A CONSENSYS DILIGENCE AUDIT REPORT

Slock.it Incubed3

Date	September 2019
Lead Auditor	Martin Ortner
Co-auditors	Shayan Eskandari

1 Summary

ConsenSys Diligence conducted a security audit on Slock.it's Trustless Incentivized Remote Node Network, in short INCUBED. Incubed is to solve the issue of connecting IoT devices to the blockchain (Ethereum). Many IoT devices are severely limited in terms of computing capacity, connectivity and often also power supply. Connecting an IoT device to a remote node enables even low-performance devices to be connected to blockchain.

2 Audit Scope

The audit scope is defined as follows:

1. The Incubed Algorithm

    - Review the specification for security considerations
    - Checking against the provided Threat Model
    - Prepare a report with additional threats and an adapted risk assessment of the currently identified

2. Smart Contract Audit of the `in3-contracts`

    - Validating the correct implementation of the specification.
    - Assuring that no funds can be lost.
    - Identifying other vulnerabilities and unexpected behaviour

3. Review of the Incubed Node RPC extension to verify that
   - The private keys are managed securely
   - The node never signs an invalid blockhash (which would lead to loss of funds)

This audit covered the contracts included in `in3-contracts` (Commit hash: `e25c758a115aef0c0640bc446027259aa7cb1a52`) and the client code included in `in3-server` (Commit hash: `fdc5eb104c4c3aba0af79222d54e6effaef6f3a7`) repositories. The `contracts` located in the `in3-server` repository are out of scope for the Smart Contract Audit. The client implementation and an `in3-server` code audit is not in scope for the review.

List of the files covered in this audit:

File	SHA-1 hash
in3-contracts/contracts/NodeRegistry.sol	85a9632c9e8e41057ae11d772c623a0fb9 2af7f8
in3-contracts/contracts/BlockhashRegistry.sol	1818e466573344fbf826fd144b96a2c6ef4f 786e

Directories covered for the Incubed Node RPC extension review:

File	Main Files
in3-server/src	server.ts, rpc.ts, EthHandler.ts, BaseHandler.ts, signatures.ts

A detailed audit of the provided proofs was not in scope for this audit.

Update (December 2, 2019): It should be noted that the IN3 architecture has changed since the original audit in September 2019. This report does not extensively reflect these changes and was written based on the original submitted code. The new changes, including, but not limited to:

- Separation of `NodeRegistry` into two contracts: `NodeRegistryData` and `NodeRegistryLogic`
- Addition of a new contract: `IN3WhiteList.sol`
- Addition of ERC20 tokens into the system
- Implementation of DoS protection for the IN3 server

The audit team evaluated that the system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three broad categories:

1. **Security:** Identifying security related issues within the contract.
2. **Architecture:** Evaluating the system architecture through the lens of established smart contract best practices.
3. **Code quality:** A full review of the contract source code. The primary areas of focus include:
   - Correctness
   - Readability
   - Scalability
   - Code complexity
   - Quality of test coverage

2.1 Documentation

The following documentation was available to the project team:

- Inline code documentation
- Talk
- Website
- Documentation
  - Specification
  - Threat Model

3 System Overview

The high level architecture of the Incubed network consists of the following three layers. Each layer is described in more detail in this section.

