits of Hash Functions
 - Hash Functions and the Blockchain
 - Security Threats to Hash Functions



Module 2: Blockchain Cryptography

Section 2.2: Public Key Cryptography

35 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- What is Public Key Cryptography?
- Benefits of Public Key Cryptography
- Blockchain Applications of Public Key Cryptography
- Security Threats to Public Key Cryptography



What is Public Key Cryptography?

- Encryption algorithms can be classified by the keys they use for encryption and decryption
 - Symmetric
 - Asymmetric
- Asymmetric or “public key” cryptography is widely used in blockchain



Encryption algorithms are designed to provide confidentiality by allowing data to be converted into meaningless gibberish (encryption) and restored to their original state (decryption). The ability to decrypt data depends on knowledge of a secret key.

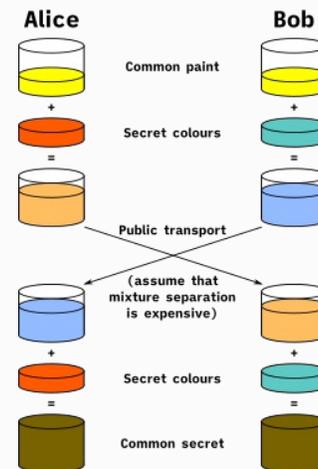
Encryption algorithms can be classified as symmetric or asymmetric. Symmetric encryption algorithms use the same secret key for both encryption and decryption. In general, symmetric encryption is more efficient (making it better for bulk data encryption), but it requires a secret key to be shared over a secure channel.

Asymmetric or “public key” cryptography uses a pair of related keys. Public keys are used for encryption, while private keys are used for decryption. As the name suggests, a public key is designed to be public information, meaning that it can be shared over an open channel without undermining the security of the encryption algorithm.

The use of a pair of related keys gives asymmetric cryptography some capabilities and benefits that symmetric encryption algorithms lack. These properties make it ideally suited to solving some of the main challenges of blockchain.

PKC: Mathematically “Hard” Problems

- Mathematically “hard” operations are much easier to perform than reverse
 - Polynomial vs. exponential complexity
- Common “hard” problems include
 - Factoring
 - Discrete logarithm



Source: https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Diffie-Hellman_Key_Exchange.svg/250px-Diffie-Hellman_Key_Exchange.svg.png

38 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Public-key cryptography is built using mathematically “hard” problems. These are mathematical operations that are much easier to do than they are to undo. More formally, one operation has polynomial complexity, while the other has exponential complexity.

In the example to the right, mixing colors together is easy, but separating them is “hard”. This makes it possible to develop a common secret over a public channel because, even with access to the blue and orange colors, an eavesdropper can’t determine the final color. Mixing them together would produce a mix with too much yellow. The only way to learn the final color is to determine Alice or Bob’s secret color, either by stealing it or separating a mixture (which is infeasible).

An example of a mathematically “hard” problem in common use in asymmetric cryptography is the factoring problem. Multiplying two numbers together is relatively easy, and the complexity of doing so grows slowly. For example, it is only a little bit more difficult to multiply two 256-bit numbers together than it is to multiply two 255-bit numbers. Factoring the product of those two numbers, on the other hand, is much harder. In fact, the best known way to factor the product of two primes is via a brute force search. While it is guaranteed that one of the factors is less than the

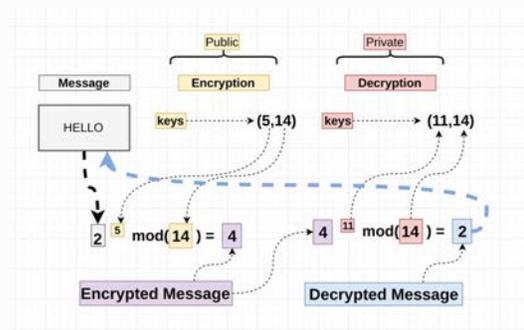
square root of the product, this is still a large space to search. Also, it grows rapidly with the length of the factors. Adding a single bit to the length of the factors doubles the search space.

These asymmetric difficulties and the differences in their growth rates make it possible to develop a system that is both usable but secure by allowing legitimate users to perform only “easy” operations while forcing an attacker to perform “hard” ones. Since the complexity of multiplication grows slowly with the length of the factors while the complexity of factoring doubles with each bit, it is possible to find a key length where a laptop could perform multiplication but all of the computing power on Earth couldn’t factor the result.

The discrete logarithm problem is another “hard” problem with a similar complexity relationship. In this case, performing exponentiation is “easy” (polynomial complexity), while calculating a logarithm is “hard” (exponential complexity).

PKC: Building a Cryptosystem

- Asymmetric cryptography uses “hard” problems to create usable but secure systems
- Legitimate users perform exponentiation (easy)
- Attackers must calculate logarithms (hard)



Source: <https://medium.com/hackernoon/how-does-rsa-work-f44918df914b>

39 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Mathematically “hard” problems are the basis for public-key cryptosystems. The image to the right shows how a system based on the discrete logarithm problem would work. In this case, there are three key pieces of information:

- The public key (5) is a public value
- The private key (11) is a secret value
- The modulus (14) is a public value

The message Hello is mapped to a value of 2 using a codebook or similar tool. Given this plaintext, we can calculate the ciphertext as $2^5 \pmod{14} = 32 \pmod{14} = 4$. This value of 4 can be sent over a public channel to the intended recipient, who can calculate $4^{11} \pmod{14} = 4194304 \pmod{14} = 2$. With the correct private key, a legitimate user can decrypt the message and retrieve the plaintext while only performing an “easy” operation (exponentiation).

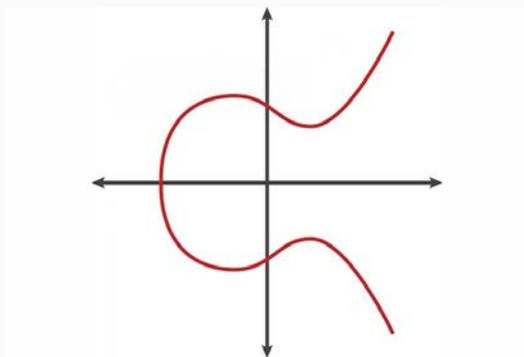
An attacker, on the other hand, only knows the public key, modulus, and the ciphertext (4). Calculating the value of 2 from this information requires either solving

a logarithm. Since this is a “hard” operation, it is infeasible for the attacker to accomplish, ensuring the security of the protocol.

In this and other asymmetric encryption schemes, the public and private keys are carefully selected to ensure that one undoes whatever the other does. In this case, $(x^y)^z = x^{(y \cdot z)} = x^{(z \cdot y)} = (x^z)^y = x$ for any x if y and z are the public and private keys.

PKC: Integer-Based vs. Elliptic Curve Crypto

- Some elliptic curve operations are equivalent to integer operations
 - Point addition/integer multiplication
 - Point multiplication/integer exponentiation
- This makes it possible to build asymmetric cryptography with elliptic curves



Source: <https://www.globalsign.com/application/files/4916/0389/7496/Elliptic-Curve-Cryptography.png>

40 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Some point operations on elliptic curves are equivalent to integer operations. For example, point addition is equivalent to integer multiplication and point multiplication is equivalent to integer exponentiation.

These are the “easy” operations used in many modern asymmetric cryptographic algorithms. Since point subtraction and division are also “hard”, it is possible to translate integer-based public key cryptography over to elliptic curves.

Making this move to elliptic curves has some benefits. In general, elliptic curve cryptography requires shorter secret keys and less power than an integer-based version with equivalent security. This makes it well-suited to resource-constrained systems, such as mobile and Internet of Things (IoT) devices.

Benefits of Public Key Cryptography

- Public key cryptography achieves all of the desired goals of cryptography to varying degrees
 - Confidentiality
 - Integrity
 - Authentication
 - Non-Repudiation
- This is because asymmetric cryptography can be used for
 - Data encryption
 - Digital signatures



Cryptographic algorithms are used to provide a few different guarantees. These include:

- **Confidentiality:** Keeping data secret
- **Integrity:** Protecting data against modification
- **Authentication:** Proving that data was created by the alleged sender
- **Non-Repudiation:** Preventing a user from denying their actions

Encrypting data with a user's public key helps to ensure that only someone with knowledge of the private key can read the data. It also provides limited integrity protections because a modified ciphertext will not decrypt to the correct plaintext.

By using public-key cryptography "in reverse", it is possible to achieve other cryptographic goals. Data "encrypted" with a private key and "decrypted" with a public key could only be generated with knowledge of that private key. Digital signatures use this fact to provide integrity, authentication, and non-repudiation

because, if a “decrypted” signature sent alongside a message matches the message, then it must have been created by someone with the right private key and not modified in transit.

Blockchain Applications of Public Key Crypto

- The security guarantees that public key cryptography provides can be invaluable in a blockchain system
- Some of the ways that asymmetric cryptography is applied in blockchain include:
 - Digital signatures
 - Account addressing
 - Data encryption



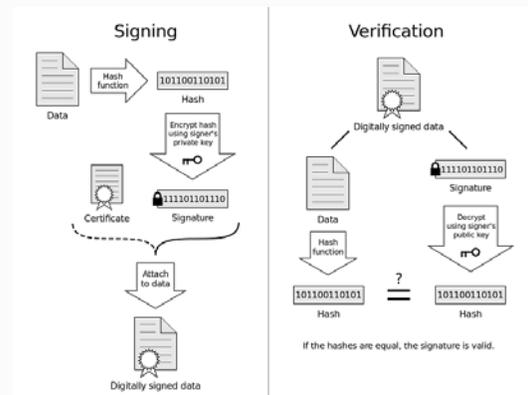
As mentioned previously, hash functions are critical to the operation of the blockchain because they provide integrity protections backed up by cryptography. Public-key cryptography is equally important to blockchain systems because of its ability to provide integrity, authentication, non-repudiation, and (to a less important extent) confidentiality.

Public key cryptography is applied in a few different ways within a basic blockchain environment, including:

- Digital signatures
- Account addresses
- Data encryption

Digital Signatures

- Digital signatures provide integrity, authentication, and non-repudiation
 - Valid signatures can only be generated with the right private key
- Blockchains use digital signatures to ensure transaction validity



Source: <https://cheapsslsecurity.com/blog/digital-signature-vs-digital-certificate-the-difference-explained/>

43 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Digital signatures were mentioned previously in the context of hash functions. However, a digital signature is composed of both hash functions and public-key cryptography. The digital signature generation process is as follows:

1. The message is passed through a hash function. Due to hash function collision resistance, this securely summarizes the data
2. The hash output is “encrypted” with the sender’s private key
3. This signature is sent alongside the message and the sender’s public key to the recipient.

Signature verification consists of the following steps:

1. The recipient hashes the message
2. Using the sender’s public key, the recipient “decrypts” the signature to extract the hash value

3. These hash values are compared. If they match, the signature is valid.

Hash function and asymmetric cryptographic security make it infeasible to forge a signature by either finding a hash collision or creating a signature without the correct private key. Therefore, digital signatures prove that a message was signed with the correct private key and has not been modified in transit. Blockchain technology uses this to validate transactions before accepting them for inclusion in the digital ledger.

Blockchain public keys are tied to a particular account. If a digital signature on a transaction is valid, it proves that someone with knowledge of the correct private key generated the transaction, which means that the transaction is authorized.

Also, it proves that the transaction has not been modified since the signature was generated. Since transactions are broadcast over a peer-to-peer network, this is essential to ensuring the integrity of these transactions.

Account Addressing

- Digital signatures require the ability to validate ownership of a public key
 - Anyone can create a valid signature with their private key
 - Need to be sure that the public key used in validation belongs to the alleged sender
- Blockchains derive account addresses from public keys
 - Often a truncated hash of the key
 - Ties public key to account



Digital signatures are designed to prove that the owner of a private key generated a signature for a piece of data. The recipient of a message can verify this fact by “decrypting” the signature with the provided public key to extract the hash it contains, which can be compared to the hash of the actual data.

The hard problem in digital signatures is proving that a particular public key belongs to the alleged sender. If an attacker intercepts a message, modifies it, generates a signature with their private key, and sends their public key along with the message, then the signature will be valid and verifiable by the recipient. Public Key Infrastructure (PKI) solves this problem by creating chains of digital certificates that attest to ownership of a particular public key.

In blockchain, it is only necessary to prove that a public key belongs to a particular blockchain address. If a transaction is signed with the private key associated with a particular account, then it presumably was created by someone with the authority to do so.

Blockchain ties public keys to account addresses by deriving account addresses from public keys. Often, the address is a truncated version of the hash of the public key.

This makes it possible to determine if a public key is associated with an account as long as hash function collision resistance holds.

Data Encryption

- Public key cryptography can also be used for data encryption
 - Provides privacy on a public ledger
- Asymmetric cryptography doesn't require a key distribution phase
 - Only need to know a user's public key
 - This is published as part of digital signatures



The blockchain is designed to maintain an open, transparent digital ledger. A level of transparency is vital to blockchain decentralization because each node in the network needs to be able to verify the validity of transactions and blocks before accepting them and including them in its copy of the distributed ledger.

However, the transparency of blockchain means that very little privacy exists on the distributed ledger. While it is possible to include messages in blockchain transactions, this is publicly accessible data that anyone with access to the network can read.

While not vital to the functioning of the blockchain, asymmetric cryptography's ability to support encrypted messaging is also a useful feature. With public-key cryptography, it is possible to send an encrypted message to anyone without first creating a shared secret key.

All that is needed to send a message encrypted with an asymmetric cryptographic algorithm is the public key of the recipient, which is published when they generate a digital signature. An encrypted message posted to the blockchain's ledger can only be read by the intended recipient as long as the algorithm and their private key remain secure.

Security Threats to Public Key Cryptography

- The security of public key cryptography can be undermined in a few different ways
 - Weak key generation
 - Insecure key storage
 - Algorithmic weaknesses
 - Quantum computing



Asymmetric cryptography provides numerous benefits. Public key encryption algorithms provide the ability to send confidential messages between parties that have never communicated previously. Digital signature algorithms provide protections for data integrity, authenticity, and non-repudiation.

However, all of these benefits rely on the algorithm being secure. Some of the ways in which public key security can be broken include:

- Weak key generation
- Insecure key storage
- Algorithmic weaknesses
- Quantum computing

PKC Security: Weak Key Generation

- Asymmetric cryptography relies on a pair of related keys
 - Private key is a random number
 - Public key is derived from the private key
- If the private key is not generated using a strong source of randomness, then an attacker may be able to guess it
- This was the case with the “Blockchain Bandit”



Public key cryptography uses related public and private keys. The public key is derived from the private key so that each reverses the effects of the other in a certain mathematical operation (like how $x^y^z=x$ in the earlier example).

The private key in a public/private keypair should be a random value generated to meet the rules of the algorithm (length, primacy, etc.). This number should be generated using a secure source of randomness. If this is the case, then the only way to guess the key is via a brute force search, trying potential values of the private key and testing them against the public key. Modern public-key cryptographic algorithms are designed to make this type of brute force search computationally infeasible.

If the private key is not generated with a strong source of randomness, then it might be possible for an attacker to guess it. If so, an attacker could generate valid digital signatures on behalf of the account with the compromised key. This could be used to steal money from the account or use it to interact with smart contracts.

The case of the “Blockchain Bandit” is a prime example of the effects of weak private key generation. By guessing weak private keys used by accounts on the Ethereum blockchain, an attacker was able to steal approximately 45,000 ETH. Read more at

<https://cointelegraph.com/news/blockchain-bandit-how-a-hacker-has-been-stealing-millions-worth-of-eth-by-guessing-weak-private-keys>

PKC Security: Insecure Key Storage

- Private keys may also be compromised if they are not securely stored
- Some ways in which a private key could be leaked include:
 - Insecure storage and backups
 - Compromised mnemonic seeds
 - Accidental exposure
 - Third-party key storage
 - Insecure hardware wallets
 - Phishing attacks
 - Compromised/malicious software
 - Authorizing malicious transactions



Even a strong private key provides no protection if that private key is known to an attacker. Private keys can be exposed in various ways, including:

- **Insecure Key Storage and Backups:** Private keys can be compromised if they are stored insecurely. For example, malware exists that is designed to search the file system of a computer for data that looks like a blockchain private key. If a private key is stored in a text file or other insecure storage, then this malware will send the key to an attacker, enabling them to generate transactions on a user's behalf.
- **Compromised Mnemonic Seeds:** Private keys can be converted into a set of 12-24 words that make them easier to remember or type into a computer. Since a user's private key can be reconstructed from this mnemonic seed, the seed needs to be protected at the same level as the key itself. Even a partially exposed key might provide an attacker with enough information to brute-force the remaining words/bits.
- **Accidental Exposure:** Some cryptocurrency ATMs and similar systems will print out the private key or a QR code for it when a user makes a deposit. Posting a picture of this on social media, etc., can lead to exposure of the private key.

- **Third-Party Key Storage:** Cryptocurrency exchanges and other service providers may store a user's private keys for them. However, if this service provider is compromised by an attacker or an attacker can learn/guess the password for a user's account with the service, then they might be able to access a user's private keys.
- **Insecure Hardware Wallets:** Hardware wallets are designed to protect private keys by storing them on a dedicated, protected device. However, if this device contains software or hardware vulnerabilities or is fake, an attacker may be able to extract private keys from the hardware wallet.
- **Phishing Attacks:** Phishing attacks are an increased risk when a blockchain user is reliant on a third-party for key storage (cryptocurrency exchange, hardware wallet, etc.). Messages claiming to be from that company may be designed to steal login information or private keys.
- **Compromised/Malicious Software:** Blockchain users commonly use software like MetaMask to interact with the blockchain network if they are not running a node. If this or other software is compromised, it may allow an attacker to steal private keys or generate transactions on a user's behalf.
- **Authorizing Malicious Transactions:** An attacker does not need access to a user's private key if the user will sign transactions generated by the attacker. Multiple hacks in the decentralized finance (DeFi) space involved users unwittingly signing transactions that allowed an attacker to steal their tokens.

PKC Security: Algorithmic Weaknesses

- Cryptographic algorithms are generally considered secure until proven otherwise
 - Algorithms in common use are trusted because no known attacks exist
- Asymmetric cryptographic algorithms may contain unknown weaknesses that might be discovered in the future



Most modern cryptographic algorithms are not provably secure. Instead, trust in them is based on the fact that these algorithms have undergone extensive cryptanalytic review and haven't been broken yet. For example, the Advanced Encryption Standard (AES) was selected as the result of a multi-year contest in which cryptographers tried to break the security of various candidate algorithms. In some cases, they even succeeded in breaking algorithms developed by professional cryptographers. AES is currently trusted because it's undergone years of review and the best-known attacks have negligible impact on its security.

The algorithms used in public-key cryptography are also trusted because there are no known attacks that are effective on modern technology. With classical (i.e. non-quantum) computing, the best-known way of solving the factoring or discrete logarithm problem is via a brute force search. The key lengths in use today make that infeasible on modern technology.

However, this is not to say that someone won't find a way to solve the factoring or discrete logarithm problem in polynomial time tomorrow. If this happens, any algorithm based on that problem is no longer secure, and blockchain systems relying on it for confidentiality, integrity, authentication, or non-repudiation are vulnerable to

attack.

PKC Security: Quantum Computing

- Asymmetric cryptography is built using mathematically “hard” problems
 - $F(x)$ has polynomial complexity
 - $F^{-1}(x)$ has exponential complexity
- Quantum computing breaks the “hardness” of some of these problems
 - Shor’s algorithm allows factoring in polynomial time
- Post-quantum cryptography uses different “hard” problems



The mathematically “hard” problems used in asymmetric cryptography are designed to have asymmetric complexity. Performing an operation has polynomial complexity while reversing it has exponential complexity. The security of the system relies on that relationship.

For some of the “hard” problems used in classical public-key cryptography, quantum computing algorithms have been developed that can solve them in polynomial time. For example, Shor’s algorithm can solve the factoring problem, breaking classical public-key cryptography.

At the moment, quantum computers large enough to run Shor’s algorithm against keys of the lengths in use today do not exist. However, the quantum computing space is advancing rapidly, and it is possible that, in the future, quantum computers will be able to break “classical” asymmetric encryption algorithms.

The factoring and discrete logarithm problems used in classical asymmetric cryptography are not the only mathematically “hard” problems in existence. Post-quantum cryptographic algorithms are designed to use problems that are believed to still be “hard” for quantum computers. By switching to these algorithms, blockchain

systems can continue to take advantage of the benefits of asymmetric cryptography.

However, these algorithms are only believed to be quantum-resistant, not proven to be. If new quantum algorithms are developed in the future that break the hardness of these problems, new “hard” problems will need to be discovered to continue using asymmetric cryptography.

Summary

- What is Public Key Cryptography?
- Benefits of Public Key Cryptography
- Blockchain Applications of Public Key Cryptography
- Security Threats to Public Key Cryptography



Module 2: Blockchain Cryptography

Section 2.3: Advanced Cryptographic Applications



Overview

- Introduction to Advanced Cryptography
- Multi-Signature Schemes
- Zero-Knowledge Proofs
- Stealth Addressing
- Ring Signatures



Introduction to Advanced Cryptography

- Cryptographic algorithms provide some of the essential properties and security guarantees of blockchain technology
 - Asymmetric cryptography
 - Hash functions
- Additional features can be included with the use of other cryptographic algorithms
 - Multi-signature schemes
 - Zero-knowledge proofs
 - Stealth addresses
 - Ring signatures



Cryptographic algorithms are the fundamental building blocks that make blockchain technology possible. Without the integrity protections of hash functions, and the confidentiality, authentication, and non-repudiation of public-key cryptography, it would be impossible to implement a distributed and decentralized digital ledger that operates without mutual trust among the nodes in the network.

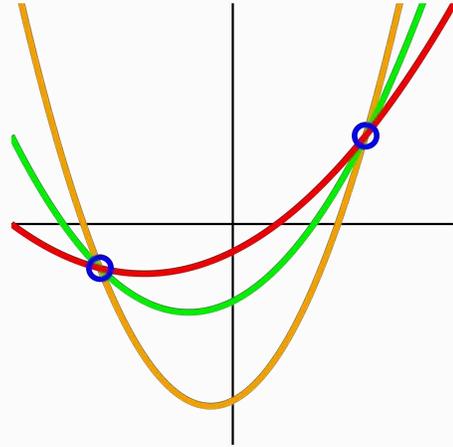
With these clear benefits of cryptography in blockchain, it makes sense that other cryptographic algorithms could add more features. Some of the ways in which some blockchains have enhanced the basic blockchain protocol using cryptography include:

- **Multi-Signature Schemes:** Digital signatures are used to indicate that a transaction by a blockchain account is approved. Multi-signature schemes require approvals by multiple parties for a transaction.
- **Zero-Knowledge Proofs:** Zero-knowledge proofs allow a party to prove knowledge of some secret without revealing the secret itself
- **Stealth Addresses:** Stealth addressing creates one-time addresses for receiving blockchain transactions

- **Ring Signatures:** Ring signatures allow digital signatures to be generated as a member of a group rather than an individual

Multi-Signature Schemes

- Anyone with a private key can generate a signature for an account
 - Risky for high-value accounts
- Multi-signature wallets require multiple inputs to generate a signature
 - Implemented algorithmically or via Shamir Secret Sharing (SSS)



Source: https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing#/media/File:3_polynomials_of_degree_2_through_2_points.svg

55 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

As blockchain systems and accounts grow more valuable, then the private key associated with an account becomes a dangerous single point of failure. If an account contains large amounts of cryptocurrency or has elevated permissions on a smart contract, an attacker may be highly motivated to steal this key. In the DeFi space, multiple hacks have already occurred against large projects that involved a compromised private key, and countless blockchain users have had private keys lost or stolen.

Multi-signature wallets attempt to address this issue by requiring multiple parties to authorize a transaction before it is performed. Often, this is implemented as an M of N scheme where any M of N key-holders can vote to take a certain action. While this may be a majority vote, M and N can be anything or can vary based on the desired application (such as requiring a super-majority for very high-risk actions).

Multi-signature schemes can be implemented in a couple of different ways:

- **Algorithmically:** Smart contracts or other systems can be coded so that a transaction must be digitally signed by M of N parties to be accepted and executed

- **Shamir Secret Sharing (SSS):** SSS encodes the secret as the y-intercept of a curve. In a scheme where M is 3, this would be a parabola like the ones shown in the image to the right. Then, each of the N users would be assigned a point on the curve before the equation of the actual curve is deleted. To regenerate the line and create a signature, at least M (3) parties would have to participate, since, as shown in the image, there are an infinite number of parabolas that pass through any two points.

The use of a multi-signature wallet is considered best practice for any high-value account, especially ones that are used to deploy and manage a valuable smart contract.

Zero-Knowledge Proofs

- Zero-knowledge proofs make it possible to prove knowledge of a secret without revealing that secret
- Valuable for blockchain systems where everything is public on the digital ledger



Source: <http://www.thefourishingbusiness.co.uk/wp-content/uploads/2014/01/Red-ball-green-ball-300x196.jpg>

56 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

A blockchain user may want to prove knowledge of some secret via the distributed ledger. This could be a password, a solution to a challenge, etc. However, if this information is posted on the blockchain's digital ledger, it will be visible to everyone in the network unless it is encrypted or otherwise protected. Even with encryption, proving knowledge of the secret requires revealing it to the verifier.

A zero-knowledge proof makes it possible to prove a secret without revealing it. For example, consider a case where someone wants to prove that two balls are different colors (red and green) to a colorblind person without revealing which ball is which color. This can be accomplished by the following series of steps:

1. The verifier (colorblind person) conceals both balls from the prover
2. The verifier shows the prover one ball and conceals it again
3. The verifier shows the prover one ball
4. The prover states whether the two balls were the same ball

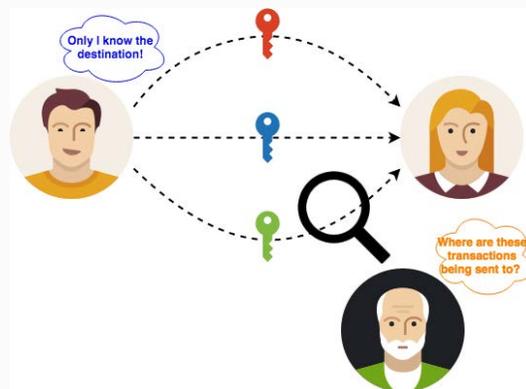
If the two balls are the same color, then the prover has a 50-50 chance of guessing correctly. However, if the two balls are different colors, then the prover should answer correctly every time. By repeating the test many times, the verifier can reduce the probability of lucky guesses to the point of statistical impossibility, proving that the balls are different colors. However, at the end of the process, the verifier has no idea which ball is which.

The security of this zero-knowledge proof relies on the fact that the two balls are identical except for color. If some other differentiator exists (size, dirt, etc.), then the prover could generate a false proof.

Blockchain systems use zero-knowledge proofs based on mathematical and cryptographic principles. However, the goal is the same: making it statistically impossible to generate a false proof.

Stealth Addressing

- Blockchains rely on decentralized validation
 - Each node checks transactions and blocks before accepting them
 - To enable this, a validator needs to see the source, destination, and value of a transaction
- Stealth addressing creates single-use destination addresses for transactions



Source: <https://hackernoon.com/blockchain-privacy-enhancing-technology-series-stealth-address-i-c8a3eb4e4e43>

57 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The blockchain's digital ledger has a high degree of transparency by necessity. Each node in the blockchain network is responsible for validating any transactions that it includes in its version of the blockchain's digital ledger. This includes being able to see the source of a transaction to ensure that the account has the required balance, permissions, etc. to generate a transaction. Account balances come from earlier transactions, making it necessary for transaction destinations to also be public on the blockchain.

The downside of this visibility is that little privacy exists on the blockchain. It is possible to see the value and transaction history of every account on the digital ledger.

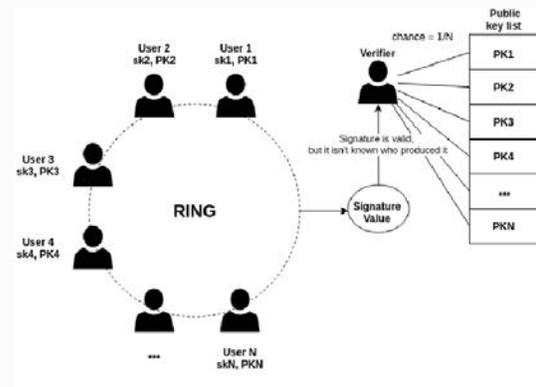
Stealth addressing provides improved privacy by enabling a user to create one-time addresses to receive transactions. A user can publish a specially-formatted address that can be used to generate a one-time address and some tag information for a transaction. Using this data, a sender can generate a transaction to the user, and the recipient can identify and claim these transactions on the blockchain (i.e. generate digital signatures for the destination accounts).

Stealth addressing improves privacy because these one-time addresses can't be

linked to a user's account, and an observer can't determine that two transactions went to the same person. This defeats many of the blockchain analytics that degrade privacy and anonymity on the blockchain.

Ring Signatures

- Ring signatures enable a signature to be generated as a member of a group
 - “Someone in the White House”, etc.
- They can be used in blockchain to provide anonymity for transaction senders



Source: <https://www.semanticscholar.org/paper/Using-Ring-Signatures-For-An-Anonymous-E-Voting-Kurbatov-Kravchenko/a2c5c700f3f06cfe6bc633b1fe73f7916bab3435/figure/2>

58 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Blockchain systems use digital signatures to protect the integrity, authenticity, and non-repudiation of transactions on the digital ledger. Only someone with knowledge of the appropriate private key can generate a valid digital signature for a blockchain account. This makes it possible to determine if a transaction is valid and authorized under the assumption that only legitimate users of an account have access to its private key.

Like transaction recipients, transaction senders are public on the blockchain to enable transaction validation. However, this also has privacy impacts for blockchain users and degrades anonymity.

Ring signatures provide privacy protections for transaction senders by enabling digital signatures to be generated as a member of a group. The digital signature algorithm takes a set of public keys and requires one private key as input. The resulting signature can then be proved to have been generated by the private key associated with one of those public keys, but it is not possible to determine which of them signed the transaction.

Summary

- Introduction to Advanced Cryptography
- Multi-Signature Schemes
- Zero-Knowledge Proofs
- Stealth Addressing
- Ring Signatures



Module 3

Blockchain Consensus Security



Module 3: Blockchain Consensus Security

Section 3.1: Introduction to Consensus



Overview

- Secure Blockchain Consensus
- Introducing the Byzantine Generals Problem
- Solving BGP with Security via Scarcity
- Major Blockchain Consensus Algorithms



Secure Blockchain Consensus

- Consensus algorithms are vital for blockchain synchronization
 - Block creation and ledger updates are a decentralized process
 - Need all nodes to agree on “official” version of the ledger
- Majority vote is ideal for decentralization, but it has issues
 - Fake accounts
 - Malicious nodes



Blockchains are designed to be decentralized systems with no authority responsible for maintaining and distributing the “official” version of the blockchain ledger. As a result, the blockchain network needs a means of defining the “official” version of the ledger via a distributed and decentralized process.

Blockchain consensus algorithms are designed to accomplish this goal. To do so, they must achieve two tasks:

- 1. Selecting a Block Creator:** Blocks in the blockchain need to be created by someone, and this person has significant control over the state of the ledger. In a decentralized system, the process of creating blocks should be decentralized to prevent anyone from having too much influence on the history of the network.
- 2. Achieving Consensus:** After a block is created, it needs to be distributed and every node in the network needs to agree on the current version of the ledger. This is especially important if multiple conflicting versions of a block can be created.

The simplest and fairest consensus system would be round-robin block creation with

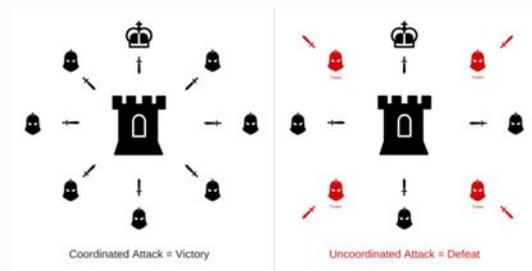
a majority vote for consensus. However, this won't work for the blockchain. Two of the main reasons for this are:

- **Fake Accounts:** Blockchain accounts are anonymous, so a person could create multiple different accounts. An attacker could use this to exploit the block creation and consensus algorithms to have undue influence over the digital ledger.
- **Malicious Nodes:** The blockchain uses a decentralized, peer-to-peer network for communications. Malicious nodes on the network may pass on fake information to influence voting, such as lying about nodes' votes in a majority vote system. This could trick nodes in accepting a fake version of the digital ledger.

Modern blockchain consensus algorithms are designed to overcome these challenges.

Introducing the Byzantine Generals Problem

- Blockchain consensus is designed to solve the BGP
- Armies must agree on a solution to a problem or risk defeat
- Generals can only communicate via messengers
- Some generals are traitors
 - Intercept messages
 - Lie about their contents



Source: <https://www.thewolffallstreets.io/bitcoin-and-the-byzantine-generals-problem/>

64 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The challenge that blockchain consensus algorithms are designed to solve predates the blockchain. The Byzantine Generals Problem is a thought experiment created by Leslie Lamport that defines this problem.

The Byzantine Generals Problem describes a scenario in which several armies are besieging a city and need to agree on a plan of attack or retreat. This decision must be coordinated because an uncoordinated attack would result in the besieging army being defeated by the defenders.

The problem is complicated by the fact that all of the generals must stay with their armies, so they must communicate via messengers. Also, some of the generals may be traitors, meaning that they could lie about their votes or the contents of messages from other generals that they relay onward. Despite this, the besiegers must not only make a decision but also have every general be confident about what the final decision is.

This problem is the same problem that blockchain nodes must solve via consensus. The blockchain network needs to make a decision of what the current version of the ledger is. They are connected only via a peer-to-peer network, so all information is

relayed via networks. Also, there is the potential that some percentage of the nodes are malicious, so the consensus algorithm must be robust enough that the group makes the right decision even if some participants try to mislead others to their own benefit.

Solving BGP with Security via Scarcity

- Blockchain consensus algorithms are designed to solve the BGP
 - Older solutions to the problem were not efficient and scalable enough
- The official version of the blockchain is selected based on majority vote
 - A scarce resource is used to represent voting power in consensus
 - This protects against Sybil attacks
- Using scarce resources makes attacking consensus expensive
 - Scarce resources are governed by laws of supply and demand
 - Buying up votes raises the price



Blockchain consensus algorithms like Proof of Work are designed to solve the Byzantine General's Problem in an efficient and scalable fashion. While solutions to the BGP predate blockchain and Proof of Work, they often lacked the scalability and efficiency required by a blockchain network.

Blockchain consensus algorithms are designed to use majority vote to decide on the "official" version of the blockchain that the network will accept. However, this is not performed using a "one account one vote" scheme. As mentioned previously, most blockchains allow anonymous accounts, so a malicious party could create many fake accounts to subvert the vote in their favor (a Sybil attack).

Instead, blockchain consensus algorithms use a scarce resource to represent votes in consensus algorithms. For example, the Proof of Work consensus algorithm uses computational power or "hashrate" as its scarce resource. The more of the scarce resource that a party controls, the more "votes" that they have regarding the state of the ledger. In practice, this translates to nodes producing blocks at a rate roughly proportional to the percentage of the scarce resource that they control. Since block producers select and organize the transactions included in blocks, this provides a significant amount of control over the state of the blockchain's ledger.

The use of scarce resources to represent voting power in blockchain consensus algorithms make attacking the vote by collecting a large amount of the scarce resource expensive. These scarce resources are governed by the laws of supply and demand, which state that increased demand for an asset with a fixed supply will cause prices to increase. This has occurred in real life as demand for GPUs by both gamers and Proof of Work miners has driven up the price.

Since increased demand drives up the price of scarce resources, an attacker attempting to control the majority vote (a 51% attack) will have to pay a significant amount of money to do so. This protects the security of the blockchain by making an attack unprofitable or too expensive to perform.

Major Blockchain Consensus Algorithms

- Bitcoin uses Proof of Work (PoW) consensus
 - Uses computational power as a scarce resource
- Other consensus algorithms have been developed to address the limitations of PoW
 - Proof of Stake
 - Proof of Capacity
 - Proof of Burn
 - Proof of Elapsed Time



Proof of Work is the original blockchain consensus algorithm described in the Bitcoin whitepaper and uses hashpower as its scarce resource. However, this consensus algorithm has its drawbacks and limitations. For example, a common criticism of Proof of Work is that it uses a great deal of energy for computational operations that do nothing but secure the blockchain. In March 2022, the Bitcoin network alone would rank 23rd among countries ranked by energy consumption.

Proof of Work's energy consumption and other limitations have prompted the creation of various alternative consensus algorithms. Some prominent examples and their scarce resources include:

- Proof of Stake (cryptocurrency)
- Proof of Capacity (computer memory)
- Proof of Burn (cryptocurrency)
- Proof of Elapsed Time (time)

Summary

- Secure Blockchain Consensus
- Introducing the Byzantine Generals Problem
- Solving BGP with Security via Scarcity
- Major Blockchain Consensus Algorithms



Module 3: Blockchain Consensus Security

Section 3.2: Securing Proof of Work



Overview

- What Makes a PoW Block Valid?
- The PoW Solution to the BGP
- Proof of Work Security Assumptions
- Proof of Work Consensus Attacks
 - The 51% Attack
 - Denial of Service Attacks
 - Selfish Mining
 - SPV Mining



What Makes a PoW Block Valid?

- In Proof of Work, a block is valid if:
 - All of the transactions that it contains are valid
 - Its header hashes to a value less than a certain threshold
- Finding a valid block requires finding a valid header
 - Miners can set the “nonce” value in a block header to anything
 - Different nonces produce very different header hashes due to hash function non-locality
 - The miner that finds a valid nonce creates a block



Proof of Work is a rare consensus algorithm in which the creator of the next block is selected after a block is created. In Proof of Work, the next block producer is selected based on their ability to find a valid version of the next block in the blockchain.

The reason for this is that Proof of Work is designed to make producing a valid block difficult and computationally expensive. The two main validity criteria for a valid block in Proof of Work are:

- **Valid Contents:** In any blockchain network, a block will only be accepted by legitimate nodes if the transactions that it contains are valid. If a block contains a transaction spending cryptocurrency that an account doesn't have or includes a “double spent” transaction that spends the same cryptocurrency as an earlier transaction, then the block is invalid.
- **Header Hash:** A criterion unique to Proof of Work blockchains is that the hash of a block's header must be below a certain threshold. This criterion is what makes creating a valid block difficult and computationally expensive.

The reason why finding a valid header hash is computationally expensive is because

the best way to do so (assuming that the blockchain is using a secure hash algorithm) is via a brute force search (i.e. “guess and check”). Proof of Work miners will take the following steps:

1. Create a block that contains valid transactions, timestamp, etc.
2. Select a nonce value in the block header
3. Calculate the hash of the header and compare to threshold
4. Repeat steps 2 and 3 until a header hash is below the threshold

Due to hash function non-locality, similar nonces produce very different header hashes. This is what makes Proof of Work possible because changing even a small part of the header (the nonce) changes its entire hash. It also makes Proof of Work mining difficult because it is impossible to predict which nonce values will produce valid headers.

If a node finds a valid block, they can submit it to the rest of the network.

Once the other nodes validate the block, they will work on building a new block on top of it, starting the process all over again. In the event of two competing versions of the blockchain, whichever one was harder to find wins. This is calculated based on the header hash. In theory, it takes more hash calculations to find a smaller header hash, the blockchain with smaller overall hashes and greater theoretical complexity is given preference. This is called the longest chain rule.

The PoW Solution to the BGP

- Proof of Work provides probabilistic consensus
 - The longest version of the blockchain is likely supported by the majority
 - This is a probabilistic solution to the BGP



The Proof of Work consensus algorithm provides a probabilistic solution to the Byzantine Generals Problem

In theory, every miner in a Proof of Work network will eventually create a valid block accepted by the network. When they do so, they explicitly endorse the history of the blockchain to that point and implicitly endorse the system. This means that, if the network only has one version of the blockchain, then every node in the network will have endorsed it at some point. Therefore, the nodes are in consensus about the state of the blockchain.

