HACKEN

INDXCOIN SECURITY ANALYSIS

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Intro

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another party. Any subsequent publication of this report shall be without mandatory consent.

Name	Indxcoin
Website	https://indxcoin.com/
Repository	https://github.com/indxcoin/indxcoin
Commit	7d52377bec95c1e0b6f23038da0785b0b33b102a
Platform	L1
Network	Indxcoin
Languages	C++
Methods	Automated Code analysis, Manual review, Issues simulation
Auditor	b.barwikowski@hacken.io
Approver	l.ciattaglia@hacken.io
Timeline	28.10.2022 - 22.12.2022
Changelog	24.03.2023 (Final Report)

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Table of contents

- **Summary**
  - Documentation quality
  - Code Quality
  - Architecture quality
  - Security score
  - Total score
  - Findings count and definitions
- **Scope of the audit**
  - Protocol Audit
  - Implementation
  - Protocol Tests
- **Issues**
  - Coinbase inflation
  - Duplicated transactions
  - Invalid transaction time
  - Transaction malleability
  - Block DoS
  - Block header DoS
  - Incorrect stake modifier cache
  - Coinstake DoS
  - Duplicated coinstake proof
  - Invalid block synchronization
  - Invalid block validation
  - CBlockIndex exception
  - Usage of floating point arithmetic
  - Slow GetLastBlockIndex
- **Long range and grinding attack against consensus**
  - Details
  - Real world example
  - Recommendations
- **Test coverage**
- **Disclaimers**
  - Hacken disclaimer
  - Technical disclaimer

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Summary

The Indxcoin network architecture is a fork from Bitcoin Core 22.0 and incorporates elements from Reddcoin and Peercoin. These technologies were selected for their track record of securely handling a high volume of transactions. The programming language used in the development of Indxcoin is C++.

Indxcoin itself is a cryptocurrency that utilizes a modified version of the Proof-of-Stake-Velocity (PoSV) consensus algorithm, which is based on Reddcoin Core 3.10.4, a variant of Litecoin. According to the INDXcoin website, the project is described as a "benefit-laden utility coin whose market value is based on the index rate of the world's 100 top-performing cryptocurrencies by market capitalization.".

Indxcoin White Paper 1.2 describes its Technology Stack as follows:

> "INDXcoin's network architecture is based on the Bitcoin 22 wallet, and the coin itself is based on Reddcoin, and its derivative, Peercoin.
>
> These technologies were chosen due to their time-tested success in handling billions of transactions safely and securely.
>
> INDXcoin takes the best aspects of all three of these legacy technologies and implements them into single, simplified versions themselves.
>
> Mainnet is the active network for INDXcoin, the peer-to-peer network which hosts the INDXcoin benchmark rate.
>
> INDXcoin as a cryptocurrency is hosted on the Mainnet and is the marker of held value used to facilitate trades on the network."
> In comparison to Bitcoin Core 22.0, there are approximately 10,000 lines of code that have been modified in the Indxcoin source code.

Documentation quality

It should be noted that there is currently no technical documentation produced by the Indxcoin development team that specifically outlines their modifications to the codebase. The only available documentation is the original documentation for Bitcoin Core, which may no longer be applicable due to the changes made by the Indxcoin team.

While the documentation for Bitcoin Core is highly regarded, it should be recognized that the quality of documentation specifically produced by the Indxcoin team is currently lacking. Based on this, the total documentation quality score is **2** out of 10.

Code Quality

Bitcoin Core is a highly respected software with exceptional code quality and extensive test coverage. However, the changes introduced by Indxcoin developers did not align with the standards upheld by Bitcoin Core. The new code lacks adequate testing, with many existing tests no longer functioning, including QA tests. Furthermore, the fuzz targets are no longer able to compile due to compiler errors. In addition, several functions were copied incorrectly from Reddcoin Core, leading to numerous critical issues. Some of the not working tests were later fixed, but many of them were just removed, significantly decreasing high code quality standards inherited from Bitcoin Core.

Based on this, the total code quality score is **4** out of 10.