- The Smart Contract Layer - **in3-Registries**
- The Node Layer - **in3-Nodes**
- The Client Layer - **in3-Clients**
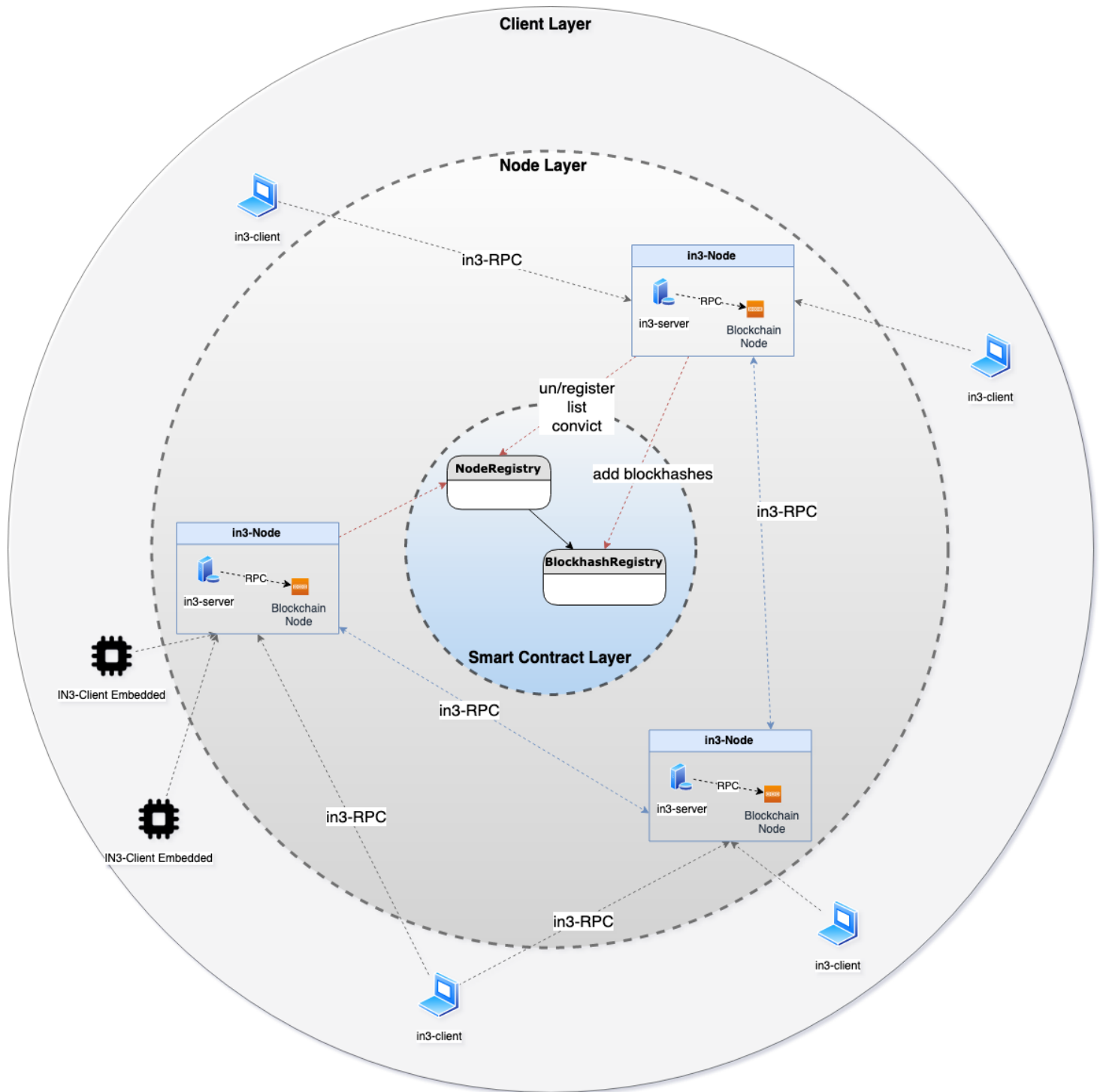
3.1 in3-Registries

Two Smart-Contracts build the backbone of the system - `NodeRegistry` and
`BlockhashRegistry` . They are deployed on the Ethereum blockchain and provide the
following functionalities:

NodeRegistry

This is the central node registry of the Incubed system. Clients start up and initially contact
a hardcoded list of boot-nodes. in3-nodes have hardcoded list of the registry contracts
and can provide clients with information about registered nodes.

- Handles the registration life-cycle of new in3-nodes.