In the case of divergent versions of the blockchain, the assumption is that whichever version of the blockchain is accepted by the network under the longest chain rule is also supported by the majority of blockchain nodes. The rationale behind this is that creating a valid block requires a certain amount of computational power. In the case of two competing versions of the blockchain, the one that wins under the longest chain rule is the one that theoretically required more computational power to create. Therefore, the winning version theoretically has majority support as demonstrated by the fact that a greater percentage of the blockchain network's hashrate was used to create it. While a minority might get lucky and create a "longer" version of the

blockchain than the majority for a short period, the probability of sustaining this success diminishes quickly over time.

Proof of Work Security Assumptions

- Proof of Work security is based on the security of the hash algorithm
 - Needs to be hard to find valid block headers
 - Can't predict the nonce that will produce a valid header hash
- A brute force search must be the only way to find valid nonces
 - Scarce asset equates to control over the blockchain
 - Must own most of the scarce resource to control the blockchain
- Difficulty targets and the longest chain rule are tied to hash function security



The security of Proof of Work depends on the assumption that the probability of creating a valid block is proportional to the amount of hashrate controlled by a node or group of nodes. This boils down to a few assumptions:

- **Hash Function Security:** The relationship between computational power and the ability to create valid blocks comes from the fact that the only way to find a hash less than the target difficulty threshold is via a brute force search. In a brute force search, a node with greater hashpower can make more guesses in a certain period of time, which increases their probability of success. If a hash function is broken in a way that allows a node to find valid blocks without performing a brute force search, then Proof of Work security is compromised.
- **Difficulty Target:** Proof of Work has a difficulty target, which defines the threshold that header hashes must be under for a block to be considered valid. This target is updated regularly and is designed so that it takes an average of the target block interval (ten minutes for Bitcoin) to find a valid block. If this threshold is set too high, it becomes too easy to find valid blocks. If it is too low, blocks will be created too slowly, which impairs the operation of the network.

- **Longest Chain Rule:** The longest chain rule states that, given two competing versions of the blockchain, the one that was “harder” to create should win. If hash function security breaks, then the longest chain rule will no longer function. Similarly, if nodes ignore the longest chain rule, then Proof of Work’s majority vote scheme no longer works.

Proof of Work Consensus Attacks

- Proof of Work can be attacked in various ways, including:
 - 51% Attack
 - Denial of Service Attack
 - Selfish Mining
 - SPV Mining



The Proof of Work consensus algorithm does a good job of protecting the blockchain against attack as demonstrated by the enduring success and security of the Bitcoin blockchain. However, while Proof of Work has not been found to be fundamentally broken, it can be attacked in a few ways, including:

- 51% Attack
- Denial of Service Attack
- Selfish Mining
- SPV Mining

The 51% Attack

74 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to The 51% Attack

- Proof of Work implements a majority vote
 - The majority of the hashpower determines the winning version of the blockchain
- A 51% attacker controls the majority of the blockchain's hashpower
 - Can create a valid blockchain faster than the rest of the network
 - Can rewrite the history of the blockchain at will



The 51% attack is the oldest and best-known attack against the Proof of Work consensus algorithm. It was described in the Bitcoin whitepaper where Proof of Work was first formally described.

A 51% attack is not a weakness in Proof of Work but arises from how the algorithm is designed to work. Proof of Work and the longest chain rule implement a majority voting scheme to select the “official” version of the blockchain’s digital ledger. This voting scheme uses hashpower to represent votes (protecting against Sybil attacks).

A 51% attack is when an attacker controls the majority (i.e. 51%) of the blockchain network’s hashpower. If this is the case, the attacker can legitimately win any vote on the official version of the blockchain’s ledger because they control more votes than anyone else.

In practice, this voting process is performed via the search for valid blocks to add to the blockchain. As mentioned previously, finding valid blocks involves finding headers whose hashes are less than a certain threshold, and greater computational power provides a high probability of accomplishing this. If an attacker controls as much hashpower as the rest of the network put together, they can build an alternative

version of the blockchain faster than the rest of the network, enabling it to be accepted under the longest chain rule.

A 51% attack is dangerous because it allows the history of the blockchain network to be rewritten by the attacker at any time. An attacker can remove a block or transaction from the ledger simply by finding versions of that block and all following. Since the attacker can find blocks faster than the rest of the network, they can catch up and build a longer chain that replaces the main chain. This enables double spend attacks because an attacker can perform one transaction and later replace the block containing it with one that contains a different transaction that spends the same cryptocurrency. Since both versions won't be accepted by a legitimate node, only the version that is on the currently accepted version of the blockchain is "real".

Mitigating 51% Attacks

- 51% attacks are a built-in threat to Proof of Work blockchains
 - Arise from PoW and the longest chain rule
- 51% attacks can be mitigated
 - Checkpointing, etc.
- However, these mitigations have tradeoffs
 - Centralization, potential permanent divergences



The 51% attack was referenced in the Bitcoin whitepaper because it is a central part of how Bitcoin’s consensus algorithm works. The purpose of Proof of Work and the longest chain rule are to ensure that the accepted version of the blockchain is the one that is supported by the majority of the network’s hashpower. A 51% attack occurs when malicious parties control this majority of hashpower and thus can fairly win the vote.

The threat of 51% attacks can be mitigated. One common suggestion is to use a checkpointing scheme where blockchain software is configured to only accept a certain version of a particular block in the blockchain. If an attacker attempts to perform a 51% attack that replaces that block, then a node would reject the new version even if it would otherwise win under the longest chain rule.

While this approach to mitigating the threat of 51% attacks could work, it also comes with its tradeoffs. To have checkpoints, someone needs to choose the “official” version of the checkpointed block. As a result, implementing a checkpointing scheme runs the risk of one of the following:

- **Centralization:** One approach to checkpointing is having a centralized authority

select the checkpoint for a particular block. For example, the creator of a particular version of the blockchain software could write the checkpoint into the code. However, this breaks blockchain decentralization because it gives a centralized authority complete control over the blockchain and the ability to rewrite history at will simply by changing the checkpoint so that the current accepted version of the blockchain is rejected by nodes.

- **Permanent Splits:** Another approach to checkpointing is to have each node set its own checkpoint independently, perhaps by automatically rejecting alternative versions of the blockchain that change a certain number of blocks. However, this approach runs the risk that different nodes may create conflicting versions of the same checkpoint. For example, if a node won't accept reorgs ten blocks deep, then an attacker can split the network into two parts and have them create different versions of the blockchain for ten blocks will force a permanent divergence. Even after reconnecting, nodes in the two parts will reject each others' versions of the blockchain because they don't match their version of the checkpoint.

Denial of Service Attacks



Introduction to Denial of Service Attacks

- Proof of Work difficulty targets are tailored to the current hashrate of the blockchain
 - Designed to create blocks at the target block rate
- A Denial of Service (DoS) attack that drops the blockchain's hashpower can result in a DoS attack on the blockchain
 - Slows creation of new blocks on the blockchain



Denial of Service (DoS) attacks can occur at most levels of the blockchain ecosystem. The Proof of Work consensus algorithm is vulnerable to DoS attacks that decrease the overall hashpower of the blockchain network.

Proof of Work blockchains have a difficulty target that determines how many header candidates must be tested on average to find a valid block header. This difficulty target is designed to cause blocks to be created at the target rate (10 minutes for Bitcoin, etc.). This is accomplished by setting the difficulty target so that the network, with its current hashpower, can search enough header candidates to find one block each interval.

If an attacker targeted nodes with high hashpower with a DoS attack, they could affect the rate at which blocks are generated on the blockchain. Without these nodes, the network would have lower hashpower, meaning that it would take longer to test the number of block header candidates needed to find a valid block with the current difficulty target. By slowing block creation, the attacker decreases the throughput of the blockchain network and the rate at which transactions can be added to the ledger.

Mitigating Denial of Service Attacks

- Differentiating between a DoS-driven drop in hashpower and a benign one can be difficult
 - Nodes enter and leave the network all the time
- Proof of Work blockchains are designed to update difficulty targets at regular intervals
 - A sustained attack will cause the network to adjust difficulty
 - This happens more slowly if an attack is occurring



Determining whether a Proof of Work network is under a Denial of Service (DoS) attack designed to slow block creation is difficult. The hashpower of blockchain networks changes frequently as nodes enter and leave the network based on the profitability of mining and other factors. Differentiating between an economically-driven exodus and one initiated by an attacker can be difficult.

The Proof of Work blockchain has built-in mechanisms for handling changes in difficulty, and the difficulty target is updated at regular intervals. If an attack is sustained, then it will be reflected and accounted for in the next difficulty update.

However, these changes occur rarely and are based on the number of blocks created (once every 2016 blocks for Bitcoin). As a result, a sustained loss of hashpower will delay the difficulty threshold update.

Selfish Mining

80 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Selfish Mining

- Block producers are expected to immediately publish a block that they find
 - Ensures that someone else doesn't beat them to it
 - Enables the network to move on to the next block
- But there is no obligation to do so or way to enforce such a rule
- A block producer can delay revealing a discovered block
 - Gives them a head start on finding the next block



In theory, a 51% attack requires the attacker to control over half of a blockchain's hashpower to carry out. In practice, various ways exist for an attacker to cheat the system and achieve the same result with slightly less hashpower.

One example of this is selfish mining, which exploits the fact that, statistically, only one version of a block should be found within a block interval if the difficulty target is appropriately set.

Each block in the blockchain contains the hash of the previous block header within its block header, linking the blocks of the blockchain together. This means that a Proof of Work miner can't start searching for block N+1 until a valid version of block N is created. In theory, a Proof of Work node will work on block N until it receives a valid version of it at which point it will abandon its work on block N and move on to trying to find a valid version of block N+1.

A selfish miner takes advantage of this fact by tricking other nodes into wasting their time and effort looking for a valid version of a block that already exists. After finding a valid version of block N, a selfish miner will not immediately publish it to the rest of the blockchain network. Instead, they immediately start work on mining block N+1

when no-one else can because they lack the header hash of block N that must be included in block N+1's header.

With the head start provided by not immediately publishing the block, the attacker can check several candidate block headers before anyone else starts the process. This means that, even with less than 50% of the blockchain network's hashpower, they might still be able to find a valid block faster than the rest of the network and perform a 51% attack. Even if their goal was not a 51% attack, the head start provided by a selfish mining attack effectively increases a node's hashpower and probability of finding a valid block. This translates into a higher probability of earning block rewards, meaning that the attacker can earn a profit from their attack.

Mitigating Selfish Mining Attacks

- Selfish miners are taking a risk
 - Another miner could find a competing version of their block
 - Other nodes may build on this version if its published first
- These risks limit the potential impact and profitability of selfish mining attacks
 - Waiting too long increases the probability of competing blocks
 - Miners can only achieve relatively small gains in effective hashrate



Selfish miners delay publishing a valid version of a block so that they can get a head start on mining the next block. The rationale is that the probability of another miner finding another valid version of the block soon after is statistically unlikely.

The main risk is that, by delaying the publishing of their block, another node could find a valid block. In Proof of Work, it is only statistically unlikely that multiple versions of a block will be found close together, but it can and does happen. If this alternative version of the block is published first and the rest of the network builds on it, the attacker may lose under the longest chain rule and lose the block rewards earned from creating block N.

This risk limits the potential impact of a selfish mining attack because a node that waits too long to publish its original block can lose the reward for that block in its attempt to increase the probability of finding the next block in the chain. As a result, selfish mining attacks can only marginally increase a node's effective hashpower and thus their probability of finding the next block in the chain as well.

SPV Mining



Introduction to SPV Mining

- Blockchain nodes will only accept valid blocks
 - No double-spent or invalid transactions
- Validating the transactions to be included in a block is time-consuming
 - Need to download previous blocks' bodies and validate all transactions before including them
- SPV miners skip this process by creating near-empty blocks
 - Only include block reward transactions in blocks
 - Provides a head start on mining that increases the probability of finding a block



SPV mining is another way in which a node can increase its probability of finding a valid block in a Proof of Work scheme and its earned rewards. SPV mining gets its name from Simplified Payment Verification (SPV) nodes, which are a “light” form of blockchain node. Instead of maintaining a complete copy of the blockchain, these nodes only track the chain of linked block headers. They use Merkle Proofs to verify that a particular transaction was contained within a given block. This combination allows them to verify that a user’s transaction was included on the blockchain without the complexity and resource requirements of tracking a complete copy of the blockchain.

SPV mining uses SPV nodes to get a head start similar to with selfish mining. As mentioned previously, blocks are only considered valid if they don’t contain double-spent or mutually conflicting transactions. To verify that a block that they are creating is valid, a node needs to download the body of the previous block, update its pool of unused transactions, and verify that each transaction that it includes in the block is valid, not double-spent, and not already included in an earlier block. This process takes time and must be completed before a node can test its first candidate nonce value for the new block.

SPV miners bypass this process by only including transactions in a block that are guaranteed not to be double-spent, which is the coinbase transaction that pays the miner the reward for creating the block. By creating an empty block, an SPV miner can skip the process of downloading block bodies and validating transactions. This provides a significant head start that can improve the probability of creating blocks, which can make a 51% attack possible with <50% of the network's hashpower and increases the probability of earning block rewards.

Mitigating SPV Mining Attacks

- SPV miners aren't breaking any rules
 - Block producers have full control over the contents of their blocks
 - Only need to contain valid transactions
- SPV mining is a tradeoff
 - Increased probability of block rewards
 - Giving up transaction fees
- As block rewards decrease over time, SPV mining becomes unprofitable



SPV miners are playing by the rules of a blockchain system. Block producers have full control over the contents of the block that they produce as long as they don't include double-spent or otherwise invalid transactions. If a miner wants to create empty blocks, that is their decision.

Many blockchains include transaction fees that are paid to miners for including a transaction within the blocks that they produce. If a block producer creates only empty blocks, then they are giving up any potential transactions fees that they could have earned. Transaction fees are the built-in incentive for block producers to make their blocks as full as possible, maximizing the throughput of the blockchain network.

SPV miners make a tradeoff where they give up transaction fees in exchange for a higher probability of creating blocks due to the head start that they get from skipping the process of validating transactions to put in their blocks. Since block rewards decrease over time, this tradeoff will not be profitable forever, and SPV miners will earn less than they would if they created full blocks. For this reason, no real mitigation is necessary since blockchain's built-in incentives will eventually correct the issue.

Summary

- What Makes a PoW Block Valid?
- The PoW Solution to the BGP
- Proof of Work Security Assumptions
- Proof of Work Consensus Attacks
 - The 51% Attack
 - Denial of Service Attacks
 - Selfish Mining
 - SPV Mining



Module 3: Blockchain Consensus Security

Section 3.3: Securing Proof of Stake



Overview

- How Proof of Stake Works
- The PoS Solution to the BGP
- Variations on Proof of Stake
- Proof of Stake Security Assumptions
- Attacking Proof of Stake
 - XX% Attack
 - The Proof of Stake “Timebomb”
 - Long-Range Attacks
 - The Nothing at Stake Problem
 - Resource Exhaustion Attacks



How Proof of Stake Works

- Proof of Stake uses its native cryptocurrency as its scarce resource
- In Proof of Stake, nodes will “stake” cryptocurrency to participate in consensus
 - Send cryptocurrency to an address where it is locked and cannot be spent
- In return, they have a chance to be selected for block creation
 - Proportional to the percentage of stake that they control



Proof of Stake was developed as an alternative to the Proof of Work consensus algorithm. It is far less resource-intensive and makes creating blocks “easy” for the legitimate block producer. This makes it possible to have very fast block rates in Proof of Stake without making block production too easy.

Instead of using hashpower as its scarce resource, Proof of Stake uses its native cryptocurrency. For example, Ethereum is making a transition to a Proof of Stake consensus scheme, so the fact that the total supply of Ether is capped creates a similar economic environment as Proof of Work.

In a Proof of Stake scheme, nodes can stake some of their cryptocurrency by sending it to a special address. By doing so, they trade the ability to use this cryptocurrency for the ability to participate in block production and to earn block rewards.

In Proof of Work, a node’s chances of creating a block are proportional to the percentage of the network’s hashpower that it controls. In Proof of Stake, a node’s probability of being selected to create a block is proportional to the percentage of stake that it controls. Different Proof of Stake algorithms implement this differently, but the ratios should match in the long term.

The PoS Solution to the BGP

- All Proof of Stake stakers should eventually be selected to create a block
- All blocks are added to an existing chain
 - Validate the history of the network to-date
 - Approve the overall system
- If all nodes create a block, they have all affirmed the system



In Proof of Stake, nodes' probability of being selected to create blocks is proportional to the percentage of stake that they control. As a result, some stakers may create blocks regularly, while others may very rarely have the opportunity to produce a block. However, in the long run, every staker should be selected to create blocks.

When a node creates a new block, it builds it on top of the existing blockchain. This relationship is explicitly encoded by the previous block hash within the block header, which indicates the previous block in the chain. By creating a new block on an existing chain, a node states that it has validated the chain and believes it to be correct and the official version of the blockchain network's ledger.

By following the rules of block production, a node also validates these rules and the blockchain system. By stating that it approves of the process of selecting block producers, it implicitly endorses future block producer's rights to create blocks and the blocks that they create as long as they are otherwise valid (no double-spends, etc.).

In the long term, every staker should create a block, explicitly validating the past history of the network and implicitly approving future blocks. Therefore, a sufficiently

old Proof of Stake blockchain implicitly has consensus across the network.

Variations on Proof of Stake

- Proof of Stake is a general framework for implementing consensus
 - Staking cryptocurrency for a chance to create blocks
- Different implementations of Proof of Stake exist
 - Coin age-based PoS is an example
- Protocols like Delegated PoS are also based on Proof of Stake
 - Stake is used to vote for delegates who do the work of block production



Proof of Stake describes a scheme in which blockchain users can stake cryptocurrency in exchange for the right to participate in consensus and potentially be selected to create blocks and earn rewards. However, Proof of Stake does not define an exact mechanism for doing so, so various versions of Proof of Stake.

Some variations on Proof of Stake use the original algorithm with tweaks to make it “fairer”. For example, coin age-based Proof of Stake takes into account the time since a staker has been selected as a block producer when selecting the creator of the next block. Stakers who haven’t been selected in a while have a higher chance of being selected than the exact ratio of stake that they control, while ones that have created a block recently have a reduced probability. However, in the long run, the percentage of blocks created by a staker should be roughly proportional to the percentage of stake that they control.

Delegated Proof of Stake is a consensus algorithm that applies the fundamental ideas of Proof of Stake in a different way. Instead of stake being used to directly select block producers, DPoS stakers use their stake to vote for delegates. A certain number of delegates with the most votes are selected to do the work of block production and validation. They then pass on a portion of block rewards to stakers.

Proof of Stake Security Assumptions

- Proof of Stake blockchains assume that:
 - The block producer selection process is secure
 - Digital signatures are secure
- These security assumptions depend on:
 - Hash function security
 - Public key cryptography security
 - Private key security
 - Algorithmic security



In Proof of Stake, creating a block is relatively easy. Once a block producer has been selected, they can collect a set of valid transactions and build a block header. There is no header hash requirement like in Proof of Work, so the computational complexity of block production is minimal.

As a result, the security of Proof of Stake blockchains depends on the correct user creating each block in the chain. This assumes that an attacker can't game the block producer selection process and that an attacker can't masquerade as the legitimate block producer by forging a digital signature.

The security assumptions of Proof of Stake depend on the security of:

- **Hash Functions:** Hash functions are often a part of the block producer selection process since they are deterministic but also pseudorandom and unpredictable. If hash function security is broken, an attacker may be able to game the system by predicting how changes in stake affects their probability of being selected as a block producer.
- **Public Key Cryptography:** Public key cryptography (along with hash functions)

make digital signatures possible. If the digital signature algorithm used by a Proof of Stake blockchain is broken, then an attacker may be able to create a fake block while masquerading as the legitimate block producer.

- **Private Key Security:** Anyone with an account's private key can generate a digital signature as that account. If an attacker has access to the private key of the legitimate block producer, they can create blocks and steal block rewards.
- **Algorithmic Security:** Proof of Stake uses various algorithms and protocols to implement its consensus algorithm. If these algorithms are insecure or incorrect, an attacker may be able to exploit or manipulate them to cheat and increase their rate of block production.

Attacking Proof of Stake

- Like Proof of Work, Proof of Stake can be attacked in a few different ways:
 - XX% Attack
 - Proof of Stake “Timebomb”
 - Long-Range Attacks
 - The Nothing at Stake Problem
 - Resource Exhaustion Attacks



Proof of Stake can be implemented in various different ways, which each have their advantages and disadvantages. Some threats to the security of Proof of Stake algorithms apply generally, while others are specific to a particular implementation. For example, Delegated Proof of Stake (DPoS) implementations sometimes have a set order in which delegates create blocks, which eliminate concerns about the predictability of the block producer selection process.

Some potential attack vectors and security risks of the Proof of Stake consensus algorithm include:

- XX% Attack
- Proof of Stake “Timebomb”
- Long-Range Attacks
- The Nothing at Stake Problem
- Resource Exhaustion Attacks

The XX% Attack

94 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to the XX% Attack

- Proof of Work has a 51% attack
 - Control over half of the network's hashpower provides complete control over the blockchain
- In Proof of Stake, controlling half of staked cryptocurrency provides a 50% chance of being selected to create blocks
 - Not the same level of control
- The percentage of stake needed to perform an attack depends on the number of blocks to be changed and the desired probability of success



The Proof of Work consensus algorithm has a sudden breakdown of security when an attacker controls at least 50% of the network's hashpower. With more computational power than the rest of the network put together, an attacker can find valid blocks faster than the rest of the network. Under the longest chain rule, this provides the ability to replace the commonly-accepted version of the blockchain at will.

In Proof of Stake, control over half of the staked cryptocurrency does not provide the same level of control. In theory, a staker's probability of being selected to create a block is proportional to the percentage of stake that they control. Control over half of the staked assets equates to a 50% probability of being selected to create a given block.

Achieving the same level of control as a Proof of Work 51% attack requires control over 100% of the staked assets, which is unlikely. However, a less sustained attack with a certain probability of success can be achieved with a lower proportion of the staked cryptocurrency.

For example, consider an attacker that wants to be selected as the block creator for three consecutive blocks as part of a double-spend attack. With 50% of the blockchain's staked assets, the probability of accomplishing this is $(1/2)^3 = 1/8 =$

.125. With 75% of the staked assets, the probability of success grows to about 42%. If an attacker accepts a certain probability of failure and only wants to replace a few blocks, it is possible to do so without full control over the staked cryptocurrency.

Mitigating the XX% Attack

- Performing an XX% attack with a reasonable probability of success requires control over a significant percentage of a blockchain's staked cryptocurrency
 - A 50% chance requires about 80% control
- Performing such an attack is likely unprofitable
- Users can also refuse to sell cryptocurrency or stake more to impede an attacker



If an attacker's probability of being selected to create a given block is proportional to the percentage of stake that they control, then a XX% attack will be difficult and expensive to perform. An attacker needs to control a significant percentage of the blockchain's staked assets to have a reasonable chance of being selected to create a series of three blocks. Additionally, the attacker would need to perform the transaction that they wish to be overwritten in the first of these three blocks with no guarantee that later blocks will be assigned to them.

For an XX% attack to be worthwhile, the blockchain's cryptocurrency needs to hold some value. However, this also means that collecting enough of the cryptocurrency to perform an attack will likely be expensive. As a result, the cost of performing the attack may outweigh the benefit that an attacker receives from a double spend.

Users can also take an active role in mitigating the threat of an XX% attack. To buy up cryptocurrency, an attacker needs to find users willing to sell. Additionally, users can drive up the cost of an attack by staking additional cryptocurrency, increasing the amount of cryptocurrency that the attacker needs to collect and stake to perform their attack.

The PoS “Timebomb”

97 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to the PoS “Timebomb”

- Proof of Stake uses cryptocurrency as both a scarce resource and a reward
 - Nodes stake cryptocurrency to participate in block production
 - Block production earns block rewards and transaction fees
- Stakers earn rewards proportional to the percent of stake they hold
 - The node with the highest stake earns the most rewards
- Over time, the rich get richer and control a higher percentage of stake



In Proof of Stake consensus, the blockchain uses its native cryptocurrency both as a scarce asset for consensus and to reward block producers. In order to participate in consensus and have the chance to create blocks, nodes must control some of the staked cryptocurrency. If they do, they are selected to produce blocks, which carries rewards in the form of transaction fees and fixed block rewards.

Proof of Stake (and other consensus algorithms) want to incentivize users to participate in consensus. They do so by making block production profitable so that, the more that a user participates, the more money they make. In Proof of Stake, a node with a higher percentage of the staked cryptocurrency is selected to create more blocks, which means they can earn more block rewards.

In the long term, this can create a problem for a Proof of Stake blockchain. In Proof of Stake, the node with the highest percentage of stake earns the most rewards. If they stake all of these rewards, then the percentage of the staked assets that they control grows. This allows them to earn even more rewards, which allows their stake to grow and so on. In the end, if this user continues the process, they can control enough the staked assets to perform an XX% attack on the Proof of Stake consensus algorithm simply by allowing their rewards to accumulate over time.

Mitigating the PoS “Timebomb”

- Exploiting the PoS “timebomb” takes time
 - Need to build up rewards over time
- With Nxt, the richest account in May 2015 would need about 1,015 years to reach 50% stake
 - Faster if not all cryptocurrency is staked, buys up additional tokens, etc.
- The “timebomb” may be an inevitable problem but doesn’t require a solution

Source: <https://cointelegraph.com/news/the-inevitable-failure-of-proof-of-stake-blockchains-and-why-a-new-algorithm-is-needed>

99 | Blockchain and Crypto Security Training | © Marin Ivezic 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The Proof of Stake “timebomb” may put an expiration date on some Proof of Stake blockchains if exploited. If an attacker controls and stakes enough of a blockchain’s cryptocurrency, then they could attack the blockchain. Also, the blockchain’s tokens would likely also become worthless.

However, exploiting the Proof of Stake “timebomb” is not actually feasible in many cases. The gradual accumulation of block rewards can take many years. For example, an analysis performed in May 2015 found that the top token holder on the Nxt blockchain would need 1,015 years to control 50% of the blockchain’s tokens and 1,275 years to claim 90% in the worst case. However, this process could be expedited through collusion, purchasing additional cryptocurrency to stake, etc.

The Proof of Stake “timebomb” demonstrates that it is certainly possible to perform an XX% attack against a PoS blockchain even without investing any additional money in doing so. However, the timescales required to do so make it a more theoretical than practical attack.

Long-Range Attacks

100 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Long-Range Attacks

- Long-range attacks enable stakeholders to take advantage of the PoS “timebomb”
- The attacker creates a divergent version of the blockchain
 - Creates blocks on this version and re-stakes rewards
 - Doesn’t create blocks on official version
- Eventually, the attacker’s chain might become the longest chain
 - Attacker can publish it and take over



The Proof of Stake “timebomb” means that the Proof of Stake node with the largest stake might eventually control the network. A long-range attack allows any node with stake in the network to eventually achieve this.

In a long-range attack, the attacker creates a divergent version of the blockchain originating from a block when they own stake. Every time that they have the opportunity to do so (based on the block producer selection process), they create a block on this chain and re-stake the rewards they earn.

Since the attacker is the only one active on their divergent chain, their stake is the only one growing. This allows the attacker to be selected as block producer more frequently over time. As a result, their divergent blockchain grows more and more quickly.

On the main version of the blockchain, the attacker will skip their opportunities to create blocks. While this involves giving up a chance at block rewards, it also slows the growth of the main version of the blockchain.

At the beginning, the main version of the blockchain will grow much more rapidly

than the attacker's version because the attacker's chain will only grow when the attacker is selected as block producer. However, over time, the attacker will control a growing percentage of stake on their version of the blockchain, enabling their version to grow more quickly. Since the attacker refuses to create blocks on the main version of the chain and other nodes may accidentally miss their opportunities (or be forced to by a DoS attack), the main chain will miss opportunities to add blocks.

If the attacker's chain catches up and becomes the longer version of the blockchain, they can replace the main version under the longest chain rule. If this occurs, then the attacker has a significant level of control over the blockchain because they control the majority of the stake.

Mitigating Long-Range Attacks

- Like the PoS “time bomb”, a long-range attack takes time
 - Depends on the amount of stake owned by the attacker
 - May make the attack more theoretical than practical
- The attack also creates an obviously malicious version of the blockchain
 - Nodes could “break the rules” and simply reject the attacker’s chain even if it is the longer version



A long-range attack provides a means by which an attacker who does not control a majority of stake in a Proof of Stake blockchain could exploit the PoS “time bomb”. By completely controlling what happens on their version of the chain, the attacker can optimize their attack. For example, as the only block producer, an attacker can ensure that every block they produce maximizes the available transaction fees, which maximizes the rate at which their stake can grow. The attacker could also cherry-pick transactions to block other users from adding stake while keeping transactions that remove existing stake.

However, despite all of these advantages, a long-range attack, as its name states, is a long-term attack. Like the PoS “time bomb”, it is likely more of a theoretical than practical attack unless the attacker controls a massive amount of stake that could expedite the process.

Additionally, the success of a long-range attack depends on the rest of the blockchain network accepting the attacker’s malicious blockchain under the longest chain rule. While a legitimate node “should” accept whichever version of the blockchain is longer, nodes could “break the rules” and reject the attacker’s chain.

The Nothing at Stake Problem

103 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to the Nothing at Stake Problem

- Block creation in a Proof of Stake system is easy and cheap
 - No expensive mining, etc.
- When presented with divergent blockchains, stakers logically should build on both
 - Whichever wins, they get their reward
- Legitimate nodes could support double-spend attacks



Creating valid blocks in Proof of Work requires a computationally expensive and environmentally unfriendly mining process. As a result, miners are incentivized to choose between divergent blockchains because attempting to build on both requires twice the expenditure of resources for the same amount of reward.

The same is not true of Proof of Stake systems. A major selling point of Proof of Stake is that blocks are easy to create and the expensive and ecologically unfriendly mining process is eliminated. However, easy block creation also eliminates the incentive for block producers to choose when presented with two conflicting versions of the blockchain.

In theory, either of two divergent blockchains could eventually win under the longest chain rule depending on which block producers build on which version. This means that, if a block producer chooses the wrong version, they could lose the rewards that they could earn from producing a block.

Logically, stakers in a Proof of Stake system should build blocks on all versions of the blockchain that they are presented with. This way, no matter what, a block that they produce ends up on the final version of the blockchain and they keep their rewards.

However, this is not ideal for the security of the blockchain. Because stakers have nothing at stake to prevent them from building on multiple versions of the blockchain, they might contribute to long-range and other attacks that could allow an attacker to perform a double-spend attack.

Mitigating the Nothing at Stake Problem

- The Nothing at Stake problem exists because stakers have nothing to lose if they create multiple, conflicting versions of blocks
- Some Proof of Stake blockchains have implemented penalties for this misbehavior
 - Nodes make a “security deposit” along with their stake
 - This deposit is lost if they break the rules
- Ethereum 2.0 slashing is an example



The “Nothing at Stake” problem arises from the fact that the Proof of Stake algorithm has no built-in penalties to prevent misbehavior by block producers. In fact, it is rational for block producers to create multiple conflicting versions of blocks to lock in their rewards on conflicting, divergent blockchains.

To address this issue, some Proof of Stake blockchains have implemented penalties for misbehavior. For example, if a block producer misbehaves, they might lose some of their stake or a “security deposit” made during the staking process. If the cost of misbehavior is greater than the potential value gained by that misbehavior (a block reward, a successful double spend, etc.), then the incentive structure of Proof of Stake changes.

With its shift to Proof of Stake, Ethereum has implemented “slashing” to solve the Nothing at Stake problem. If validators misbehave, they are penalized, which helps to ensure that nodes are incentivized to behave properly. However, this can also cause issues, such as nodes being slashed due to misconfigurations or other issues that cause them to create or validate multiple versions of the same block (see <https://cointelegraph.com/news/expensive-lesson-75->

eth2-validators-slashed-for-introducing-potential-chain-split-bug).

Resource Exhaustion Attack

106 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Resource Exhaustion Attacks

- Proof of Stake blocks are easy to create
 - No expensive mining like in Proof of Work
- Proof of Stake blockchains are computationally expensive to validate
 - Block producers are selected based upon stakes
 - Stakes are contained within block bodies
- Fake blockchains are easy to create but expensive to validate
 - Denial of Service attack on blockchain nodes



Because of the longest chain rule, blockchain nodes are obliged to validate different versions of the blockchain that are presented to them. If a divergent blockchain is valid and longer than the one that the node is currently processing, then the node should switch to the new version.

However, blockchains using different consensus algorithms have different costs of validation. In Proof of Work, it is hard to create a block with a valid header but easy to detect invalid headers. This makes creating a divergent blockchain that passes validation difficult, but validating it is easy.

In Proof of Stake, the opposite is true. A Proof of Stake block is easy to create. The legitimate block producer just needs to collect a set of valid transactions together and create the block header. Unlike Proof of Work, there is no nonce value that is difficult and expensive to validate.

In contrast, validating a potential blockchain in Proof of Stake is expensive. In Proof of Stake, the legitimate block producer is selected based on the allocation of stake within the blockchain. Since staking transactions are contained within the body of a block, validation of a Proof of Stake blockchain candidate forces a Proof of Stake node

to download the entire blockchain (or at least the divergent part) and inspect each block body for staking transactions to calculate the legitimate producer of each block. This process consumes significant time and resources, but a node must complete it to ensure that it is working with the correct version of the blockchain. As a result, an attacker could perform a Denial of Service attack on a blockchain node by sending it invalid blockchains to validate. This attack could result in a node failing to create blocks if it is unsure which version of the blockchain it should be building on due to an incomplete validation process and a desire to avoid penalties under the Nothing at Stake problem.

Mitigating Resource Exhaustion Attacks

- Resource exhaustion or “fake stake” attacks take advantage of the design of Proof of Stake consensus
 - Critical information is included in block bodies
- Heuristics exist for determining whether a candidate blockchain is likely to be valid before performing in-depth analysis
 - These heuristics can be exploited by an attacker



Just like the 51% attack exploits the design of Proof of Work, resource exhaustion attacks take advantage of the design of Proof of Stake. Staking transactions are transactions, which means they are included in the body of Proof of Stake blocks. As a result, the process of validating a candidate PoS blockchain is much more computationally intensive than in Proof of Work, where high-level validation can be performed with only the block headers.

Various heuristics exist for performing such a high-level validation for Proof of Stake algorithms, but they are imperfect and can be exploited by an attacker. For example, one option is to check if a stake transaction used when producing a block exists on the blockchain. This heuristic checks that a transaction exists but has not already been spent, so an attacker could amplify their apparent stake by spending their stake to buy new stake. The heuristic would validate that the previously-spent stake exists but miss that it has already been spent. Detecting this would require the in-depth and computationally expensive full analysis of the blockchain.

Source: https://medium.com/@dsl_uiuc/fake-stake-attacks-on-chain-based-proof-of-stake-cryptocurrencies-b8b05723f806

Summary

- How Proof of Stake Works
- The PoS Solution to the BGP
- Variations on Proof of Stake
- Proof of Stake Security Assumptions
- Attacking Proof of Stake
 - XX% Attack
 - The Proof of Stake “Timebomb”
 - Long-Range Attacks
 - The Nothing at Stake Problem
 - Resource Exhaustion Attacks



Module 4

Blockchain User, Node, and Network Security



Introduction to Infrastructure Security

- Blockchain whitepapers define theoretical protocols
 - Explain how blockchain systems should work
- Blockchains need to be implemented to be usable
 - This creates new security risks
- Blockchain implementation security includes the security of:
 - Users
 - Nodes
 - Networks



Whitepapers like the Bitcoin whitepaper define overall protocols that describe how blockchains are supposed to work in theory. These theoretical protocols have security risks associated with them that can lead to attacks, such as the risk of a 51% attack in Proof of Work blockchains.

However, these theoretical descriptions do not describe all of the potential security risks of blockchain systems. To be useful, blockchains must be implemented, and the details of the implementation impact their security. For example, implementing Bitcoin using pen, paper, and carrier pigeon involves a very different threat model than using modern computers and networks to support the blockchain.

At the implementation and infrastructure level, blockchains can be exposed to various threats. These include security risks from three main categories:

- User security
- Node security
- Network security

Module 4: Blockchain User, Node, and Network Security

Section 4.1: User Security



Overview

- Introduction to User Security
- Non-Random Private Keys
- Exposed Mnemonic Seeds
- Nonexistent/Insecure Backups
- Third-Party Key Management
- Phishing Attacks
- Compromised Hardware Wallets
- Unverified Transactions
- DeFi Spend Approvals



Introduction to User Security

- Blockchain users' main security responsibilities revolve around account security
- Blockchain user security can fall apart in various ways
 - Non-Random Private Keys
 - Exposed Mnemonic Seeds
 - Nonexistent/Insecure Backups
 - Third-Party Key Management
 - Phishing Attacks
 - Compromised Hardware Wallets
 - Unverified Transactions
 - DeFi Spend Approvals



Blockchain users are a key component of the blockchain ecosystem. They generate and approve transactions and are responsible for securing their blockchain accounts. From a security perspective, the main roles that blockchain users play are the protection of the private keys associated with a blockchain account and ensuring that the transactions that they digitally sign with these keys are correct.

Blockchain user security can fail in a number of different ways. Some examples of user security failures include:

- Non-Random Private Keys
- Exposed Mnemonic Seeds
- Nonexistent/Insecure Backups
- Third-Party Key Management
- Phishing Attacks

- Compromised Hardware Wallets
- Unverified Transactions
- DeFi Spend Approvals

Non-Random Private Keys

- Blockchain accounts' private keys are designed to be private
 - Only known to the account owner and those authorized to use it
- Random keys are an essential part of this
 - Keys that are easily guessable won't remain private
- Use of non-random, weak private keys endangers the security of user accounts and the blockchain as a whole
 - Accounts' value, permissions, etc. can be used in attacks



As their name suggests, the private keys for blockchain accounts are intended to be private. In digital signature schemes, the only secret value is the private key, meaning anyone with knowledge of an account's private key can generate valid digital signatures and transactions on its behalf. For this reason, the private key of a blockchain account should only be known to the account owner and those authorized to use that account.

An essential part of keeping private keys private is generating these keys using a cryptographically secure random number generator. If keys are generated using a weak source of randomness (such as hashing a word or phrase), then it is possible that an attacker will be able to guess the key and steal the value stored in the blockchain account.

Blockchain accounts may also be compromised to take advantage of their access and privileges on the blockchain. For example, multiple decentralized finance (DeFi) projects have suffered hacks where the attacker gained access to the private key used to manage the project's smart contract and used the permission associated with that account to steal from the protocol and its users.

One example of weak private keys leading to theft of funds is the “Blockchain Bandit”. The “Bandit” learned that some Ethereum users were using a tool that generated predictable private keys. The attacker searched the Ethereum blockchain for accounts addresses associated with these weak keys and stole the cryptocurrency that was stored within.

Source: <https://cointelegraph.com/news/blockchain-bandit-how-a-hacker-has-been-stealing-millions-worth-of-eth-by-guessing-weak-private-keys>

Exposed Mnemonic Seeds

- Blockchain private keys are not easy to remember or type into blockchain software
 - Even encoded are a long series of letters and numbers, making typos easy
- Mnemonic seeds are designed to address this problem
 - 12-24 words that encode the complete private key
- If mnemonic seeds are partially or wholly exposed, an attacker may be able to learn the entire private key



To perform transactions with a blockchain account, the blockchain software needs access to the private key associated with it. While blockchain software can generate and store these keys internally, a user might need to type in a key in some cases. However, even encoded, private keys are a long series of letters and numbers, which makes it easy for a user to make a mistake when typing it in.