Architecture quality

Changes in the consensus algorithm introduced many new threats which did not exist in Bitcoin Core or were hard to execute, especially denial of service attacks and long-range & grinding attacks. In the audit, we described an attack scenario that allows the attacker to take control of the network with less than 0.01% of existing coins and very low computing power. Since Bitcoin Core improvements like for e.g. SegWit are disabled, others like Simplified Payment Verification are no longer secure to use. After the audit, consensus mechanism was improved based on our suggestions, but it is still vulnerable to attacks using less than 10% of all existing coins.

Based on this, the architecture quality score is **6** out of 10.

Security score

During the audit, 8 critical and 3 medium severity issues and relatively easy attacks against consensus were found. All of the previously identified critical issues were fixed in the version from commit fa7bcd0ec2a5431a6b356b1e9655b906ee579828. However, due to the high number of critical issues and a lot of code changes introduced after the audit it is highly recommend an additional audit in the nearly future, because the chance of finding additional issues in the code is high.

All issues detected during the audit were correctly fixed by Indxcoin team and verified.
Still, due to the increased likelihood of new undetected vulnerabilities introduced during the audit's fix period, the score was reduced. The overall security score is **6** out of 10.

Total score

Considering all metrics, the total score of the report is **5.4** out of 10.

Findings count and definitions

Severity	Findings	Severity Definition
Critical	8	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	0	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	3	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	2	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality.
Total	13	

Scope of the audit

Protocol Audit

Code Audit

- Review of all code changes since fork of bitcoin 22.0
- Detailed review on P2P
- Detailed review of consensus changes
- Detailed review of chainparameters changes
- Detailed review of wallet changes
- Review of proof of stake algorithm

Implementation

Code Quality

- Static analysis
- Dynamic analysis
- Test coverage
- Quality assurence tests review

Protocol Tests

Node Tests

- Setup of test network (10 nodes)
- Extensive tests of test network
- Fuzz testing of new data structs (PoS blocks)

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Issues

Coinbase inflation

When proof-of-stake mining in enabled, value of reward in coinbase is not validated which allows to generate any number of new coins.

ID	INDX-101
Scope	Block validation
Severity	CRITICAL
Vulnerability Type	Improper validation
Reproduction Test	issues_reproduction/coinbase_inflation.py
Status	Fixed in 533427

Details

In `validation.cpp`, in function `CChainState::ConnectBlock` when a new block is validated in proof-of-stake mode, blockReward from coinbase (1st transaction in block) is not validated. It allows to create a coinbase transaction with any reward which can be later used in other transactions. In practice, it allows to generate an infinite number of new coins.

```
//! subsidy checks
if (block.IsProofOfWork()) {
    CAmount blockReward = nFees + GetBlockSubsidy(pindex->nHeight, m_params.GetConsensus());
    if (block.vtx[0]->GetValueOut() > blockReward) {
        LogPrintf("ERROR: ConnectBlock(): coinbase pays too much (actual=%d vs limit=%d)\n", block.vtx[0]->GetValueOut(
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-cb-amount");
    }
}
else if (block.IsProofOfStake())
{
    // PoSV: coinstake tx earns reward instead of paying fee
    uint64_t nCoinAge = GetCoinAge(this, *block.vtx[1], m_params.GetConsensus());
    if (!nCoinAge)
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-posv-coinage");
    CAmount nCalculatedStakeReward = 0;
    nCalculatedStakeReward = GetProofOfStakeReward(nCoinAge, nFees);
    if (nStakeReward > nCalculatedStakeReward) {
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-posv-amount");
    }
}
```

The problem is caused by the lack of `if (block.vtx[0]->GetValueOut() > blockReward) {` check in `else if (block.IsProofOfStake())` code block.

Recommendation

When a block is proof-of-stake, coinbase transaction should not be able to generate any new coins (`block.vtx[0]->GetValueOut()` should be 0 or `block.vtx[0]->vout.size()` should be 1).

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Duplicated transactions

It is possible to include the same transaction in more than one block.