- The account registering a node is set as the owner of the node information. The owner manages the node, can update its properties, un-register it and transfer the ownership to someone else. One account can be the owner of multiple nodes in the system. It should be noted that each owner can registerNodeFor multiple `signers`, which will be signing from their associated nodes.
- The node information includes an account that is specified as the `signer`'s address for the node. There are two entry points for nodes to be submitted, one will register the node with the `signer`'s address to be set to the account that is registering the node (which is also the owner), and the other allows an account to submit a node for a different `signer` address. For the latter to succeed the signer has to provide the signed node properties as a proof that an account is indeed allowed to submit the node for this signer. Each `signer` can only be associated with one unique URL (node).
- Registration requires a minimum deposit (`10 finney` - hard-coded).
- Unregistering a node removes it from the list of registered nodes. Funds previously deposited by the node cannot immediately be withdrawn as they are locked for a duration of time chosen by the node upon registration `timeout` (minimum 1 hour - hard-coded).

- Provides access to the list of registered in3-nodes.
- Allows other participants (not restricted to actual nodes) to convict nodes if they signed wrong blockhashes. Conviction follows a commit-reveal scheme in an attempt to prevent front-running.
  - The system incentivizes nodes to convict other nodes that misbehave (signing wrong information in responses on the in3-Nodes layer) by awarding half of `node.depost` to the successful convictor, while burning the other half to avoid nodes from convicting themselves in an attempt to unregister before others convict them.
- The registry is deployed with an `admin` -key named `unregisterKey`, managed by the Slock.it team, that allows them to remove nodes from the registry. This functionality is automatically revoked after one year from the deployment of the smart contract.
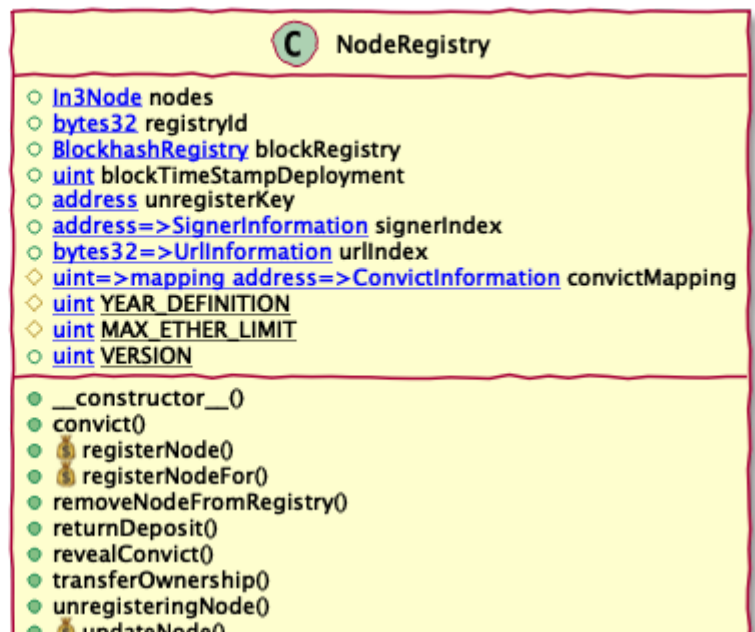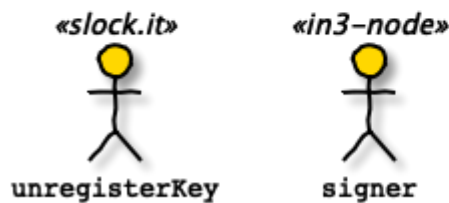
BlockhashRegistry

The ethereum virtual machine (`evm`) can only provide the blockhash of the most recent 256 blocks, excluding the current block. The ethereum block times [vary](#) and are typically within 10s and 20s. This means that the `evm` can provide blockhashes back in time for only around 1 hour. This means that nodes would only be able to convict rogue nodes that return wrong blockhashes within that window. In order to tackle this problem and allow nodes to convict even older responses for blockhashes the system provides a `BlockhashRegistry` .