Mnemonic seeds are designed to make private keys easier to memorize or type by converting a random string of bits into a set of 12-24 words (depending on the key length). These words are drawn from a dictionary of words where each word is accessed based on its location in the list. This makes it possible to convert private keys into mnemonic seeds by breaking the key into chunks, converting the binary value into a decimal number, and looking up the word at that offset within the list.

Mnemonic seeds are designed to be used to derive private keys, so they need to be protected just as strongly. Even a partially compromised mnemonic seed could expose an account if the attacker has sufficient knowledge to brute-force the remaining words/bits.

For example, Alistair Milne ran a contest in which he exposed 8 of the 12 words of a

mnemonic seed over time. John Cantrell wrote a script that brute forced approximately 1 trillion possibilities for the remainder of the key within 30 hours. He successfully cracked the key, enabling him to claim the 1 Bitcoin stored in the account.

Source: <https://medium.com/@johncantrell97/how-i-checked-over-1-trillion-mnemonics-in-30-hours-to-win-a-bitcoin-635fe051a752>

Nonexistent/Insecure Backups

- Private keys are literally the key to a blockchain account
 - Anyone with the key can access the account
 - Anyone without the key can't
- Keeping a backup of a private key is a good way to prevent losing it
 - Roughly 20% of Bitcoin is believed lost forever due to lost keys
- Backups must also be secure
 - Insecure backups can expose accounts to attackers

Source: <https://blog.chainalysis.com/reports/money-supply>

117 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Blockchain account security boils down to the security of the account's private key. The account is only useful if transactions can be generated that spend its value or take advantage of its influence. Creating transactions requires the ability to generate digital signatures, and valid digital signatures can only be made with the appropriate private key.

Anyone with access to the private key of a blockchain account can use that key to generate digital signatures and valid transactions. Without the key, there is no way to use a blockchain account.

A user's copy of a blockchain account's private key can be lost due to a dead/discarded hard drive, forgetting a mnemonic seed, etc. An estimated 20% of Bitcoin has been lost forever because the private keys of the accounts holding this Bitcoin have been lost/forgotten.

Creating a backup copy of a private key is a good idea and can help protect against these losses. However, it also comes with its risks. A blockchain account is only as secure as the least secure copy of its private key. If a backup copy of a key is not securely stored, it may be compromised by an attacker who uses it to take over the

associated blockchain account.

Third-Party Key Management

- Managing blockchain account private keys can be a lot of work
 - Need to store it securely, type it in to perform transactions, etc.
- Some third-party services offer key management
 - Cryptocurrency exchanges, etc.
- Third-party key management carries significant risks
 - Key is protected by user's password and MFA
 - Private keys may be compromised if provider is hacked



The “be your own bank” philosophy of blockchain doesn’t appeal to everyone. While the control is nice, the complexity of managing and securing their own private keys can be overwhelming. If users want to maintain full control over their blockchain accounts, they need to take responsibility for storing and securing their own keys and will need to provide these keys to blockchain software any time that they want to perform a transaction. For frequent traders, users with multiple accounts, and non-technical users, the overhead required for this may be too much.

Many third-party providers offer key storage and management as part of their portfolio. For example, many cryptocurrency exchanges will allow a user to generate, store, and use their private keys all on the same site. This makes it much easier to use blockchain-based solutions because the service provider is handling all of the technical details. However, entrusting private keys to third-party providers means that a user is giving up control over their blockchain account. Some implications of this include:

- **Reduced Security:** Ideally, a blockchain account’s private key is a long, random value that is essentially impossible to guess. However, when using a third-party key management service, a user’s account is accessed via a password and potentially a

multi-factor authentication (MFA) code. These are much easier to guess, phish, or hack than a full private key, and compromising them gives an attacker full access to all of a user's blockchain accounts.

- **Supply Chain Risks:** If a third-party provider is managing a user's private keys, then that provider has access to those private keys. If the provider is hacked, then an attacker may be able to gain access to users' keys. Even if those keys are appropriately encrypted when not in use, an attacker could potentially insert malicious code into the provider's system that steals and exfiltrates keys when they are unlocked (such as when a user performs a transaction).

Phishing Attacks

- Phishing attacks are some of the easiest and most effective cyberattacks
 - Trick the user into handing over sensitive information
- Phishing attacks exploit trust relationships
 - Such as those between users and their third-party service providers
- Blockchain-focused phishing attacks attempts to steal passwords, MFA code, private keys, etc.



Phishing attacks are a major threat that has existed for decades. Most cyberattacks involve a phishing attack at some point because these attacks are easy to perform, highly effective, and provide an attacker with much-needed access.

Phishing attacks work by targeting the user and exploiting trust relationships. For example, an email from an unknown sender requesting someone's bank account information or other sensitive data is likely to be rejected. However, an email that looks like it comes from the bank and that asks the user to log in and review anomalous activity is much more likely to succeed and has the same result.

Blockchain is theoretically trustless, but trust relationships definitely exist. For example, users of a third-party service for storing private keys trust their service provider and are likely to react to their emails. Similarly, blockchain users who store keys on hardware wallets will pay attention to emails from the manufacturer of that wallet.

Phishing attacks are a common threat in the blockchain space. Attackers attempt to steal passwords, MFA codes, and private keys. Phishing messages may also be designed to point users to malicious pages designed to steal this information or trick

them into performing malicious transactions that give an attacker access to their blockchain accounts or the tokens stored within.

Compromised Hardware Wallets

- Hardware wallets are designed to provide secure storage for private keys
 - Dedicated microprocessor stores keys and performs calculations
 - Keys never leave the protected device
- However, hardware wallets have their security risks
 - Vulnerabilities and defects
 - Supply chain exploits
 - Fake hardware wallets
 - Phishable trust relationships



Storing private keys on a hardware wallet is a good idea, especially compared to many other options for private key storage. Hardware wallets incorporate dedicated, hardened microprocessors that store private keys and perform the computations needed to digitally sign a blockchain transaction. By performing all operations that use a private key on the device, hardware wallets ensure that the key never leaves the device where it could be compromised.

Hardware wallets are a valuable tool for blockchain account security. However, they do carry certain security risks, including:

- **Vulnerabilities and Defects:** Hardware wallets are designed to securely store and use private keys. However, vulnerabilities have been discovered in hardware wallets in the past. Exploitation of these vulnerabilities could result in the compromise of a private key or generation of malicious transactions.
- **Supply Chain Exploits:** Hardware wallets are built out of various components, and completed wallets are shipped from the manufacturer to the user. If an attacker has access to the original components or can intercept a delivery, they might be able to make modifications that could expose private keys placed on these devices.

- **Fake Hardware Wallets:** Hardware wallets have known and respected brands that users implicitly trust. If an attacker can create lookalike or knockoff wallets that look like the real thing, then they could potentially trick users into accepting and storing private keys on these fake wallets. These wallets could transmit compromised keys to the attacker or generate malicious transactions themselves.
- **Phishable Trust Relationships:** Hardware wallet users have an established trust relationship with their wallet manufacturer, which an attacker may be able to exploit in a phishing attack. For example, a phishing email could claim that a user's wallet needs to be upgraded/replaced for some reason and request a shipping address to which the attacker could send a fake wallet onto which the user might transfer private keys.

Unverified Transactions

- Creating a blockchain transaction is a multi-stage process
 - Define transaction data
 - Generate a digital signature
 - Transmit the transaction to the blockchain
- In some cases, this process involves multiple systems or devices
 - Digital signatures are performed on hardware wallet
- Failing to verify transactions at each stage enables malicious modifications



A blockchain user will go through multiple different steps while creating a blockchain transaction:

- **Defining Transaction Data:** A blockchain transactions has one or more intended recipients, associated data, etc. Anyone can define transaction data for any account, it is the digital signature that proves authenticity and integrity.
- **Digital Signature Generation:** After the transaction data is defined, the user generates a digital signature for that data. This signature proves that the transaction was created by a legitimate user of the account (or someone that knows the account's secret key)
- **Transaction Broadcast:** To be included in the blockchain's digital ledger, a transaction must be known to block producers. This involves broadcasting the transaction via the blockchain's peer-to-peer network.

This process is a mix of non-privileged actions (defining transaction data and transaction broadcast) and privileged ones (digital signature generation). In some cases, these actions may be performed by different systems or even on different

devices. For example, a hardware wallet may be used to digitally sign a transaction that is created and broadcast by software running on the user's computer.

Up until the digital signature is generated, an attacker can modify the transaction at will. Unless the user detects the modification before the transaction is transmitted, changing the transaction data can change the effects of the transaction. An attacker might redirect a transfer to their account or add malicious actions to a user's transaction.

An example of this is an attack targeting the CEO of Nexus. An attacker infected the CEO's computer with a malicious version of MetaMask that modified transaction data before sending it on to the CEO's hardware wallet for signature. Since the CEO missed the modifications to their transaction, the attacker was able to steal \$8 million out of his account. This attack didn't require access to the hardware wallet or private key, just the ability to modify the transaction data without detection.

Source: <https://decrypt.co/51355/hacker-steals-8-million-from-nexus-ceo-by-remotely-changing-metamask>

DeFi Spend Approvals

- DeFi projects have the concept of spend approvals
 - Allow a project's contract to extract tokens from a user's account without explicit approval
 - Make trades faster and easier to perform
- Unlimited spend approvals mean that a DeFi smart contract can extract approved tokens at any time
 - Including if they are compromised by an attacker
- Several DeFi hacks have exploited spend approvals to drain user funds



A spend approval for a DeFi project is designed to make it faster and easier to perform certain types of trades. A user can authorize the project to pull certain types of tokens from their account without explicit approvals. This makes it faster to perform a transaction in the future because the user doesn't need to explicitly authorize the transfers. However, spend approvals also carry significant risks. If a DeFi user authorizes unlimited spends for certain tokens, then the DeFi project's smart contract can extract these tokens at any time. This includes if the contract is compromised by an attacker or if a malicious project team performs a rug pull.

Multiple DeFi projects have been the victims of hacks that included exploitation of unlimited spend approvals. One example is a November 2021 hack against bZx. A spear phishing attack targeting a bZx developer allowed the attacker to steal the private keys of the developer's personal account and the ones used to launch bZx's smart contracts on Polygon and Binance Smart Chain (BSC). The attacker used these contract keys to drain value from the contracts and to exploit unlimited approvals to steal values from bZx user's wallets.

Summary

- Introduction to User Security
- Non-Random Private Keys
- Exposed Mnemonic Seeds
- Nonexistent/Insecure Backups
- Third-Party Key Management
- Phishing Attacks
- Compromised Hardware Wallets
- Unverified Transactions
- DeFi Spend Approvals



Module 4: Blockchain User, Node, and Network Security

Section 4.2: Node Security



Overview

- The Role of the Blockchain Node
- Attacking Blockchain Nodes
 - Blockchain Breakouts
 - Denial of Service Attacks
 - Malware
 - Man-in-the-Middle Attacks
 - Software Misconfigurations



The Role of the Blockchain Node

- The blockchain is a decentralized system
 - No central authority updating and maintaining the digital ledger
- Blockchain nodes are responsible for keeping the blockchain going
 - Digital ledger storage
 - Transaction creation and validation
 - Block production and verification
 - Communications
 - Smart contract execution



Blockchains are designed to be completely distributed and decentralized systems. This means that there is no central authority that maintains a copy of the digital ledger, including building blocks, validating and executing transactions, etc.

Instead, the nodes in the blockchain network each perform this role independently. Each node in the blockchain network is responsible for maintaining its own copy of the digital ledger, and nodes participate in every aspect of the blockchain's operations. Some of the functions that a blockchain node might perform include:

- **Digital Ledger Storage:** Without a centralized, “official” version of the ledger, nodes need to keep their own copies of the blockchain. This enables them to track the history of the network and to validate the transactions that are contained within new blocks.
- **Transaction Creation and Validation:** Any account within the blockchain network can create transactions for inclusion in the digital ledger. Nodes are responsible for validating these transactions before including them in their copy of the ledger to protect against double-spends or other fraudulent transactions.

- **Block Production and Validation:** Blockchain nodes have the option to participate in the blockchain consensus and block production processes, which involves creating valid blocks from verified transactions. Other nodes in the network, after receiving these blocks must validate them before adding them to local copies of the distributed ledger.
- **Communications:** The blockchain uses a peer-to-peer network for communication. Nodes in the network may receive blocks and transactions from their peers and should forward them on to their other connections.
- **Smart Contract Execution:** Smart contract platforms allow code to run on top of the blockchain. Nodes in a smart contract platform need to host a virtual machine in which they run smart contract code.

Attacking Blockchain Nodes

- Blockchain nodes are crucial to the operation of the blockchain network
 - Attacks against nodes can impact the security of the blockchain
 - The reverse is also true
- Some security risks of blockchain nodes include
 - Blockchain Breakouts
 - Denial of Service Attacks
 - Malware
 - Man-in-the-Middle Attacks
 - Software Misconfigurations



Blockchain nodes are what make a decentralized system like the blockchain possible. In a decentralized system, there is no central server responsible for processing and storing data. Instead, the blockchain nodes do so.

The importance of nodes within the blockchain make them a crucial part of blockchain security. If an attacker can compromise a node or disrupts its operations, then this can impact the security and performance of the blockchain network. Similarly, attacks within the blockchain ecosystem can affect the security of blockchain nodes.

Blockchain nodes face various security risks. Some of these include:

- Blockchain Breakouts
- Denial of Service Attacks
- Blockchain-Specific Malware
- Man-in-the-Middle (MitM) Attacks

- Software Misconfigurations

Blockchain Breakouts

- Blockchain protocols are implemented as software
 - Process transaction and smart contract data in an isolated environment
- Transactions and smart contracts are untrusted data
- Vulnerabilities in processing code could enable a malicious transaction to break out of the blockchain sandbox
 - Run malicious code on blockchain node processing the code



Blockchain protocols like Bitcoin and Ethereum are implemented as software running on a blockchain node. This software implements all of the core functions of the blockchain node.

One of these core functions is processing and executing transactions (which also include smart contract code). The blockchain is designed to allow anyone to create transactions to be included in blocks and processed by every node within the blockchain network. This means that every node in the network will process untrusted user data contained within transactions and blocks.

If the blockchain software contains a buffer overflow or other vulnerability, a specially crafted transaction could exploit this vulnerability to allow malicious code execution on any node that runs the vulnerable blockchain software and processes the transaction.

Before the EOS network went live, researchers from Qihoo 360 identified a blockchain breakout vulnerability in the code. This vulnerability could have allowed an attacker to take over every node in the network if they processed malicious data. The researchers created a Proof of Concept (PoC) that exploited the vulnerability to

create a reverse shell. Due to the report of the vulnerability EOS was able to correct the issue before launch.

Source

<https://thehackernews.com/2018/05/eos-blockchain-smart-contract.html>

Denial of Service Attacks

- Blockchain nodes need resources to do their jobs
 - Storage space
 - Computational power
 - Network connectivity
- Denial of Service (DoS) attacks can disrupt the blockchain's operation by denying access to these resources
 - Consuming CPU and/or memory
 - Distributed Denial of Service (DDoS) attack



Blockchains' security and performance depend on the performance of the nodes in the blockchain network. A very decentralized network with many active nodes is protected against dangerous centralization and has greater resiliency. Also, with many active nodes, the probability of missed blocks because the selected block producer is unavailable is lower.

A Denial of Service (DoS) attack against a blockchain node can impact the operations of the blockchain. If a node lacks storage space, it can't save new blocks or pending transactions. Without computational power, a node can't create and validate transactions or blocks. Without network connectivity, the node can't receive transactions and blocks from its peers or send out new transactions and blocks.

An attacker can decrease blockchain security and performance by targeting nodes in the network. Targeting blockchain nodes in general can have an effect, but the impact of an attack is amplified if the attacker can target temporary single points of failure within the blockchain network.

For example, a DoS attack against the selected block producer in a given interval will have a significant effect on the blockchain's performance because an attacker can

delay or prevent the creation of a block. Another target might be a critical link in a blockchain's peer-to-peer network, which can prevent the propagation of transactions and blocks through the network.

Malware

- Blockchain protocols are implemented as software
 - Run on a computer alongside other software, which could be malware
- Malware can be designed to target blockchain systems in various ways
 - Theft of private keys
 - Malicious transactions
 - Traffic Filtering
 - Denial of Service Attacks



Blockchain protocols are implemented as software, which, like other software, run on a computer. Like other software, blockchain software's security can be impacted by other, malicious applications running on the same computer (i.e. malware).

Blockchain software is complex with various different functions. This means that blockchain-specific malware can impact the security of this software and the blockchain in several different ways, including:

- **Theft of Private Keys:** Blockchain software is how users interact with the blockchain, which includes generating transactions. Blockchain transactions are digitally signed, which requires access to the private key associated with the blockchain account. Malware can be designed to steal this private key, enabling it to generate transactions on behalf of the compromised account.
- **Malicious Transactions:** Only someone with knowledge of a blockchain account's private key can digitally sign transactions for that account. However, an attacker could create malicious transactions and trick a legitimate user into signing them. Some blockchain malware is designed to change the content of transactions created by the user, redirecting transfers or adding malicious functionality before

the user signs the transaction.

- **Transaction Filtering:** A blockchain node relies on communications from its peers in the blockchain network to monitor the current state of the blockchain ecosystem. Malware that can intercept and selectively drop data being sent to a node can manipulate its view of the state of the blockchain network.
- **Denial of Service Attacks:** As mentioned previously, blockchain nodes need computational power, storage space, and network connectivity to do their jobs. Malware on a node can disrupt any of these, impeding the node's ability to perform core functions.

Man-in-the-Middle Attacks

- Blockchain nodes are heavily reliant on communications over the network
 - Sharing transactions and blocks for inclusion in the ledger
- A Man-in-the-Middle (MitM) attacker can't create fake transactions and blocks without an account's private key
- However, it can filter traffic entering and leaving a blockchain node
 - Impacts its view of the state of the blockchain



The blockchain is designed to be a distributed and decentralized system where nodes in the network are responsible for independently updating and maintaining their own copies of the blockchain's digital ledger. This makes the blockchain system heavily reliant on network communications to access the new transactions and blocks that it needs to produce new blocks and to add blocks to its copy of the digital ledger.

A Man-in-the-Middle (MitM) attacker has the ability to intercept blockchain transmissions en route to/from a blockchain node. This attacker can't create fake versions of transactions because each transaction is digitally signed, which requires access to the private key of the account generating the transaction.

However, the ability to filter and drop data being sent to/from a node can have an impact on its and other nodes' view of the state of the blockchain. For example, dropping information about some transactions and blocks could cause a node to start building a conflicting version of the blockchain. In this way, a MitM attacker could manipulate a node with a significant amount of consensus power into supporting an attempted double-spend by tricking it into building a divergent version of the blockchain.

Missed or Malicious Updates

- Blockchain software is software
 - It needs regular updates to add functionality, fix vulnerabilities, etc.
- Updates can create risk for blockchain nodes
 - Missed updates
 - Insecure updates
 - Malicious updates
- Updates are harder in a decentralized system



Blockchain software is like any other software. It isn't perfect in the first release and needs regular updates. These updates might be driven by the desire to add new features and functions to the blockchain implementation or to fix vulnerabilities and other security issues

Software updates can create significant risks to a blockchain node's security and performance. Some update-related risks include:

- **Missed Updates:** Blockchain nodes should promptly install both security and functionality-focused updates. A failure to install security updates could leave a node vulnerable to exploitation. Missed functionality updates might leave a node unable to interact with the blockchain, especially if the update implements a "hard fork" that includes features that are not backward-compatible.
- **Insecure Updates:** Software updates change the code of the software that they are updating. If the update fails to fix an existing vulnerability or introduces a new one, then it could create a false sense of security or create an opportunity for an attacker to exploit the node.

- **Malicious Updates:** Blockchain software is often open-source and relies on open-source libraries and components. This creates potential opportunities for an attacker to insert malicious code into a software update. If this update is installed by a node, it could create vulnerabilities, insert a backdoor, or add other malicious functionality that could hurt the node, the blockchain, or the security of a user's blockchain account.

Update and patch management is a challenge across IT, but it is especially difficult for blockchain systems. The decentralization of the blockchain means that there is no means for compelling nodes to install updates other than community pressure and the risk of missing out on the rewards of participating in the blockchain.

This makes security updates especially dangerous and difficult. Often, the importance of an update is downplayed for some time after release because revealing the details could allow an attacker to exploit the vulnerability that the update is designed to fix. This creates a delicate balancing act between getting node operators to install updates without drawing the attention of attackers.

Software Misconfigurations

- Blockchain software is a mix of vital and optional features
 - Some functionality (such as blockchain communications) is essential for all nodes
 - Other features are optional
- Some features of blockchain nodes can create security risks if misconfigured
 - Leaves nodes vulnerable to exploitation



Blockchain software has both necessary and optional functionality. For example, not every node participates in consensus, but all nodes in the blockchain network should receive transactions and blocks from their peers and forward them on via the peer-to-peer network. Necessary features are enabled by default, while optional ones might be disabled until a user activates them.

Different features of blockchain software carry different levels of security risks. Even vital functionality such as receiving and processing transactions carries risk in the form of potential injection vulnerabilities that could be exploited by malformed and malicious transactions and blocks. Optional features, such as support for integration with external systems, can also introduce security risks.

An example of this is the JSON-RPC interface available in some Ethereum node software. This interface is designed to expose an API on port 8545 that allows remote interaction with the Ethereum client. While this feature is disabled by default, some users enabled it without using a firewall to block access to this port from the public Internet. As a result, attackers who scanned the Internet for exposed port 8545 were able to steal over \$20 million in ETH from users that enabled this feature without properly protecting it.

Source: <https://thehackernews.com/2018/06/ethereum-geth-hacking.html>

Summary

- The Role of the Blockchain Node
- Attacking Blockchain Nodes
 - Blockchain Breakouts
 - Denial of Service Attacks
 - Malware
 - Man-in-the-Middle Attacks
 - Software Misconfigurations



Module 4: Blockchain User, Node, and Network Security

Section 4.3: Network Security



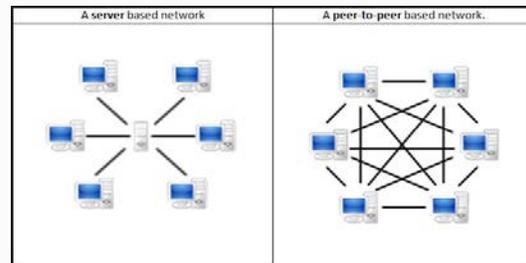
Overview

- Structure of the Blockchain Network
- Attacking Blockchain Networks
 - Denial of Service Attacks
 - Eclipse/Routing Attacks
 - Sybil Attacks



Structure of the Blockchain Network

- Blockchain decentralization demands a peer-to-peer network
- Each node in the network is connected to a few neighbors
- Data moves through the network via multiple hops



<https://en.bitcoinwiki.org/wiki/Peer-to-peer>

137 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The blockchain is designed to be a distributed, decentralized system. This is why all of the core functions of the blockchain are duplicated across many independent blockchain nodes.

It is also why the blockchain uses a peer-to-peer network instead of a more traditional client-server model. A client-server architecture is more efficient and can offer higher performance than a peer-to-peer network. However, it also relies on a centralized authority that can abuse its power or be targeted in an attack designed to take down the blockchain network.

The blockchain does not use a fully-connected peer-to-peer network due to scalability considerations. The number of links in a fully-connected network scales exponentially with the number of nodes, making it unworkable for blockchain networks the size of Bitcoin. Also, the complexity of maintaining a list of connections when nodes are frequently coming online and going offline would consume resources that blockchain nodes need to perform their other functions.

Instead, the blockchain uses a partially-connected peer-to-peer network in which each node is connected to only a few other nodes. When a node receives data from

one of its peers, it forwards that transmission on to the rest of its neighbors. In this way, data percolates across the blockchain network via multiple hops. The variety of potential routes between any two nodes provides redundancy and decentralization.

Attacking Blockchain Networks

- The blockchain is heavily dependent on its underlying network infrastructure
 - Need reliable communications for consensus, block production, etc.
- The blockchain can be attacked at the network level in various ways, including:
 - Denial of Service Attacks
 - Eclipse/Routing Attacks
 - Sybil Attacks



Blockchain systems are heavily reliant on the robustness and performance of their underlying network infrastructure. Blockchains' decentralization makes them inefficient with multiple copies of transaction and block data being received and sent by each node. To perform regular updates to the digital ledger, blockchain nodes need network infrastructure that is capable of rapidly and reliably distributing new data (such as transactions and blocks) throughout the blockchain network.

Blockchain's dependence on its underlying network infrastructure makes it a good target for attacks against the system. Some examples of attacks that target blockchain systems at the network level include:

- Denial of Service Attacks
- Eclipse/Routing Attacks
- Sybil Attacks

Denial of Service Attacks



How a Denial of Service Attack Works

- Blockchain decentralization should improve resiliency
 - No permanent single points of failure
 - Resistance to Denial of Service attacks
- However, blockchains have temporary points of centralization
 - Block producers
 - Communications links
- Denial of Service attacks targeting these can impact a blockchain's operations



One of the main benefits of blockchain decentralization is that it provides a level of resiliency to the blockchain system. By eliminating centralized authorities and distributing crucial operations across many nodes, blockchain technology makes it harder to disrupt operations with a Distributed Denial of Service (DDoS) attack. No node in the blockchain network is essential to its operations, so taking down one or more nodes in attack only serves to reduce performance or security, not bring the system down entirely like a DDoS attack against a centralized system might.

By eliminating permanent points of centralization or single points of failure, a blockchain becomes more resistant to DDoS and other DoS attacks. However, the blockchain does have temporary points of centralization, such as:

- **Block Producers:** Blockchain consensus algorithms are designed to select a single node to produce the next block in the blockchain. For consensus algorithms where the block producer is selected in advance (like Proof of Stake), an attacker that predicts the block producer could perform a DDoS attack against that node to slow or prevent the creation of that block.
- **Communications Links:** While a peer-to-peer network should create multiple

paths between various parts of the network, this is not guaranteed. If a network is organized so that only a few links exist between different parts of a network, then a DoS attack against the nodes that maintain these links could divide the network. These divisions can enable double-spend and other attacks.

Mitigating Denial of Service Attacks

- Denial of Service attacks are a traditional IT threat
 - Can be addressed with traditional IT solutions (anti-DDoS, antimalware, etc.)
- Blockchain systems can also be designed to make these attacks harder
 - More connected network, etc.
- Deploying DoS protections can be difficult in a decentralized environment
 - No means of enforcement



Denial of Service (DoS) and Distributed DoS (DDoS) attacks are not a threat unique to blockchain systems. These attacks have been around for decades; the primary difference is what might be targeted during an attack. Since these are established threats, there are also solutions for addressing them. For example, a potential target of a DDoS attack (such as a block producer) could deploy anti-DDoS solutions, and blockchain nodes could install antimalware solutions to protect against malware-driven DoS attacks.

Blockchain systems can also be designed to mitigate the effects of these attacks. For example, networks with more connectivity between various segments are more difficult to divide and isolate with a DoS attack against the nodes maintaining these links. A consensus algorithm where the chosen producer of the next block is less predictable can protect against DDoS attacks against block producers.

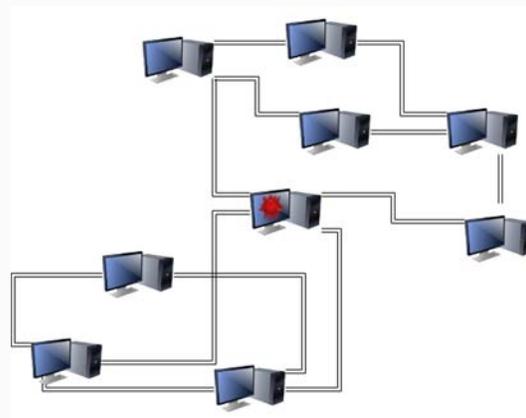
The main challenge in protecting against DoS attacks in a blockchain environment is blockchain decentralization. Without a centralized authority capable of forcing nodes to implement these protections, nodes individually choose whether or not to do so. As a result, individual nodes' decisions can have an impact on the functioning of the blockchain network as a whole.

Eclipse/ Routing Attacks



How an Eclipse/ Routing Attack Works

- Eclipse/routing attacks are designed to isolate one or more nodes from the rest of the network
 - Allows attacker to control these nodes' view of the state of the blockchain
- These attacks can be used to enable various other attacks



The blockchain network is implemented as a partially-connected peer-to-peer network. Each node in the network is directly connected to a few peers, who are connected to a few peers of their own, and so on. Information flows through the network via a series of hops across these peer-to-peer connections.

In an eclipse/routing attack, an attacker isolates one or more nodes from the rest of the network. An eclipse attack targets a single node, while a routing attack breaks the blockchain network into multiple, isolated segments. These attacks can be accomplished in various ways, such as:

- An attacker controlling all of a node's peers
- A Border Gateway Protocol (BGP) hijacking attack
- Disruption of critical peer-to-peer links via a DoS attack

If successful, an attacker can take advantage of an eclipse/routing attack in various ways, these include:

- **51% Attack:** By breaking a blockchain network into chunks, an attacker reduces the computational power needed to perform a successful 51% against each of the network segments. For example, a 51% attack against a segment containing 60% of the network's hashpower only requires 30% of the blockchain's hashpower. If an attacker's version of the blockchain is accepted and built upon by this 60%, it could grow faster than and supplant the blockchain version maintained by the other 40% under the longest chain rule as well.
- **Double-Spend Attack:** An attacker could send conflicting versions of a transaction to two isolated network segments, each of which contains the intended recipient of the version sent to it. After the recipients act on the transaction, the attacker could end the eclipse/routing attack, causing one version of the blockchain to defeat the other under the longest chain rule.

Mitigating Eclipse/Routing Attacks

- An eclipse/routing attack requires the attacker to intercept communications to/from one or more nodes
- Nodes can make this more difficult by
 - Connecting to more peers
 - Randomly reconnecting
 - Multi-homed nodes
 - Network statistics monitoring
 - Encrypted, authenticated “heartbeats”



An eclipse or routing attack requires the attacker to have control over the links between one or more nodes and the rest of the network. This can be accomplished by attacking the node itself, controlling a node’s peers, exploiting network routing (BGP, etc.), and other means.

Completely preventing an eclipse/routing attack is difficult because a blockchain node only has control over its direct connections. If all of the peers of a node’s peers are malicious, then an attacker can perform a routing attack without the node connecting to a single malicious node.

However, a node can take some steps to make an eclipse/routing attack more difficult, including:

- **More Neighbors:** To isolate a node from the rest of the network, an attacker needs to control all of its connections to the rest of the network. If a node connects to more peers, doing so becomes more difficult for an attacker.
- **Random Reconnects:** To perform a sustained eclipse/routing attack, an attacker needs to cut off a node or nodes for the entire duration of the attack. Otherwise,

the targets may receive data that the attacker does not want them to see. Reconnecting to random peers at random intervals makes it more difficult for an attacker to ensure continued control over a node's communications.

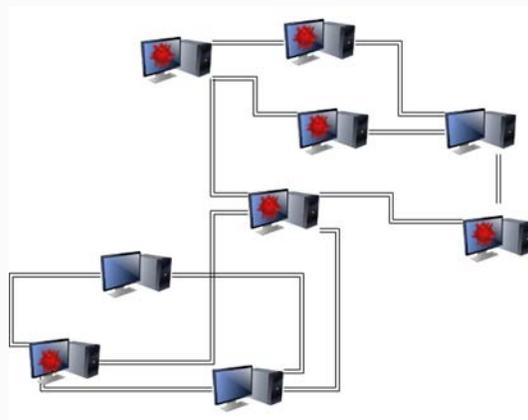
- **Multi-Homed Nodes:** Border Gateway Protocol (BGP) hijacking attacks can enable a routing attack by forcing traffic between different IP prefixes to pass through the attacker's machine. Multi-homed nodes with interfaces with different IP prefixes can make it more difficult to perform a routing attack using BGP hijacking.
- **Network Statistics Monitoring:** An eclipse/routing attack is likely to have an impact on the rate of transaction and block production and the latency of traffic that needs to pass through an attacker's system for analysis before being forwarded on to its destination. By monitoring these statistics, a node can detect if it is likely currently the victim of an eclipse/routing attack.
- **Encrypted, Authenticated "Heartbeats":** An attacker can only filter blockchain traffic without detection if they can differentiate between traffic that they do and do not want to reach the target node(s). If blockchain traffic is encrypted and contains "heartbeat" traffic, then an eclipse/routing attack can be detected if an attacker blocks the heartbeat traffic by accident.

Sybil Attacks



How a Sybil Attack Works

- Blockchain accounts are anonymous, meaning that anyone can create one
- Sybil attackers create many malicious accounts
- These malicious accounts can be used to support other attacks



The original blockchain systems were designed to be completely open and anonymous. Anyone could create a blockchain account and participate in the blockchain's operations. This anonymity is enabled by the fact that identity on the blockchain is managed by account addresses. Account addresses are derived from public keys, which are calculated from private keys, which are random numbers. Since these random numbers are not linked to real-world identities, blockchain accounts are anonymous.

A Sybil attacker takes advantage of this anonymity to create many malicious accounts. This is accomplished simply by generating private keys, public keys, and account addresses. Creating a blockchain account and running a fake blockchain node requires minimal resources, so an attacker could create many fake accounts.

Blockchain consensus algorithms are designed to protect against Sybil attacks, using a scarce resource for voting instead of a "one account, one vote" scheme. However, Sybil attacks can be used to enable other attacks, such as eclipse/routing attacks. By controlling many malicious accounts/nodes, an attacker increases the probability that all of a node's peers will be controlled by an attacker. If this is the case, then the attacker controls that node's view of the current state of the blockchain network.

Mitigating Sybil Attacks

- Sybil attackers take advantage of blockchain anonymity to create many malicious accounts
 - Blockchains without anonymity are more protected against Sybil attacks
- Blockchains and nodes can also attempt to mitigate the impacts of Sybil attacks
 - Connecting to more or trusted peers to decrease the probability of having all malicious neighbors



A Sybil attack is possible because blockchain accounts and nodes are anonymous. If no-one knows the owner of a particular account or node, it is difficult to determine if other accounts/nodes are owned by the same person.

Sybil attacks can be made impossible by stripping away blockchain anonymity. Some blockchains, such as private and permissioned blockchains have the ability to do so by tying blockchain identity to real-world identity. For example, a blockchain user may be required to have a digital certificate that ties their public key to their real-world identity.

For blockchains with anonymity built in, another option is to focus on the effects of a Sybil attack. Having multiple accounts on the blockchain owned by the same party does no harm unless they can be used in some malicious way. Protecting against eclipse/routing attacks, for example, can help to mitigate the effects of Sybil attacks. As mentioned previously, blockchain consensus algorithms are designed to be immune to these attacks, which limits the potential impact that they can have on the blockchain's operations and security.

Summary

- Structure of the Blockchain Network
- Attacking Blockchain Networks
 - Denial of Service Attacks
 - Eclipse/Routing Attacks
 - Sybil Attacks



Module 5

Smart Contract Security



Module 5: Smart Contract Security

Section 5.1: Introduction to Smart Contract Security

150 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- What is a Smart Contract?
- How Do Smart Contracts Work?
- Security Challenges of Smart Contracts
- Introduction to Smart Contract Security Vulnerabilities



What is a Smart Contract?

- The original blockchains were designed to create a decentralized financial system
 - Recorded transaction data on a distributed ledger
- Smart contract platforms dramatically expanded the capabilities of the blockchain
 - Turing-complete programs run “on top of” the blockchain
 - Leverage the benefits of blockchain technology
 - Implements a distributed and decentralized computer



Bitcoin, the original blockchain, was designed to implement a fully distributed and decentralized financial system that operated without relying on a centralized authority to maintain a trusted record of the transactions performed on the system. The blockchain created an immutable digital ledger that could be maintained by a distributed and decentralized network of mutually-distrusted nodes.

Blockchain technology provided a new way of accomplishing various business goals, and smart contract platforms dramatically expanded their capabilities. With smart contracts, it is possible to develop Turing-complete programs that run “on top of “ the blockchain, making it theoretically possible to implement any computer program as a smart contract. These programs take advantage of blockchain decentralization and other benefits to create a distributed and decentralized computer that is synchronized across all nodes in the blockchain network.

How do Smart Contracts Work?

- Smart contracts run “on top of” the blockchain
- What does this mean?
 - Executable code is embedded in blockchain transactions
 - Block producers organize transactions into blocks
 - Nodes validate transactions and blocks
 - Code is executed in virtual machines hosted by nodes
- Identical code run in identical environments should create identical results



Smart contracts are programs that are run “on top of” the blockchain. This provides smart contracts with all of the benefits of blockchain technology.

Smart contracts run on top of the blockchain by:

- **Executable Transactions:** Instead of simply recording transfers of value between blockchain accounts, transactions include executable code such as commands defining smart contract code or calls to smart contract functions.
- **Organized Transaction:** Blockchain protocols are designed to have block producers organize transactions into blocks to be added to the digital ledger. This creates a defined sequence in which the code contained within these transactions will be executed.
- **Transaction Validation:** Nodes in the blockchain network are responsible for validating the transactions in the blocks that they create and the blocks that they add to their copies of the digital ledger. This ensures that all smart contract code within a block is valid.

- **Virtual Machine Execution:** Smart contract platforms' software incorporates a virtual machine that creates identical, deterministic execution environments for smart contract code on various nodes. This helps to ensure that each node running the code gets the same result.

Smart contract platforms use the blockchain's digital ledger to organize and achieve consensus on the code to run. This code is then executed in identical, deterministic virtual machines to ensure that every node gets the same result from running the code. As a result, each node can independently track the state of the smart contract virtual machine, implementing a decentralized computer.

Security Challenges of Smart Contracts

- Blockchain systems provide significant benefits
- However, they also carry security risks
 - Turing-complete programs
 - Untrusted code execution
 - Immutable contract code
 - External integrations
 - Fragmented ecosystem
 - Immature platforms and languages



Smart contract platforms dramatically expand the capabilities of blockchain technology. The ability to run programs on top of the blockchain makes it possible to implement Decentralized Finance (DeFi) and other unique applications of blockchain.

However, while smart contracts provide valuable features, they also create security risks. Some of the security risks associated with smart contracts include:

- **Turing-Complete Programs:** Smart contracts are Turing-complete, meaning that any program that can be implemented on a computer could be written as a smart contract. This means that complex programs are running in a distributed and decentralized environment where they are uniquely vulnerable to attack.
- **Untrusted Code Execution:** Nodes in a smart contract platform must execute untrusted and potentially malicious code as they add transactions to the blockchain's digital ledger. While these transactions are run within an isolated virtual machine, there is the potential that an unknown vulnerability could allow malicious contracts to attack the node itself.
- **Immutable Contract Code:** Smart contracts are recorded on the blockchain's

digital ledger, meaning that they are immutable and impossible to update if they were not designed to do so. Software commonly contains bugs and vulnerabilities, and a poorly implemented smart contract may be impossible to update to fix exploitable vulnerabilities.

- **External Integrations:** Smart contracts run within the smart contract environment and can interact with one another; however, some smart contracts also interact with external systems and applications. This internal connectivity introduces new security risks as vulnerabilities in the smart contract or the external application could enable attacks against the other.
- **Fragmented Ecosystem:** Many smart contract platforms exist, many with their own programming languages and virtual machines. This fragmented ecosystem complicates smart contract security and vulnerability research as efforts are spread over many different platforms.
- **Immature Platforms and Languages:** Smart contracts are a relatively new technology with Ethereum, the oldest smart contract platform only launching in 2015. The relative youth of these platforms and their programming languages means that developers and security auditors lack experience with these platforms and languages, increasing the probability that vulnerabilities will be overlooked in smart contracts released on the blockchain.