ID	INDX-102
Scope	Block validation
Severity	CRITICAL
Vulnerability Type	Improper validation
Reproduction Test	issues_reproduction/duplicate_transaction.py
Status	Fixed in 533427

Details

In `validation.cpp`, in function `CChainState::ConnectBlock` when a new block is validated in proof-of-stake mode, BIP30 is not correctly enforced for coinbase, which allows duplicate transactions.

```
if (fEnforceBIP30 || pindex->nHeight >= BIP34_IMPLIES_BIP30_LIMIT) {
    bool fProofOfStake = pindex->nHeight > Params().GetConsensus().nLastPowHeight;
    for (const auto& tx : block.vtx) {
        for (size_t o = 0; o < tx->vout.size(); o++) {
            if(fProofOfStake && tx->IsCoinBase()) {
                o++;
                continue;
            }
            if (view.HaveCoin(COutPoint(tx->GetHash(), o))) {
                LogPrintf("ERROR: ConnectBlock(): tried to overwrite transaction\n");
                return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-txns-BIP30");
            }
        }
    }
}
```

For some reason, when a transaction is a coinbase (1st transaction in the block), it is not being checked if it is a duplicate of an already existing transaction.

Recommendation

If there are no blocks in the network with duplicate transactions, then the following part of the code should be removed:

```
if(fProofOfStake && tx->IsCoinBase()) {
    o++;
    continue;
}
```

If there are blocks with duplicate transactions, then a more complicated solution must be enforced. Otherwise, it will break the consensus.

Invalid transaction time

It is possible to create a transaction that will not be accepted when it is in a block, but it will be accepted by the mempool, making the generation of new blocks impossible.

ID	INDX-103

Scope	Mempool
Severity	CRITICAL
Vulnerability Type	Denial of service
Reproduction Test	issues_reproduction/invalid_transaction_time.py
Status	Fixed in 4a5268

Details

A transaction with `nVersion > 1` has a new field, `nTime`, that contains the timestamp of when a transaction was created. It is being used by proof-of-stake mining. In `validation.cpp`, in `CheckBlock`, when a new block is being validated, the following check is used:

```
for (const auto& tx : block.vtx) {
    ...
    if (block.IsProofOfStake() && block.GetBlockTime() < (int64_t)tx->nTime)
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-tx-time", "block timestamp earlier than transa
}
```

which disallow a block to have transactions with `nTime` higher than the block creation time.

However, in the mempool, there are no checks validating transaction `nTime` so it is possible to create a transaction with `nTime` far into the future and such transaction will be accepted. By having such transaction in the mempool, the build-in miner won't be able to correctly create a new block, because it will always include this transaction in new block, making it invalid.

Recommendation

When a transaction is validated by the mempool, check that `nTime` is not greater than the current time.

Transaction malleability

Transaction `nTime` field is not cryptographically secured, it can be correctly modified by anyone.

ID	INDX-104
Scope	Transactions
Severity	CRITICAL
Vulnerability Type	Improper cryptographic signing implementation
Status	Fixed in 4a5268

Details

In comparison to bitcoin, a transaction has one new field (`nTime`), which tells when the transaction was created. This field is used by proof-of-stake mining to determine the age of the transaction. The new field was added incorrectly, and it's not being signed and verified (`CTransactionSignatureSerializer` was not updated); thus, anyone can modify it, and the transaction will still be valid. This behavior is similar to the past bitcoin transaction malleability issue, but in this case, it allows to manipulate proof-of-stake consensus.

Recommendation

Verification of transaction `nTime` field should be added to `CTransactionSignatureSerializer`. This will make existing transactions invalid, so a proper migration strategy must be implemented.

Block DoS

In proof-of-stake mining, a node able to mine a new block can generate an infinite number of new, valid blocks causing a denial of service.

ID	INDX-105
Scope	Block validation
Severity	CRITICAL
Vulnerability Type	Denial of service
Reproduction Test	issues_reproduction/block_dos.py
Status	Fixed

Details

When a node mines a new proof-of-stake block, it can generate an infinite number of new, valid blocks for the same height just by adding different transactions to the block (The hash of the block will change, but it will still be valid). In proof-of-work mining, it is very expensive to generate new blocks, so `Bitcoin Core` lacks a system to protect against receiving a lot of valid blocks for the same height (tip). Because of that, a node spamming a lot of valid blocks can easily cause a denial of service attack. It also makes Simplified Payment Verification impossible to use.

Recommendation

There is no simple solution to this problem. A system to deal with nodes sending too many valid blocks for a given height must be implemented. It is also recommended to redesign the proof-of-stake mining algorithm to limit such behavior.