The only incentive for someone to actually spend gas on adding new blockhashes to the `BlockhashRegistry` is to convict a node in the `NodeRegistry` . In case a participant attempts to convict a `signer` for a blockhash that cannot be provided by the `evm` , the `NodeRegistry` will call out to `BlockhashRegistry` and check if the blockhash is available there. If it is not available in the `BlockhashRegistry` the `signer` cannot be convicted. In order to successfully prove that the `signer` signed a wrong blockhash and is to be convicted - burning half of the `signer` 's deposit and awarding the other half to the `convictor` - the `convictor` has to provide the correct blockhash to the `BlockhashRegistry` . Adding new blockhashes is not

trivial and requires the `convictor` to provide a chain of rlp-encoded blockheaders up to an already trusted blockhash in the registry (trust anchor).

The registry provides the following functionality:

- Search for a blockhash in the registry within a given range. This is meant to be used offline by directly talking to an Ethereum node
- Store the blockhash of the previous block or a given `block.number` available to the `evm` in the registry.
- Store an arbitrary blockhash for a block by re-calculating it from a list of provided rlp-encoded blockheaders. The blockheaders must be anchored to an existing entry in the registry.
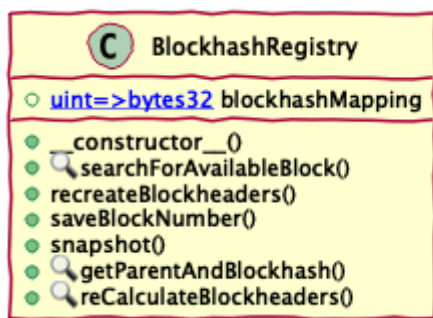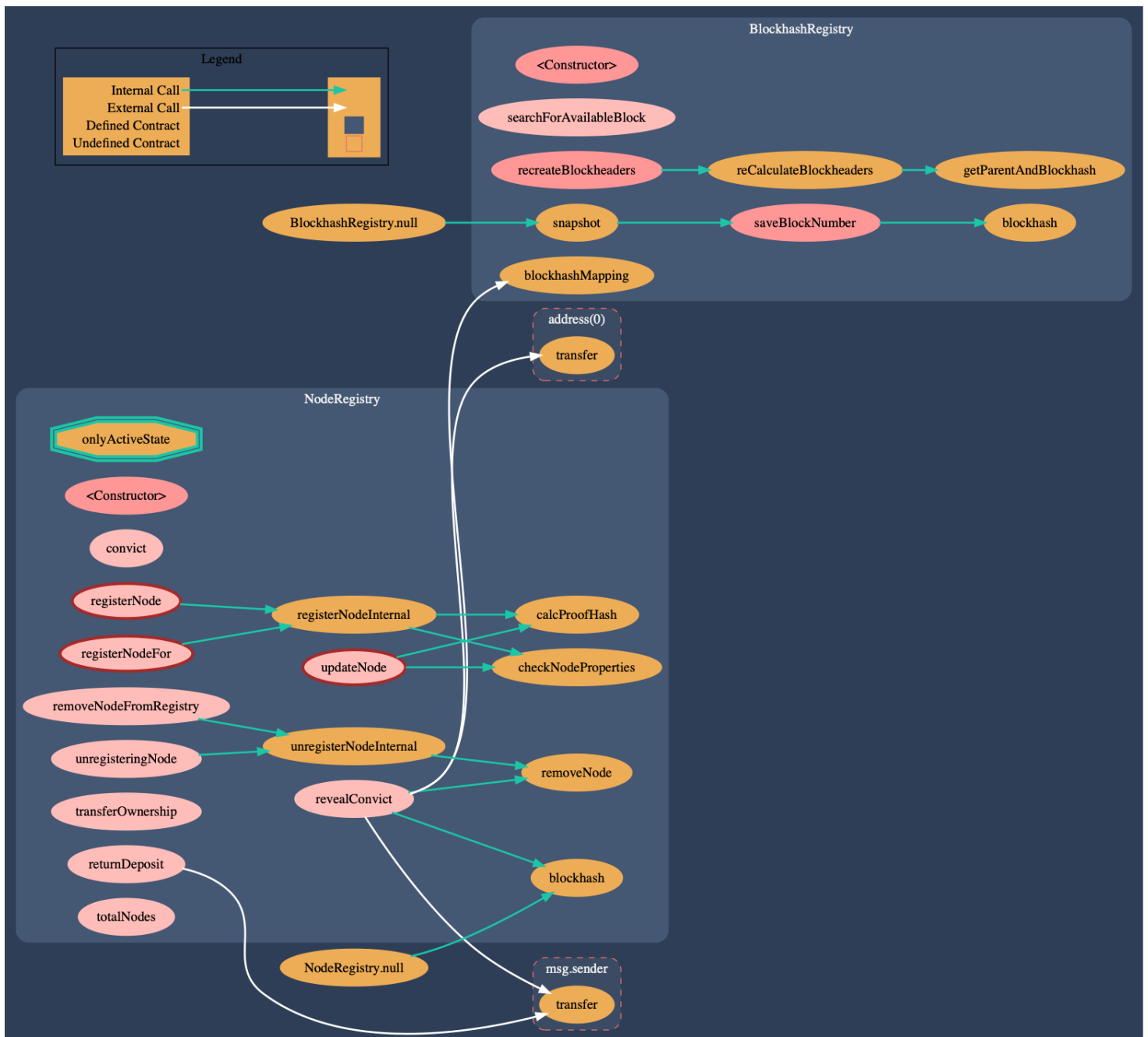