Introduction to Smart Contract Vulnerabilities

- Smart contracts can contain a wide range of potential vulnerabilities
- These vulnerabilities can be broken into four classes
 - General programming vulnerabilities
 - Blockchain-specific vulnerabilities
 - Platform-specific vulnerabilities
 - Ethereum, etc.
 - Application-specific vulnerabilities
 - Decentralized Finance (DeFi)
 - Non-Fungible Tokens (NFTs)



Smart contracts are programs that can derive vulnerabilities from a variety of different sources. Some vulnerabilities could exist in any Turing-complete application, while others are unique to smart contracts' deployment environment or their particular use case.

This module will divide smart contract vulnerabilities into four main classes:

- **General Programming Vulnerabilities:** Some vulnerabilities could exist in any application, regardless of deployment environment. Integer overflows and underflows are an example of these vulnerabilities.
- **Blockchain-Specific Vulnerabilities:** Smart contracts are programs that run on top of the blockchain, rather than on a traditional computer. This unique deployment environment introduces some security risks.
- **Platform-Specific Vulnerabilities:** Different implementations of smart contract platforms can have smart contract vulnerabilities. This module will explore vulnerabilities specific to the Ethereum smart contract platform.

- **Application-Specific Vulnerabilities:** The various uses to which smart contracts are put can create application-specific vulnerabilities and risks. The risks associated with DeFi and NFT smart contracts will be explored in this module.

Summary

- What is a Smart Contract?
- How Do Smart Contracts Work?
- Security Challenges of Smart Contracts
- Introduction to Smart Contract Security Vulnerabilities



Module 5: Smart Contract Security

Section 5.2: General Programming Vulnerabilities



Overview

- Introduction to General Programming Vulnerabilities
- Arithmetic Vulnerabilities
- Decimal Precision
- Digital Signature Vulnerabilities
- External Dependencies
- Text Direction
- Unsafe Serialization



Introduction to General Programming Vulnerabilities

- Smart contracts are programs
 - Programs can have bugs, which can be exploitable vulnerabilities
- General programming vulnerabilities are risks that could exist in any program:
 - Arithmetic Vulnerabilities
 - Decimal Precision
 - Digital Signature Vulnerabilities
 - External Dependencies
 - Text Direction
 - Unsafe Serialization



Smart contracts are Turing-complete programs that happen to run on top of a blockchain platform. Programs, regardless of their execution environment, can contain bugs, and some of these bugs are vulnerabilities that could be exploited by an attacker.

General programming vulnerabilities are potential security risks that could exist in smart contracts simply because they are programs. Examples include:

- Arithmetic Vulnerabilities
- Decimal Precision
- Digital Signature Vulnerabilities
- External Dependencies
- Text Direction
- Unsafe Serialization

Arithmetic Vulnerabilities

160 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Arithmetic Vulnerabilities

- Arithmetic vulnerabilities arise from how computers store data
 - Fixed-sized variables
 - Signed and unsigned variables
- These vulnerabilities occur when the value stored in a variable falls outside the range that it can hold
 - Integer overflows and underflows
- This happens for various reasons
 - Computations
 - Unsafe typecasts



Programming languages can store various different numeric data types, such as integers, floats, etc. For integers, there are two main ways in which variables are differentiated:

- **Length:** Different integer types use different numbers of bits to store values. This allows developers to make tradeoffs based on storage efficiency vs. the range of values to be stored.
- **Signed vs Unsigned:** Signed integers can hold negative numbers and use their most significant bit to indicate whether a value is positive or negative. Unsigned integers can hold only positive numbers, and the most significant bit is part of their value. Signed integers can be below zero, while unsigned integers can be larger values that fit within a signed variable of the same size.

Arithmetic vulnerabilities occur when the value stored within a variable does not fall within the range of values that it can store. The two types are:

- **Integer Overflows:** The value is larger than can be stored in the variable. For example, the value 2^9 would overflow an eight-bit variable because it requires

nine bits of storage.

- **Integer Underflows:** The value is smaller than can be stored in the variable. For example, any negative value will underflow an unsigned variable.

Integer overflows and underflows can occur in a smart contract for various reasons:

- **Computations:** The result of a computation may not fit within the intended variable. For example, multiplication and addition can cause integer overflows, while subtraction can cause integer underflows.
- **Unsafe Typecasts:** Typecasting converts the variable where a value is stored from one type to another and can be problematic if the value does not fit in the new type. For example, downcasting (decreasing the variable length) and conversions between signed and unsigned types can be dangerous depending on the value stored. Upcasting, on the other hand, is safe because any value that fits in a variable will also fit in a larger variable of the same signedness.

Integer Underflow Example

```
1 function withdraw(uint _amount) {
2     require(balances[msg.sender] - _amount > 0);
3     msg.sender.transfer(_amount);
4     balances[msg.sender] -= _amount;
5 }
```

Source: <https://dasp.co/>

162 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Integer overflow and underflow vulnerabilities are especially dangerous in smart contracts because these contracts commonly work with transfers of value. An integer overflow or underflow vulnerability in a check that determines if a transfer is valid could allow an invalid transfer to go through.

For example, this code sample implements a withdraw function that allows a user to extract the value that they have stored within the contract. In theory, line 2 validates the transfer, line 3 performs it, and line 4 updates the contract's internal balance sheet.

In practice, line 2 will permit a transfer of almost any value that is not exactly equal to the amount that the user has in their account in the contract. The reason for this is that the result of the subtraction in line 2 will be stored in an unsigned variable, which cannot hold a negative value. Therefore, the only cases where the require statement will fail are if the result of the subtraction is exactly equal to zero.

This flaw could allow an attacker to withdraw a value of `_amount` that is greater than the value stored in their account. After passing the require statement, this withdrawal would succeed as long as the contract contains sufficient value. Any

excess value extracted by the attacker would be taken from other users' accounts or any value owned by the contract itself.

Case Study: PIZZA

- In December 2021, the PIZZA DeFi project was the victim of a \$5 million attack
- The attack exploited an integer overflow vulnerability in the eCurve smart contract
 - Created a large amount of Tripool tokens out of thin air
- These Tripool tokens were deposited into PIZZA
 - Attacker could then extract valuable tokens and drain value from the project

Source: <https://twitter.com/PizzaProFi/status/1468869822389768192>.

163 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In December 2021, the PIZZA DeFi protocol was a victim of an attack that exploited an integer overflow vulnerability but not one in the project's smart contract. The attacker exploited a vulnerability in another project and used it to steal \$5 million in tokens from PIZZA.

The vulnerability was actually in the eCurve smart contract, which is an EOSIO-based implementation of Curve. This integer overflow vulnerability allowed an attacker to mint a large number of Tripool tokens.

These Tripool tokens were then deposited into the PIZZA contract, which provided the attacker with the ability to withdraw other tokens from the contract in exchange. The attacker used the Tripool token deposits to extract the other tokens deposited within the PIZZA contract. This allowed them to drain approximately \$5 million worth of other tokens from the project.

Arithmetic Vulnerability Mitigations

- Arithmetic vulnerabilities occur when it is possible that a value stored within a variable could fall outside of the range of values that it could store
 - Caused by computations, unsafe typecasts, etc.
- These vulnerabilities can be avoided by
 - Validating values before performing operations
 - Avoiding unsafe type conversions
 - Avoiding dangerous operations when possible



Every integer data type has a fixed range of values that it can hold. This means that an integer overflow or underflow vulnerability is always a possibility if, for some reason, the value stored within a variable falls outside of the range of values that it can hold. This can happen due to computations, unsafe typecasting, and other issues.

Some ways to manage the risk of arithmetic vulnerabilities include:

- **Data Validation:** Before performing operations that could result in an integer overflow/underflow, check that the result will still fit within the target variable. For example, before multiplying, check that the product will not be too large.
- **Avoid Unsafe Type Conversions:** Whenever possible, don't typecast a value between variables of different types. However, this may not be possible if, for example, different external functions require different variable types. Before typecasting a value, verify that the result will be able to fit in the target variable type.
- **Avoid Dangerous Operations:** Integer overflows and underflows can be caused by even simple mathematical operations like addition, subtraction, and multiplication.

While avoiding these entirely is infeasible, their use should be minimized. For example, in the vulnerable code sample earlier, testing if `balances[msg.sender] > _amount` is safer than `balances[msg.sender] - _amount > 0` and produces the same result.

Decimal Precision



Introduction to Decimal Precision

- Solidity supports floating point numbers
 - These numbers have limited precision
 - The results of division may be rounded
- The sequence in which operations are performed can affect the result
 - Divide before multiply
 - Multiple before divide
- Rounding errors amplified by multiplication can have significant impacts



Solidity supports floating point numbers, meaning that it is not limited to integer data types. Like all programming languages, the size of a floating point variable is fixed, so Solidity can only support a certain number of digits after the decimal point. In the event that a division operation's fractional part does not fit within the supported digits, the value is rounded towards zero. This can decrease positive numbers and increase negative ones.

In most cases, the effects of rounding are negligible due to the size of the numbers involved. However, if division is performed as part of a sequence of calculations, then the effects of rounding could be amplified. For example, consider the case where we can support two digits after the decimal point and want to calculate the product of $2/3$ and 1000. Calculating the value of $2/3$ first and then multiplying produces $(2/3)*1000 = .66 * 1000 = 660$. In contrast, performing multiplication and then division results in $(2*1000)/3 = 2000/3 = 666.66$. Due to rounding, the results are significantly different.

The effects of rounding can be especially significant in DeFi contracts. As a result of decimal precision errors, a user performing a trade may receive too few or too many tokens in exchange.

Decimal Precision Example

```
1 function calcLiquidityShare(uint units, address token, address pool, address member) {
2     uint amount = iBEP20(token).balanceOf(pool);
3     uint totalSupply = iBEP20(pool).totalSupply();
4     return(amount.div(totalSupply)).mul(units);
5 }
```



This code sample implements a liquidity share calculation for a DeFi smart contract. Based on the type and number of tokens that a user deposits and the value of the corresponding liquidity pool, the function calculates the value of their share that could be withdrawn from the contract.

This calculation is based on three values: amount, totalSupply, and units. The calculation is performed as $(\text{amount}/\text{totalSupply}) * \text{units}$.

If the result of the division operation is rounded, then it is possible that the user will receive significantly less tokens than they should. The reason for this is that the division is followed by a multiplication by the number of tokens held by the user (units). If this is a large value, then the impact of the rounding is amplified, cheating the user out of their full liquidity share.

Case Study: Formation Fi

- In November 2021, the Formation Fi DeFi protocol was the victim of an attack
 - Attacker stole \$100,000 in tokens from the protocol
- The attacker exploited vulnerabilities in the protocol's calculation of token values
 - Inflated perceived value of the reward token
- This exploit took advantage of decimal precision issues when comparing token values

Source: <https://hacked.slowmist.io/en/>



In November 2021, the Formation Fi DeFi protocol was the victim of a flashloan attack. This type of attack inflates the perceived value of a token and then uses these overvalued tokens to drain value from a smart contract. In this case, the attacker was able to steal \$100,000 in tokens from the vulnerable smart contract.

Formation Fi's performed calculations of the perceived value of various tokens within its smart contract, which is dangerous and vulnerable to manipulation. In this case, the calculation also contained a decimal precision error when comparing the values of various tokens. Rounding errors caused the protocol to miscalculate the relative value of different tokens, enabling an attacker to extract more value from the protocol than they were due when claiming rewards. As a result, the attacker was able to drain \$100,000 in tokens from the protocol.

Decimal Precision Mitigations

- Decimal precision issues arise when the effects of rounding are amplified
 - This occurs when multiplication follows division
- Smart contracts should follow a divide-after-multiply code pattern
 - Rounding is only performed at the very end
 - Requires consideration of cross-function series of calculations
- However, this code pattern risks integer overflow vulnerabilities



Decimal precision issues arise when division is performed as part of a series of calculations. The impacts of rounding on a value are minimal unless division is followed by multiplication. In this case, the effects of the rounding can be amplified and significant, depending on the size of the other input to the multiplication.

Avoiding decimal precision issues requires designing calculations to follow a divide-after-multiply code pattern. If division is performed at the end of a series of calculations, then the effects of rounding on the final result are minimized.

However, this is not always as simple as looking at a single line of calculations within a smart contract. If a value is processed in many different locations, then a location that ends one sequence of calculations may be followed by a multiplication somewhere else that amplifies the impact of rounding errors. The overall flow of the smart contract must be scrutinized to ensure that division truly comes last whenever possible.

When following this code pattern, it is also important to consider the potential for other programming vulnerabilities. For example, performing several multiplications first may create risk of an integer overflow vulnerability when a developer is

attempting to avoid a decimal precision issue.

Digital Signature Vulnerabilities

170 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Digital Signature Vulnerabilities

- Digital signatures are a fundamental part of blockchain technology
 - Prove transaction integrity and authenticity
 - Smart contracts use digital signatures for the same reason
- Smart contracts use of digital signatures can go wrong in various ways
 - Failure to Validate
 - Cryptographic Errors
 - Malleable Signature



Digital signatures are one of the features that make blockchain technology possible. Blockchains are distributed, decentralized systems that need a means of validating that a transaction is authentic and has not been modified during its passage through the peer-to-peer network or storage by a node. Digital signatures provide this by guaranteeing that a transaction was created by someone with access to a blockchain account's private key.

Smart contracts, especially in the DeFi space, also make use of digital signatures to validate the authenticity and integrity of data. However, this use of digital signatures can go wrong in various ways, including:

- **Failure to Validate:** A digital signature only protects data authenticity and integrity if the recipient validates that it is correct. If a smart contract fails to validate that a digital signature matches the associated data and public key, then an attacker can submit forged data with fake, invalid digital signatures.
- **Cryptographic Errors:** Digital signature schemes are cryptographic algorithms, and these algorithms can be fragile with only a small error breaking their security. If the implementation of a digital signature algorithm has errors (such as the reuse of

single-use values), then this can invalidate the protections provided by the digital signatures.

- **Malleable Signatures:** A digital signature guarantees that someone with access to a particular private key generated a certain sequence of bits. If this sequence of bits can be interpreted in multiple different ways, then an attacker may be able to exploit this fact to reuse a legitimate digital signature generated by a user for another purpose. For example, is a chunk of serialized data could have multiple interpretations, then an attacker could use an existing signature to achieve a different purpose.

Case Study: Anyswap

- In July 2021, Anyswap was the victim of a hack
 - The attacker stole an estimated \$7.87 million in tokens
- The hack was enabled by a digital signature implementation error
 - Anyswap uses the Elliptic Curve Digital Signature Algorithm (ECDSA)
 - ECDSA signature calculations include a single-use value, K
- Anyswap reused the same value of K across multiple signatures
 - Created signatures that had the same value of R (part of the signature)
 - The attacker was able to calculate the private key used in the signatures

Source: <https://medium.com/multichainorg/anyswap-multichain-router-v3-exploit-statement-6833f1b7e6fb>

172 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In July 2021, the Anyswap protocol suffered a \$7.87 million hack. This hack was made possible by an error in how the protocol implemented the Elliptic Curve Digital Signature Algorithm (ECDSA).

Cryptographic algorithms commonly include single-use values that are designed to ensure that every ciphertext or digital signature is unique. In the case of ECDSA, this is a value called K. An ECDSA signature consists of two parts: R and S. If a blockchain account uses the same private key and K value when creating digital signatures, it will produce signatures that have identical values for R.

Anyswap's implementation of ECDSA caused the reuse of K values. This meant that multiple signatures created by the same account had the same R component, which made it easy to detect this reuse of K across multiple signatures.

The reuse of K is significant because it allows an attacker to recalculate the private key used to generate the digital signatures. This is accomplished by the following equation:

$$\text{private key} = (z1*s2 - z2*s1)/(r*(s1-s2))$$

Where z1 and z2 are the hashes of the two messages to be signed (see

<https://web.archive.org/web/20160308014317/http://www.nilsschneider.net/2013/01/28/recovering-bitcoin-private-keys.html>). With the private key associated with a user's account, the attacker could create transactions that drained value from that account. The attacker exploited this access to steal an estimated \$7.87 million worth of tokens.

Digital Signature Vulnerability Mitigations

- Digital signature errors can have significant impacts
 - Forged transactions
 - Exposed private keys
- Some methods for mitigating digital signature vulnerabilities include:
 - Use of trusted cryptographic libraries
 - Signature verification
 - Fixed input formats



Digital signature algorithms provide invaluable guarantees about the integrity and authenticity of data. However, these guarantees only apply if the digital signature algorithms are implemented and used properly. Some potential effects of digital signature errors include:

- Forged Transactions:
- Exposed Private Keys:

To achieve the full benefits of digital signatures, smart contracts need to use them properly. Some best practices for avoiding digital signature issues include:

- **Use of Trusted Libraries:** Cryptographic algorithms are fragile, and small errors can break them entirely. When including digital signatures in a smart contract, use trusted library implementations rather than creating a custom implementation.
- **Signature Verification:** Digital signatures are only useful if the digital signature is verified. Smart contracts should ensure that signature verification is performed and cannot be bypassed.

- **Fixed Input Formats:** If the data protected by a digital signature can be interpreted in different ways, then an attacker can reuse existing signatures for different purposes. When serializing data, ensure that it has only one potential interpretation.

External Dependencies



Introduction to External Dependencies

- Most software has external dependencies
 - Libraries, DLLs, etc. that implement common, shared functionality
- Smart contracts can also have shared library contracts
- Insecure library contracts create security risks
 - Denial of Service attacks
 - Inherited vulnerabilities



Code reuse is a development best practice. If a high-quality library exists for a particular function, then it is likely to offer a more functional and secure implementation of a function than a developer could build in-house (since developers aren't experts in all subjects). Also, calling a library function is much faster and cheaper than writing and testing the required functionality from scratch.

The use of external libraries is common for desktop applications, and they can be used in smart contracts as well. Smart contracts can interact with one another, so a function from one contract can call publicly-accessible functions within another. This makes it possible to deploy a library contract that client contracts can use to access key, shared functionality.

While this design pattern is efficient, it also creates security risks. Some of these include:

- **Denial of Service Attacks:** Client contracts are dependent on the library contract to implement key functionality. If the library contract becomes unavailable – due to a call to a self-destruct function – then the client contracts may no longer be able to do their jobs.

- **Inherited Vulnerabilities:** Client contracts will call the library contract and execute the code that it contains. If the code that a client contract calls contains a vulnerability, then an attacker may be able to exploit this vulnerability to attack the client contract as well.

Case Study: Parity Wallet

- Parity wallet was a smart contract-based wallet
 - Published a contract that users could launch and use as a wallet
- Parity wallets used a shared library contract
 - User wallets called its functions to implement crucial functionality
- An attacker exploited an access control vulnerability to take over and self-destruct this library contract
 - User wallets no longer had ability to perform transactions
 - 513,774.16 ETH plus other tokens in 587 wallets lost forever

Source: <https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>

176 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Parity wallet implemented a blockchain wallet as a smart contract. Users could launch a copy of the Parity wallet contract and claim ownership of it. They could then use this contract to receive, store, and send transactions.

The Parity wallet was designed to use a shared library contract to implement the core functionality of the smart contract-based wallet. The user wallet contracts were relatively lightweight and called functions implemented in the library contract to perform various actions. This made the design more efficient and made it easier to implement functionality since only the library contract needed to be updated.

An attacker exploited an access control vulnerability in the library contract that allowed them to claim ownership of that contract. Using their new permissions, the attacker was able to call the self-destruct function within the contract, rendering it inaccessible to user wallets.

Without the shared library contract, the user wallet contracts lacked the ability to perform transactions and could not be updated to point to a new library contract. As a result, all tokens stored in affected wallets was lost forever. At the time, there were 587 Parity wallet users and these wallets contained 513,774.16 ETH plus various

other tokens.

External Dependency Mitigations

- Using external library contracts is a good idea
 - Makes contracts more usable and updatable
 - Minimizes bloat on the blockchain
- The security of library contracts is of paramount importance
 - Access management
 - Ability to update library contract address
 - Vulnerability management



The use of external library code is a good idea in any context. On a desktop computer, the fact that shared libraries implement important shared functionality makes applications more efficient and can improve the overall quality of code.

Shared libraries are especially important on the blockchain due to the immutability of the blockchain's distributed ledger. Updating contracts can be complex, so having only a single contract to change when modifying library functions is more efficient. Also, blockchain immutability means that nothing can be deleted from the ledger, so minimizing blockchain bloat, in the form of repeated code, is also a good idea.

Centralizing key functionality within an external contract does create some risks. Ways of managing these risks include:

- **Access Management:** The library contract has a great deal of power and can have serious impacts if deleted or attacked. This contract should have strong access management built into the code and support by multi-signature wallets.
- **Address Updates:** The address of the library contract where client contracts look to for shared code should be updatable from within the client contract. This

protects against cases like the Parity wallet hack where, without the library contract, the client contracts became unusable.

- **Vulnerability Management:** Vulnerabilities within library contracts can have significant impacts because they affect many client contracts. These contracts should undergo in-depth security analysis to identify and remediate as many vulnerabilities as possible before release. Code review should be ongoing post-release and check for new vulnerabilities as they are discovered.

Text Direction

178 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Text Direction Vulnerabilities

- Different languages write text in different directions
 - English is left-to-right
 - Arabic is right-to-left
- Computers support both text directions
 - Non-printable control characters are used to indicate changes in text direction
- Reversals in text direction can create deceptive smart contract source code



Different languages use different conventions for the direction in which text is written. While English text is written left-to-right, Arabic text goes from right-to-left, and other languages use columns rather than rows of text.

Computers support both left-to-right and right-to-left text within a document. Switches between different text directions are accomplished via non-printable characters. This makes it possible for text to switch direction at any time, which is useful for quoting text in another language in the middle of a document, etc.

However, the ability to switch the direction of text can also be abused by malicious smart contract developers to create deceptive and misleading source code. Smart contracts are often open source, which allows potential users to review the source code if they wish to look for potential vulnerabilities, malicious functionality, etc. Since this code review is typically performed within applications that obey text direction control characters, a smart contract developer can make code appear to do something different than it actually does.

Text Direction Example

```
1 contract GuessTheNumber {
2   uint _secretNumber;
3   address payable _owner;
4   event success(string);
5   event wrongNumber(string);
6
7   constructor(uint secretNumber) payable public {
8     require(secretNumber <= 10);
9     _secretNumber = secretNumber;
10    _owner = msg.sender;
11  }
12
13  function getValue() view public returns (uint) {
14    return address(this).balance;
15  }
16
17  function guess(uint n) payable public {
18    require(msg.value == 1 ether);
19
20    uint p = address(this).balance;
21    checkAndTransferPrize("The prize/p , n/guessed number"/
22    /*The user who should benefit */,/msg.sender);
23  }
24
25  function checkAndTransferPrize(uint p, uint n, address payable guesser) internal returns(bool) {
26    if(n == _secretNumber) {
27      guesser.transfer(p);
28      emit success("You guessed the correct number!");
29    }
30    else {
31      emit wrongNumber("You've made an incorrect guess!");
32    }
33  }
34
35  function kill() public {
36    require(msg.sender == _owner);
37    selfdestruct(_owner);
38  }
39 }
```

Source: <https://skylightcyber.com/2019/05/12/ethereum-smart-contracts-exploitation-using-right-to-left-override-character/>

180 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The code sample shown implements a smart contract-based guessing game. The smart contract contains a secret number and will pay a reward if the user correctly guesses it. Looking at the smart contract code, the value of `_secretNumber` is limited to a value less than 10 (line 8), which means that a player has a 1 in 10 chance of getting it right. The `guess` and `checkAndTransferPrize` functions implement the actual guessing game. A player can call the `guess` function, which verifies that they have paid at least 1 ETH to play (line 18). If so, it determines the current balance of the contract and passes this amount, the guessed value, and the guesser's address to the `checkAndTransferPrize` function.

In line 26, the `checkAndTransferPrize` function checks if the argument `n` equals the secret number. If so, the contract transfers the value of `p` to the player and prints a success message. Otherwise, the contract emits a message indicating that the guess is incorrect.

This contract seems relatively straightforward, but it is actually unwinnable. The problem exists at line 21 in the call to `checkAndTransferPrize`. It appears that this passes the prize value as `p` and the guess as `n` to the `checkAndTransferPrize` function. However, text direction control characters in the two comments just make it look this

way.

In reality, `p` in the `checkAndTransferPrize` function is the guess, while `n` is the prize amount. Since `n` is the value checked against `_secretNumber`, a contract with a balance greater than 10 will be unwinnable since `_secretNumber` is less than or equal to 10.

This contract uses text direction control characters to defeat code review. While the contract looks fair and potentially winnable, it can easily be made unwinnable and the user's guess has no impact on whether or not they win.

Text Direction Mitigation

- Text direction control characters are unprintable and can make a contract's apparent functionality different than the reality
- While these control characters are unprintable, they still exist within the contract's code
 - Scanning the code for these characters can reveal the attempted deception



The malicious sample source code used unprintable text direction control characters to obscure the true functionality of the contract. While it looked like a winnable game, it actually was completely unfair and a scam.

This form of deception relies on unprintable characters within the source code of the contract. These characters cannot be detected via a visual review of the smart contract's code but have a significant impact on the interpretation of the code.

While the text direction control characters are unprintable, they still are present within the smart contract's code. Scanning the contract for these characters can reveal the attempted deception.

However, the presence of these characters within the code does not automatically mean that it is deceptive. For example, a developer may make comments in a language that is printed right-to-left, which requires changes in text direction from the left-to-right Solidity commands. If these control characters are identified, they need to be analyzed to determine if they have a potential impact on the interpretation of the smart contract's code.

Unsafe Serialization

182 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Unsafe Serialization

- Serialization allows data stored in multiple variables to be packed into a string of bits
 - Used for data transmission or storage
 - Recipient unpacks the data based on known structure
- Multiple valid interpretations may exist for serialized data
 - Variable-length arguments
- Digital signatures authenticating one version can be reused for another



Data serialization is designed to make a structure or collection of variables easier to transmit or store. Serialization converts the set of values into a single string of bits. After being transmitted or retrieved from storage, the recipient can reconstruct the original set of variables based on knowledge of the underlying structure.

Serialization is a useful tool, but it can be problematic when applied to certain types of variables. For example, data structures that contain variable-length fields can be dangerous if the length of the fields is not clearly defined within the serialized data. If a structure has two adjacent arrays and flattens them as part of the packing process (like Ethereum's `abi.encodePacked` does), then it can be difficult to determine where the boundaries of the two arrays actually are.

An attacker can take advantage of serialized data with multiple interpretations, especially when that data is digitally signed. A digital signature only validates that a series of bits has not been modified, it provides no guarantees about how that data is interpreted. If the same series of bits can be interpreted in multiple different ways, an attacker can exploit this fact by submitting data with a valid digital signature to a system that will interpret it differently than intended.

Unsafe Serialization Example

```
1 contract AccessControl {
2   using ECDSA for bytes32;
3   mapping(address => bool) isAdmin;
4   mapping(address => bool) isRegularUser;
5   // Add admins and regular users.
6   function addUsers(address[] calldata admins, address[] calldata regularUsers, bytes calldata signature) external
7   {
8     if (!isAdmin[msg.sender]) {
9       // Allow calls to be relayed with an admin's signature.
10      bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
11      address signer = hash.toEthSignedMessageHash().recover(signature);
12      require(isAdmin[signer], "Only admins can add users.");
13    }
14    for (uint256 i = 0; i < admins.length; i++) {
15      isAdmin[admins[i]] = true;
16    }
17    for (uint256 i = 0; i < regularUsers.length; i++) {
18      isRegularUser[regularUsers[i]] = true;
19    }
20  }
21 }
```

Source: <https://medium.com/swlh/new-smart-contract-weakness-hash-collisions-with-multiple-variable-length-arguments-dc7b9c84e493>

184 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample contains an `addUsers` function at line 6 that adds new user accounts to a system. It requires administrator-level permissions to execute, so the function checks if the account calling the function is an administrator account (using the `isAdmin` function) or if the message is digitally signed by an administrator.

This sample contains an unsafe serialization vulnerability in line 10. It calculates the Keccak256 hash of a list of administrator and regular user accounts after serializing that data with `abi.encodePacked`.

The issue is that both `admins` and `regularUsers` are arrays, and `abi.encodePacked` flattens these arrays as part of the serialization process. For example, the following two lines of code would return the same hash:

```
abi.encodePacked([admin1,admin2],[user1,user2,user3])
abi.encodePacked([admin1,admin2,user1],[user2,user3])
```

In the first line of code, `user1` is listed as a user-level account. In the second, `user1` is granted administrator-level permissions. If a valid administrator signed the serialized data in the first example (with `user1` as a regular user), then an attacker could submit the signature and the second example (with `user1` as an administrator) to provide `user1` with administrator-level permissions.

Case Study: Superfluid

- In February 2022, Superfluid was the victim of a \$13 million hack
- The attacker took advantage of unsafe serialization in the protocol
 - Arguments to Superfluid agreements are serialized
 - This serialized data included a context value that begins as a placeholder
- The attacker called the callAgreement function with a malicious context
 - This function added its own ctx variable and passed the arguments to the target
 - When deserializing, the attacker context was used and the legitimate one was ignored as superfluous data
 - Malicious context involved transfer of tokens to attacker from legitimate user

Source: <https://medium.com/superfluid-blog/08-02-22-exploit-post-mortem-15ff9c97cdd>

185 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The February 2022 hack of the Superfluid protocol was an example of an exploitation of an unsafe serialization vulnerability. Superfluid’s callAgreement function acts as a handler for its “agreements”, which perform various functions. Data sent to an agreement is sent in a serialized format, which is unpacked by the recipient.

Included in the arguments to an agreement is a variable named ctx, which provides contextual data that can be shared across multiple agreements. This variable is defined as a placeholder by Superfluid’s callAgreement function before data is serialized and sent to the first agreement.

The Superfluid attacker called callAgreement with data that included a malicious version of the ctx variable. When the callAgreement function prepared to serialize the data, it appended its own, empty version of ctx at the end of the data.

When this data was deserialized by the agreement, the malicious ctx variable appeared before the legitimate one. As a result, the agreement accepted and used the malicious one and discarded the legitimate one as excess data. While this agreement had the ability to check the validity of the arguments sent to it based on their hash, it did not do so because the data came from a trusted source

(callAgreement).

The malicious context provided by the attacker included a transfer of tokens from a legitimate user to the attacker's account. By exploiting this unsafe serialization vulnerability, the attacker was able to steal tokens worth an estimated \$13 million from Superfluid users.

Unsafe Serialization Mitigations

- Unsafe serialization vulnerabilities exist when serialized data has multiple interpretations
 - Typically caused by variable length inputs
- Mitigations for unsafe serialization include:
 - Using fixed-size inputs
 - Explicitly defining input lengths
 - Avoiding array flattening
 - Validating data before deserialization
 - Performing complete matching



Serialization becomes dangerous when serialized data can have multiple different interpretations. If serialized data has variable sized inputs, then different interpretations of the same data can produce very different results.

Serialization is a useful tool, but it needs to be used safely and correctly. Best practices for implementing safe data serialization include:

- **Use of Fixed-Size Inputs:** Using fixed-size inputs whenever possible decreases the potential risks of serialization. These inputs cannot be interpreted in different ways by the recipient, reducing the probability of error.
- **Explicitly Defining Input Lengths:** If serialized data has variable-length inputs, the lengths of these inputs should be explicitly defined. For example, an array within the data should be preceded by a variable that specifies the length of the array.
- **Avoiding Array Flattening:** When arrays are flattened, the boundaries between variables becomes fuzzy because elements of one are right next to elements of another. If array boundaries are clearly defined, then the serialized data has a clear interpretation.

- **Validating Data Before Deserialization:** In the case of the Superfluid hack, code existed to validate the serialized data before unpacking it, but this code was not executed. Smart contracts should always validate data integrity and authenticity before unpacking it.
- **Performing Complete Matching:** In the Superfluid hack, the real context variable was ignored because a fake version preceded it. If excess data exists after deserialization, a smart contract should raise an error rather than just discarding the excess.

Summary

- Introduction to General Programming Vulnerabilities
- Arithmetic Vulnerabilities
- Decimal Precision
- Digital Signature Vulnerabilities
- External Dependencies
- Text Direction
- Unsafe Serialization



Module 5: Smart Contract Security

Section 5.3: Blockchain-Specific Vulnerabilities



Overview

- Introduction to Blockchain-Specific Vulnerabilities
- Access Control
- Denial of Service
- Frontrunning
- Rollback Attacks
- Timestamp Dependence
- Weak Randomness



Introduction to Blockchain-Specific Vulnerabilities

- Smart contracts are programs that run on top of the blockchain
 - Very different execution environment from traditional computers
 - These differences can create potential vulnerabilities
- Examples of blockchain-specific smart contract vulnerabilities include:
 - Access Control
 - Denial of Service
 - Frontrunning
 - Rollback Attacks
 - Timestamp Dependence
 - Weak Randomness



Smart contracts run in a very different environment than traditional computer programs. Instead of running on a computer, they execute in a decentralized set of virtual machines with the instructions to be executed organized into transactions on the digital ledger. The underlying blockchain technology that plays host to smart contracts creates new potential security threats that do not exist or pose the same threat in traditional environments.

Some examples of security risks that smart contracts face due to the position running on top of a blockchain include:

- Access Control
- Denial of Service
- Frontrunning
- Rollback Attacks
- Timestamp Dependence

- Weak Randomness

Access Control Vulnerabilities

191 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Access Control Vulnerabilities

- Anyone can create a blockchain account and perform transactions
 - This includes interacting with public functions of smart contracts
- Smart contracts have permissioned, private functionality
 - Value withdrawals, self-destruction, etc.
- Access to these privileged functions must be protected
 - Otherwise an attacker may be able to access these functions



Accessibility is a core tenet of the blockchain ethos. The original blockchains were designed to allow anyone to create an account, and these accounts were anonymous to ensure privacy and true global accessibility. Blockchains also lacked built-in distinctions between accounts that determined what they were allowed to do, so any account can perform transactions on the blockchain as long as they are valid.

Smart contract platforms follow the same rules with any account being permitted to interact with any other account, including those hosting smart contract code. However, while anyone can interact with a smart contract, many smart contracts have privileged functionality. For example, while a contract may accept deposits from anyone, it wants to restrict who can extract value from the contract and under what conditions they may do so.

This balance between open accessibility and privileged functionality makes it necessary for smart contracts to implement internal access controls that govern a user's ability to access and execute functions within the smart contract. These access controls must be securely designed and implemented to ensure that they effectively protect privileged functionality. Access control vulnerabilities could be exploited by an attacker to attack the smart contract and its users.

Access Control Vulnerabilities

```
1 function initContract() public {  
2     owner = msg.sender;  
3 }
```



The `initContract` function shown is a common function for claiming ownership within a smart contract. Smart contract functions can be labeled as only accessible to the contract's owner (or other tags). To enforce these access controls, the smart contract needs to know who the owner is, which is commonly tracked based on the owner's account address.

While the owner's address could be hardcoded into the contract, this is not ideal. A better approach is the one shown above, which includes a function that is called after the launch of the contract. This function assigns ownership to the address of the account that called it, which presumably belongs to the actual contract owner.

The issue here is that the `initContract` function does not work as intended. A secure `initContract` function will assign ownership to the first person that calls it. In contrast, this one assigns ownership to the last person who calls this.

The reason for this is that the function includes no test to verify that this is the first time that the function has been called. While the true contract owner might have called this function and initially claimed ownership, anyone can call this function again after them. Due to the missing check, these later calls would be accepted and

assign ownership of the contract to the address of the caller. This attacker could then use these new permissions to access owner-only functions, which include dangerous and privileged functionality like draining value from the contract or self-destructing it.

Case Study: Poly Network

- The Poly Network was the most expensive DeFi hack to date
 - Attackers stole over \$611 million in tokens
- The attacker bypassed access controls by exploiting inter-function relationships
 - The `verifyHeaderAndExecuteTx` function provided access to the private `PutCurEpochConPubKeyBytes` function
 - This function had control over the contract's "keeper" role
 - With keeper privileges, the attacker could steal value from the contract

Source: <https://slowmist.medium.com/the-root-cause-of-poly-network-being-hacked-ec2ee1b0c68f>

194 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

At the time, the Poly Network hack was the biggest and most expensive DeFi hack to date. The attacker was able to exploit access control vulnerabilities to steal over \$611 million in tokens from the project.

In this case, the vulnerability exploited by the attacker was not as straightforward as the vulnerable sample code in the previous slide. The attacker took advantage of relationships between functions to gain access to privileged functionality.

The `PutCurEpochConPubKeyBytes` function had the ability to update the role of keeper (a privileged role within the contract) to point to a new account address. This function was properly configured as private to prevent unauthorized changes to this role.

However, the `verifyHeaderAndExecuteTx` function was callable by an external party and has the ability to execute the `PutCurEpochConPubKeyBytes` function. The attacker took advantage of this fact to change the keeper role to point to their account.

With the keeper role, the attacker had privileged access within the smart contract.

This included the ability to drain over \$611 million in tokens from the project.

Access Control Vulnerability Mitigations

- The blockchain has no built-in controls to restrict access to smart contracts and their functions
 - Smart contracts must implement these controls internally to protect privileged functions
- Access control best practices for smart contracts include:
 - Use of private functions by default
 - Modularize functionality
 - Multi-signature wallets for privileged accounts
 - Inter-function relationship analysis



Blockchain systems allow anyone to create an account and perform transactions that interact with smart contracts. Since this makes all smart contracts publicly accessible, smart contracts must manage their own access controls. This includes identifying and managing access to functions that include dangerous and privileged functionality. Some best practices for implementing access control for smart contracts include:

- **Default Private Functions:** “Secure by default” is always a better design pattern than starting in an insecure state and adding in access and security controls later. Defining all functions as private by default and then exposing functions that need to be accessible decreases the probability that an oversight will leave a sensitive function exposed.
- **Modularize Functionality:** Breaking functionality into smaller pieces enables more granular security controls. Privileged functionality can be marked as private and only accessible after appropriate validation of requests from publicly accessible functions.
- **Multi-Signature Wallets:** Access controls are designed to allow only certain accounts to access privileged functionality within a smart contract, making these

accounts a prime target for attack. The accounts should be protected using multi-signature wallets to make it difficult for an attacker or a malicious insider with access to the account's private key to abuse its access and permissions.

- **Inter-Function Relationships:** A private, protected function within a smart contract may be callable from other functions that have less stringent access controls. The relationships between various smart contract functions should be analyzed to determine if a relationship allows access controls to be bypassed or undermined.

Denial of Service



Introduction to Denial of Service Vulnerabilities

- Denial of Service vulnerabilities can occur at lower levels of the blockchain ecosystem
 - These vulnerabilities can impact smart contracts
- Examples of DoS attacks that can affect smart contracts include:
 - DDoS against transaction creators
 - Traffic dropped at network level
 - Transactions excluded from blocks



Smart contract platforms allow programs to run on top of the blockchain. This means that these smart contracts are dependent on the underlying blockchain infrastructure. Denial of Service (DoS) attacks against the blockchain can impact the availability and usability of the smart contracts hosted on it.

Smart contracts can be impacted by various DoS attacks. Some examples include:

- **Transaction Creator DDoS:** Smart contract functions are executed within blockchain transactions. If an attacker can prevent a user from creating or broadcasting their transaction, via a DDoS attack or similar, then they will not be able to execute the smart contract code.
- **Dropped Network Traffic:** The blockchain's peer-to-peer network carries transactions from their creators to the block producers that will create the blocks that add these transactions to the digital ledger. If an attacker can block these communications (using an eclipse/routing attack), then they can delay or prevent the transaction from being added to the blockchain.
- **Transaction Exclusion from Blocks:** Block producers have full control over the

contents of the blocks that they create, including which transactions are included and how they are ordered. A malicious block creator could refuse to include a transaction within a block that they create, delaying its addition to the blockchain's digital ledger.