Block header DoS

In proof-of-stake mining, it is impossible to validate whether the block header is valid or not. Anyone can generate an infinite number of new block headers, causing a denial of service attack.

ID	INDX-106
Scope	Block header validation
Severity	CRITICAL
Vulnerability Type	Denial of service
Status	Fixed

Details

In proof-of-work mining, correctness of block header can be easily validated by checking proof-of-work value. However, in proof-of-stake mining, there is no way to tell if block header is valid or not unless whole block is fully downloaded.

`CheckBlockHeader` in `validation.cpp`:

```
static bool CheckBlockHeader(const CBlockHeader& block, BlockValidationState& state, const Consensus::Params& consensusP
{
    // Check proof of work matches claimed amount
    if (block.IsProofOfWork() && !CheckProofOfWork(block.GetHash(), block.nBits, consensusParams))
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-pow", "proof of work failed");
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

```
    return true;
}
```

Lack of validation allows anyone to spam the network with infinite number of "valid" block headers causing denial of service. In case of proof-of-work such attack would be extremly expensive, impossible in practice. Because of that `Bitcoin Core` does not have any system to deal with huge number of block headers.

Recommendation

A system to deal with nodes sending too many block headers must be implemented.

Incorrect stake modifier cache

Caching mechanism implemented for `GetKernelStakeModifier` function is not correctly implemented, which can lead to accepting invalid blocks and network split.

ID	INDX-107
Scope	Consensus
Severity	CRITICAL
Vulnerability Type	Invalid caching mechanism
Reproduction Test	issues_reproduction/incorrect_stake_modifier_cache.py

In `pos/kernel.cpp`, the function `GetKernelStakeModifier` has the following caching mechanism:

```
static bool GetKernelStakeModifier(CChainState* active_chainstate, uint256 hashBlockFrom, uint64_t& nStakeModifier, int&
{
    const Consensus::Params& params = Params().GetConsensus();
    nStakeModifier = 0;
    const CBlockIndex* pindexFrom = active_chainstate->m_blockman.LookupBlockIndex(hashBlockFrom);
    if (!pindexFrom)
        return error("GetKernelStakeModifier() : block not indexed");
    nStakeModifierHeight = pindexFrom->nHeight;
    nStakeModifierTime = pindexFrom->GetBlockTime();
    int64_t nStakeModifierSelectionInterval = GetStakeModifierSelectionInterval();
    // Check the cache first
    uint64_t nCachedModifier;
    cachedModifier entry { nStakeModifierTime, nStakeModifierHeight };
    if (cacheCheck(entry, nCachedModifier)) {
        nStakeModifier = nCachedModifier;
        LogPrint(BCLog::POS, "%s: nStakeModifier=0x%016x cache hit!\n", __func__, nStakeModifier);
        return true;
    }
    // loop to find the stake modifier later by a selection interval
    while (nStakeModifierTime < pindexFrom->GetBlockTime() + nStakeModifierSelectionInterval) {
        ...
        pindex = active_chainstate->m_chain.Next(pindex);
        ...
    }
    nStakeModifier = pindex->nStakeModifier;
    cacheAdd(entry, nStakeModifier);
}
```

The problem is, that `nStakeModifier` may be different for different active_chainstate, especially when validating a new chain forking from some older block. This can be used to make the node accept an invalid block or reject a valid one. The test incorrect_stake_modifier_cache.py was created to prove this behavior.

Recommendation

The cache should be reimplemented to work correctly with different branches (taking into account different CChainStates).

Coinstake DoS

There is a out-if-bonds array read in `CheckProofOfStake` which can crash the application.

ID	INDX-108
Scope	Block validation
Severity	CRITICAL
Vulnerability Type	Out-of-bounds read (denial of service)
Reproduction Test	issues_reproduction/coinstake_dos.py
Status	Fixed in f07ee9

Details

In `pos/kernel.cpp`, in function `CheckProofOfStake`, the following code can be found:

```
const CTxOut& prevOut = txPrev->vout[tx->vin[nIn].prevout.n];
```

in the `CheckProofOfStake` function:

```
bool CheckProofOfStake(CChainState* active_chainstate, BlockValidationState& state,  const CTransactionRef& tx, unsigned
{
    ...
    {
        int nIn = 0;
        const CTxOut& prevOut = txPrev->vout[tx->vin[nIn].prevout.n];
        TransactionSignatureChecker checker(&(*tx), nIn, prevOut.nValue, PrecomputedTransactionData(*tx), MissingDataBel
        if (!VerifyScript(tx->vin[nIn].scriptSig, prevOut.scriptPubKey, &(tx->vin[nIn].scriptWitness), SCRIPT_VERIFY_P2S
            return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "invalid-pos-script", "VerifyScript failed on 
    }
    ...
}
```

When the code is accessing the output of the previous transaction in `txPrev->vout[tx->vin[nIn].prevout.n]`, the value of `tx->vin[nIn].prevout.n` is not validated and can be any value. It can easily crash the node when this value is out of bounds, for example, 1000000000.

Recommendation

Add the missing validation for preventing wrong format inputs or large numbers in `tx->vin[nIn].prevout.n`

Duplicated coinstake proof

If is possible to create the same coinstake proof for different transactions, when one of them will be valid (to be used as coinstake), all the other transactions with the same proof will be valid too.

ID	INDX-109
Scope	Block validation
Severity	MEDIUM

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Vulnerability Type	Seed manipulation for pseudo-random function

Details

In `pos/kernel.cpp`, in function `CheckStakeKernelHash` there is the following pseudo-random number generator to determine if the stake is valid or not.

```
CDataStream ss(SER_GETHASH, 0);
ss << nStakeModifier;
ss << nTimeBlockFrom << nTxPrevOffset << nTimeTxPrev << prevout.n << nTimeTx;
hashProofOfStake = Hash(ss.begin(), ss.end());
```

It is possible to create multiple transactions with the same parameters for this pseudo-random function. When one transaction is a valid coinstake, all of them will be valid, allowing someone to mine multiple valid blocks.

The problem exists because `nTxPrevOffset` is equal to `prevout.n`. In original peercoin implementation `nTxPrevOffset` is an equal position of transaction in the block, which makes it (almost) impossible to create multiple transactions with the same parameters for this pseudo-random function.

However, the blockchain node can accept only 6 blocks with the same `nTimeTx`, so this vulnerability allows to mine only 6 blocks in the same time slot instead of 1 block.

Recommendation

Restore original peercoin implementation of `nTxPrevOffset`, or add a new parameter that should be unique for each transaction.

Invalid block synchronization

When a block is being synchronized by using BLOCKTXN network message, it will be synchronized incorrectly due to the lack of `vchBlockSig` synchronization.

ID	INDX-110
Scope	P2P
Severity	MEDIUM
Vulnerability Type	Invalid serialization
Status	Fixed in e578f0

Details

Block field `vchBlockSig` containing the block signature is not transferred when a block is synchronized using CMPCTBLOCK.

Recommendation

Block field `vchBlockSig` should be moved from full block to BlockHeader or `vchBlockSig` should be added to `BlockTransactions` network message and correctly synchronized.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Invalid block validation

Proof-of-stake block may be set as valid (checked) even when it has an invalid signature.

ID	INDX-111
Scope	Block validation
Severity	MEDIUM
Vulnerability Type	Improper block validation
Status	Fixed in bd0f6c

Details

In `CheckBlock` in `validation.cpp`, the following part of code:

```
bool CheckBlock(const CBlock& block, BlockValidationState& state, const Consensus::Params& consensusParams, bool fCheckI
{
    ...
    if (fCheckPOW && fCheckMerkleRoot)
        block.fChecked = true;
    // check block signature
    if (block.IsProofOfStake() && !CheckBlockSignature(block)) {
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-blk-sign", "bad block signature");
    }
    return true;
}
```

will set `block.fChecked` to true even when `CheckBlockSignature` returns false. Even if it is not possible to abuse this behavior at this point, it is still an invalid implementation and, therefore, can represent a threat in future changes.

Recommendation

The block should be set as checked only after `CheckBlockSignature` validation.

CBlockIndex exception

Function `CBlockIndex::ToString()` will always throw an exception due to invalid parameters.

ID	INDX-112
Scope	Debugging
Severity	LOW
Vulnerability Type	Unhandled exception
Status	Fixed in 15bf19

Details

Function `CBlockIndex::ToString()` in `chain.h`:

```
std::string ToString() const
{
```

```
    return strprintf("CBlockIndex(nprev=%08x, nFile=%d, nHeight=%d, nMint=%s, nMoneySupply=%s, nStakeModifier=%016llx, 
        pprev, nFile, nHeight,
        FormatMoney(nMint), FormatMoney(nMoneySupply),
        GeneratedStakeModifier() ? "MOD" : "-", GetStakeEntropyBit(), IsProofOfStake()? "PoS" : "PoW",
        nStakeModifier, nStakeModifierChecksum,
        hashProofOfStake.ToString(),
        prevoutStake.ToString(), nStakeTime,
        hashMerkleRoot.ToString().substr(0,10),
        GetBlockHash().ToString().substr(0,20));
}
```

will always throw a tiny format exception because the number of strprintf parameters is greater than the number of parameters required by format string (15 > 12). However, this function is only being used when the blockchain node is being tested/debugged, so it cannot be currently abused in a production environment.

Recommendation

Validate and format the amount of parameters passed to the function.

Usage of floating point arithmetic

Floating point arithmetic used by consensus may work differently on various platforms.

ID	INDX-114
Scope	Consensus
Severity	LOW
Vulnerability Type	Usage of implementation/platform specific code

A lot of new functions, introduced with the new consensus like `GetCoinAgeWeight` in `pos/kernel.cpp` for example contains a floating point arithmetic:

```
int64_t GetCoinAgeWeight(int64_t nIntervalBeginning, int64_t nIntervalEnd, const Consensus::Params& params)
{
    if (nIntervalBeginning <= 0) {
        LogPrint(BCLog::POS, "%s: WARNING *** GetCoinAgeWeight: nIntervalBeginning (%d) <= 0\n", __func__, nIntervalBeg
        return 0;
    }
    int64_t nSeconds = std::max((int64_t)0, nIntervalEnd - nIntervalBeginning - params.nStakeMinAge);
    double days = double(nSeconds) / (24 * 60 * 60);
    double weight = 0;
    if (days <= 7)
        weight = -0.00408163 * pow(days, 3) + 0.05714286 * pow(days, 2) + days;
    else
        weight = 8.4 * log(days) - 7.94564525;
    return std::min((int64_t)(weight * 24 * 60 * 60), (int64_t)params.nStakeMaxAge);
}
```

The problem is that in C++, standard floating point arithmetic is implementation-defined and may work differently on some platforms. It won't be a problem in the case of most modern platforms, but it may not work in the same way on other platforms, causing an unintended network fork.

Recommendation

Avoid the usage of floating point arithmetic in any consensus-related function.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Slow GetLastBlockIndex

Function `GetLastBlockIndex` in `chain.cpp` has `O(n)` complexity where n is a number of blocks, it may be very slow in the future.

ID	INDX-113
Scope	P2P
Severity	LOW
Vulnerability Type	Inefficient function

Function `GetLastBlockIndex`:

```
const CBlockIndex* GetLastBlockIndex(const CBlockIndex* pindex, bool fProofOfStake)
{
    while (pindex && pindex->pprev && (pindex->IsProofOfStake() != fProofOfStake))
        pindex = pindex->pprev;
    return pindex;
}
```

which is being used by `PeerManagerImpl::ProcessGetBlockData` in `net_processing.cpp`:

```
...
vInv.push_back(CInv(MSG_BLOCK, GetLastBlockIndex(m_chainman.ActiveChain().Tip(), false)->GetBlockHash()));
...
```

searches for last proof-of-work, starting from latest blocks. Proof-of-work mining ends at block with height 2200, so it will need to iterate over almost all blocks, which may be slow.

Recommendation

Height of the last proof-of-work block is fixed, it should be returned in `O(1)` time.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Long range and grinding attack against consensus

Indxcoin's proof of stake consensus implementation is vulnerable to long range and grinding attacks. An attacker with a very low stake (less than 0.01% of all existing coins) can potentially perform such attack.

Details

In Indxcoin consensus, any transaction can be used to create a new block, known as a coinstake. The likelihood of a transaction successfully mining a new block is determined by three factors: the number of coins involved in the transaction, the age of the coins since they were last used as a coinstake, and a random value generated based on the transaction parameters and the stake modifier of the blockchain.

The random value is divided by the number of coins and the age of the coins, and if it is lower than the current difficulty, the block with the given coinstake is considered valid. The block interval is set to 60 seconds.

The most important part of this consensus algorithm is the pseudo-random function:

```
CDataStream ss(SER_GETHASH, 0);
ss << nStakeModifier;
ss << nTimeBlockFrom << nTxPrevOffset << nTimeTxPrev << prevout.n << nTimeTx;
uint256 hashProofOfStake = Hash(ss); // Hash is SHA-256
```

which is used to generate the random value. This function takes the following six parameters as input:

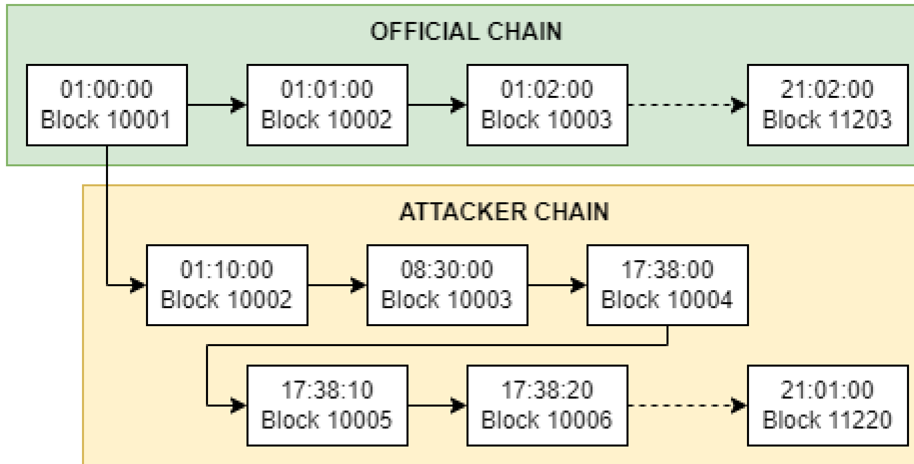
- `nStakeModifier`: a value computed every 13 minutes from the last bit of selected block hashes from the last 8 hours
- `nTimeBlockFrom`: the time of the block that mined the transaction used to create the coinstake
- `nTxPrevOffset`: the position of the output used to create the coinstake transaction
- `nTimeTxPrev`: the nTime field of the transaction used to create the coinstake
- `prevout.n`: has the same value as nTimeBlockFrom
- `nTimeTx`: the time of the new block

The idea behind this function is that all parameters except `nTimeTx` and `nStakeModifier` are not modifiable. The `nStakeModifier` is changing on average every 13 minutes, and a user must wait at least 8 hours to be able to use their transaction as a coinstake to mine a new block. This makes it theoretically impossible to predict the random value at the time of creating a transaction.

However, there is a special case that allows for the prediction of all parameters used by this pseudo-random function: when the attacker is the only miner for a given chain. In this case, the random value can be calculated into the future, making it possible to prepare transactions that will be valid coinstakes with a very low stake. Using a medium-range modern GPU makes it possible to compute 2^32 SHA-256 hashes within a few seconds, which allows for the typical stake requirement to be reduced by a factor of 1,000 million. This allows the attacker to mine thousands of correct blocks within a few hours, making their chain the longest in the network.

An attacker can start their own chain at any point in time. All they need to do is mine at least 3 blocks where the time between the first and last block is greater than the minimum stake period of 8 hours. The diagram below illustrates such an attack: the attacker needs 16 hours to mine their 3 blocks, but then they gain control over the stake modifier, allowing them to mine new blocks 6 times faster than the official chain. Within a few hours, the attacker has the longest chain in the network, making it the new official chain. The big advantage for the attacker is that they can check at any moment if they are able to generate a given number of new blocks within a given time period.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io



During our audit, we created a tool that calculates all valid coinstakes for a given set of parameters. This tool demonstrates that a potential attacker who gains control over the stake modifier can easily compute the transactions needed to efficiently mine a large number of valid blocks using a minimal amount of stake. The tool is located in coinstake_gpu_finder audit repository.

Real world example

As a real-world example, let's consider the blockchain Reddcoin, which uses the same consensus algorithm and has a market cap of approximately $10M. Reddcoin has 30B coins in circulation, with a coin value of $0.33. The current block difficulty is 118. Using our tool, we calculated that an attacker with 1M coins aged 30 days would have more than an 80% chance of mining three blocks within 24 hours. This would allow the attacker to take control over the stake modifier and subsequently mine new blocks every 2 seconds using an average of 500 coins per block for coins aged 30 days. To mine the subsequent 2,000 blocks within 2 hours, the attacker would need to have an additional 1M coins.

Based on these parameters, it is possible to perform a successful attack against Reddcoin with only 2M coins, representing less than 0.007% of all coins in circulation and a value of approximately $700. This demonstrates the feasibility of executing such an attack with a relatively small amount of stake.