Inheritance

[dot file]

Call Graph

[dot file]

3.2 in3-Nodes

This layer is a network of in3-nodes that can communicate with each other via an extension to the Ethereum RPC protocol. in3-nodes facilitate access to ethereum blockchain nodes by transparently proxying Ethereum RPC interface to other in3-nodes.

The node-to-node and client-to-node communication follows a request-response schema. A client (or node) can use a node to get access to the ethereum blockchain using the Ethereum RPC interface. An in3-node extends the Ethereum RPC interface with the following methods:

- `in3_sign` - request the node to sign a specific blockhash for a given blocknumber
- `in3_nodeList` - request the nodelist

- `in3_stats` - provides node statistics
- `in3_validatorlist` - request the validatorlist
- `in3_call` - is an alias for `eth_call`

In addition to that, merkle-proofs can be requested for specific Ethereum RPC calls:

- **blockProof** - Verifies the content of the BlockHeader: `eth_getBlockByNumber` , `eth_getBlockByHash` , `eth_getBlockTransactionCountByHash` , `eth_getBlockTransactionCountByNumber`
- **transactionProof** - Verifies the input data of a transaction: `eth_getTransactionByBlockHashAndIndex` , `eth_getTransactionByBlockNumberAndIndex` , `eth_getTransactionByHash`
- **receiptProof** - Verifies the outcome of a transaction `eth_getTransactionReceipt`
- **logProof** - Verifies the response of `eth_getLogs`
- **callProof** - Verifies the result of call response: `eth_call` , `in3_call`
- **accountProof** - Verifies the state of an account: `eth_getCode` , `eth_getBalance` , `eth_getTransactionCount` , `eth_getStorageAt`

Only blockhashes are signed by nodes when they are requested to do so by calling `in3_sign` . Proofs are not signed but can contain a signed blockhash. Unhonest nodes can only be convicted for signing wrong blockhashes.

3.3 in3-Clients

1. The in3-network layer - A network of `in3-server` nodes that can communicate with each other via an extended Ethereum RPC protocol, providing access to Ethereum blockchain data.
2. The in3-client layer - Further information about the general concept and architecture can be found here.

The in3-Client is not in scope for this audit.

4 Key Observations/Recommendations

3.1 in3-Registries

- The smart contract specifications at in3.readthedocs.io (github) is inaccurate.
  - The specification does not accurately describe the data-structures and their relationship in the smart contracts.
  - Node structure mentions an `unregisterTime` property that is not available in the smart contract's source code.