Case Study: Sia

- Sia is a blockchain-based project designed to implement decentralized data storage
- In June 2021, it was the target of a DDoS attack
 - Impacted approximately a quarter of the project's network hosts and storage providers
- The attack did not impact the protocol's file storage capabilities
 - It did impact availability though
 - Approximately 30% of host connections were affected

Sources: <https://hacked.slowmist.io/en/>
<https://sia.tech/ddos2021>

198 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Sia implements decentralized file storage using a blockchain-based system. By decentralizing data storage, systems like Sia can improve resilience, accessibility, and censorship-resistance.

In June 2021, the Sia protocol was the victim of a Distributed Denial of Service (DDoS) attack. The attackers targeted roughly 25% of the protocol's storage providers and network hosts

The attackers did not succeed in affecting Sia's ability to provide file storage. However, it did have a marked impact on the availability and performance of Sia's services. Approximately 30% of Sia host connections were impacted by the DDoS attack.

Denial of Service Mitigation

- Many Denial of Service attacks against smart contracts target the underlying infrastructure
 - Difficult or impossible to mitigate within a contract
 - Would require revisions to the blockchain protocol to fix
- Some of these DoS threats can be mitigated in various ways, such as:
 - Deploying anti-DDoS protections
 - Expanding a node's peer group
 - Participating in block production



Denial of Service (DoS) attacks that target the underlying blockchain infrastructure are difficult to protect against. Smart contracts rely on this infrastructure and have no means of affecting it. Also, many of the potential DoS threats exploit the design of the blockchain protocol and would require revisions to this protocol to prevent.

That said, nodes can take actions to mitigate the risk of some of these threats, including:

- **DDoS Mitigation Solutions:** An attacker may perform a DDoS attack against a node to prevent it from publishing transactions or to disrupt the transmission of a transaction through the peer-to-peer network. Deploying DDoS protections can help protect against these threats.
- **Changing Peer Groups:** An attacker may attempt to delay or prevent a transaction from being added to the ledger using an eclipse/routing attack. Connecting to additional or more distributed peers can help a node to mitigate this threat.
- **Block Production:** A block producer controls the transactions that are included in their blocks, including the ability to include or exclude transactions. Participating in

consensus allows a node to ensure that its transactions make it onto the blockchain's ledger whenever that node is selected to create a block.

Frontrunning

200 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Frontrunning

- Transactions pass through a multi-step process before being added to the ledger
 - This includes being broadcast to the entire blockchain network
- Block producers organize transactions into blocks based on transaction fees
 - Transaction creators can pay for priority
- Later transactions may be added and processed before earlier ones



Transactions are added to the blockchain as part of blocks, which are created by block producers. This means that, before a transaction can be included in the blockchain's immutable ledger, a block producer needs to see it. To make this possible, blockchain protocols have transactions broadcast to all of the nodes within the blockchain network. These nodes put these transactions into pools of unused transactions to be included in later blocks.

When creating a block, the block producer gets to decide which transactions are included and how they are organized within the block. The most common way of doing so is based on transaction fees. All transactions pay a minimum fee, but transaction creators can pay extra fees if desired. Block producers can (and are expected to) organize transactions into blocks based on these fees to maximize the rewards that they earn from block production.

Transactions are publicly broadcast before being added to blocks and are organized into blocks based on fees, not the order in which they were received. This makes frontrunning attacks possible where an attacker observes a transaction and creates their own based upon it.

If this later transaction has a higher fee and is received in time, it is likely to be processed first, which can benefit the attacker. For example, an attacker could win a contest by frontrunning a transaction containing a valid solution or take advantage of price slippage in a decentralized exchange (DEX) by frontrunning another user's buy or sell of an asset.

Frontrunning Example

```
1 contract EthTxOrderDependenceMinimal {
2   address public owner;
3   bool public claimed;
4   uint public reward;
5
6   function EthTxOrderDependenceMinimal() public {
7     owner = msg.sender;
8   }
9
10  function setReward() public payable {
11    require (!claimed);
12
13    require(msg.sender == owner);
14    owner.transfer(reward);
15    reward = msg.value;
16  }
17
18  function claimReward(uint256 submission) {
19    require (!claimed);
20    require(submission < 10);
21
22    msg.sender.transfer(reward);
23    claimed = true;
24  }
25 }
```

Source: <https://swcregistry.io/docs/SWC-114>

202 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample is a smart contract that is vulnerable to frontrunning also known as a Transaction Order Dependence vulnerability. The contract implements a contest where the first user with a valid submission wins the prize. In this case, a valid submission is any value less than 10 based on line 20 in the code.

This sort of contract that implements a “first come first served” approach to distributing prizes is inherently vulnerable to frontrunning attacks. Due to the design of the blockchain, the first transaction with a valid submission to be created is not necessarily the first transaction to be added to the digital ledger and processed. An attacker willing to pay a higher transaction fee can have their transaction processed before other contenders.

In this case, the complexity of determining a valid solution to the problem is low since any call to claimReward with a submission argument less than 10 is valid. However, other contracts may implement more complex puzzles. In those cases, the ability to monitor pending transactions for a valid submission and then frontrun it is probably easier than solving the actual puzzle and achieves the same result.

Case Study: DODO and Punk Protocol

- The DODO and Punk Protocol smart contracts were victims of flashloan attacks
 - Attacker manipulated perceived value of assets to steal from the protocol
- Frontrunning bots observed the exploit transaction and frontrun them
 - Enabled the bots to steal value from the protocol before the attackers could
- Bot owners returned most of the stolen tokens
 - Decreased the impact of the attack

Sources: <https://medium.com/punkprotocol/punk-finance-fair-launch-incident-report-984d9e340eb>

<https://dodoexhelp.zendesk.com/hc/en-us/articles/900004851126-Important-update-regarding-recent-events-on-DODO>

203 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Frontrunning attacks are common, especially in the DeFi space. For some DeFi transactions, advance knowledge and the ability to frontrun the transaction can provide a guaranteed profit for the frontrunner. For this reason, multiple parties have deployed frontrunning bots that scan the blockchain for frontrunnable transactions and automatically create and transmit their own transactions to take advantage of these opportunities.

While frontrunning bots can be a nuisance to legitimate traders who lose money due to them, they are not always a bad thing. In the case of the DODO DEX and Punk Protocol hacks, these bots actually decreased the impact of an attack.

Both of these protocols were targeted by flashloan exploits in which an attacker manipulated the perceived price of a token to drain value from the protocol. These exploit transactions were observed and frontrun by bots that performed the exploit before the attacker could. As a result, the bots drained a significant portion of the value in the vulnerable protocols before the attacker's transaction was processed.

This was a benefit because, in both cases, bot operators elected to return most of the extracted funds. This dramatically decreased the cost of the attack to the impacted

protocols and their users.

Frontrunning Mitigations

- Smart contracts are vulnerable to frontrunning if the order in which transactions are processed affects the result
 - First come first served
 - Internal state changes as a result of transactions
- Eliminating transaction order dependence can help to mitigate a contract's vulnerability to frontrunning
 - This may not always be possible



If a smart contract's operations are not dependent on the order in which transactions are received and processed, then it is not vulnerable to frontrunning. For example, a contract that takes some action after a set of users have all made a deposit likely doesn't care in which order the deposits are made.

However, contracts where transaction ordering does matter are potentially vulnerable to frontrunning. This includes contracts that have a "first come first served" policy or ones where the internal state of the contract changes as a result of transactions and affects future ones. For example, buys and sells of a token on a decentralized exchange impact the perceived value of the token, impacting future trades.

Ideally, contracts can eliminate their vulnerability to frontrunning by eliminating their dependence on transaction ordering. However, this is not always possible. In those cases, the contract can either accept the risk of frontrunning or implement patches to make it less profitable and effective, such as randomly picking a winner from a set of transactions rather than operating under a first-come-first-served policy.

Rollback Attacks

205 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Rollback Attacks

- Blockchain transactions often operate under an “all or nothing” model
 - If exceptions cannot be or are not handled, then the entire transaction fails
 - The transaction state rolls back as if it never happened
- An attacker can take advantage of this “all or nothing” approach to transactions
 - Perform an action and test for success
 - Revert the transaction upon an unfavorable result



The blockchain is designed to be a decentralized system where all nodes are in consensus on the current state of the digital ledger and – in the case of smart contract platforms – the execution state of the “world computer”. To accomplish this, smart contract platforms use blockchain technology to achieve consensus on the code to run and execute it within identical, deterministic environments.

To avoid issues, smart contract platforms must also have a process in place for error handling. If something goes wrong, all nodes in the network should react in the same way to protect their shared state. In most cases, smart contract platforms will completely roll back transactions that contain unhandled errors. By restoring execution state to the point before a problematic transaction started, the platform eliminates any undesirable effects of that transaction.

However, this approach to error handling could also potentially be exploited by an attacker. When performing a high-risk transaction (such as betting in a smart contract-based casino game), a player could make a bet and test for a desirable result. If the play was not a winner, the attacker could then revert the transaction (using a false assert, thrown exception, etc.), causing the transaction to be rolled back as if it never happened.

Case Study: Betdice, EOSMax, Tobet

- In December 2018, multiple EOSIO-based gambling apps were targeted by a rollback attack
 - Betdice, EOSMax, Tobet, etc.
- The attackers exploited EOSIO's blacklist feature
 - Bet transactions were sent from an address blacklisted by block producer
 - Servers associated with the project processed the transaction and created a reward transaction
 - Attacker's transaction was rejected based on the blacklist
 - Reward transaction was valid and recorded on the blockchain

Source: <https://slowmist.medium.com/roll-back-attack-about-blacklist-in-eos-adf53edd8d69>

207 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

On December 19, 2018, an attacker performed a rollback attack against multiple gambling apps hosted on the EOSIO blockchain. These included Betdice, EOSMax, and Tobet among others.

This rollback attack exploited a couple of features of the EOSIO blockchain. One is that the block producer could maintain a blacklist of accounts from which it would automatically reject transactions, even if they were otherwise valid. Another is that servers associated with a project might process and respond to a transaction before it was recorded on the blockchain.

The attacker took advantage of these features by sending a transaction to the gambling app from a blacklisted address. When the node servers associated with the game saw the transaction, they processed it, and, in the event of a winning bet, created a reveal transaction that paid a reward to the player.

When these two transactions reached the block producer, the initial transaction was rejected because it originated from an address on the block producer's blacklist. However, the reveal transaction came from a legitimate address associated with the game and was accepted and recorded on the blockchain.

By performing a rollback attack, the attacker never risked losing a bet because betting transactions would always be rejected by the block producer. However, in the event of a win, a separate transaction paid the reward, which meant that it could still be recorded on the blockchain even if the betting transaction was rejected.

Rollback Attack Mitigations

- Rolling back failed transactions is a built-in feature of blockchain technology
 - Helps to ensure that the shared computer is in a valid state
 - Also makes rollback attacks possible
- In some cases, rollback attacks can be prevented by a smart contract
 - Some calls enable exceptions to be handled before they cause the transaction to fail entirely



Smart contract platforms intentionally take an “all or nothing” approach to failed transactions. By stating that any transaction that includes unrecoverable error will be completely rolled back, they make it easy to ensure that all implementations of the blockchain software handle these errors identically and that the smart contract platform’s virtual machine ends up in a valid state. However, this also creates the potential for rollback attacks.

A smart contract’s vulnerability to rollback attacks depends on the platform and the control flow within the contract. Some smart contract languages and platforms (such as Solidity on Ethereum) allow a smart contract to handle most exceptions that are raised in a function that it calls. This enables a smart contract to protect itself against rollback attacks by refusing to allow an invalid reversion, such as a player trying not to pay their wager on a failed bet.

Timestamp Dependence

209 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Timestamp Dependence

- Some blockchains contain time dependencies
 - Contest starting after a certain time
 - Token sale running for a particular period
- Smart contracts need access to time information to implement these functions
 - Block headers contain a creation timestamp
- These header timestamps are not reliable
 - Flexible and under control of the block producer



Some smart contracts are designed to perform certain functions at a certain time. For example, a contest may be designed to award a prize to the first person to submit a valid solution after a certain time. Alternatively, a smart contract may implement a token sale where users are able to purchase the native tokens of a particular project for a certain rate during a particular time period.

To implement this type of functionality, these smart contracts need access to timing information. Since they run within a virtualized environment, they do not necessarily have access to a system clock. To ensure that smart contracts running on different nodes have access to a synchronized clock, they use the creation timestamps included in the header of a block.

While these header timestamps provide a good approximation of the current time, they are not exact or reliable. Some of the issues include:

- **Timestamp Flexibility:** The timestamps in block headers are designed to be flexible to account for unsynchronized clocks on different nodes and the latency of transmitting data across the peer-to-peer network. Most blockchains have an algorithm for determining if the timestamp contained within a block header is

“close enough”. However, these can create wide windows of acceptable timestamps and do not ensure that the header timestamp is accurate.

- **Producer Control:** The timestamp value included in a block header is set by the producer of that block. As long as the timestamp falls within the acceptable interval, the block producer can put any value that they want there. This means that a block header could have a timestamp from the past or the future and even contain one that is earlier than the previous block.

This combination of timestamp flexibility and block producer control over timestamps creates an opportunity for block producers to exploit timestamp-dependent smart contracts. A block producer could create a block claiming to meet the contract’s timing criteria and include a transaction in it that provides some benefit to the block producer (winning the contest, buying tokens with favorable terms, etc.).

Timestamp Dependence Example

```
1 function play() public {
2     require(now > 1521763200 && neverPlayed == true);
3     neverPlayed = false;
4     msg.sender.transfer(1500 ether);
5 }
```

Source: <https://dasp.co/>

211 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This sample code is from a smart contract that implements a contest. The first user to call the play function after a certain point wins the contest. The two criteria for a valid play are shown in line 2, which requires that the timestamp is greater than 1521763200 and that the contract has never been played before. If these two criteria are met, the contract updates its state to indicate that someone has already won the contest and transfers a 1500 ETH prize to the winner.

This smart contract implements the type of contest that can be exploited by a malicious block producer. In theory, no-one will create a transaction calling the play function and attempting to win the contest until the indicated time or shortly before to ensure that they are included in the first block after that timestamp.

However, a block producer creating a block near to the indicated time could win the contest before anyone else starts playing. If the timestamp 1521763201 is within the interval of valid timestamps for the next block, then the block producer could create a block with that timestamp that includes a transaction by the block producer that calls the play function and wins the prize. As long as the block is valid, then it will be accepted by the nodes of the blockchain network and executed, sending the prize to the malicious block producer.

Case Study: GovernMental

- GovernMental was a Ponzi scheme implemented as a smart contract
 - The last account to send a transaction to the contract is the current leader
 - If no transactions were received for 12 hours, the leader is rewarded
- GovernMental used block header timestamps to track the 12 hour window
 - The block producer controls these timestamps
 - A block producer could force the contract to pay out early with a forged timestamp

Source: <https://eprint.iacr.org/2016/1007.pdf>

212 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

GovernMental implemented a Ponzi scheme game on the blockchain. Users could pay into the contract to be the current leader within the game. If 12 hours elapsed without a transaction to the contract, then the smart contract paid out a reward to the current leader.

The issue with the GovernMental smart contract is that it needed a method for determining when the 12-hour window had elapsed. The smart contract was written to use the timestamp values contained within block headers to track the current time. However, these values are under the control of the producer of the given block, which means that a block producer could cheat the system.

Assume that the current leader in the GovernMental game produces a block at a point when the 12-hour window will terminate within the acceptable range of block header timestamp values. The block producer could create a block with a forged timestamp indicating that the window has completed and a transaction triggering the GovernMental contract to pay the reward. This would allow the malicious block producer to cheat and win the game before the full 12-hour window had elapsed.

Timestamp Dependence Mitigations

- Smart contracts are vulnerable if they rely on the timestamps in block headers for timing information
 - These are flexible and under control of the block producer
- These vulnerability can be mitigated by using a more reliable source of timing information
 - Block heights
 - External oracle



Timestamp dependence vulnerabilities exist because a smart contract uses an unreliable source of information about the current time. While block header timestamps provide a rough approximation of the current time, they are deliberately flexible and are completely under the control of the producer of the given block. As a result, a malicious block producer can use them to cheat a timestamp-dependent smart contract.

While block header timestamps are the most convenient source of timing information, they are not the only option. A couple of more secure options include:

- **Block Heights:** Blockchains have target block intervals, which means that a particular block should be produced around a certain time. For example, Bitcoin's target interval is ten minutes, so a block height six higher than the current block should be reached in about an hour. While block heights are not the most precise source of timing information, they are less manipulable by a block producer, making it more difficult to cheat.
- **External Oracle:** Smart contracts can also use an external timestamp oracle to provide a more reliable value for the current time. Decentralized timestamp

oracles exist and provide a means for getting a more reliable timestamp that cannot be manipulated by the block producer.

Weak Randomness



214 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Weak Randomness

- Random values are necessary for many smart contracts
 - Especially those implementing contests or games
- Smart contracts operate in a deterministic environment
 - Not many sources of randomness
- Many methods of generating “random” values are insecure
 - “Secret” values
 - “Secret” code
 - Block metadata



Many smart contracts want a source of random or pseudorandom values. For example, a smart contract implementing a casino-style contest or game needs a way of selecting winners in an unpredictable way.

The problem is that smart contracts do not have easy access to good sources of random values. Smart contract code is executed within a deterministic virtual machine designed to help maintain consensus across the blockchain ecosystem. If nodes in the network generated true random numbers, then different nodes would have different results when executing the same code, breaking consensus. As a result, smart contracts’ access to random values is deliberately limited.

Smart contract developers have implemented various workarounds to try to generate random numbers; however, many of these are insecure and easily bypassed. Some common options include:

- **“Secret” Values:** Some smart contract platforms and languages enable variables to be private, which prevents other smart contracts for seeing them. Smart contract developers occasionally use these private variables to hold “secret” values used to generate “random” numbers by using them to seed a pseudorandom number

generator (PRNG), etc. However, any smart contract data or code is broadcast as a transaction and added to the public blockchain ledger. As a result, an attacker can find these “secret” values and predict the contract’s “random” numbers.

- **“Secret” Code:** Like the use of “secret” data, smart contract developers may also use “secret” code, such as a custom PRNG, to calculate “random” values. However, like the “secret” data, “secret” code is publicly visible in blockchain transactions and on the digital ledger, enabling an attacker to reverse engineer the code and predict the values that it will generate.
- **Block Metadata:** Blockchain blocks contain seemingly-random values such as the hash of a particular block. Smart contracts may use these as unpredictable values for random number generation. However, any value that is accessible to one smart contract is also accessible to another. An attacker can win the game by deploying a smart contract that checks if the current block would be a winner, and, if so, calls the play function to win the contest.

Weak Randomness Example

```
1 function play() public payable {
2     require(msg.value >= 1 ether);
3     if (block.blockhash(blockNumber) % 2 == 0) {
4         msg.sender.transfer(this.balance);
5     }
6 }
```

Source: <https://dasp.co/>

216 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The play function shown above demonstrates a smart contract game that uses a weak source of randomness. The contract uses a “random” value in line 3 when it is determining whether or not a player has won the game. In this case, a “random” number that is even is a winner, while odd ones are not.

This contract uses the hash of a particular block to determine whether or not a player is a winner. While the number of the block used isn’t shown, both potential cases are problematic:

- **Recent Blocks:** For blocks less than 256 blocks from the current block, Solidity provides access to the hash of the indicated block. In this case, both this smart contract and any other smart contract can look up the block hash. An attacker could create a contract that copies line three and calls the play function if the result is true, only making a bet when they are guaranteed to win the reward.
- **Older Blocks:** Solidity only keeps the hashes of the previous 256 blocks. For all older blocks, a call to the blockhash function will return a value of zero. Since a hash of zero would pass the test in line 3, then using an older block for the test case would always result in a winner.

As a result, this smart contract can easily be won by an attacker who understands how it decides what is a win or a loss and acts appropriately.

Case Study: SmartBillions Lottery

- SmartBillions was a lottery based on guessing 6 lucky numbers
 - After guessing, the player could trigger a comparison to a “random” number
 - Each correct guess earned a greater reward
- SmartBillions used the hash of a past block as its “random” number
 - Only the last 256 hashes are accessible in Solidity
- One player guessed all zeros and called `won()` after 256 blocks
 - Earned 400 ETH before the contract owner drained the contract of remaining 1100 ETH

Source: https://www.reddit.com/r/ethereum/comments/74d3dc/smartbillions_lottery_contract_just_got_hacked/

217 | Blockchain and Crypto Security Training | © Marin Ivezic 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

SmartBillions was a blockchain-based lottery game. Players would guess six “lucky numbers”. Later, they could call the `won()` function to check if the next number in the sequence matched a “random” value generated by the smart contract. If that value matched, then the player earned a reward and the chance to see if their next number matched as well for a larger reward.

The SmartBillion smart contract was vulnerable to exploitation because it used a weak source of randomness. The contract code used the hash of an earlier block on the blockchain to compare against a player’s guess when they called the `won()` function.

As mentioned previously, any use of an earlier block hash provides a weak random number, but, in this case, a player exploited the fact that Solidity only retains the hashes of the previous 256 blocks. The player guessed all zeros as their lucky number and then waited until the block whose hash was used in the comparison was at least 256 blocks in the past. At this point, Solidity would provide a hash of zero, which matched the player’s guess.

In this case, the player was able to win 400 ETH from the game before the contract

creators shut them down. The creator used a backdoor to drain the remaining 1100 ETH from the contract before the player could test their remaining guesses and win the rest of the prize.

Mitigating Weak Randomness Vulnerabilities

- Smart contracts execute in deterministic environments
 - Little access to sources of randomness
 - Most approaches to random number generation are insecure
- Generating random values within the blockchain is always problematic
 - Smart contract code and data is publicly accessible
 - Anything visible to one smart contract is visible to another
- An external source of random values is needed



Smart contracts' execution environments are deliberately deterministic. To maintain consensus and coordination across blockchain nodes, smart contract platforms need to ensure that executing the same code produces the same result in all nodes. As a result, smart contracts don't have direct access to truly random values. This inspires many smart contract developers to use insecure and predictable methods of generating "random" numbers.

Any pseudorandom numbers generated within the blockchain ecosystem will always be insecure and predictable. The data and code that smart contracts use for random number generation is publicly visible on the blockchain, enabling an attacker to reverse engineer it at will. Also, any data accessible to one smart contract is accessible to another as well, making it possible to write contracts that call a vulnerable contract only when it would produce a desirable result (winning a contest, etc.).

Secure random number generation requires sources of random numbers that originate from outside of the blockchain environment. External random number oracles can provide smart contracts to truly random values for use in their calculations.

Summary

- Introduction to Blockchain-Specific Vulnerabilities
- Access Control
- Denial of Service
- Frontrunning
- Rollback Attacks
- Timestamp Dependence
- Weak Randomness



Module 5: Smart Contract Security

Section 5.4: Platform-Specific Vulnerabilities



Overview

- Introduction to Platform-Specific Vulnerabilities
- Denial of Service: Block Gas Limits
- Denial of Service: Unexpected Revert
- Forced Send of Ether
- Missing Zero Address Checks
- Reentrancy
- Short Addresses
- Token Standards Compatibility
- Unchecked Return Values
- Unsafe External Calls



Introduction to Platform-Specific Vulnerabilities

- Ethereum is the first smart contract platform
 - Largest adoption and most opportunity for security research
- Many Ethereum-specific vulnerabilities have been discovered
 - Denial of Service: Block Gas Limits
 - Denial of Service: Unexpected Revert
 - Forced Send of Ether
 - Missing Zero Address Checks
 - Reentrancy
 - Short Addresses
 - Token Standards Compatibility
 - Unchecked Return Values
 - Unsafe External Calls



Smart contract platforms build on the blockchain protocol by embedding executable code in transaction data that can be run within virtualized environments. Different smart contract platforms have their own approach of implementing this model, potentially with custom virtual machines and smart contract programming languages. The idiosyncrasies of these different implementations can create platform-specific smart contract vulnerabilities.

Ethereum is the oldest smart contract platform and has been widely adopted, providing the most opportunity for the discovery of platform-specific vulnerabilities. Some examples of smart contract vulnerabilities unique to the Ethereum platform include:

- Denial of Service: Block Gas Limits
- Denial of Service: Unexpected Revert
- Forced Send of Ether
- Missing Zero Address Checks

- Reentrancy
- Short Addresses
- Token Standards Compatibility
- Unchecked Return Values
- Unsafe External Calls

Denial of Service: Block Gas Limits

223 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Gas Limit DoS

- The Ethereum blockchain has the concept of “gas”
 - All Ethereum VM instructions consume gas
 - Transactions pay gas fees to run smart contract code
- Transactions that run out of gas are reverted
- The Ethereum VM defines a maximum amount of gas contained within a block
 - Smart contract functions can be rendered unrunnable



The Ethereum virtual machine (EVM) has the concept of “gas”, which works like gas in an automobile. Each instruction in the EVM costs a certain amount of gas to execute. This amount depends on the complexity of the instruction, so a simple addition operation would be cheaper than a token transfer, etc. When creating an Ethereum transaction to run smart contract code, a user needs to include gas, which is a fraction of an ETH, along with it.

Like with gas in a car, running out of gas in an Ethereum transaction means that nothing further can be done. However, in the EVM, running out of gas is an exception that can cause a transaction to be reverted. This makes it like the transaction never happened, but the transaction creator is not refunded the gas spent.

The EVM defines a maximum amount of gas that can be included within a block, which also limits the amount of gas that a particular transaction can hold. This creates an upper limit on the operations that can be performed within a call to a smart contract. If a function requires more gas than is allowed in an Ethereum block, then that function can be rendered unrunnable since any transactions calling it would throw an out of gas exception and be reverted.

Block Gas DoS Example

```
1 function selectNextWinners(uint256 _largestWinner) {
2     for(uint256 i = 0; i < largestWinner, i++) {
3         // heavy code
4     }
5     largestWinner = _largestWinner;
6 }
```

Source: <https://dasp.co/>

225 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample demonstrates a potential Block Gas Limit DoS vulnerability. As shown, the code includes a loop whose number of iterations is governed by the `largestWinner` variable. This `largestWinner` variable is set in line 5 based upon user-provided input. Additionally, line 3 of the sample indicates that the loop contains computationally-intensive code that can be expected to consume a large amount of gas.

If the value of `largestWinner` grows large enough that all gas is consumed by the loop, then no gas will remain to execute the instruction on line 5. This will trigger an out of gas exception that would result in the transaction being reverted and the execution state being rolled back to before the transaction started.

The design of this function is dangerous because its DoS vulnerability can be triggered not only by an attacker but also as part of legitimate operations. The names of the variables (`largestWinner`) indicate that the value of `largestWinner` is expected to grow over time. Therefore, enough users playing the game will eventually drive up the value enough that the function requires more gas than can fit within an Ethereum block.

Case Study: GovernMental

- Timestamp dependence was not GovernMental's only issue
- When someone won the contest, the contract performed some cleanup
 - Included clearing the contents of two arrays
- Since the array lengths depended on the number of players, they became long
 - Reward claiming transactions ran out of gas
 - Clearing memory and claiming reward failed

Source: <https://eprint.iacr.org/2016/1007.pdf>

226 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The GovernMental smart contract was a case study for Timestamp Dependence. The contract implemented a game where users could send transactions to the contract, and, if no transaction was received for 12 hours, then the current leader earned a reward.

In addition to its use of insecure time information, GovernMental was also vulnerable to out-of-gas vulnerabilities in its reward claiming function. This is because the contract was configured to perform cleanup operations as part of the reward transaction. The contract maintained a pair of arrays whose length depended on the number of accounts that had participated since the last time a reward was successfully claimed.

Eventually, these array grew so long that the amount of gas required to clear them caused an out-of-gas exception in the reward claiming transaction. As a result, transactions attempting to claim the reward would fail and be reverted, making it impossible for a winner of the game to claim their payout.

Block Gas Limit DoS Mitigations

- Ethereum's block gas limits are built into the design of the Ethereum VM
 - Ethereum smart contracts need to work within these limits
- Smart contracts should be designed to stay within gas limits
 - Avoid loops and recursion
 - Modularize when possible



Ethereum's block gas limits are an intentional feature of the Ethereum VM. They limit the number of computations that a blockchain node can be asked to perform while running the code contained within an Ethereum block. While the relative costs of various instructions and the overall limit may vary over time, this feature is unlikely to go away. As a result, smart contracts need to be designed to work within these limits and protect themselves against accidentally reaching a state where it is impossible for a function to complete execution without running out of gas.

While a function might be accidentally designed to use too much gas, these types of DoS vulnerabilities usually occur when the amount of gas used by a function is unpredictable. One way to mitigate the risk of these vulnerabilities is to avoid loops and recursion, especially when the number of iterations is determined by the user. Another best practice for avoiding these vulnerabilities is to modularize code whenever possible. If a particular sequence of events can be split across multiple transactions instead of being executed as part of a single, massive transaction, this reduces the probability that that transaction will require more gas than the limit.

Denial of Service: Unexpected Revert

228 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Unexpected Reverts

- Ethereum smart contracts can send and receive value from one another
 - Operate just like a user address
- When a smart contract is sent value, it has the opportunity to run code in its fallback function
- This fallback function could revert the transfer of value
 - This could cause a DoS in the calling function



In Ethereum, user accounts and smart contract accounts are virtually identical. The primary difference is that a smart contract account contains runnable code while a user account does not.

This means that smart contracts can send value to one another and receive value from user or smart contract accounts. In the event that a smart contract is sent Ether, its fallback function is triggered, giving it the opportunity to run some code. This fallback function could allow the smart contract to update its internal balance (crediting the transaction to the sender's account), perform some response to the transaction, etc.

However, a smart contract's fallback function can also be designed to revert when it receives a transfer of value, causing that transaction to fail. Depending on the mechanism used for the transfer, this could throw an exception or just indicate failure to the sender. If the sending smart contract function assumes or requires that the transaction will go through, this could result in a Denial of Service (DoS) for that smart contract.

Unexpected Revert Example

```
1 contract Auction {
2     address currentLeader;
3     uint highestBid;
4
5     function bid() payable {
6         require(msg.value > highestBid);
7
8         require(currentLeader.send(highestBid)); // Refund the old leader, if it fails then revert
9
10        currentLeader = msg.sender;
11        highestBid = msg.value;
12    }
13 }
```

Source: https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-unexpected-revert

230 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample implements an auction contract that contains a bid function. Users can call the bid function and deposit the amount that they are willing to pay for the item being sold. If the caller's bid is higher than the current winning bid, then the previous leader is refunded their bid and the new bidder is saved as the current leader.

This smart contract is vulnerable to unexpected reversion due to the require statement on line 8. This line of code sends the previous leader's bid back to them and requires it to go through successfully. The intent is to ensure that the previous leader is not cheated by having a failed transfer result in the smart contract keeping their money and them losing their position as the current leading bidder.

However, if the current leader is a smart contract with a reverting fallback function, then they will always win this auction. After they place their bid, any attempt to outbid them will fail due to the require statement on line 8. If this transfer fails (due to reversion), then the require statement will fail and throw an exception. This will result in the transaction being rolled back, and the new bidder will not be able to claim the position of the current leader.

Case Study: King of the Ether

- The King of the Ether contract allowed player to pay to be named King/Queen of the Ether
 - The current monarch could be unseated by someone paying 50% more than they did
 - All but 1% of the new monarch's payment goes to the one they unseat
- The contract required the payment to the previous monarch to go through
 - A monarch could have a fallback function that reverted payments
 - This would make them impossible to depose

Source: <https://www.kingoftheether.com/postmortem.html>

231 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The King of the Ether smart contract was a game in which users could be named King/Queen of the Ether. Someone can depose the current monarch by paying half again as much as they did for the title. All but 1% of this payment would go to the previous monarch, enabling them to make a significant (48.5%) profit on their initial payment when they are unseated.

The contract code was designed to ensure that a previous monarch could not be unseated without receiving their payment. To accomplish this, the call to the function that transferred the value was wrapped in a require statement. If the transfer failed, the require statement would resolve to false, causing an exception to be thrown that reverted the transaction.

However, this design created an unexpected reversion vulnerability within the King of the Ether smart contract. If the previous monarch's address contained a smart contract with a reverting fallback function, then any attempts to pay them would fail. This would cause the entire transaction to fail, making it impossible to unseat them.

Mitigating Unexpected Revert Vulnerabilities

- Smart contracts may be designed to ensure that a transaction goes through before continuing execution
 - Acts as a failsafe against failed transactions
- This can create unexpected reversion vulnerabilities
- Unexpected reversion vulnerabilities can be mitigating by testing rather than requiring transfer success and implementing error handling



In many cases, the code creating an unexpected reversion vulnerability is an intentional feature within a smart contract. When sending value to another account, the smart contract wants to ensure that the value transfer goes through successfully before updating its internal accounts ledger, removing that account as the current winner of a contest, etc.

However, this code pattern can create unexpected reversion vulnerabilities. If a transaction to a smart contract function only succeeds if a value transfer succeeds, then the recipient of that value transfer can cause the transaction to fail by reverting the transaction.

A smart contract risks a DoS attack if it requires success of a transfer. A more secure code pattern is to perform the transfer, check if it succeeded, and implement error handling if it did not. This enables a smart contract function to manage an issue gracefully rather than being vulnerable to a DoS attack if the recipient of a transfer decides to revert payments to it.

Forced Send of Ether

233 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Forced Send of Ether

- Ethereum fallback functions can be designed to revert or reject any transfers to them
- However, some types of transfers don't trigger fallback functions
 - Self-destruction
 - Prefunding an address
 - Mining Ether to the address
- This forces Ether into the target contract
 - Can cause unexpected behavior



As mentioned previously, Ethereum smart contracts have the opportunity to run code if value is sent to them. These fallback functions can be designed to reject or revert any transfers of value to them. The intent is to prevent any Ether from being deposited into the contract.

However, while fallback functions are executed as part of normal value transfers, some transfers do not trigger them. These include:

- **Self-Destructing Contracts:** A smart contract that is self-destructing can indicate that any Ether that it contains should be sent to another address. This transfer does not trigger the beneficiary's fallback function if it is a smart contract.
- **Prefunding:** Smart contracts are deployed to Ethereum addresses that "exist" before they are claimed and have the code deployed to them. If the address of a smart contract can be predicted, value can be transferred to it before the fallback function exists, making it impossible for it to revert the transaction.
- **Mining Rewards:** Mining cryptocurrency generates rewards for the user that creates a valid block. Mining rewards do not trigger fallback functions, so an

attacker could use them to send value to a particular address.

All of these methods bypass the reverting fallback function, enabling Ether to be sent to an address and contract that assumes that value cannot be sent to it or only accepts certain transfers of value (such as only allowing transfers from a set of allowlisted addresses and reverting all others). If the contract's behavior is based on this assumption, then a forced send of Ether can cause it to behave unexpectedly and undesirably.

Forced Send of Ether Example

```
1 contract Vulnerable {
2     function () payable {
3         revert();
4     }
5
6     function somethingBad() {
7         require(this.balance > 0);
8         // Do something bad
9     }
10 }
```

Source: https://consensys.github.io/smart-contract-best-practices/known_attacks/#forcibly-sending-ether-to-a-contract

235 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample shows a smart contract that is designed to revert all transfers to it. The fallback function in this code could trigger the unexpected reversion vulnerability discussed previously. As shown, the fallback function, indicated by the payable keyword, automatically reverts all attempts to send value to it.

The somethingBad function in this example operates based on the assumption that it is impossible for anyone to send value to this contract. The require statement in line 7 states that the balance of the contract's account must be greater than zero, which should be false due to the reverting fallback function. As a result, the undesirable functionality in line 8 should never be triggered.

However, an attacker that forces Ether into this contract using one of the methods described previously could increase the contract's account balance while bypassing the reverting fallback function. If this occurs, then a call to somethingBad would pass the require check on line 7, allowing the undesirable code on line 8 to execute. While this is a toy example, smart contracts may have balance-dependent functionality. For example, a contract might make a strict check of its balance (i.e. `this.balance == threshold`) before taking some action and only allow deposits of fixed sizes that would eventually cause the condition to be true. If so, an attacker could prevent the

condition from being met and the action from occurring by forcing Ether into the contract and exceeding that threshold without equaling it.

Case Study: Edgeware

- Edgeware was a project that implemented “lockdrop” contracts
 - Users could lock ETH in a contract for some time (3-12) months and earn a reward
- This contract created individual lock contracts for each user
 - This contracts asserted upon creation that the value of the contract was equal to the value locked by the user
 - This created a forced send of Ether vulnerability
- The smart contract would try to use the same address for a lock contract until it succeeded
 - An attacker that sent Ether to the next addresses could cause it to fail forever

Source: <https://medium.com/@nmcl/gridlock-a-smart-contract-bug-73b8310608a9>

236 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Edgeware was a blockchain-based project that allowed users to lock Ether in a contract in exchange for a reward. If users didn't touch their Ether for 3-12 months, they would receive a payout.

The Edgeware project was implemented as a primary lockdrop contract that spun off lock contracts for each of the users that deposited ETH in it. As part of the lock contract creation process, the contract asserted that the value of the address of the new lock contract was equal to the value that the user was agreeing to lock within the protocol. This assertion created a Forced Send of Ether vulnerability in these lock contracts. The algorithm that Edgeware used to select the address of the next lock contract was predictable, and the contract would not move on to a new address until a contract was created at the previous one.

Therefore, an attacker who calculated the address of the next lock contract could cause it and all future lock contracts to fail. By sending a small amount of Ether to that address, the attacker could ensure that the assertion that the address' balance equaled the amount deposited into the contract was incorrect. As a result, the transaction creating the lock contract would fail. Since the lockdrop contract would not move on to the next address until a contract was created at the current one,

performing the attack once prevented the creation of all future lock contracts.

Forced Send of Ether Mitigations

- Having Ether deposited into a smart contract is not usually a bad thing
 - Only undesirable if it impacts a contract's behavior in an undesirable way
- Forced Send of Ether vulnerabilities can be mitigated by
 - Avoiding strict balance comparisons
 - Internally tracking contract balance



In most cases, an unanticipated deposit of Ether into a blockchain address would be considered a positive thing since it is presumably free money. However, these deposits can be problematic if the control flow of a smart contract is based on the balance of its account. If the smart contract code uses account balance to trigger certain actions and attempts to regulate deposits, then an attacker could trigger these actions at an undesirable time or prevent them from occurring at all.

Most Forced Send of Ether vulnerabilities are entirely avoidable in smart contracts. Some best practices that can help to mitigate this risk include:

- **Avoiding Strict Balance Comparisons:** Most contracts with this vulnerability are vulnerable because they attempt to trigger a particular action when the contract's account balance reaches a certain threshold. If this is implemented with a strict comparison, it is possible that the balance could go above/below the threshold without being exactly equal. Using GTE (\geq) or LTE (\leq) instead of strict equality achieves the desired effect while eliminating the vulnerability.
- **Use Internal Balance Tracking:** Forced Send of Ether works by depositing value into a contract's account without allowing the contract to run code to manage that

deposit. If the contract's control flow depends on the current balance, tracking this balance in an internal variable, which is only updated by transfer that trigger the fallback function, can help to protect against this vulnerability.

Missing Zero Address Checks

238 | Blockchain and Crypto Security Training | © Marin Ivezic 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Missing Zero Address Checks

- Ethereum uses the genesis or zero address (0x0) for burning tokens
 - No known key exists for this address
 - Contracts will often allowlist the zero address for this reason
- A failure to check if an address used in a function is the zero address can cause unexpected behavior
 - A call to `address.call` with the zero address as address will try to run the fallback function of the contract at 0x0
 - This fallback function does not exist and no exception will be thrown



Blockchain security is based on the assumption that guessing the private key associated with a particular public key or blockchain account address is effectively impossible. If an attacker could calculate a private key for any given address, then they could generate valid digital signatures and transactions for that address.

Most Ethereum account addresses have no known private key, and the genesis or zero address (0x0) is one of these. Since no one knows a private key for this address and it is unlikely ever to be discovered, this address is commonly used for burning tokens. Tokens sent to the address can never be retrieved because the probability of finding a valid private key for it are essentially zero.