Recommendations

There is no simple solution to mitigate the potential for a long-range & grinding attack. However, there are several measures that can be taken to make such an attack more expensive for the attacker. These include:

- Setting a minimum stake requirement for mining a block, such as 0.001% of all coins
- Increasing the minimum age of coins from 8 hours to several days
- Modifying the calculation of the stake modifier to include not only the most recent 8 hours of blocks, but also the 64-128 most recent blocks regardless of their age
- Choosing different parameters for the pseudo-random function that are more difficult for the attacker to calculate
- Using a slower random number generation algorithm, such as one that is at least 1 million times slower than the current SHA256 algorithm, to make it much harder for the attacker to generate large numbers of random values.

Implementing these measures can help to make this attack more expensive and less likely to succeed.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Test coverage

At the beginning of the audit, the code coverage of tests could not be determined due to the majority of the tests not functioning correctly.

Following the introduction of changes to Bitcoin Core 22.0 fork, the project did not rectify pre-existing tests or incorporate new ones to verify these changes. This inadequacy also extends to Quality Assurance tests, which are presently non-functional.

The tests (command `make coverage`) were fixed few months later with commit fa7bcd0ec2a5431a6b356b1e9655b906ee579828. Below are the results of test coverage for that commit.

Directory	Indxcoin line coverage	Bitcoin Core 22.0 line coverage
src	69.2%	87.4%
src/compat	52.5%	87.5%
src/consensus	85.6%	99.5%
src/crypto	82.5%	81.0%
src/node	55.1%	76.9%
src/policy	68.8%	95.1%
src/pos	75.8%	n/a
src/pos/wallet	0.8%	n/a
src/primitives	86.7%	94.3%
src/rpc	47.1%	94.3%
src/script	76.7%	85.9%
src/support	90.0%	94.7%
src/wallet	44.0%	78.8%

Directory	Indxcoin functions coverage	Bitcoin Core 22.0 functions coverage
src	62.2%	79.0%
src/compat	44.4%	100%
src/consensus	87.0%	100%
src/crypto	93.6%	93.8%
src/node	53.9%	65.8%
src/policy	85.1%	97.9%
src/pos	88.0%	n/a
src/pos/wallet	5.6%	n/a
src/primitives	80.5%	95.5%
src/rpc	60.6%	97.4%

src/script	75.9%	85.9%
src/support	84.0%	88.0%
src/wallet	53.2%	72.5%

Compared to Bitcoin Core 22.0, the test coverage is significantly lower. During the audit, it was unable to make some tests work. Instead, they were removed or commented. New functionalities could be better tested; negative scenarios are usually not tested. This also applies to the migration to the new protocol version (IsProtocolV02), as it is not currently tested.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken disclaimer

The code base provided for audit has been analyzed according to the latest industry code quality, software processes and cybersecurity practices at the date of this report, with discovered security vulnerabilities and issues the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functional specifications). The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code (branch/tag/commit hash) submitted to and reviewed, so it may not be relevant to any other branch. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits, public bug bounty program and CI/CD process to ensure security and code quality. English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical disclaimer

Protocol Level Systems are deployed and executed on hardware and software underlying platforms and platform dependencies (Operating System, System Libraries, Runtime Virtual Machines, linked libraries, etc.). The platform, programming languages, and other software related to the Protocol Level System may have vulnerabilities that can lead to security issues and exploits. Thus, Consultant cannot guarantee the explicit security of the Protocol system in full execution environment stack (hardware, OS, libraries, etc.)