- The specification assumes one Node struct that contains all fields while Node, Signer, URL and Convict information is split up into multiple linked structures. Nodes are stored in an array and the `In3Node` struct references a signer. Signers are stored in a signer mapping that maps addresses to a `SignerInformation` struct. URLs are stored in an `UrlInformation` mapping indexed by `bytes(url)`. the `In3Node` struct stores the `url` as string and this string used as a reference to the `UrlInformation` struct. Note that `url` is stored in two different types, `bytes` and `string`.
  - A Node's `owner` is not directly stored in the `In3Node` struct but available by resolving the `In3Node`'s reference to `SignerInformation`.
  - While the specification declares `props` as `uint64` it is declared as `uint128` in code.
  - The property `proofHash` from the source code is not available in the specification.
- The contracts are not resilient to error conditions and input provided to methods should be more strictly controlled and kept within allowed ranges.
  - Conditional checks and input validation should be generally at the beginning of functions. Even though it may not lead to a vulnerability, it is still recommended to refactor functions to follow coding best practices (More details on this here).
  - Bookkeeping for deposits & payouts is done via a field in `In3Node` and `SignerInformation`. Complexity especially when it comes to dealing with funds should be avoided.
  - Assertions (`assert()`) should be used to verify invariants while `require()` should be used to validate inputs (SWC-110).

4.2 in3-server

- The code does not appear to be production ready.
  - The specification at in3.readthedocs.io is inaccurate (github) and does not list all available RPC commands.
  - Some methods are only documented in the api-cmd documentation but should be in a general API overview.
  - `in3_call` is an **undocumented** alias for `eth_call` (`/in3-server/src/modules/eth/EthHandler.ts#L96-L99`)
  - API naming is inconsistent: `in3_nodeList` vs. `in3_validatorlist` (case)
  - `watcher.ts` attempts to call method `cancelUnregisteringServer` which is not existent in `NodeRegistry` (`/in3-server/src/chains/watch.ts#L251-L252`)
  - `in3-server` includes code that may unexpectedly attempts to deploy a `NodeRegistry` in `registerNodes` if no registry was configured (