For this reason, smart contracts are accustomed to transactions dealing with the zero address but must be careful to validate addresses for certain functions. One example is when executing the `call` function for a particular address. Invoking `address.call` when address is the zero address will attempt to execute the fallback function of the contract at the zero address. This contract does not exist, but the function will not cause a reversion, which could make a contract believe that the transfer performed using `call` succeeded.

Missing Zero Address Checks Example

```
1 function safeTransferFrom(address token, address from, address to, uint value) internal {
2     (bool success, bytes memory data) = token.call(abi.encodeWithSelector(@x23b872dd, from, to, value));
3     require(success && (data.length == 0 || abi.decode(data, (bool)), "!safeTransferFrom");
4 }
```

Source: <https://blocksecteam.medium.com/when-safetransfer-becomes-unsafe-lesson-from-the-qbridge-security-incident-c32ecd3ce9da>

240 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample implements a safe transfer function similar to those that are common in DeFi smart contracts. This function is designed to safely transfer an ERC-20 token from one address to another.

However, this implementation is slightly different from the standard one created by OpenZeppelin. This function uses the `address.call` function to perform the transfer in line 2 rather than the `address.functionCall` function used by OZ. One of the major differences between the two is that `address.functionCall` checks that a smart contract exists at the indicated address while `address.call` does not.

In the case where `token` points to the zero address, this makes a big difference in the result of this function. This code would attempt to call the fallback function at the zero address and succeed despite the fact that no such contract exists. The OZ implementation, on the other hand would identify the issue and indicate an error.

Case Study: Qubit

- In January 2022, the Qubit's QBridge was the victim of a hack
 - This contract implements a bridge between Ethereum and BNB Chain
 - The attacker managed to steal \$80 million in BNB
- The vulnerable function from the previous slide is from QBridge
 - The attacker performed a fake deposit of ETH that exploited the missing zero address validation vulnerability
 - This fake deposit was interpreted as a real deposit of Ether
 - The contract unlocked tokens on BNB Chain, which the attacker exchanged for \$80 million in BNB

Source: <https://blocksecteam.medium.com/when-safetransfer-becomes-unsafe-lesson-from-the-qbridge-security-incident-c32ecd3ce9da>

241 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The Qubit QBridge implements a bridge between the Ethereum and BNB Chain blockchains. Users can deposit tokens into a smart contract on one chain, which causes tokens to be unlocked and sent to their accounts on the other.

A QBridge smart contract contained the vulnerable function shown in the previous slide that used `call` instead of `functionCall` to perform a token transfer. The attacker exploited this to perform a fake deposit of Ether into the contract that took advantage of this missing address check.

QBridge interpreted this fake deposit as a real deposit of Ether, which caused a `Deposit` event to be emitted and caught by the BNB Chain contract. This contract then minted `qxETH` tokens that were sent to the attacker's address on the BNB Chain blockchain. The attacker was able to exchange these `qxETH` tokens for BNB tokens, allowing them to drain approximately \$80 million in BNB tokens from QBridge.

Missing Zero Address Checks Mitigations

- The zero address is a valid address for some Ethereum transactions
 - Burning tokens, etc.
- However, the zero address can cause unexpected behavior in some cases
- Smart contracts should always check for the zero address and validate that a smart contract exists before calling its functions



The zero address (0x0) in Ethereum has legitimate uses for smart contracts. The fact that no private key is or is likely ever to be known for this address makes it an ideal address for transactions to be sent when burning them. For this reason, smart contracts may be configured to accept the zero address for certain types of transactions.

However, in other cases, performing certain operations with the zero address can result in unexpected behavior. For example, the low-level call function in Ethereum will not revert when trying to call a fallback function in a non-existent smart contract located at Ethereum's zero address.

Assuming that calls to non-existent functions will cause reversions is dangerous for Ethereum-based smart contracts. Smart contracts should always check for the zero address and that a smart contract exists at a given address before trying to interact with functions within that contract with `call()` and similar functions.

Reentrancy



Introduction to Reentrancy

- Reentrancy is another vulnerability related to fallback functions
 - Smart contracts can run code when they receive value
- Ethereum fallback functions can call other smart contract functions
 - Including the ones that called them
- A reentrant call may occur before internal state updates occur
 - Function may process call based on outdated internal state



Ethereum fallback functions are the source of several potential Ethereum-specific vulnerabilities. Reentrancy is another vulnerability that exists because of fallback functions and smart contracts' ability to execute code when they receive a transfer of value.

When a fallback function is triggered by a value transfer, it executes immediately, before the next instruction of the calling function is run. This fallback function is able to call other smart contract functions, including the one that performed the value transfer.

This ability to reenter the sending function can be dangerous if the sending function has not yet updated its internal state. For example, the sender may be waiting to see if the transaction succeeded and was not reverted before updating the balance of the recipient's account with the smart contract.

If this is the case, then the reentrant call might be processed using an outdated account balance that does not reflect the recent transfer. This could allow an attacker to extract more value from the smart contract than is actually in their account with it.

Reentrancy Vulnerability Example

```
1 function withdraw(uint _amount) {
2     require(balances[msg.sender] >= _amount);
3     msg.sender.call.value(_amount)();
4     balances[msg.sender] -= _amount;
5 }
```

Source: <https://dasp.co/>

245 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample implements a withdraw function for a smart contract. This smart contract presumably allows users to deposit funds with it in exchange for some benefit (interest payments, participation in some activity, etc.). When the user no longer wishes to take advantage of the contract's services, they can call this function to withdraw their funds.

This function's code follows a logical control flow. In line 2, the function verifies that the request is valid by requiring that the user's account contain the amount to be withdrawn. Then, the transfer is performed in line 3. Finally, when the transfer succeeds, the function updates its internal state to reflect the successful transfer.

However, with Ethereum's fallback functions and the potential for reentrancy, this function is vulnerable to attack. Assume that the user's account contains 5 ETH and they attempt to withdraw 4. The test at line 2 would succeed because $5 > 4$.

At line 3, the smart contract transfers 4 ETH to the user, which will trigger the fallback function of a smart contract at that address. This fallback function could then reenter this withdraw function, starting over a line 1.

At this point, line 4 has not been executed, so the user's account still holds 5 ETH. The require statement at line 2 succeeds because $5 > 4$. Therefore, the transfer at line 3 occurs again, transferring 4 more ETH for a total of 8 and triggering the recipient's fallback function.

The main limitation on the number of iterations of this loop that can be performed is the amount of available gas. When the fallback function elects not to reenter this withdraw function, it returns, allowing line 4 to execute for the most recent call. This updates the account's internal balance and returns, working up the chain of recursive calls and updating the balance at each step until the transaction completes.

After the second execution of line 4, the balance of the user's account within the smart contract will be negative. However, at this point, the transfer has already occurred. With this vulnerable code pattern and lack of error handling, the smart contract can transfer much more Ether to the attacker than their account contains.

Case Study: The DAO

- The DAO hack is the most famous hack in blockchain history
 - Caused Ethereum community to break the rules to save the blockchain
- The DAO was a smart contract-based, crowdsourced venture capital program
 - Holders of the DAO token could vote on proposals
 - Proposals could be approved or denied by a quorum of 20% of token holders
 - Dissenters could retrieve their tokens via a split function
- The DAO split function contained a reentrancy vulnerability
 - Attacker exploited this to steal 3.6 million ETH from the contract

Source: <https://oguccluturk.medium.com/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562>

246 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The DAO was a highly successful project that implemented crowdsourced venture capital function as a blockchain-based decentralized autonomous organization (DAO). By investing ETH in the DAO, users could receive tokens that gave them the right to vote on proposals about how the funds would be spent. A proposal could be approved or denied by a quorum of 20% of the token holders, and, if approved, provided the proposed project with funding.

Because of the design of the DAO, it is possible that a minority of token holders may have their money invested into a project that they didn't approve of. To remedy this, the DAO contract included a split function that allowed dissenting token holders to retrieve their tokens.

In June 2016, an attacker exploited a reentrancy vulnerability in this split function. This vulnerability allowed malicious "child DAO" smart contract to withdraw tokens from the DAO contract multiple times, draining value from the contract. The attacker exploited this to steal 3.6 million ETH from the DAO contract.

This was the most significant hack of a project on the Ethereum blockchain at the time since it affected 14% of all tokens in circulation. In response to the hack, the

Ethereum community decided to perform a hard fork in which the history of the blockchain was rewritten to roll back to the block before the one containing the transaction that hacked the DAO. This enabled DAO token holders to retrieve their ETH and undid the hack.

However, this hard fork broke the rules of blockchain by breaking the immutability of the Ethereum blockchain. This was a controversial decision that prompted a split between the Ethereum (ETH) blockchain that followed the hard fork and the Ethereum Classic (ETC) blockchain that did not.

Reentrancy Mitigations

- Reentrancy vulnerabilities are made possible by an insecure code pattern
 - Check validity
 - Perform transfer
 - Update state
- A more secure code pattern is
 - Check validity
 - Update state
 - Perform transfer
 - Verify success



Smart contract functions that perform value transfers can be vulnerable to reentrancy attacks if they use a logical but insecure code pattern. When making a transfer to a user, it is logical to verify that the transfer is valid, make the transfer, and then update internal state. This flow ensures that state updates are only made if the transaction succeeds.

However, this sequence of events also creates a vulnerability to reentrancy attacks. Since the recipient's fallback function allows the vulnerable function to be reentered before the state update, a vulnerable function could be tricked into making transfers based on outdated balances.

Mitigating reentrancy vulnerabilities requires implementing a different control flow in these types of functions. This flow is:

- **Check Validity:** Verify that the transfer request is valid (i.e. the user's account contains the requested balance)
- **Update State:** Update the contract's internal balance tracking to reflect the transfer

- **Perform Transfer:** Transfer value to the indicated account and allow execution of the fallback function
- **Verify Success:** Check that the transfer completed successfully, and, if not, perform error handling

While this process requires more steps (and gas), it provides protection against reentrancy. When the recipient's fallback function attempts to reenter the function, its account balance has already been updated. This means that the function will only allow additional withdrawals that would be valid even after the initial transfer is performed.

Short Addresses



Introduction to Short Addresses

- The Ethereum virtual machine (VM) serializes arguments to a function
 - Caller packs the arguments into a single value
 - Callee unpacks the arguments based on expected lengths
- If an argument is shorter than expected, it will steal from the next argument
- Some functions implementing value transfers can be valuable to exploitation
 - Accepts target address and value
 - Sends them to function that enforces length requirements



The Ethereum virtual machine (VM) serializes arguments when making calls between smart contract functions. The caller packs the arguments into a single chunk of data, and the callee unpacks the data based on the anticipated lengths of the various arguments to the function.

This scheme can be problematic if one of the arguments to a function is shorter than expected. If this is the case, the argument steals any missing bits from the next argument. This continues down the chain until the last argument. If this argument is shorter than expected, the Ethereum VM right-pads it with zero bytes, which can dramatically impact its value.

Ethereum's method for handing function arguments can be problematic for a function that validates a transfer of value and then calls another function to perform it. If the initial function does not enforce the expected lengths of the function arguments, then it performs validation on one version of the arguments. When it calls the transfer function, the arguments may be interpreted differently, resulting in a different transfer than the one that is approved.

Short Addresses Example

```
1 event Transfer(address _from, address indexed _to, uint256 _value);
2 function sendCoin(address to, uint amount) returns(bool sufficient) {
3     if(balances[msg.sender] < amount) return false;
4     balances[msg.sender] -= amount;
5     balances[to] += amount;
6     Transfer(msg.sender, to, amount);
7     return true;
8 }
```



This code sample demonstrates a short address vulnerability in which inconsistent enforcement of argument lengths can cause issues. In this case, an attacker would call the `sendCoin` function with an account address that is one byte too short. This address will be designed so that, if a zero byte is appended to it, it would produce a valid address that the attacker controls.

The `sendCoin` function will validate the transaction using the too-short address and the intended value of `amount`. If this amount is less than the balance of the message sender, the function will update internal state first (to protect against reentrancy attack) and then call the `transfer` function to perform the actual transfer.

However, this `Transfer` function will actually enforce the intended lengths of the `to` address and value to be transferred. Since the `to` address is too short, it will steal a zero byte from value, resulting in the address owned by the attacker. This will cause value to be too short, which will cause the Ethereum VM to zero-pad it.

The result of this zero-padding is to multiply the value being transferred by 256. As a result, the attacker receives significantly more value than was validated and approved in the `sendCoin` function.

Mitigating Short Address Vulnerabilities

- Short address vulnerabilities arise from Ethereum's handling of function arguments
 - Arguments are serialized and then unpacked based on expected lengths
 - Too-short arguments result in the last argument being right-padded with zeros
- Short address vulnerabilities can be managed by validating arguments
 - Ensure that the total length of arguments is the expected amount
 - Addresses should be 20 bytes long



Short address vulnerabilities are made possible by an idiosyncrasy in how the Ethereum virtual machine handles function arguments. Arguments are transferred in a serialized form, and too-short arguments are zero-padded. While this might be logical for strings, which may be missing their null terminator, it can dramatically change the value of a numeric argument.

Short address vulnerabilities can be mitigated by testing argument lengths, but this is not as easy as checking the length of each argument. By the time arguments are accessible to the called function, the unpacking and padding has already been performed.

Short address vulnerabilities can be prevented by testing the length of the serialized arguments as a whole. If this length matches expectations (with addresses being 20 bytes long), then the arguments should unpack correctly and no theft of bytes or padding should have occurred.

Token Standards Compatibility Issues

252 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Token Standards

Compatibility Issues

- Ethereum Improvement Proposals (EIPs) define standard for the Ethereum community
- EIP-20 lays out prototypes for common functions in tokens
 - Smart contracts processing these tokens should use these prototypes as a guide
- Not all tokens follow the standard
 - EIP-20 mandates that the transfer and transferFrom functions return True/False indicating success or failure
 - USDT returns void
- Non-compliant tokens can break smart contracts that assume compliance



Requests for Comment (RFCs) are the documents in which the Internet Engineering Task Force (IETF) creates standards for how the Internet should work. Ethereum Improvement Proposals (EIPs) and Ethereum Requests for Common (ERCs) are proposed and finalized versions of standards for the Ethereum blockchain.

EIP-20 is a standard for certain types of tokens that run on top of the Ethereum blockchain. This includes defining function prototypes for common functions that these tokens should implement and that smart contracts that work with these tokens can invoke. Smart contracts using these tokens use these function prototypes as a guide and should comply with the standard.

Smart contracts that assume that tokens follow the EIP-20 standard can experience issues if they process a token that is non-compliant. For example, the EIP-20 standard states that the transfer and transferFrom functions within token contracts should return a Boolean value of True/False to indicate whether or not the transfer succeeded.

However, the USDT stablecoin does not comply with the standard, returning void instead of a Boolean.

These non-compliant tokens can cause issues for smart contracts that process them and assume compliance. For example, any contract that performs error handling based on the transfer function's return value while assume that USDT tokens always fail (because void resolves to false).

Token Standards Compatibility Example

```
1 require(IERC20(inputToken).transfer(msg.sender, _amountIn),errorMessage);
```



This line of code might appear in a smart contract that processes EIP/ERC-20 tokens. The goal of this line is to perform a transfer of a token to the address that called the function (`msg.sender`). It also implements error handling in the form of a `require` statement, which will throw an exception if the result of the transfer function resolves to `false`.

While this implements programming best practices, it will have issues with tokens like USDT that are not compliant with the EIP/ERC-20 standard. A void function with no return value will resolve to a Boolean of `false` within the `require` statement. This would cause the function to assume that the transfer failed and likely would cause the entire transaction to be reverted unless higher-level error handling can catch and manage it.

Case Study: Force DAO

- In April 2021, Force DAO's xFORCE contract was exploited by an attacker
 - Approximately \$367,000 in tokens were stolen from the contract
- The xFORCE contract assumed that transaction would revert if transferFrom failed
 - The attacker deposited tokens that indicated errors by returning false
 - Contract did not check return value and minted xFORCE tokens
- Attacker traded xFORCE tokens for other tokens, draining value from the contract



In April 2021, the FORCE DAO project was exploited shortly after it was launched. The project's xFORCE contract was a fork of xSUSHI and had issues with token standard compatibility that enabled an attacker to exploit it for \$367,000 in tokens.

The xFORCE contract assumed that a failure within the transferFrom function would result in the transaction being reverted. However, the attacker deposited tokens that indicated issues by returning false rather than reverting the transaction.

Since the xFORCE contract assumes that failed transfers will result in the transaction being reverting, it does not check the return value from call to the transferFrom function. As a result, failed transfers were seen as successes and the contract minted xFORCE tokens to the attacker's account.

The attacker was then able to exchange these xFORCE tokens for other tokens that were deposited within the smart contract. This allowed the attacker to drain about \$367,000 worth of tokens from the contract in exchange for fake deposits.

Token Standards Compatibility Mitigations

- EIP-20 and other token standards are intended to define standard interfaces between tokens and the contracts that use them
 - Token contracts and contracts processing these tokens should follow them
 - Non-compliance can cause security issues
- Smart contracts should be written to comply with the standard
 - But should be capable of handling non-compliant tokens



The purpose of EIP/ERC-20 is to standardize interfaces for token contracts. By defining function prototypes and expected behaviors, these standards are intended to make it easier to process tokens without needing to consider every potential implementation and edge case.

Ideally, both token contracts and the smart contracts that accept and process these tokens should follow the standard, but this does not always happen. For example, the EIP/ERC-20 standard states that a token's `transferFrom` should return `false` on failure, a requirement that neither the USDT token or the xFORCE contract comply with.

A lack of compliance with the token standard can create security issues. Smart contracts should be designed to follow the interfaces defined in EIP/ERC-20 (or other standards for applicable tokens) but not to assume compliance and implement appropriate error-handling code.

Unchecked Return Values

257 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Unchecked Return Values

- Ethereum smart contract functions can report failure in various ways
 - Exceptions
 - Return values
- These different methods have different impacts on the calling function
 - Exceptions can cause a transaction to revert
 - Returning false allows execution to continue
- Smart contract functions need to test return values to know if some called functions have succeeded



The Ethereum virtual machine allows a smart contract function to report errors in a few different ways. A function can indicate failure by throwing an exception or returning a Boolean value of false.

These two ways of reporting issues have very different effects on the calling function. A thrown exception can cause execution to terminate and the entire transaction to be rolled back, depending on how the function throwing the exception was called. In contrast, returning false has no impact on the execution of the calling function unless it triggers error handling code such as causing a require statement to fail.

The different effects of these two means of indicating failure can cause issues for smart contract developers. In some cases, no error handling code is needed in a calling function because the called function will revert on failure. Eliminating extraneous error handling saves gas. However, if a called function returns a value of false to indicate an issue, then failing to check the return value can result in an invalid state for the calling contract. These different methods of indicating issues are especially confusing when functions with similar purposes, such as send and transfer, indicate failure in different ways.

Unchecked Return Value Example

```
1 function withdraw(uint256 _amount) public {
2     require(balances[msg.sender] >= _amount);
3     balances[msg.sender] -= _amount;
4     etherLeft -= _amount;
5     msg.sender.send(_amount);
6 }
```

259 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



Source: <https://dasp.co/>

This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample implements a withdraw function similar to earlier examples. Note that this version is resistant to reentrancy attacks because the internal balance updates are performed on lines 3 and 4 before the value transfer on line 5. Attempted reentrancy in the fallback function of the recipient contract will be processed using updated balance values.

The issue with this withdraw function is that it lacks essential error handling code. In line 5, the value transfer is performed using the send function, which returns a value of false upon failure rather than throwing an exception like transfer would.

As a result, this withdraw function will successfully complete execution regardless of whether or not the transfer succeeds unless an out-of-gas or other exception is thrown.

Additionally, since the contract updates its internal balance tracker and the balance of the user account, a failed transfer will be recorded as a success. As a result, the contract will contain more value than indicated by the etherLeft variable, and the account balance of msg.sender will be recorded as lower than it should be.

Case Study: ForceDAO

- The ForceDAO hack was made possible by incompatibility with the EIP-20 token standard
- Standard states that tokens' transferFrom functions should return false upon failure
 - xFORCE contract assumed that transferFrom would revert on failure
- xFORCE didn't check transferFrom's return value
 - Minted tokens in exchange for fake deposits

Source: <https://blog.forcedao.com/xforce-exploit-post-mortem-7fa9dcb2ac3>

260 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The ForceDAO hack was introduced previously in the context of non-compliance with token standards. The EIP-20 token standard defines prototypes for common functions in token contracts that are used to interface with other smart contracts. Among these is a function prototype for the transferFrom function, which states that transferFrom should return a Boolean value of True/False indicating success or failure.

The xFORCE smart contract called tokens' transferFrom functions while assuming that a failure in the transfer would result in a thrown exception and reversion. As a result, it did not check the return value of the call to transferFrom under the assumption that a transaction including a failed transfer would never reach the code following transferFrom.

The attacker called the xFORCE contract with a token that returned false in its transferFrom function rather than reverting. Under the assumption that the transfer succeeded, the xFORCE contract minted xFORCE tokens to the attacker's address that the attacker exchanged for the other tokens deposited into the protocol. This allowed the attacker to extract \$367,000 in tokens without depositing anything.

Unchecked Return Value Mitigations

- Not all errors within the functions called by a smart contract will result in exceptions that revert the transaction
 - Some smart contract functions indicate issues by returning false
- Smart contracts need to check return values to ensure that called functions succeeded



The fact that thrown exceptions can cause the execution state of a transaction to be rolled back can inspire insecure programming practices. If errors in code execution cause that execution to be undone, then there is no need for error handling code, especially on a smart contract platform where running that code costs money.

However, transaction-reverting exceptions are not a universal method for Ethereum smart contract functions to indicate issues. Some functions return false instead, which allows the calling function to continue execution. A failure to check these return values can leave the calling contract in an invalid state.

When performing calls to external functions, error handling code is essential for functions that do or could indicate issues by returning a value of false. Checking return values ensures that the calling function does not make incorrect assumptions regarding the success of the callee.

Unsafe External Calls

262 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Unsafe External Calls

- Ethereum's Solidity provides multiple options for calling one smart contract function from another
 - Some provide greater or lesser error handling capabilities
 - Some provide the callee with extra privileges
- The delegatecall function runs the callee within the context of the caller
 - Provides access to the caller's address, balance, and storage
- Delegatecall is a dangerous and easily exploitable function



Ethereum smart contract functions are designed to interact with one another, and Ethereum's Solidity smart contract programming language provides a few different options for one smart contract function to call another. Some of these options provide more error handling capabilities and protection against a transaction being reverted due to an exception thrown in the callee. Others provide the callee with more permissions and access than it would otherwise have.

The delegatecall function is Solidity function that calls a smart contract function within the context of the calling function. For all intents and purposes, the called function is essentially the caller. If it performs calls to other functions, then the caller's address is the source of that call, enabling the callee to take advantage of the caller's privileged access. The callee can also access the caller's balance and access or modify its storage.

The use of delegatecall is extremely dangerous within an Ethereum smart contract. If a function delegatecalls a malicious function, then the malicious function can masquerade as that function, drain it of value, and modify its internal variables and state.

Unsafe External Call Example

```
1 contract Proxy {
2
3   address owner;
4
5   constructor() public {
6     owner = msg.sender;
7   }
8
9   function forward(address callee, bytes _data) public {
10    require(callee.delegatecall(_data));
11  }
12 }
```

Source: <https://dasp.co/>

264 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample implements a forwarding function for calls between different smart contract functions. If a smart contract function calls the forward function with an address and data, this function will delegatecall the indicated function with the provided data.

The intent of the forward function is to allow the caller to use the permissions, data, and balance of this Proxy contract when calling other functions. This could allow the caller to access functions that it could not call otherwise (due to access controls).

However, this function is extremely dangerous because it implements a public forwarding function that can be called by anyone. While a private forwarding function with access controls is dangerous enough, this essentially hands over the private key of this smart contract to anyone that asks and allows them to perform any smart contract calls that they want with the permissions of this account.

Case Study: Furucombo

- Furucombo implements a web-based GUI for DeFi trading
 - Users can preapprove transfers of certain tokens to expedite later trades
- An attacker exploited a chain of delegatecall uses
 - A public batchExec function can delegatecall functions in Aave contracts
 - Among these is the Aave fallback function, which can delegate call Aave's implementation logic contract
 - Calling initialize using delegatecall made the attacker's contract the Aave implementation logic contract
- Attacker could then delegatecall their contract masquerading as Furucombo
 - Allowed theft of preapproved tokens

Sources: <https://cmichel.io/replaying-ethereum-hacks-furucombo/>

<https://slowmist.medium.com/slowmist-analysis-of-the-furucombo-hack-28c9ae558db9>

265 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Furucombo is designed to make it easier for traders to build chains of trades to take advantage of arbitrage opportunities. The drag-and-drop interface makes it easier to identify and define a series of transactions that would earn a profit for the trader.

As part of this, Furucombo allows users to preapprove transfers of certain tokens. This makes it faster and easier to perform transactions with these tokens later.

Furucombo made use of delegatecall, which the attacker exploited. The batchExec function is a public function in Furucombo and has the ability to call certain functions within AaveV2's proxy contract. This included the fallback function within Aave's proxy contract, which had the ability to delegatecall functions within the protocol's implementation logic contract.

The attacker used delegatecall to set the address of the implementation logic contract to that of a malicious contract controlled by the attacker. This allowed the attacker to exploit the two delegatecalls to delegatecall their malicious contract while in the context of Furucombo.

Access to Furucombo's context means that the attacker's contract has access to

users' preexisting approvals for token transfers. The attacker was able to call `transferFrom` for these tokens, sending them to the attacker's account.

Unsafe External Call Mitigations

- Ethereum's Solidity language provides multiple options for one function to call another
 - Delegatecall is a dangerous option because it gives the callee complete access to the context of the caller
- Ideally, smart contracts should never use delegatecall
 - Especially dangerous when calling untrusted functions
 - Even internal functions can be dangerous to call



Solidity provides a range of options for one function to call another. The various options have different features and associated advantages and disadvantages.

The use of delegatecall can make some operations easier because it provides the callee with direct access to the context of the caller. However, this function is very dangerous because a malicious callee can abuse the caller's permissions, drain value from it, and modify internal storage.

Ideally, a smart contract should never use delegatecall due to its significant potential for abuse. Calling untrusted, external functions is extremely dangerous due to the potential that they are malicious, but even internal delegatecalls can be risky. An attacker may be able to exploit the use of delegatecall to evade access controls as in the case of the Furucombo case study.

Summary

- Introduction to Platform-Specific Vulnerabilities
- Denial of Service: Block Gas Limits
- Denial of Service: Unexpected Revert
- Forced Send of Ether
- Missing Zero Address Checks
- Reentrancy
- Short Addresses
- Token Standards Compatibility
- Unchecked Return Values
- Unsafe External Calls



Module 5: Smart Contract Security

Section 5.5: Decentralized Finance (DeFi) Vulnerabilities



Overview

- What is Decentralized Finance
- Introduction to DeFi Vulnerabilities
- Access Control
- Centralized Control and Governance
- Cross-Chain Bridge Vulnerabilities
- Frontend Vulnerabilities
- Price Manipulation



What is Decentralized Finance?

- Bitcoin created a decentralized system for value transfers
 - Recorded transactions on the distributed ledger
- Decentralized finance uses smart contracts to implement other features of financial systems
 - Investing
 - Lending
 - Trading



The primary goal of Bitcoin was to provide an alternative to centralized financial systems in which users were dependent on a bank or other centralized authority to maintain a trusted ledger of asset exchanges. The blockchain creates a system in which the digital ledger is maintained by a distributed and decentralized network of mutually-distrusting blockchain nodes.

Tracking exchanges of value is only one function performed by financial institutions. Decentralized Finance (DeFi) uses the expanded capabilities of smart contract platforms to implement other functions, such as investing, lending, and trading. These DeFi projects are designed to run on the blockchain in a distributed and transparent fashion that provides clear visibility into their internal operations, unlike traditional financial institutions.

Introduction to DeFi Vulnerabilities

- DeFi is a rapidly-growing application of smart contracts
- Its unique use cases introduce new application-specific security risks and threats:
 - Access Control
 - Centralized Control and Governance
 - Cross-Chain Bridge Vulnerabilities
 - Frontend Vulnerabilities
 - Price Manipulation



Smart contracts are Turing-complete programs, meaning that they can be used to implement a wide variety of potential applications. However, certain use cases have gained more attention and widespread adoption than others.

DeFi is an example of a popular, rapidly-growing, and valuable application of smart contracts. DeFi projects have received a massive amount of investment. However, the security issues and vulnerabilities in these contracts have also contributed to huge hacks.

DeFi smart contracts inherit all of the same potential vulnerabilities and security risks as smart contracts designed for other applications that run on a particular platform.

However, the unique nature and value of DeFi creates new security threats and amplifies others, such as:

- Access Control
- Centralized Control and Governance

- Cross-Chain Bridge Vulnerabilities
- Frontend Vulnerabilities
- Price Manipulation

Access Control

272 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Access Control

- Access control vulnerabilities exist in all smart contracts
- The unique nature of DeFi makes them especially dangerous
 - DeFi contracts are extremely valuable and dangerous
- DeFi introduces new types of functions that require protection
 - Burn functions
 - Mint functions



Access control vulnerabilities are a topic covered in an earlier discussion of blockchain-specific smart contract vulnerabilities. The design of the blockchain makes internal access management essential within a smart contract.

All of the vulnerabilities and best practices discussed previously also apply to DeFi smart contracts. However, the unique nature of DeFi makes access control vulnerabilities especially dangerous. These contracts commonly hold massive amounts of value, and privileged functions are extremely powerful. Exploitation of access control vulnerabilities has resulted in numerous DeFi hacks with price tags in the millions.

In addition to amplifying the impact of a potential access control vulnerability, DeFi also expands the range of functions that need to be protected. For example, DeFi contracts often create native tokens that can represent debt or a share in the value of a DeFi project.

In a DeFi contract, new tokens can be created out of thin air using mint functions, and burn functions can destroy existing ones. Both of these affect the overall supply of the tokens, and, therefore, their intrinsic value. Restricting access to these and other

privileged functions is essential to the security of DeFi smart contracts and projects.

Access Control Vulnerability Example

```
1 function mint(address to, uint256 amount) public virtual {  
2     _mint(to,amount);  
3 }
```

Source: <https://twitter.com/RugDocIO/status/1451067795140005891>

274 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample demonstrates an access control function relating to an unprotected mint function. As mentioned previously, a mint function has the ability to create new tokens out of nothing. By expanding the supply of these tokens, a mint function can decrease the value of each token. However, it also places the new tokens in the hands of an attacker, which means that the value lost by other tokens is transferred to them, enabling them to steal value from a DeFi protocol.

In this case, the fact that the mint function is labeled as public is a significant access control vulnerability. With a public mint function, anyone can create any amount of new tokens. Exploiting this vulnerability could provide an attacker with tokens that they could trade for other tokens deposited within the project's smart contract, enabling them to drain value from the protocol.

Case Study: Zenon Network

- The Zenon smart contract included an unprotected burn function
 - The publicly accessible function allowed anyone to destroy wZNN tokens
- The attacker made a deposit to earn some wZNN tokens
- They destroyed over 26k wZNN tokens by calling the public burn function
 - Other wZNN tokens are more valuable as a result
 - Attacker redeemed their wZNN tokens for more than \$1 million in WBNB

Source: <https://twitter.com/peckshield/status/1462165620506742784>

275 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The Zenon Network was a DeFi project that suffered a hack in November 2021. The attacker took advantage of the fact that the burn function in the protocol's smart contract was inappropriately labeled as public.

Burn functions are designed to destroy instances of a protocol's native token. This increases the value of the remaining tokens because the value of the protocol is distributed over fewer tokens. With an unprotected burn function, anyone could destroy wZNN tokens to manipulate their value.

To benefit from their attack, the attacker needed to be a holder of wZNN tokens. They deposited tokens into the protocol and received wZNN in exchange. Then, they called the public burn function, triggering the destruction of over 26,000 wZNN tokens.

By significantly depleting the supply of wZNN tokens, the attacker increased the value of the remaining tokens. This allowed the attacker to trade in the tokens that they held for over \$1 million in WBNB tokens.

Access Control Vulnerability Mitigations

- Most smart contracts have privileged functionality that needs to be protected
 - Smart contracts should manage access to these functions
 - Functions inappropriately marked as public are the most common access control vulnerability
- The access management best practices for smart contracts in general also apply to DeFi contracts in particular
 - Default private functions
 - Modularize functionality
 - Multisignature wallets
 - Inter-function relationship analysis



Blockchain systems allow anyone to create a blockchain account and interact with smart contracts. Since many smart contracts have privileged functionality not intended for public access, they must implement internal access management that controls access to these functions. The most common access control vulnerability is inappropriately marking a function as public, allowing anyone to call it.

DeFi smart contracts are just another type of smart contract albeit one holding significant amounts of value. Most DeFi access management mistakes are the same as those for smart contracts in general, and the same access management best practices apply:

- **Default Private Functions:** Functions should be set to private by default and only made public if appropriate and necessary.
- **Modularize Functionality:** Breaking functions into smaller pieces enables more granular access controls.
- **Multisignature Wallets:** Privileged accounts should be protected by multiple private keys to prevent abuse of privileges.

- **Inter-Function Relationship Analysis:** Relationships between functions should be analyzed to ensure that private functions cannot be called by public ones, etc.

Centralized Control and Governance

277 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Centralized Control and Governance

- Access controls are designed to restrict access to certain functionality
 - Only privileged accounts can call certain functions
- Effective access control mechanisms centralize control
 - Certain parties have significant power
- This creates security risks
 - Private key theft
 - Rug pulls
 - Governance exploits



The goal of access control mechanisms is to limit who can access certain privileged and dangerous functionality. For example, an effective access control mechanism might be designed to allow only the owner of a smart contract to withdraw value from it or self-destruct it.

While strong access control protects privileged functions from public access, it also has the effect of centralizing power within the smart contract. An effective access control mechanism will only allow a few parties to access certain functionality. This gives these parties more power, which also makes them a target for attackers.

Some of the security risks associated with the centralization of control and governance created by access control include:

- **Private Key Theft:** Smart contract access control mechanisms indicate that only certain addresses should be able to call certain functions, and running these functions requires generating and signing transactions with the account's private key. If this private key is compromised or abused, then it could be used to steal from the smart contract or take other malicious actions.

- **Rug Pulls:** DeFi projects are often developed and maintained by a small team that has elevated permissions on the project's smart contract(s). In some cases, these projects are a scam where, after users have invested money in the contract, the team will use their privileged accounts to drain this money and disappear, deleting the project's website, social media accounts, etc.
- **Governance Exploits:** DeFi smart contracts commonly have access control mechanisms built in that restrict access to certain functions and functionality. If these access control mechanisms have a flaw, then an attacker can take advantage of them to gain control over the contract and have unrestricted access to it.

Centralized Control and Governance Example

```
1 function mint(uint256 amount) public onlyOwner returns (bool) {  
2     _mint(_msgSender(), amount);  
3     return true;  
4 }
```

Source: <https://letmeape.medium.com/how-to-spot-a-potential-rug-clear-signs-something-is-sketchy-169fb84c7084#59e7>

279 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This code sample is an example of a corrected version of the unprotected mint function discussed in the DeFi access control section. Instead of being completely public, this function is labeled as `onlyOwner`, which indicates that only the owner of the smart contract has the ability to call this function and create new tokens.

However, while this is desirable from an access control standpoint, it also has the effect of centralizing power in the contract owner's account. If an attacker can gain access to the owner's account by stealing their private key or by compromising the mechanism that defines the ownership label, then they can access this mint function. This would then allow them to create new tokens, enabling them to drain value from the protocol.

Case Study: bZx

- bZx is a DeFi protocol that suffered a \$55 million hack in November 2021
 - This includes losses from both the smart contract and its users' accounts
- The bZx hack began as a spear phishing email to the bZx developer
 - Installed malware on their computer
 - Stole private keys belonging to the developer and to two bZx contracts
- These keys were used to steal from the developer's account, the smart contract, and users who had authorized unlimited spends

Source: <https://bx.network/blog/preliminary-post-mortem>

280 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In November 2021, the bZx DeFi protocol was the victim of a hack. The attacker gained access to the contract's account and stole approximately \$55 million in tokens. These losses were distributed across a bZx developer, the contract's funds, and the accounts of bZx users.

The bZx hack began as a phishing attack targeting one of the developers behind the bZx protocol. This phishing email delivered malware to that developer's computer. Due to the developer's privileges on the bZx contract, they had access to the private keys of the accounts used to deploy bZx contracts on the blockchain. The attacker stole both of these keys as well as the private key of the bZx developer's private account.

With the privileges associated with the private keys of the bZx accounts, the attacker was able to drain the project's funds from the contract and to extract tokens from the accounts of users who had authorized unlimited spends of those tokens. The attacker also accessed the developer's personal account and drained it of funds as well.

Centralized Control and Governance Mitigations

- Centralized control is a significant security issue in DeFi smart contracts
 - Several DeFi hacks have involved private key thefts or rug pulls
 - Centralization creates single points of failure
- Decentralized access control can help to manage these risks:
 - Multi-signature wallets
 - Separation of power
 - Decentralized governance mechanisms



The centralization of access and power to a DeFi project's team creates significant security risks for the protocol. Many DeFi hacks have involved the private keys of the account governing the project's smart contract. This includes both theft/leakage of private keys and rug pulls in which the team abuses their access to steal from the project and its users.

Centralization of power is a common result of implementing access control mechanisms, but it also creates security risks. Centralization produces single points of failures that can be prime targets for attack or make it possible for privileged parties to abuse their power.

Implementing decentralization in access control mechanisms can help to protect against these types of threats. Some ways to do so include:

- **Multi-Signature Wallets:** Multi-signature wallets require multiple private keys to generate a valid signature for a blockchain account. A multi-signature wallet raises the bar for an attack or abuse of power because the attacker needs access to all of the private keys for a controlling account to carry out their attack.

- **Separation of Power:** In some cases, DeFi smart contracts define a single privileged role (like owner) that has full access and control over all privileged functionality. Defining multiple roles and assigning privilege accordingly can make attacks more difficult to perform because an attacker may need to gain access to several sets of permissions to achieve their goals
- **Decentralized Governance Mechanisms:** While giving the project team elevated privileges is the simplest way to implement access management, it is not the only one. DeFi projects could weaken the power of the project team by requiring some actions (such as draining value from the protocol) to require a vote by holders of the project's native token or some similar decentralized mechanism.

Cross-Chain Bridge Vulnerabilities



Introduction to Cross-Chain Bridge Vulnerabilities

- Bridges are designed to link two blockchains together
 - Deposits into the smart contract on one bridge release funds on the other
 - Allow users to take advantage of the various benefits of different blockchains
- Bridges are often implemented as smart contracts
 - Code listens for events on one blockchain and releases tokens in response
- Blockchain bridges can allow cross-chain exploits
 - Exploitation of a vulnerability on one end of bridge can release tokens on other end



Different blockchains have different strengths and weaknesses. For example, Bitcoin is a well-established and secure blockchain but it lacks support for smart contracts. Ethereum offers smart contract support, but it has currently significant scalability and speed limitations.

Cross-chain bridges create links between independent blockchains, allowing assets to be transferred from one blockchain to another. If a user sends cryptocurrency to an account controlled by the bridge, this unlocks cryptocurrency at the other end. This is how “wrapped” tokens (wBTC, etc.) work, and some bridges use liquidity pools where they provide users with the requested amount of a particular token based on what is available. With these bridges, a user can take advantage of the benefits of different blockchains.

While bridges can be implemented as centralized systems, they are often implemented using smart contracts. A contract on each blockchain manages its pool of tokens, accepting deposits and performing withdrawals based on events on the other blockchain. This makes it possible to implement a decentralized system for cross-chain transfers.

While blockchain bridges have their advantages, they also create security risks. If the blockchain bridge's code contains design flaws, implementation errors, or other vulnerabilities, an attacker can exploit these and produce impacts on multiple blockchains. For example, previous hacks of cross-chain bridges have involved fake deposits of tokens into one bridge contract that release real tokens at the other end.

Case Study: Axie Infinity/Ronin

- In March 2022, Axie Infinity's Ronin Bridge was the victim of a \$600 million hack
- The Ronin bridge used a set of nine validators responsible for approving cross-chain transfers
 - The attacker was able to gain access to the signing keys for five of these validators
 - Four controlled by Sky Mavis and a third-party Axie DAO node that delegated signing powers to Sky Mavis
- With a majority of validators, the attacker approved two malicious transactions that drained \$600 million worth of tokens from the protocol

Source: <https://rekt.news/ronin-rekt/>

284 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

The Ronin bridge is a cross-chain bridge designed to link the Ronin sidechain to the Ethereum network. This bridge used nine validators to approve cross-chain transfers, and a majority vote was needed for a transfer to go through.

In March 2022, these validators were extremely centralized with four of the nine under the control of Sky Mavis. An attacker managed to steal the private keys associated with these nodes, meaning that they only needed one more key to validate any transaction. Several months earlier, scalability issues had inspired an arrangement where Sky Mavis could perform validations on behalf of a third-party node operated by Axie DAO. While this arrangement had ended, the permissions has not been revoked, and the attacker was able to use these delegated permissions to gain the validator majority that they needed.

In March 2022, the attacker generated two malicious transactions worth a combined \$600 million. With control over a majority of the validators, they were able to independently validate these transactions, allowing them to withdraw the tokens on the Ethereum network. The hack was only discovered a week later when a user reported an issue making a large withdrawal from the bridge, which failed because the available assets had been drained.

Cross-Chain Bridge Vulnerability Mitigations

- Cross-chain bridges are high-value targets where attacks can have an outsized effect
- Cross-chain bridge best practices highlighted by recent hacks include:
 - Decentralized Validation
 - Smart Contract Code Review
 - Change Management Processes



Cross-chain bridges can be a valuable asset, but they are also high-value targets to attackers. Since the bridge contract holds the tokens sent to it when a user transfers value to another contract, the value of these contracts can be substantial. Also, these are centralized targets where attacks can impact the security of multiple different blockchains.

Development best practices for cross-chain bridge contracts are similar to those of other smart contracts and DeFi projects. Some best practices highlighted by previous hacks of cross-chain bridges include:

- **Decentralized Validation:** Some cross-chain bridges have a system for validating and approving cross-chain transfers over the bridge. As demonstrated by the Axie Infinity/Ronin hack described previously, a centralized validation process can leave a cross-chain bridge vulnerable to attack.
- **Smart Contract Code Review:** Decentralized cross-chain bridges rely on smart contracts to properly manage transfers of tokens between the two linked blockchains. If these smart contracts have vulnerabilities, then they could put the bridge at risk. Some hacks of cross-chain bridges, such as the Meter.io hack

(<https://rekt.news/meter-rekt/>), were made possible by vulnerabilities that could have been detected and fixed during a smart contract audit.

- **Change Management Processes:** Another contributor to the Meter.io hack was poor change management processes in which outdated code was permitted to remain in the smart contract, leaving it vulnerable to attack. The code of cross-chain bridges should only be updated as part of a structured process that involves code reviews and security audits.

Frontend Vulnerabilities

286 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Frontend Vulnerabilities

- DeFi projects are not just a smart contract
- They also commonly include a frontend website
 - This makes it easier for users to interact with the service
- This web frontend is part of the DeFi project's attack surface and security model
 - Frontend vulnerabilities impact smart contract security
 - The reverse can also be true



Most users of DeFi projects are not writing transactions and directly interacting with the project's smart contract. While this is certainly possible, it requires a certain level of technical expertise and is not very user-friendly.

Instead, most DeFi projects implement a web frontend for their users to interact with. This works similarly to a traditional website, which is composed of a frontend website and backend infrastructure hosted in a datacenter somewhere. The primary difference is that a DeFi decentralized application (DApp) is composed of a traditional web frontend supported by a backend smart contract.

When evaluating the security of DeFi projects and other DApps, both the smart contract and the web frontend are part of the attack surface. If a web frontend contains vulnerabilities or the interface between it and the smart contract backend is incorrect, then there might be opportunities for exploitation by an attacker.

Case Study: BadgerDAO

- In December 2021, an attacker exploited the BadgerDAO project's web frontend
 - Allowed them to steal over \$120 million in tokens from the protocol's users
- The attacker gained access to the frontend and inserted malicious scripts
 - These scripts added approvals for sending tokens to the attacker
- When users generated transactions, these approvals were included
 - Over 500 wallets were affected, leading to over \$120 million in losses

Source: <https://rekt.news/badger-rekt/>

288 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In December 2021, the BadgerDAO DeFi project suffered a frontend exploit. The attacker took advantage of access to the project's web frontend to insert malicious functionality that allowed them to steal tokens from users' accounts.

The attacker is believed to have gained access to a compromised API key for BadgerDAO's Cloudflare deployment. With this access, they were able to inject malicious scripts into the project's web frontend.

When users visited the website and attempted to build transactions, the malicious scripts added additional functionality to these transactions. This included unlimited approvals for extracting certain tokens from the user's accounts. After the user signed these transactions and sent them to the blockchain, the approvals were in place.

With these approvals, the attacker was able to withdraw tokens from affected user accounts. In total approximately 500 BadgerDAO users generated the malicious transactions, allowing the attacker to extract over \$120 million in tokens from their accounts.

Frontend Vulnerability Mitigations

- Web frontends are part of many DeFi projects
 - Make the system more accessible and user-friendly
 - Expand the attack surface
- The security of frontend websites is a known challenge
 - Code and design review
 - Cross-site scripting (XSS) protections
 - HTTP Strict Transport Security (HSTS)



Most DeFi projects consist of a web frontend and a smart contract backend. The addition of a web frontend makes the system more accessible to users, especially compared to reading the smart contract's code and the blockchain's state and creating transactions accordingly.

However, a web frontend also has the effect of expanding the potential attack surface of a DeFi DApp. With a web frontend comes the potential for frontend vulnerabilities and for interface mismatches between the web frontend and smart contract backend that can result in undesirable effects.

Securing web frontends and validating their interfaces with backend systems is not a new problem that originates with the blockchain. Security best practices such as performing code and design review, implementing protections against common attacks like cross-site scripting (XSS), and implementing security best practices such as the use of HSTS apply to DApps as well as traditional web apps.

Price Manipulation

290 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Price Manipulation

- Flashloan exploits are one of the most common attacks against DeFi projects
 - These exploit price manipulation vulnerabilities
- DeFi projects often calculate tokens' values based on relative supply and demand
- On-chain price calculations can be manipulated by large deposits
 - Causes "slippage" that creates arbitrage opportunities



Flashloans are a concept that was introduced with the DeFi ecosystem. The design of the blockchain makes it possible to make risk-free loans that don't require the borrowers to provide collateral.

Normally, lenders require collateral for loans to hedge against the risk of a borrower defaulting on a loan. If the loan is not repaid, the lender can seize the collateral to recoup some of their losses. Since the collateral needs to be proportional to the loan, this creates a cap on the size of loan that a borrower can request.

Flashloans eliminate the need for collateral by eliminating the risk of a default. This is possible because flashloans are taken out and paid off within the same transaction. If something goes wrong and the borrower would be unable to pay off the loan, then the whole transaction is reverted and the initial loan never happened. Without risk, there is no need for collateral and no collateral-driven cap on loan amounts. Therefore, flashloans allow borrowers to take out massive loans.

This is relevant because of the means by which many DeFi projects calculate the value of tokens. Often, a DeFi token will calculate a token's value based on the supply of that token and the overall value of the protocol. If a protocol's smart contracts hold

a large amount of investment and there are relatively few tokens that represent a share of that investment, then the value of each token is higher than for a protocol with less investment or more tokens.

This type of price calculation can be manipulated by large deposits of value into a protocol, which are enabled by flashloans. By taking out a massive loan and using it to manipulate the perceived value of a DeFi project's tokens, an attacker can drain value from the protocol by taking advantage of the inflated value.

Price Manipulation Example

```
1 function calcLiquidityShare(uint units, address token, address pool, address member) {
2     // share = amount * part/total
3     // address pool = getPool(token);
4     uint amount = iBEP20(token).balanceOf(pool);
5     uint totalSupply = iBEP20(pool).totalSupply();
6     return(amount.mul(units)).div(totalSupply);
7 }
```

Source: <https://peckshield.medium.com/the-spartan-incident-root-cause-analysis-a0324cb4b42a>

292 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

This sample code is designed to calculate the share of a DeFi project's liquidity that belongs to a particular user. The token has a certain number of tokens (units) that are invested in a particular liquidity pool.

This function calculates the value of the user's liquidity share based on the value of the liquidity pool and the percentage of tokens that the user holds. Therefore, the value of the user's share is dependent on the total value of the pool and the number of tokens that exist.

An attacker can manipulate either of these values. In a flashloan exploit, the attacker can take out a massive loan and use that to inflate the total balance of the pool. This inflates the perceived value of each token held by the attacker, enabling them to extract an outsized liquidity share.

Other attacks could allow an attacker to manipulate the number of tokens in existence. For example, an access control vulnerability that creates an unprotected mint or burn function could allow an attacker to create or destroy tokens. This also provides the potential for an attacker to claim more liquidity than they are entitled to.

Case Study: Cream Finance

- In October 2021, Cream Finance suffered a \$130 million hack
- The attackers performed repeated actions to build up tokens
 - Deposited tokens (initially from a flashloan) into Cream
 - Used deposit as collateral for a loan, which was then deposited
 - In the end had ~1.5 billion in crYUSD and ~500M in yUSDVault
- Redeeming yUSDVault for yUSD decreased total supply to \$8 million
 - Depositing \$8M in yUSD doubled perceived value of crYUSD
 - Redeeming crYUSD paid off loans and enabled theft of \$130 million

Source: <https://rekt.news/cream-rekt-2/>

293 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In October 2021, the Cream Finance DeFi project suffered a flashloan attack. The attacker profited by \$130 million due to their attack.

After taking out a flashloan, the attacker began a series of operations designed to build up a large supply of tokens. This included depositing tokens into the vault and using that deposit as collateral for a loan. The attacker could then deposit the loan to use as more collateral, etc. In the end, the attacker held approximately \$1.5 billion worth of crYUSD tokens and about \$500 million in yUSDVault tokens.

After building up this token supply, the attacker's next move was to manipulate the perceived value of their crYUSD tokens. To do so, they redeemed their yUSDVault tokens for yUSD, which decreased the vault's total supply of yUSD to \$8 million. The attacker then deposited \$8 million of yUSD into the vault.

This deposit doubled the amount of yUSD in the vault, which also doubled the perceived value of the crYUSD token. The attacker took advantage of this slippage by redeeming the crYUSD tokens that they held for the contents of the vault. Since their tokens were overvalued, the attacker was about to extract enough tokens to pay back their loans and flashloan and to make a profit of \$130 million (all that was left in the

vault.

Price Manipulation Mitigations

- Price manipulation vulnerabilities arise from on-chain price calculations
 - These could occur inside of a contract or in other contracts
 - Attackers can manipulate these calculations to create slippage
- Price manipulation vulnerabilities can be mitigated by using off-chain sources of pricing information
 - Chainlink, etc.



Calculating the value of assets on-chain creates the potential for price manipulation vulnerabilities. These calculations are based on the current value of a protocol and the quantity of its native tokens. Both of these values can be manipulated by an attacker through the use of flashloan or access control exploits. Whether this calculation is performed within the contract itself or in another contract that uses a similar mechanism doesn't change vulnerability to price manipulation.

Any on-chain price calculation is potentially vulnerable to price manipulation attacks. For more accurate token valuations, smart contracts should use an off-chain price oracle like Chainlink. These offline oracles have wider context and cannot be manipulated by flashloan attacks, making them a more reliable source of token pricing information.

Summary

- What is Decentralized Finance
- Introduction to DeFi Vulnerabilities
- Access Control
- Centralized Control and Governance
- Cross-Chain Bridge Vulnerabilities
- Frontend Vulnerabilities
- Price Manipulation



Module 5: Smart Contract Security

Section 5.6: Non-Fungible Token (NFT) Vulnerabilities



Overview

- What are NFTs?
- Introduction to NFT Vulnerabilities
- Centralized NFT Platforms
- Forged NFTs
- Off-Chain Assets
- Malicious NFTs
- Unlimited Token Supplies



What Are NFTs?

- Most blockchain tokens are fungible
 - Identical tokens carry identical value
 - Can be exchanged 1:1
 - Similar to dollar bill
- Non-fungible tokens have unique value
 - Two similar NFTs can carry wildly different values
 - Similar to baseball cards
- NFTs can track ownership of assets on the blockchain



Most blockchains have their own native cryptocurrency or token. For example, Bitcoin has Bitcoin (BTC), Ethereum has Ether, etc. One valuable feature of smart contract platforms like Ethereum is the ability to quickly and easily create new tokens that are hosted on an existing blockchain.

The native tokens used by various blockchains and many others hosted on smart contract platforms are fungible tokens. This means that two tokens of the same “type” carry the same value and can be exchanged 1:1 with no loss of value. The paper money or coinage of a country is an example of a fungible asset since any dollar bill is worth the same as any other dollar bill. Similarly, each Bitcoin or Ether has the same value.

Non-fungible tokens (NFTs) are a type of token in which each token is unique and has a unique value. Two largely similar NFTs can have wildly different values. A baseball card or other collectible card is an example of a non-fungible asset since cards of certain players are much more valuable than others.

NFTs on the blockchain can be used for various purposes. One of the most popular is tracking ownership of various digital and real-world assets.

Introduction to NFT Vulnerabilities

- NFTs are a rapidly-growing market
 - Commonly used to track ownership of digital art
- These NFTs are created using smart contracts
- NFTs have unique security risks, such as:
 - Centralized NFT Platforms
 - Forged NFTs
 - Off-Chain Assets
 - Malicious NFTs
 - Unlimited Token Supplies



NFTs can be used to track various assets, including land, luxury good, items in a supply chain, etc. However, the most popular application is to track ownership of digital art. While anyone can download a copy of a piece of digital art, NFTs confer “ownership” of that piece.

NFTs are generally hosted on smart contract platforms and created using smart contracts. Ethereum and other platforms have made it easy to create a new NFT, and blockchain technology is ideally suited to tracking and transferring ownership of tokens.

NFTs are a unique application of smart contract technology and they come with security risks. Some of the primary risks associated with NFTs include:

- Centralized NFT Platforms
- Forged NFTs
- Off-Chain Assets

- Malicious NFTs
- Unlimited Token Supplies

Centralized NFT Platforms

300 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Centralized NFT Platforms

- NFTs are stored as tokens on the blockchain
- Most users manage their NFTs via centralized, web-based platforms
 - Open Sea, Axie Marketplace, etc.
- NFT platform centralization introduces security risks
 - Denial of Service (DoS) attacks
 - Platform hacks



NFTs are designed to track ownership of assets on the blockchain. They are implemented as blockchain-based tokens that are stored on the blockchain's decentralized digital ledger.

However, most NFT owners don't interact directly with the blockchain and NFT contracts. Instead, they use a web-based frontend to view and manage their NFTs. Examples of these platforms include Open Sea and Axie Marketplace.

These platforms are centralized, which creates security issues, including:

- **Denial of Service (DoS) Attacks:** The centralization of a NFT platform creates a single point of failure for users of that platform. A Distributed DoS (DDoS) attack could render the platform inaccessible and prevent NFT owners from buying, selling, or viewing their assets.
- **Platform Hacks:** NFT users commonly link their blockchain wallets to NFT platforms, providing them access to the user's private keys and the ability to sign transactions. If the NFT platform is compromised, an attacker may be able to access users' keys or insert malicious scripts into the platform's website to steal

sensitive information.

Case Study: Open Sea

- In January 2022, Open Sea users were the victim of a hack exploiting the design of the NFT platform
- On the frontend, transferring an NFT between accounts removed sale listings
 - This did not trigger transactions to the smart contract backend
- Old listings were still valid on the smart contract
 - Attacker interacted with it directly to buy NFTs at old list prices

Source: <https://decrypt.co/91076/opensea-exploit-sees-bored-ape-yacht-club-nft-sell-1700>

302 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In January 2022, several Open Sea users had their NFTs sold for far under their current value. This attack took advantage of the interface between Open Sea's web frontend and smart contract backend.

To save gas fees, some NFT owners took advantage of a trick to cancel old token sale listings on the Open Sea marketplace. Instead of sending a transaction to delist the tokens, they swapped them between wallets that they owned. This resulted in the tokens being no longer listed for sale on the Open Sea web frontend.

However, this trick was designed to save money by not having the delisting information recorded on the blockchain, which would cost gas. As a result, the backend smart contract was not informed that the token listing was cancelled and still listed it for sale at the old price.

An attacker identified this inconsistency and exploited it by sending transactions buying NFTs directly to the smart contract, which accepted them and transferred the NFTs. This enabled them to buy them at old sale prices when they were worth significantly less and then sell them for current values, making a substantial profit.

NFT Platform Centralization Mitigations

- Most NFT users interact with them via centralized platforms
 - This creates security risks
- The risks of platform centralization can be mitigated by
 - Anti-DDoS protections
 - Web frontend vulnerability scanning
 - Business logic validation



NFTs track ownership of assets on the decentralized blockchain. However, the platforms that are used to buy, sell, and view common NFTs, such as those tracking ownership of digital assets, are centralized. The centralization of NFT platforms creates security risks; however, these platforms are unlikely to become less centralized.

Mitigating the risks of NFT platform centralization requires directly addressing the threats that it poses. Some potential mitigations include:

- **Anti-DDoS Protections:** Centralized NFT platforms are potentially at risk of DDoS attacks that exploit the fact that all users are reliant on the accessibility of the centralized NFT website for buying and selling their NFTs. Deploying DDoS mitigation solutions can help to reduce these platforms' exposure to DDoS attacks.
- **Frontend Vulnerability Scanning:** NFT platforms' web frontends expose them to common web security threats such as cross-site scripting (XSS) attacks. NFT platforms' frontends should be scanned for vulnerabilities and malicious scripts that place their users at risk.

- **Business Logic and Implementation Validation:** Interface mismatches such as the one that allowed the unauthorized sale of NFTs on OpenSea put NFT owners and their assets at risks. Code should undergo security audits that validate that it works as expected.

Forged NFTs

304 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Forged NFTs

- NFTs are designed to track ownership of assets on the blockchain
 - Transparent digital ledger makes it easy to verify ownership
 - Using tokens makes ownership transfers easy
- No good mechanism exists to prove that holding an NFT actually confers ownership
- NFTs made without the real owner's consent are valueless



The major selling point of NFTs is that they provide a way to transparently and securely track ownership of assets on the blockchain. The blockchain has a transparent, immutable digital ledger, which means that anyone can view and verify ownership information without the risk that records could be modified. Tying ownership to tokens makes transfers of ownership easy since they leverage the blockchain's underlying functionality.

With NFTs, it is possible to track ownership of assets that would otherwise be difficult to prove ownership of. For example, many modern NFTs are used to track ownership of digital art.

A major issue with NFTs is that it is difficult to prove that the true owner of an asset consented to it being tracked on the blockchain. Anyone can mint an NFT, and the existence of the NFT does not necessarily mean that its owner actually owns the associated asset. This makes the NFT essentially valueless because it does not actually prove ownership of anything

Case Study: Derek Laufman

- Derek Laufman is an artist whose work was being sold as NFTs on the Rarible marketplace
- These NFTs were created without his knowledge or consent
- Rarible took down the account of the NFT forger after Laufman contacted him
 - At least one fan had already bought one of the fake NFTs

Source: <https://www.theverge.com/2021/3/20/22334527/nft-scams-artists-opensea-rarible-marble-cards-fraud-art>

306 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

One case of forged NFTs was discovered in March 2021. Derek Laufman is an artist who learned via messages from fans that NFTs of his art were currently being offered for sale on the Rarible NFT marketplace.

However, Laufman had not created the NFTs or consented to their creation. As a result, they provided no rights of ownership over his art. The scammer had gone through the process of impersonating Laufman through the entire Rarible account verification process to create a verified account purporting to be owned by Laufman himself. This account then offered the fake NFTs for sale.

After Laufman contacted them, Rarible deleted the fake profile and forged NFTs. However, by this point, at least one fan had already purchased one of the NFTs on the platform, wasting money on an asset that provided no true rights or proof of ownership.

Forged NFT Mitigations

- Proving that an NFT confers ownership of an asset is difficult
 - Anyone can mint an NFT
 - Multiple NFTs exist for some assets
- Background research can help in identify obvious fakes
- True confirmation requires validation from the real owner



NFTs are designed to track and prove ownership of assets on the blockchain; however, they don't do a very good job of that in some cases. While NFTs secure ownership of an NFT after the asset is created, there is no security built into the NFT generation process. Anyone can create an NFT, whether or not they are the actual owner of the asset and have the right to do so. As a result forged NFTs exist, and, in some cases, multiple versions of the same NFT exist.

Proving that ownership of an NFT actually proves ownership of the associated asset is a difficult task. In some cases, background research can help if an NFT is an obvious fake. If the website and other online presence of the NFT creators is sketchy or non-existent, then the NFT is likely fake.

However, more sophisticated forged NFTs may be harder to detect. In the end, only an attestation from the true owner of the NFT can prove that ownership of the NFT proves ownership of the asset.

Off-Chain Assets

308 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Off-Chain Assets

- Most modern NFTs are used to track ownership of digital art
 - Anyone can download the art but only one person “owns” it
- Storing NFT images on-chain is infeasible
 - Limited space on-chain
 - Contributes to blockchain bloat
- NFTs contain a URL or IPFS hash pointing to the image
 - NFTs prove ownership of a URL or hash, not the actual asset
 - Images can be taken down or changed (for a URL)



NFTs can be used to track ownership of any asset on the blockchain. However, the most common application of NFTs today is tracking the ownership of pieces of digital art. While anyone can download a copy of the file, the NFT indicates who the true “owner” is.

Blockchain platforms have significant space constraints due to block sizes. Also, the immutability of the digital ledger means that data cannot be deleted from the blockchain. This makes storing NFTs on-chain infeasible because copies of each image would be duplicated each time the asset was transferred, contributing to bloat on the blockchain.

As a result, many NFTs contain a URL or an Interplanetary File System (IPFS) hash pointing to the asset that the NFT asserts ownership over. This means that, in reality, an NFT proves “ownership” of a particular URL or IPFS hash, not the asset itself. This creates issues, including:

- **Deleted Images:** The URL or IPFS hash that an NFT asserts ownership over is only valuable if the digital image purchased by the NFT owner is still at that URL. However, URLs can go down at any time, and data only remains on the IPFS as long

as an IPFS gateway continues to host it.

- **Changed Images:** In the case of the IPFS, changing an image changes the hash, so IPFS-based NFTs are protected against modification. However, the content located at a particular URL can be changed by the owner of that URL at any time.

Case Study: Cent and Other Platforms

- Many NFT platforms have taken down NFTs that have been found to infringe on the rights of the artist
 - Rarible and Derek Laufman is an example
- In February 2022, Cent stopped sales of many NFTs due to “rampant fakes and plagiarism” on their platform

Source: <https://www.reuters.com/business/finance/nft-marketplace-shuts-citing-rampant-fakes-plagiarism-problem-2022-02-11/>

310 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Deletion of NFTs is common practice in the event that an NFT is found to be illegitimate and infringes on the rights of the artist or true owner of the art. One example of this is the deletion of the fake verified account for Derek Laufman on the Rarible NFT platform.

Another example of NFTs being taken down en masse is the decision by the Cent platform in February 2022 to halt the sales of many NFTs on its platform. The rationale for this decision was that “rampant fakes and plagiarism” existed on the platform and they wanted to halt the infringement on the rights of the true owners.

These are only a couple examples of NFTs being taken down by NFT platforms for legitimate reasons.

Off-Chain Asset Mitigations

- NFTs prove ownership of a URL or IPFS hash, not the image at that location
 - Content at a particular URL or hash can be deleted at any time
 - Content indicated by a URL can be changed
- Protecting an NFT investment requires an owner to take over hosting of their NFTs
 - Claim the URL or become an IPFS gateway
 - Not always possible



In theory, the NFTs in common use today prove ownership of a particular piece of digital art. In reality, they only lay claim to a URL or IPFS hash. The content at that location can be taken down at any time, and, in the case of URLs, the owner of the URL can change the image or content hosted at that location.

An NFT is only valuable if it still points to the image that the owner bought the rights to. Protecting that investment requires the owner to ensure that the image remains online, often by hosting it themselves.

However, this is not always possible. For example, if a party owns a URL and refuses to hand it over to the person with an NFT pointing to it, then they can delete or change the image at that location at any time.

Malicious NFTs

312 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Malicious NFTs

- Many modern NFTs track ownership of digital art
- This art is often stored off-chain
 - Users need to follow a link to view the art
- Links to images are an ideal vector for phishing attacks
 - Malicious code embedded in the image could steal users' private keys or perform transactions on their behalf



NFTs can be used to track ownership of a variety of different assets. However, the most common application today is tracking ownership of digital art.

As mentioned previously, this art is often stored off-chain for various reasons. Instead of storing the image itself immutably on the blockchain, an NFT contains a URL or IPFS hash that points to the asset. This minimizes the bloat on the blockchain caused by NFTs.

To view the image associated with an NFT, a user needs to follow the link indicated by it. This is an ideal setup for phishing attacks in which an attacker can convince a user to visit an image, which may contain malicious code, within their browser.

Attackers can embed malicious scripts in the image itself or the page hosting it. When a user visits, these scripts could potentially access the user's private keys of the user's wallet, especially if the page claims that providing them is required to "claim" an NFT sent to them. Alternatively, an attacker could add content to a user-signed transaction that extracts value or tokens from their account. In either case, the malicious NFT allows the theft of valuable NFTs and other tokens from the user.

Case Study: Check Point and Open Sea

- In October 2021, Check Point Research identified vulnerabilities that permitted malicious NFTs in the Open Sea marketplace
- An attacker could generate SVG files that generated malicious popups
 - Prompted users to enter their credentials or private key
- No evidence exists that this vulnerability was exploited
 - Open Sea implemented protections against this vector on their platform

Source: <https://blog.checkpoint.com/2021/10/13/check-point-software-prevents-theft-of-crypto-wallets-on-opensea-the-worlds-largest-nft-marketplace/>

314 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

In October 2021, Check Point Research (CPR) revealed the result of their research on malicious NFTs. This research was kicked off based on curiosity about NFT owners claiming that their wallets were hacked after receiving airdropped tokens.

CPR's research revealed that attackers could create crafted malicious SVG image files for NFTs. On the Open Sea marketplace, users viewing the malicious tokens airdropped into their wallets would be presented with a popup that requested their login credentials or private key. With this information, an attacker could drain value from their accounts.

While CPR was able to develop a working proof of concept (PoC) of their attack, Open Sea found no evidence that the vulnerability had actually been exploited by an attacker. After CPR's report, Open Sea made modifications to their platform to protect against this type of attack.

Malicious NFT Mitigations

- Malicious NFTs include URLs that point to images containing malicious code
 - Viewing the image executes the code in the user's browser



Unlimited Token Supplies

316 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Introduction to Unlimited Token Supplies

- NFTs have value because they are a scarce resource
 - Only so many NFTs exist in a collection
- Some NFT smart contracts lack limits on token supplies
 - New NFTs could be added to the collection at will
- This could undermine the value of existing NFTs



Digital art NFTs are valuable because there are only a limited number of them in a given collection. If something is scarce, then demand for it increases its value, which is why some digital art NFT collections have such high values. The collection gains name recognition or is made by a talented or famous artist, so people are willing to pay more to have one.

Ideally, the smart contract creating an NFT would have a built-in constraint on the size of the collection, making it impossible for additional NFTs to be minted after the initial collection created. However, this is not always the case.

In some instances, NFT smart contracts have the ability to mint additional NFTs after the initial sale. Since this can make the asset no longer scarce – or at least creates the potential for breaking its scarcity – this can undermine the value of the NFT collection.

Case Study: Bored Apes Yacht Club

- Bored Apes Yacht Club (BAYC) is one of the most famous and expensive NFT collections
- The BAYC is composed of 10,000 images generated from 172 possible traits
- BAYC contract code includes a reserveApes function
 - Allows minting of 30 new apes at a time
 - Could be used to inflate the supply indefinitely

Source: <https://etherscan.io/address/0xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d#code>

318 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Bored Apes Yacht Club (BAYC) is one of the more popular and expensive NFT collections. The average price for a bored ape in March 2022 was approximately \$350,000. The entire collection is valued at over \$3.5 billion.

Like many digital art NFTs, BAYC NFTs derive their value from scarcity. In the case of BAYC, there are 10,000 apes, each of which is algorithmically generated based on 172 possible traits.

However, the BAYC smart contract does not enforce the scarcity of these digital assets. In fact, it included a function called `reserveApes` that would allow the owner of the contract to mint 30 new apes at any time. Since this contract could be called repeatedly, the owner could create as many apes as they wished.

Since BAYC's value depends on scarcity, this could have a dramatic impact on the value of the NFTs. Inflating the supply could devalue the tokens by destroying public trust in their value.

Unlimited Token Supply Mitigations

- Digit art NFTs like Bored Apes Yacht Club (BAYC) derive value from their scarcity
 - However, this scarcity is not always enforced in the contract code
 - Contract owners may be able to mint new NFTs at will
- Protecting the value of these NFTs requires protecting their scarcity
 - Collection sizes should be limited in code
 - Future planned upgrades should be governed by code that prevents unauthorized expansions



NFTs like the Bored Ape Yacht Club (BAYC) are valuable because they are rare. This NFT collection enjoys enormous popularity, and anyone that wants in on the hype needs to buy one of the limited number of existing apes. With a scarce asset, increased demand drives up the price, which is why bored apes are so expensive.

However, some NFT smart contracts, like BAYC, do not enforce this scarcity in their code. Contract owners have the ability to mint new NFTs at will, which risks the value of the collection.

Protecting the scarcity-based value of NFTs requires mechanisms that ensure that they remain rare. The maximum sizes of collections, which are a major selling point, should be written into the smart contract code to prevent unauthorized expansions. Any planned future minting should also be governed by code to prevent unauthorized expansion of a collection.

Summary

- What are NFTs?
- Introduction to NFT Vulnerabilities
- Centralized NFT Platforms
- Forged NFTs
- Off-Chain Assets
- Malicious NFTs
- Unlimited Token Supplies



Module 5: Smart Contract Security

Section 5.7: Securing Smart Contracts



Introduction to Smart Contract Security Testing

- Security testing is especially important for smart contracts
 - Blockchain immutability
 - Open source philosophy
 - Exposure to attack
- Security should be integrated into every stage of the smart contract lifecycle
 - Secure development
 - Smart contract security audits



Security testing is important for all types of software. The number of vulnerabilities in production applications rises each year, contributing to growing numbers of security incidents and data breaches.

Security testing is especially important for smart contracts for various reasons. Some of these include:

- **Blockchain Immutability:** The immutability of the blockchain makes it more difficult to remediate vulnerabilities in deployed contracts. Also, it makes it impossible to reverse an attack after it occurs.
- **Open Source Philosophy:** Many smart contracts are open source, which helps to build trust in contracts and aids development of new project. However, access to smart contracts makes it easier for attackers to identify vulnerabilities and develop exploits.
- **Exposure to Attack:** Many smart contracts are deployed on public blockchains where anyone can create anonymous accounts and interact with them. This means that these programs are easy for attackers to exploit with limited repercussions.

Smart contracts should be analyzed for vulnerabilities in every stage of their lifecycle. The sooner that a vulnerability is detected, the lower the cost of remediation. Both developers and security auditors should take advantage of the variety of smart contract security resources that are publicly available.

Secure Smart Contract Development Resources

323 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- Introduction to Security Resources
- Resources and Style Guides
- Vulnerability Listings
- Testnets
- Vulnerable Applications



Introduction to Security Resources

- Identifying vulnerabilities in development minimizes the cost
 - Cheaper to fix
 - No chance of exploitation
- Smart contract developers can take advantage of various resources
 - Resources and style guides
 - Vulnerability listings
 - Testnets
 - Vulnerable applications



The earlier that vulnerabilities are identified, the lower the cost to a smart contract project and its users. If vulnerabilities are identified during development, they can be fixed without creating and pushing an update to a smart contract. Also, vulnerabilities remediated before release have no chance of being exploited by an attacker.

Smart contract developers have access to a wide variety of resources that can help them to identify and fix vulnerabilities before release. Examples include:

- Resources and style guides
- Vulnerability listings
- Testnets
- Vulnerable applications

Resources and Style Guides

- Different smart contract platforms have their own idiosyncrasies and design patterns
 - What works for one doesn't for another
- These platforms often have developer documentation describing how the system works, known security issues, etc.
- Ethereum developer documentation is at <https://ethereum.org/en/developers/>



Smart contracts platforms dramatically extend the functionality of the basic blockchain. Each implements a virtual machine and a language for writing smart contracts in addition to any unique protocol changes that it makes. While some blockchains may use existing VMs and languages, others have custom implementations (like Ethereum's Solidity).

The wide variety in smart contract platforms means that each has its own idiosyncrasies and best practices. What might seem like a good idea and best practice in one language or environment may not work for another, and vulnerabilities may vary from one platform to another. For example, reentrancy in Ethereum and EOSIO are vulnerabilities with similar effects that work in very different ways. Also, the fact that transfer and send do essentially the same thing but have different methods of error handling is an Ethereum-specific quirk.

When working with or auditing a smart contract platform, reviewing the provided developer documentation is essential. This documentation will include descriptions of the theory behind the platform, example use cases and design patterns, and, often, information about known vulnerabilities and common coding errors.

Vulnerability Listings

- Vulnerabilities in traditional software are tracked as CVEs and CWEs
 - This is also true for some blockchain vulnerabilities
- Smart contract platforms also often have their own repositories of vulnerability information
 - Describe common errors, idiosyncrasies, etc.
 - Ethereum has https://consensys.github.io/smart-contract-best-practices/known_attacks
- Smart contracts may also have vulnerable dependencies



Listings of known vulnerabilities and insecure coding patterns can be invaluable for secure development and security auditing. For traditional software, known vulnerabilities in software are tracked as Common Vulnerabilities and Exposures (CVEs) and known common mistakes are Common Weaknesses Enumeration (CWEs). Some vulnerabilities in blockchain software may also be assigned a CVE.

Smart contract platforms often provide their own information on known vulnerabilities, insecure coding patterns, etc. Gaining familiarity with these is essential for writing secure code and identifying common errors in smart contract code.

When auditing a smart contract, it is vital to consider its dependencies and external integrations. Many smart contracts use forked or copied code from other smart contracts to implement common functionality. If the original code is insecure or the fork is modified from the source, it may contain vulnerabilities.

Testnets

- Live code analysis is necessary to detect some errors
- Smart contract code is designed to run on the blockchain
 - Nodes can run their own blockchain nodes to test code
- Most blockchains also offer testnets
 - Designed to accurately simulate real-world network
 - Provides a safe environment to test contracts before live deployment



Static code analysis can identify some vulnerabilities and implementation errors in a smart contract, but it doesn't catch everything. In some cases, the best way to identify if anything is wrong with a smart contract is to run it and interact with it live.

Smart contracts aren't traditional computer programs that can be run with a click from the desktop. They're designed to be hosted and executed on top of smart contract platforms. A computer can run an instance of blockchain node software to perform some testing, but this is not a fully accurate simulation of how the code would run on a real blockchain.

Many of the major smart contract platforms offer testnets for testing smart contract code. These are fully-functional blockchains designed to simulate the real environment as closely as possible. Testnets allow code to be tested live in a no-risk environment.

Vulnerable Applications

- Example vulnerable applications are essential for learning to identify vulnerabilities in smart contract code
- Many sample vulnerable contracts are available, such as:
 - ScrawlID: <https://github.com/sujeetc/ScrawlID>
 - Damn Vulnerable Ethereum Smart Contract: <https://github.com/mixbytes/DVESC>
 - Damn Vulnerable DeFi: <https://www.damnulnerabledefi.xyz/>



Reading about smart contract vulnerabilities and code patterns only gets a potential smart contract auditor so far. To be able to accurately identify vulnerabilities in smart contract code, an auditor needs practice doing so.

Several resources are available to provide this much-needed, real-world experience, including:

- **ScrawlID:** ScrawlID is a database that indicates potential vulnerabilities in real-world Ethereum smart contracts based on automated analysis by various tools.
- **Damn Vulnerable Ethereum Smart Contract (DVESC):** DVESC is a collection of Ethereum smart contracts that is deliberately designed to contain known Ethereum vulnerabilities
- **Damn Vulnerable DeFi:** Damn Vulnerable DeFi is a series of challenges designed to teach about common attack techniques used against DeFi smart contracts

Summary

- Introduction to Security Resources
- Resources and Style Guides
- Vulnerability Listings
- Testnets
- Vulnerable Applications



Smart Contract Security Audit Tools

331 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- Tools for Smart Contract Audits
- Code Coverage Testing
- Static Analysis
- Control Flow Analysis
- Dynamic Analysis
- Symbolic Execution
- Fuzzing
- Threat Intelligence



Tools for Smart Contract Audits

- Smart contract auditors can use various tools to identify vulnerabilities in both source code and live smart contracts
- Some of the tools used for smart contract auditing include:
 - Code Coverage Testing
 - Static Analysis
 - Control Flow Analysis
 - Dynamic Analysis
 - Symbolic Execution
 - Fuzzing
 - Threat Intelligence



Security audits are vital to ensuring smart contract security. Smart contract auditors can use a variety of different tools and techniques to identify vulnerabilities within smart contracts.

Different tools and techniques are capable of detecting different types of vulnerabilities. For this reason, using various tools can help to identify vulnerable code before it is deployed and potentially exploited. Some examples of tools that smart contract security auditors might use during an assessment include:

- Code Coverage Testing
- Static Analysis
- Control Flow Analysis
- Dynamic Analysis
- Symbolic Execution

- Fuzzing
- Threat Intelligence

Code Coverage Testing with solidity-coverage

- Code coverage tools measure how well a smart contract is covered by unit testing
- solidity-coverage is a great code coverage tool for Ethereum smart contracts written in Solidity

```
token_test_1 Contract: SushiToken
token_test_1 ✓ has a name
token_test_1 ✓ has a symbol (41ms)
token_test_1 ✓ has 18 decimals
token_test_1 ✓ assigns the initial total supply to the creator (69ms)
token_test_1 ✓ can transfer 10 tokens from token owner to not owner (68ms)
token_test_1 contract burnable
token_test_1 ✓ burn 10 tokens (50ms)
token_test_1 ✓ can not burn if burner doesn't have tokens (50ms)
token_test_1 contract mintable
token_test_1 ✓ mint 10 tokens by owner
token_test_1 ✓ can not mint by not owner
token_test_1 total sushi eaten
token_test_1 ✓ can add total sushi eaten value by sushi type
token_test_1
token_test_1 10 passing (553ms)
token_test_1
token_test_1 -----
token_test_1 | File | % Stats | % Branch | % Funcs | % Lines | Uncovered Lines |
token_test_1 |-----|-----|-----|-----|-----|-----|
token_test_1 | contracts/ | 42.86 | 100 | 50 | 42.86 | |
token_test_1 | SushiToken.sol | 42.86 | 100 | 50 | 42.86 | 28,29,33,34 |
token_test_1 |-----|-----|-----|-----|-----|
token_test_1 | All Files | 42.86 | 100 | 50 | 42.86 | |
token_test_1 |-----|-----|-----|-----|-----|
```

Source: <https://medium.com/hara-engineering/smart-contract-testing-with-coverage-9fd17476c0d4>



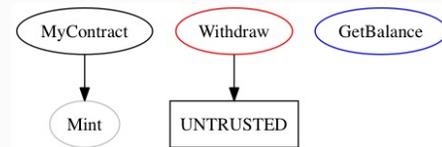
Unit testing is an essential component of secure design. As developers design and write code, they should also create unit tests that verify that the code works as anticipated. Unit tests can evaluate functionality, security, and other aspects of an application's performance.