`/in3-server/src/util/registry.ts#L86-L87`). This is currently prevented by a check in `checkRegistry` , which is a function that seems to be under development.
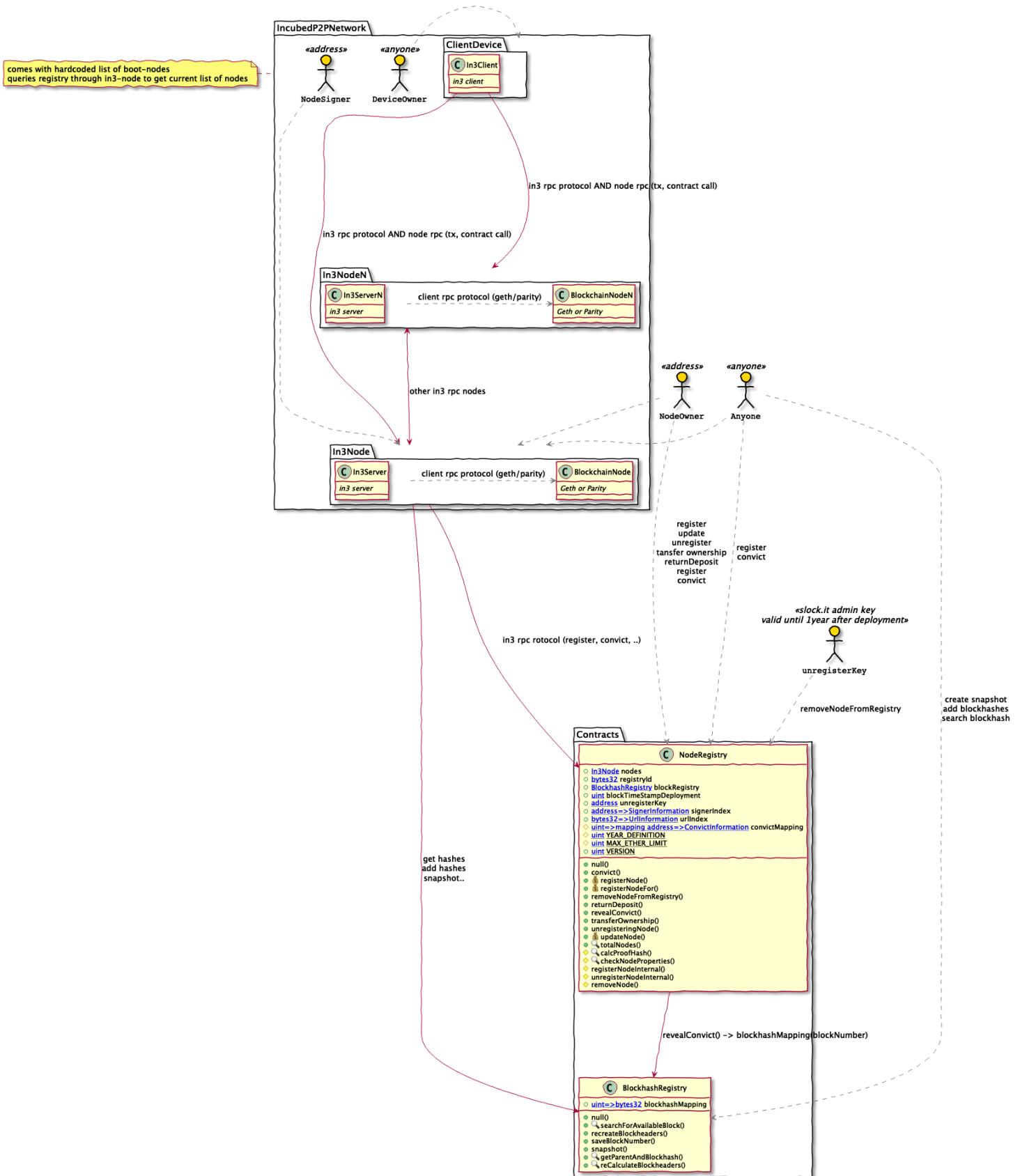
- Incubed specific methods discovered:
  - `in3_call`
  - `in3_nodeList`
  - `in3_validatorlist`
  - `in3_stats`
  - `in3_sign`
- Incubed nodes relay unhandled calls to blockchain nodes they are connected to.
- Incubed nodes can provide proofs for the following eth API calls:
  - blockProof: `eth_getBlockByNumber` , `eth_getBlockByHash` , `eth_getBlockTransactionCountByHash` , `eth_getBlockTransactionCountByNumber`
  - transactionProof: `eth_getTransactionByBlockHashAndIndex` , `eth_getTransactionByBlockNumberAndIndex`
  - transactionProof: `eth_getTransactionByHash`
  - receiptProof: `eth_getTransactionReceipt`
  - logProof: `eth_getLogs`
  - callProof: `eth_call`
  - accountProof: `eth_getCode` , `eth_getBalance` , `eth_getTransactionCount` , `eth_getStorageAt`
    - Note: `eth_getBalance` will only work with archive nodes. (Tested on `https://in3.slock.it/kovan/nd-2` and failed to respond)
- Incubed nodes do not raise an error condition if a node requests a proof for a method that does not support proofs.
- Some methods are blacklisted from being called. Any method that is not on the blacklist is forwarded to the blockchain node the `in3-server` is attached to. Blacklists can often be trivially circumvented. If blockchain nodes support custom api commands that are not in the `in3-server` blacklist, or if the Ethereum node is updated with new new RPC commands, this blacklist fails to function as intended. API commands can be implementation specific or even provided with future updates and not meant to be exposed to the public for security reasons. The incubed node method blacklist is insufficient.
  - blacklisted: `eth_sign` , `eth_sendTransaction` , `eth_submitWork` , `eth_submitHashrate`
  - node specific API that might get exposed: parity, geth-eth, geth-mgmt
- The server assumes a hardcoded configuration for the ethereum kovan chain using `https://kovan.infura.io/` as an RPC endpoint
- Node messages are not signed.