However, unit tests are only useful for the portions of the code that they cover. Code coverage tools can identify parts of a smart contract for which unit tests have not been created as well as validating that a smart contract passed existing unit tests.

For Ethereum smart contracts written in Solidity, solidity-coverage (<https://github.com/sc-forks/solidity-coverage>) provides code coverage testing. It validates that all test cases are passed and identifies lines within a smart contract where test cases are missing.

Control Flow Analysis with Solgraph

- Control flow graphing can help to identify anomalous or undesired control flows within an application
- Solgraph analyzes the source code of a Solidity smart contract and creates a control flow graph



Source: <https://github.com/raineorshine/solgraph>

336 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

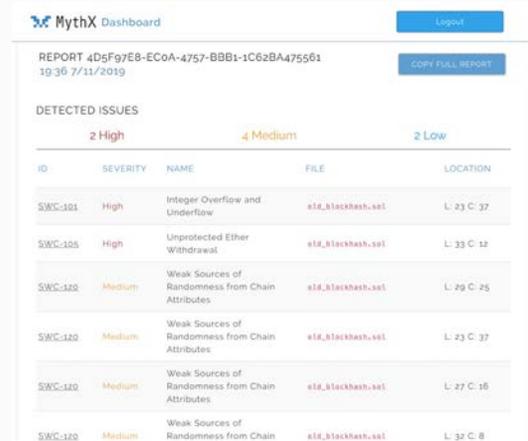
Errors in conditional statements can cause unexpected and anomalous control flows within an application. These issues could allow an attacker to gain unauthorized access to privileged functionality, perform a Denial of Service (DoS) attack, or otherwise exploit a vulnerable application.

Control flow analysis tools analyze source code and generate a visual representation of the control flow of the application. These visualizations can help understand the overall structure of an application, identify how a user can interact and move through it, and determine if the application contains any anomalous and incorrect conditional statements.

Solgraph is a control flow analysis tool for Solidity smart contracts. It graphs control flow between the functions in a smart contract and color codes them based upon the actions that they perform.

Dynamic Analysis with MythX

- Dynamic analysis tools analyze the security of a running application
 - Can detect configuration flaws and other runtime vulnerabilities
- MythX is an online dynamic analysis service for Ethereum smart contracts
 - Requires an API key



MythX Dashboard

REPORT 4D5F97E8-EC0A-4757-BBB1-1C62BA475561
19:36 7/11/2019

DETECTED ISSUES

2 High 4 Medium 2 Low

ID	SEVERITY	NAME	FILE	LOCATION
SWC-101	High	Integer Overflow and Underflow	src_blockhash.sol	L: 23 C: 37
SWC-105	High	Unprotected Ether Withdrawal	src_blockhash.sol	L: 33 C: 12
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	src_blockhash.sol	L: 29 C: 25
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	src_blockhash.sol	L: 23 C: 37
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	src_blockhash.sol	L: 27 C: 16
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	src_blockhash.sol	L: 32 C: 8

Source: <https://mythx.io/>



Dynamic analysis involves interacting with a running application to identify if it contains known vulnerabilities, configuration errors, and other issues. Some classes of vulnerabilities can only be detected at runtime; however, dynamic analysis can provide lower test coverage than static analysis. For this reason, dynamic analysis tools are a good complement for static code analysis.

MythX is a web-based analysis service for Ethereum smart contracts that offers static analysis, dynamic analysis, and symbolic execution. Like other tools presented here, it provides an in-depth listing of discovered vulnerabilities, including their severity, location in the code, and the associated Smart Contract Weakness Classification (SWC) record. MythX requires an API key and is a subscription-based service.

Symbolic Execution with Oyente

- Symbolic execution performs abstract execution of a program
 - Identifies ranges of inputs that cause certain code paths to be followed
- Oyente is an open-source symbolic execution tool for Ethereum smart contracts

```
root@6092bf150c9f:oyente/oyente# python oyente.py -s greeter.sol
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.21. The latest supported version is
0.4.19
INFO:root:contract greeter.sol:greeter:
INFO:symExec: ===== Results =====
INFO:symExec: EVM Code Coverage: 99.5%
INFO:symExec: Integer Underflow: False
INFO:symExec: Integer Overflow: False
INFO:symExec: Parity Multisig Bug 2: False
INFO:symExec: Callstack Depth Attack Vulnerability: False
INFO:symExec: Transaction-Ordering Dependence (TOD): False
INFO:symExec: Timestamp Dependence: False
INFO:symExec: Re-Entrancy Vulnerability: False
INFO:symExec: ===== Analysis Completed =====
```

Source: <https://medium.com/haloblock/how-to-use-oyente-a-smart-contract-security-analyzer-solidity-tutorial-86671be93c4b>



Symbolic execution is a technique designed to abstract away the details of a program's execution. Instead of sending explicit values through a program and observing the result, symbolic execution examines the program's code to determine which ranges of values would cause it to follow certain code paths. This enables comprehensive test coverage without the overhead of running each potential input through the code.

Oyente is one example of a symbolic execution tool for analysis of Solidity smart contracts (MythX also performs symbolic execution). It analyzes smart contract code to check for test coverage and the presence of various known vulnerabilities such as integer overflows/underflows, reentrancy, timestamp dependence, and more.

Fuzzing with Echidna

- Fuzzers send random and malformed input to an application and observe results
 - Can detect buffer overflows, integer overflows and other issues
- Echidna is a smart fuzzer for Ethereum smart contracts
 - Tailors inputs to code

```
Echidna 2.0.0
Tests found: 11
Seed: -7920091280766187841
Unique instructions: 1355
Unique codehashes: 1
Corpus size: 13

-----Tests-----
assertion in checkInvariant(): FAILED: with ErrorUnrecognizedOpcode
Call sequence:
1. lock()
2. voteLate("\161\231\t\0LE\225+5UB
\508->CS\2s\158\250\147p0L\223\170\215\*\1241\155\152IV\161")
3. etch(0x30000)
4. checkInvariant()

AssertionFailed(.): fuzzing (28325/50000)
assertion in deposits(address): fuzzing (28325/50000)
assertion in voteLate(bytes32): fuzzing (28325/50000)
assertion in lock(uint256): fuzzing (28325/50000)
assertion in free(uint256): fuzzing (28325/50000)
assertion in votes(address): fuzzing (28325/50000)
assertion in sLates(bytes32): fuzzing (28325/50000)
```

Source: <https://github.com/crytic/echidna>

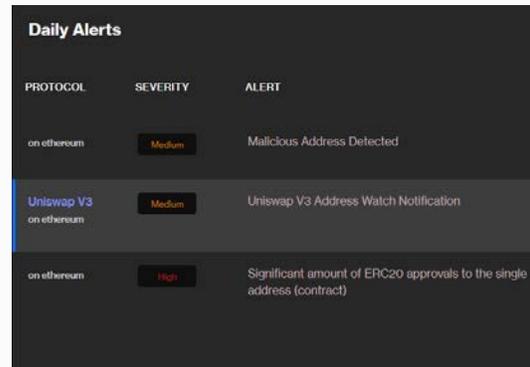


Fuzzers are designed to take a brute-force approach to security testing. They send a variety of random, known malicious, and malformed inputs to an application and observe its responses. This makes it possible to detect integer overflows/underflows, injection vulnerabilities, buffer overflows, and other issues.

Echidna is a fuzzer designed for the Ethereum blockchain. It uses Slither as a preprocessor to identify important features of the smart contract and then tailors its fuzzed inputs to the smart contract being analyzed.

Threat Intelligence with Forta

- Live monitoring and threat detection can be invaluable
 - Rapid response can mitigate effects of an attack
- Forta is a live threat detection engine
 - Enables creation of custom agents and alerts



PROTOCOL	SEVERITY	ALERT
on ethereum	Medium	Malicious Address Detected
Uniswap V3 on ethereum	Medium	Uniswap V3 Address Watch Notification
on ethereum	High	Significant amount of ERC20 approvals to the single address (contract)

Source: <https://explorer.forta.network/>



After a smart contract has been deployed on the blockchain, live monitoring can help with detection and remediation of attacks. In some cases, quickly identifying an ongoing attack could enable the affected protocol to take action to mitigate the damage.

Forta is a decentralized platform for monitoring deployed smart contracts for vulnerabilities and attempted exploitation. Forta allows anyone to develop agents to monitor for certain threats and provides alerts regarding potential malicious activity on the blockchain. Each alert indicates the affected protocol, severity, alert name, details, and timestamp of the incident.

Summary

- Tools for Smart Contract Audits
- Code Coverage Testing
- Static Analysis
- Control Flow Analysis
- Dynamic Analysis
- Symbolic Execution
- Fuzzing
- Threat Intelligence



Module 6

Developing Secure Blockchain Systems



Security Considerations for the Blockchain

- Blockchain technology offers an alternative to traditional, centralized systems
 - Provides numerous benefits for various use cases
- However, blockchain is not always the right solution
- Some key factors when considering a blockchain solution include:
 - Blockchain Architecture
 - Balancing Blockchain Benefits and Risks
 - Regulatory Considerations for Blockchain Systems



Blockchain is a revolutionary technology that provides a viable alternative to traditional, centralized processes. Instead of relying on and trusting in a centralized authority to maintain a digital ledger, blockchain creates and maintains the ledger using a decentralized network backed by cryptographic guarantees. Smart contracts expand this functionality to support programs running on top of this decentralized platform.

However, blockchain is not the ideal solution to every problem, and various blockchain solutions exist. In some cases, one type of blockchain may be a better fit than another, and, in others, blockchain technology may be more of a liability than an asset.

When determining whether a blockchain is the right choice for a particular problem, some key considerations include:

- Blockchain Architecture
- Balancing Blockchain Benefits and Risks

- Regulatory Considerations for Blockchain Systems

Module 6: Developing Secure Blockchain Systems

Section 6.1: Blockchain Architecture



Overview

- Architectural Considerations for Blockchain
- Public vs. Private Blockchains
- Open vs. Permissioned Blockchains
- Rollups
- Other Architectural Features



Architectural Considerations for Blockchain

- The Bitcoin whitepaper defined the original blockchain architecture
 - Open, permissionless network
- Since the creation of Bitcoin, alternative architectures have been proposed to meet different goals
 - Public vs. Private
 - Open vs. Permissioned
 - Rollups
 - Other Architectural Features



Bitcoin is the original blockchain, and the Bitcoin whitepaper defined the architecture used by the Bitcoin network. Since then, many blockchain platforms have been implemented based on the Bitcoin model, in some cases directly forking the Bitcoin code and making tweaks and modifications to fit the goals or ethos of the platform.

Some changes to the blockchain model have focused on Bitcoin's consensus algorithm, defining alternatives to Proof of Work. Others have focused on the architecture of the Bitcoin blockchain, making changes to the accessibility of the network and the distribution of power across blockchain accounts.

Today, blockchains can be classified in a couple of different ways. Blockchain platforms can be distinguished based on their accessibility and users' ability to create a blockchain account (public vs. private). They can also be classified based on how power is allocated within the network (open vs. permissioned). Blockchains might also implement other changes to the blockchain architecture, such as incorporating checkpointing to improve resistance to 51% attacks.

Public vs. Private Blockchains

- Bitcoin is an example of a public blockchain
 - Anyone can create an account and perform transactions on the blockchain
- Not all applications of blockchain are appropriate for a public blockchain
 - Some organizations have launched private blockchains for internal use
- The public/private choice comes with tradeoffs
 - Anonymity, centralization, data control, etc.



Bitcoin was designed to be a public and openly accessible platform. Anyone could create an account on the blockchain and perform transactions. This public design also provided anonymity to owners of Bitcoin accounts. Since anyone could create an account, there was no need to know the real-world identities behind a Bitcoin account. As a result, Bitcoin could use public key cryptography and account addresses on the blockchain. If you know the private key of an account, then you're assumed to have legitimate access to it.

However, the fully public model of Bitcoin is not well-suited to all potential applications of blockchain technology. Blockchain's immutable, decentralized digital ledger is useful for audit tracking or monitoring a company's supply chain. However, these applications include storing sensitive corporate information on the blockchain ledger. A fully public blockchain with a globally visible digital ledger is not ideal for this, so some organizations have deployed their own private blockchains that take advantage of the technology but move away from the public accessibility of Bitcoin.

Both public and private blockchains can use the same technology; however, there are tradeoffs between the two. Some example considerations include:

- **Anonymity:** Public blockchains can support fully anonymous user accounts because there is no vetting process needed to determine whether or not a user has a legitimate right to use the blockchain. Private blockchains, on the other hand, cannot be fully anonymous because users need to prove that they are valid users at least at the account creation stage.
- **Centralization:** Private blockchains are inherently more centralized than public ones. For access controls to exist, some means of enforcing them must exist, and the system that enforces these access controls is a potential single point of failure within the system. Also, by limiting the users that can be part of a blockchain, private blockchains limit the potential set of nodes that support it, making it more centralized and less resilient.
- **Data Control:** A major disadvantage of public blockchains is that all information is stored on a public digital ledger, which means that organizations have no control over their data that is placed on the blockchain or ability to delete that data. A private blockchain enables an organization to restrict access to the blockchain's ledger and control all nodes in the network, making deletion of data possible if needed.

Open vs. Permissioned Blockchains

- Bitcoin used an open permissions model
 - Any Bitcoin account could participate in any aspect of the network
- Other blockchains have implemented permissioned models
 - Certain nodes have special privileges
- The choice between open and permissioned blockchains comes with advantages and disadvantages
 - Centralization, control, etc.



In addition to being private or public, blockchains can be open or permissioned. This distinction depends on the privilege model used by the blockchain.

Bitcoin is an example of an open blockchain. All Bitcoin accounts are theoretically equal and have the ability to participate in all aspects of the blockchain protocol. While certain nodes may have more power over consensus, any node in the network can acquire some of the scarce resource and participate.

Other blockchains have built permissions into the blockchain architecture, only permitting certain accounts and nodes to perform certain actions. The most common example of this is masternodes, where certain nodes control the consensus process and no other nodes can participate.

Both open and permissioned blockchains have their pros and cons. Some examples include:

- **Centralization:** Permissioned blockchains are more centralized than open ones because only a subset of the blockchain's nodes and accounts have certain access and privileges. This creates increased risk of Denial of Service (DoS) attacks and

abuse of these privileges by a malicious or compromised account.

- **Control:** Permissioned blockchains provide greater control over the data and functionality of the blockchain. In open blockchains, anyone can access or do anything, which could lead to exposure of sensitive information or attacks by malicious users.

Rollups

- Rollups execute transactions outside of a blockchain and then record information about those transactions on the blockchain
 - **Optimistic Rollups (ORs):** Assumes transaction validity until a challenge triggers computation of a fraud proof
 - **ZK Rollups:** Off-chain computation includes a proof of validity recorded on the blockchain

- Rollups carry security risks as well



Rollups are a blockchain scalability solution explored by blockchains like Ethereum. Instead of performing all transaction computation and validation on the blockchain, they allow computation to be performed off-chain and certain data about the computation to be recorded on-chain.

Two main types of rollups exist: Optimistic Rollups (ORs) and Zero-Knowledge Rollups (ZK Rollups)

Optimistic Rollups

In an Optimistic Rollup (OR), all computation is performed off-chain and a node notarizes an updated state that results from executing the transaction in question. With ORs, the assumption is that a transaction is legitimate until it is challenged. A challenger would compute a fraud proof that they post to the blockchain. If a transaction is found to be fraudulent, then the fraudsters lose a bond that they had locked up with the promise that they would tell the truth.

The main security assumption of an OR is that, if a fraudulent transaction exists, then someone will create a fraud proof and that this proof will be posted to the

blockchain. The first part of this assumption, that transaction validation will occur, is the same as with the rest of the blockchain network. Building on an invalid block could result in the loss of block rewards.

The posting of a fraud proof is a bit more difficult since this is a type of transaction. Malicious block producers, Denial of Service attacks, or similar threats could delay or prevent the posting of a fraud proof within the challenge window for a transaction. If this occurs, then a fraudulent transaction may remain on the blockchain.

ZK Rollups

A ZK Rollup, like an OR, allows transactions to be executed off-chain and bundled up into a single state update. Unlike ORs, a ZK Rollup includes a zero-knowledge proof (ZKP) that a series of valid, digitally-signed transactions exists that would produce the state update. As a result, a ZK Rollup securely summarizes and validates the series of rolled-up transactions because it is only computable if those transactions actually existed.

The security of ZK Rollups depends on the security of the cryptographic algorithms used to generate the underlying ZKP. These algorithms are complex, and occasionally assume a trusted setup. If these algorithms are proven to be insecure or the participants in the trusted setup are malicious, then it may be possible to develop false proofs for malicious transactions.

Other Architectural Features

- The choice of private vs. public and open vs. permissioned can dramatically change how the blockchain operates
- Different blockchains can also make smaller architectural changes to achieve various goals
 - Checkpointing, block sizes, Segregated Witness, etc.
- These other changes also come with their pros and cons



Some architectural differences between various blockchain implementations are dramatic. Changes in the consensus algorithm or the differences between private and public, open and permissioned blockchains have a significant impact on who can use the blockchain, what they can do, and how the system works.

However, different blockchains might incorporate other changes to the architecture of the blockchain as well. Some examples include:

- **Checkpointing:** The longest chain rule means that no version of the blockchain is final and that a block or transaction can always be replaced. Checkpointing locks in a certain version of a given block so that a node will not accept an alternative version of the blockchain that lacks this block.
- **Block Sizes:** Blockchains commonly have a maximum block size that limits the number of transactions that can be contained within a block. Different blockchain implementations use different block sizes to optimize throughput vs. other considerations.
- **Segregated Witness:** Segregated witness (SegWit) separated the witness (digital

signature) information from the input field of the block. This helped to protect against transaction malleability and allowed more transactions to fit into a single fixed-sized block.

These and other architectural changes to blockchain protocols have their benefits, but they can have downsides as well. For example, checkpoint protects against 51% attacks at the cost of centralization or the potential for permanently divergent blockchains. This is because the checkpoint must be defined by someone. If a single authority does so, it centralizes control over the blockchain. If each node does so independently, then the potential exists for permanent forks if different nodes define different versions of the same checkpoint and reject each others' version of the blockchain.

Summary

- Architectural Considerations for Blockchain
- Public vs. Private Blockchains
- Open vs. Permissioned Blockchains
- Rollups
- Other Architectural Features



Module 6: Developing Secure Blockchain Systems

Section 6.2: Balancing Blockchain Benefits and Risks

352 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- Blockchain Features and Risks
- Decentralized Architecture
- Distributed Infrastructure
- Immutable Blockchain
- Signed Transactions
- Transparent Digital Ledger
- Smart Contract Support



Blockchain Features and Risks

- Blockchain technology has several built-in features that provide significant benefits:
 - Decentralized Architecture
 - Distributed Infrastructure
 - Immutable Blockchain
 - Signed Transactions
 - Transparent Digital Ledger
 - Smart Contract Support
- While these features are useful in some contexts, they also carry risks



Blockchain technology provides an alternative solution to solving common business problems. The underlying technology of the blockchain provides several valuable benefits, including:

- Decentralized Architecture
- Distributed Infrastructure
- Immutable Blockchain
- Signed Transactions
- Transparent Digital Ledger
- Smart Contract Support

All of these features can be valuable tools in the right scenario; however, they are not always positives. Along with the benefits, these features can also carry significant security risks.

Decentralized Architecture

- Blockchain technology decentralizes power across many nodes
- Benefits:
 - Resiliency
 - Separation of power
- Downsides:
 - System cannot be taken offline
 - Expanded attack surface and supply chain



Decentralization is one of the major selling points of blockchain technology. A primary goal of the blockchain is to eliminate reliance on a centralized authority to maintain the blockchain's digital ledger. Instead, the blockchain relies on a collection of decentralized nodes that are encouraged to follow the rules via financial incentives.

Blockchain's decentralization is a major selling point of the technology because it brings significant benefits. A couple of examples include:

- **Resiliency:** Blockchain decentralization eliminates single points of failure within the system because no node is crucial to the operation of the blockchain. This increases the resiliency of the system because Denial of Service attacks or other events that bring one or more nodes down do not bring down the entire network.
- **Separation of Power:** Entrusting a centralized authority with control over the digital ledger creates the potential for abuse of power. Theoretically, the blockchain's decentralization eliminates the risk that a powerful node could control the blockchain system and ledger.

Blockchain decentralization has significant benefits, but it has downsides as well. Some of these include:

- **Inability to Shutdown the Blockchain:** In the event of a hack or leakage of sensitive information onto the blockchain, it might be desirable to take down the system to remediate the issue. With a decentralized system, an organization lacks the ability to do so.
- **Expanded Attack Surface:** Centralized systems can protect the database server(s) with strong defenses and access controls, making them more difficult to attack. Blockchain data and services are hosted on many different nodes, making it more difficult to implement comprehensive cyber defenses.

Distributed Infrastructure

- The blockchain is designed as a distributed system with each node in the network keeping an independent copy of the ledger
- Benefits:
 - Resiliency
 - Transparency
- Downsides:
 - Data control
 - Inefficiency



Blockchain networks are designed to be distributed, especially for public blockchain networks. Each node in the blockchain network keeps its own copy of the digital ledger and is responsible for validating and adding new blocks to it.

Blockchain's distributed infrastructure has its benefits, including:

- **Resiliency:** Blockchain systems commonly have nodes spread across different organizations, countries, and even continents. This makes it much less likely that a cyberattack, natural disaster, or other event would take all of the nodes offline, increasing the resiliency and availability of the blockchain's ledger.
- **Transparency:** Every node in the blockchain network maintains its own copy of the digital ledger, and new transactions and blocks are broadcast across the entire peer-to-peer network. This makes the blockchain's ledger extremely transparent, reducing opportunities for fraud or deletion of private data.

While blockchain's distribution has its advantages, it also has disadvantages, including:

- **Data Control:** Every node in the blockchain network has a copy of every transaction included on the blockchain's digital ledger. This means that the creator of a transaction has no control over that data once it has been broadcast to the blockchain network and added to the digital ledger.
- **Inefficiency:** Blockchain's distribution and decentralization mean that every node in the blockchain network is performing identical functions as transactions are validated, processed, and added to the digital ledger. This makes blockchains hugely inefficient in the name of achieving the benefits of decentralization and distribution.

Immutable Blockchain

- The blockchain maintains an immutable digital ledger, making it difficult for transactions to be modified or deleted after they are recorded on the ledger
- Benefits:
 - Immutable record
 - Decentralization
- Downsides:
 - Permanent data storage
 - Irreversible transactions



The immutability of the blockchain's digital ledger is a major benefit of the blockchain and a key feature that makes the technology possible. Using digital signatures and hash functions, the blockchain's design makes it infeasible for an attacker to significantly rewrite the blockchain's ledger.

This immutability provides certain benefits, including:

- **Immutable Record:** The immutability of the blockchain's record is a major benefit in itself. An immutable ledger is invaluable for creating audit logs and other records where the knowledge that the record cannot be forged is a major benefit.
- **Decentralization:** The immutability of the blockchain's ledger is essential to the decentralization of the blockchain. If the nodes hosting the ledger had the ability to modify the data that they stored, then it would be impossible to create a consistent, shared, and trustworthy record of the blockchain's history.

Blockchain immutability can provide significant benefits. However, it does have its downsides, such as.

- **Permanent Data Storage:** The fact that data cannot be deleted from the ledger is not ideal in the case that sensitive data is included in transactions. If this is the case, then blockchain immutability can hurt data privacy, security, and regulatory compliance.
- **Irreversible Transactions:** All blockchain transactions are recorded on the blockchain's digital ledger. This means that incorrect transactions or one that are part of a hack cannot be reversed after they are recorded on the immutable ledger.

Signed Transactions

- All transactions on the blockchain are digitally signed, proving their integrity and authenticity
- Benefits:
 - Data authentication
 - Data integrity
 - Anonymity
 - Blockchain decentralization
- Downsides:
 - Lack of privacy
 - False sense of security



All transactions stored on the blockchain's digital ledger are digitally signed. This guarantees that the transaction was generated by someone with the account's private key and that the transaction has not been modified since it was created.

Blockchain's use of digital signatures provides certain benefits, including:

- **Data Authentication:** If a transaction carries a valid digital signature, then it was generated by someone with knowledge of the account's private key. This is useful for proving the authenticity of data and for generating audit logs of a user's activities on the blockchain.
- **Data Integrity:** Once a digital signature has been created, it is infeasible to modify the signed data without invalidating the digital signature. This ensures that transaction data cannot be tampered with after it is signed by the account owner.
- **Anonymity:** Digital signatures validate that someone with knowledge of an account's private key generated a particular transaction. This helps to support anonymity because it provides a means of authenticating data that does not require knowledge of a user's real identity.

- **Decentralization:** Digital signatures prove the integrity and authenticity of transaction data. This is vital to decentralization because, without them, it would be infeasible to implement a system where data transmitted over a peer-to-peer network and stored on decentralized nodes could be trusted.

Digitally signed transactions have disadvantages as well, these include:

- **Lack of Privacy:** On the blockchain, all actions are publicly associated with a blockchain account. While this account may be intended to be anonymous, analysis of “patterns of life” across an account’s transactions can reveal the owner’s identity, allowing anyone to know the value of their blockchain account and what they do with it.
- **False Sense of Security:** Digital signatures can create a belief that a transaction with a valid digital signature was created by the account owner. However, anyone who learns an account’s private key can create a valid signature, and the data included in a signature may be modified before the signature is generated without the user’s awareness or consent.

Transparent Digital Ledger

- The blockchain's digital ledger is stored on every node in the blockchain network and transactions are visible to all users
- Benefits:
 - Decentralized validation
 - Transparent operations
- Downsides:
 - Access controls
 - Lack of privacy



The blockchain's digital ledger is designed to be completely transparent. Every transaction is broadcast to the entire blockchain network and stored on each node, making it possible for anyone to view any transaction in the blockchain, including source, destination, value, and associated data.

The transparency of the blockchain's digital ledger provides some benefits including:

- **Decentralized Validation:** Blockchain nodes are responsible for ensuring that all transactions that they put into the blocks that they produce and that they include in their copy of the ledger are valid. The transparency of the blockchain ledger helps them perform this critical validation. However, full transparency is not necessary for validation if a blockchain uses privacy-preserving solutions such as ring signatures, stealth addressing, and zero-knowledge proofs.
- **Transparent Operations:** Centralized systems commonly have opaque internal processes, making it difficult to trust the validity of their internal ledgers. In blockchain, everything is out in the open, engendering trust by allowing anyone to audit operations and transactions.

Blockchain's transparent ledger also has its disadvantages, such as:

- **Access Controls:** On the blockchain, all nodes and users have access to all data that is stored on the blockchain. This makes it infeasible to implement the access controls required for some forms of data under data protection regulations or corporate policy.
- **Lack of Privacy:** Blockchain transparency makes it possible to determine the exact value of any blockchain account and its complete transaction history. While these accounts are intended to be anonymous, deanonymized accounts provide in-depth information about their owners.

Smart Contract Support

- Smart contract platforms allow programs to run on top of the blockchain in a decentralized “world computer”
- Benefits:
 - Decentralized processing
 - Support for DeFi and other projects
- Downsides:
 - Irreversible code execution
 - Smart contract vulnerabilities



Smart contract platforms dramatically expanded the capabilities of blockchain systems. The original blockchains like Bitcoin were largely designed as a decentralized financial system that could track exchanges of value with no central intermediary. Smart contract platforms allow Turing-complete programs to run on the blockchain, which provides certain benefits:

- **Decentralized Processing:** A smart contract platform runs code within virtual machines that are hosted on each node in the blockchain network. This brings the resiliency, separation of power, and other benefits that blockchain brings to data storage to data processing as well.
- **DeFi and Other Projects:** Smart contracts expanded blockchain technology from tracking value transfers to being able to support a fully-fledged financial system including loans and other functionality. It also provides support for various other blockchain-based initiatives.

Smart contracts are powerful tools that change how many common business processes are performed. However, they also have their downsides:

- **Irreversible Code Execution:** Smart contracts are hosted on the blockchain's immutable ledger and executed via transactions that are also recorded on the ledger. Smart contract code, once executed, cannot be reversed, which is why DeFi and other hacks cannot be undone.
- **Smart Contract Vulnerabilities:** Smart contract platforms are relatively immature, and many projects have not adopted common software development best practices. As a result, insecure smart contracts are being deployed to the blockchain where they cannot be easily updated and are vulnerable to exploitation.

Summary

- Blockchain Features and Risks
- Decentralized Architecture
- Distributed Infrastructure
- Immutable Blockchain
- Signed Transactions
- Transparent Digital Ledger
- Smart Contract Support



Module 6: Developing Secure Blockchain Systems

Section 6.3: Regulatory Considerations for Blockchain Systems



Introduction to Compliance

- Most organizations are subject to various regulations
 - Data protection laws
 - Industry regulations
 - Anti-fraud legislation
- These regulations mandate that an organization implement certain security controls and policies
- Blockchain technology can help with regulatory compliance
 - It can also be a liability



For many organizations, the regulatory compliance landscape is growing rapidly. Some data protection regulations and anti-fraud legislation has existed for years, such as the Payment Card Industry Data Security Standard (PCI DSS) and national anti-money laundering (AML) laws. The European Unions General Data Protection Regulation (GDPR) kicked off a rapid expansion of the data privacy law landscape, inspiring many jurisdictions to pass their own laws.

All of these various laws impose requirements on businesses. Some common ones include:

- **Access Control:** Companies should limit access to protected data to authorized parties
- **Data Encryption:** Data should be protected against exposure by encryption and other security controls
- **Records and Reporting:** Companies should keep records of certain events and processes to report to auditors and regulators

As organizations' compliance responsibilities expand, maintaining compliance can become more complex. In some cases, the unique nature of blockchain technology is well suited to meeting an organization's regulatory compliance obligations. In others, its design can be a significant liability for corporate regulatory compliance.

Compliance Benefits of Blockchain

364 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- Blockchain Provides Compliance Benefits
- Immutable Access Logging
- Transparent Corporate Operations
- Data Retention Policy Support



Blockchain Provides Compliance Benefits

- Blockchain technology implements an immutable digital ledger supported by a decentralized network
- The design of the blockchain makes it useful for certain regulatory compliance applications, such as:
 - Immutable Access Logging
 - Transparent Corporate Operations
 - Data Retention Policy Support



Blockchain technology offers an alternative method of accomplishing common business goals. Instead of maintaining a centralized digital ledger that can potentially be manipulated by the authority controlling it, blockchain technology embraces decentralization. The digital ledger is maintained by a network of independent nodes, and the integrity and immutability of the ledger are protected by cryptographic algorithms.

The design of the blockchain and its digital ledger mean that it is well suited to meeting some of an organization's regulatory compliance needs. Some compliance-focused applications of blockchain technology include:

- Immutable Access Logging
- Transparent Corporate Operations
- Data Retention Policy Support

Immutable Access Logging

- The blockchain's ledger creates an immutable log of user actions
 - All transactions are digitally signed
 - Transactions are recorded on the immutable digital ledger
- This can be valuable for meeting regulatory requirements
 - Immutable audit log for access to sensitive data, etc.
 - Demonstrate that no unauthorized users have accessed sensitive data



Access management and logging is a core requirement of many data protection laws. A major goal of the regulation is to ensure that unauthorized users do not have access to sensitive and protected data. To accomplish this, regulators want organizations to prove that only certain individuals and devices can and did access that protected data.

The design of the blockchain makes it ideal for access logging, creating an immutable digital record of all actions performed within the blockchain system. Blockchain transactions are digitally signed, meaning that every action is authenticated (based on knowledge of a blockchain account's private key) and associated with a particular blockchain account. Before being processed, these transactions are recorded on the blockchain's immutable ledger, making it infeasible for an organization or an attacker to modify and remove records without detection.

The blockchain's immutable digital ledger can be a useful tool for regulatory compliance. For example, systems can be designed so that certain operations, such as access to sensitive data, require requests made via blockchain transactions. These requests will be logged on the blockchain ledger, making it possible for an organization to prove to regulators that only certain authorized parties had access to

sensitive information.

Transparent Corporate Operations

- Regulations like Sarbanes-Oxley in the US are focused on corporate transparency
 - Transparency helps to protect against fraudulent activities
- Blockchain implements a transparent, immutable digital ledger
 - Regulators can easily view records
 - Smart contracts can encode needed functionality
 - Ledger immutability protects prevents concealment of fraud



The Sarbanes-Oxley (SOX) Act was passed in the United States in response to the Enron scandal, where deceptive financial reporting concealed fraudulent operations and the impending failure of the business. The SOX Act is designed to protect shareholders in publicly traded companies by requiring companies to make financial reports and testify to their accuracy.

Corporate transparency helps to protect against fraud and build trust in customers and regulators. Blockchain can help to support corporate transparency efforts due to its transparent and immutable digital ledger. Blockchain transactions are accessible to all nodes and accounts in the network, and data cannot be removed once added to the digital ledger. Additionally, critical operations can be encoded in smart contracts, which are also triggered by transactions that are recorded on the blockchain's ledger. This can engender trust by making it impossible for a company to lie in its records.

Data Retention Policy Support

- Many regulations mandate that companies retain certain records for a period of time
 - Financial records, etc.
- Blockchain's immutable ledger helps to ensure that data is not accidentally deleted
 - Ledger is stored on several distributed nodes
 - Ledger immutability makes it impossible to delete data without detection



Data retention policies are a common requirement in regulations. Regulators want records of an organization's financial activities, security controls, and other events to be retained for a few years.

Blockchain implements an immutable digital ledger that is hosted by a distributed and decentralized network of nodes. This makes it extremely difficult for data to be lost because nodes cannot delete it intentionally, and the resiliency and redundancy of the network makes it unlikely that all copies of the blockchain ledger will be affected or deleted by a cyberattack or other event.

The immutability and resiliency of the blockchain's digital ledger makes it a good choice for meeting regulatory data retention requirements. Placing records that must be retained on the blockchain ledger leverages its built-in protections to meet regulatory requirements.

Summary

- Blockchain Provides Compliance Benefits
- Immutable Access Logging
- Transparent Corporate Operations
- Data Retention Policy Support



Blockchain Compliance Challenges

371 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Overview

- Blockchain Creates Compliance Challenges
- Sensitive Data Exposure
- Data Control and Jurisdictions
- Data Encryption Requirements
- Deleting Unneeded Data



Blockchain Creates Compliance Challenges

- Blockchain technology has some properties that are useful for regulatory compliance
 - Immutability, transparency, distribution, etc.
- However, these and other properties can also be liabilities
- Some disadvantages of blockchain for compliance include:
 - Sensitive Data Exposure
 - Data Control and Jurisdictions
 - Data Encryption Requirements
 - Deleting Unneeded Data



Blockchain technology has several advantages for regulatory compliance. Its immutable and transparent digital ledger and the distribution and decentralization of the blockchain nodes have their advantages for regulatory compliance.

However, blockchain is not perfectly aligned to an organization's compliance needs. In some cases, these same features that provide benefits for organizations seeking to implement regulatory compliance policies also create significant challenges.

Some examples of compliance challenges that organizations may need to address if using blockchain technology for business operations include:

- Sensitive Data Exposure
- Data Control and Jurisdictions
- Data Encryption Requirements
- Deleting Unnecessary Data

Sensitive Data Exposure

- Blockchain stores all transaction data on a publicly-accessible, immutable digital ledger
 - Every blockchain node has a copy of the ledger
 - Any blockchain user can view the contents of any transaction
- Regulations commonly mandate that organizations restrict access to protected data
 - Companies have no control over data on the blockchain
 - Ledger immutability prevents deletion of sensitive data



The blockchain's digital ledger is designed to be completely transparent. Due to the decentralization of the blockchain, every node in the blockchain network has a copy of the blockchain's ledger. They also have a certain level of visibility into the contents of the transactions on the ledger to allow them to validate each transaction before accepting it. This distributed and decentralized storage means that all blockchain accounts have visibility into transactions on the blockchain ledger.

This visibility and transparency can be an issue for corporate regulatory compliance. A common requirement in data protection regulations is that an organization protect access to sensitive data by limiting to authorized users. By design, no one has control over the blockchain and the data that it contains, meaning that an organization surrenders control over data when it is placed on a public blockchain. Additionally, due to blockchain distribution and immutability, it is impossible for an organization to delete any sensitive data accidentally exposed within a transaction.

Data Control and Jurisdiction

- Data control is a core requirement of many data protection regulations
 - Companies must control access to sensitive data
 - Some regulations restrict data to certain jurisdictions
- Blockchains allow all nodes access to transaction data
 - No access control
 - Globally distributed nodes



Data protection regulators want companies to demonstrate that they retain control over the sensitive customer data entrusted to them. A few important aspects of this requirement include:

- **Access Control:** Data privacy laws mandate that access to sensitive data be restricted to authorized users. This access should be based on job roles and the principle of least privilege.
- **Geographic Control:** Some regulations, like the GDPR, restrict where the data of protected entities can be stored, processed, and transmitted. For example, GDPR only allows EU citizens' data to be shared within the EU, with countries with "adequate" data privacy laws, and with countries where a data sharing framework has been created. In the US, which doesn't have a federal data privacy law, setting up Privacy Shield to allow EU-US data sharing has been a long and complex process.

Using blockchain technology can make it difficult to comply with these requirements. Once data is included in a blockchain transaction, it is publicly accessible and an organization has no control over how it is accessed and used. Also, many blockchains

are globally distributed, which means that they break the jurisdictional boundaries imposed by GDPR and similar regulations.

Data Encryption Requirements

- Data privacy laws commonly mandate that protected data be encrypted
 - May specify encryption algorithms
 - Expect data to remain protected
- Blockchain's design makes data encryption complex
 - Transaction validation
 - Blockchain immutability



Data protection is a primary goal of data privacy laws, and encryption is one of the most effective ways to protect data against unauthorized access. Data protection regulations commonly mandate that the sensitive data that they protect be encrypted and may even specify which encryption algorithms that are acceptable. The intent is for the sensitive data to be protected indefinitely against unauthorized exposure.

The design of the blockchain can make protecting data with encryption complicated. Some challenges include:

- **Transaction Validation:** Blockchain nodes are expected to validate that transactions are valid before incorporating them into their copy of the blockchain digital ledger. If any of the data needed for validation is protected under regulations, encrypting it would break blockchain's decentralization.
- **Blockchain Immutability:** Blockchain's digital ledger is immutable, meaning that information cannot be removed from it. In the future, modern encryption algorithms may be broken, or an organization's keys may be compromised. If this is the case, the organization cannot prevent the exposure of encrypted data.

Deleting Unneeded Data

- Blockchain technology is ideal for corporate data retention requirements
 - Immutable ledger ensures that transaction data is kept
- However, it is less ideal for meeting data deletion requirements
 - Many regulations mandate that protected data be deleted when it is no longer needed
 - Ledger immutability makes data deletion infeasible



Blockchain technology is ideal for data retention. The blockchain's digital ledger is immutable, meaning that data cannot be deleted either intentionally or by accident. This immutability, combined with the resiliency provided by blockchain's distributed and decentralized architecture, makes it ideal for data retention.

However, blockchain immutability becomes a problem when the time comes for this data to be deleted. Many regulations mandate that certain records be destroyed after a set period or, in the case of laws like GDPR, that collected data be deleted once it is no longer required for the original purpose for which it was collected. With blockchain immutability and the distributed and decentralized nature of blockchain nodes, complying with these requirements is infeasible for any sensitive data placed included in transactions on a blockchain not fully under the organization's control.

Summary

- Blockchain Creates Compliance Challenges
- Sensitive Data Exposure
- Data Control and Jurisdictions
- Data Encryption Requirements
- Deleting Unneeded Data





Thank you!

379 | Blockchain and Crypto Security Training | © Marin Ivezić 2022



This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).