- It is up to the node to decide whether it wants to receive a proof or not. Nodes can just not provide an `in3.verification` field and the node will act like a normal blockchain node RPC endpoint.
- Code-complexity for the `in3-server` codebase is high and appears to be hard to maintain
  - Inline comments are sparse.
  - Coding style should be enforced.
  - Checking the existence of protocol specific values is too lax. E.g. strictly enforce that `request.in3.verification` is of a valid type instead of assuming everything that starts with `proof` requires a proof at `in3_nodeList` (`/in3-server/src/server/rpc.ts#L84-L84`)
  - Some method names are misspelled (e.g. `handeGetTransaction` , `handeGetTransactionFromBlock` , `handeGetTransactionReceipt` (`/in3-server/src/modules/eth/proof.ts#L269-L367`))
  - Redundancies - `EthHandler` is used in case `rpcConf.handler` is `eth` or `null` and also as default value (`(/in3-server/src/server/rpc.ts#L50-L61`). This can be reduced to a `switch` statement with one case, a default and one without the autoset to `eth` in case `rpcConf.handler` is empty.
  - RPC method handling is done across multiple files instead of inheriting implementations.
    - `in3_nodeList` , `in3_validatorlist` , `in3_stats` in `rpc.ts`
    - `in3_call` , `in3_sign` in `EthHandler.ts` (and `BaseHandler.ts`)
  - RPC methods should return an error on unexpected input instead of assuming defaults
  - Input validation - Especially when forwarding parts of user tainted data or complete calls to back-end services it is important to strictly control input that can reach these endpoints. For example, for parameters that are supposed to be numbers it should be checked that they are actually numbers within a valid range before forwarding them. For hashes it can be validated that they adhere to a certain format and length. Strings can be limited to a maximum allowed length.
- The documentation for api-cmd recommends unsafe key-management.
  - Keys should never be made available in a shell's environment as this environment variable may leak to other processes.
  - Keys should not be passed in plaintext via commandline options as they will be available to privileged accounts (procview) and likely leak to shell history files. It also encourages users to store keys in start-up scripts for their nodes.
  - It should be investigated whether the blockchain node can be used to securely sign messages for the incubed node. This way the incubed node would not have

to deal with private key handling. However, this would require a strong trust relationship between the incubed node and the blockchain node which is already weakened by the fact that the incubed node transparently forwards Ethereum RPC requests to the node.

5 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

comes with hardcoded list of boot-nodes
queries registry through in3-node to get current list of nodes

5.1 Actors

The relevant actors are as follows:

- **NodeOwner** - An Ethereum account that manages the node's life-cycle (register, unregister, update, withdraw deposit, and transfer ownership)

- **NodeSigner** - An Ethereum account used to sign in3-node responses. Can either be NodeOwner or a different account. Note that NodeOwner can be a multisig account,

however NodeSigner must be a normal (externally controlled) account.

- **UnregisterKey** - A time limited admin key. This account is managed by Slock.it.
- **Convictor** - An Ethereum account (or in3-node) that tries to convict a signer for signing wrong responses.
- Nodes **in3-node** - An implementation of the node in in3 protocol that connects in3-clients to the in3-network and the Ethereum blockchain. Can verify blockhashes directly with a connected node. May convict other nodes/signer in case they sign wrong blockhashes.
- Clients **in3-client** - An implementation of the client in in3 protocol that can verify blockhash proofs.
- **Watchdog** - Similar to an in3-node. Secures the network by randomly querying for blockhashes and verifying them with a local node or by asking other nodes in the system with the purpose of convicting nodes that provide dishonest signed responses.

5.2 Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- *in3-clients* are typically not directly connected to the blockchain, they can however request proofs from *in3-nodes* that certain *blockhashes* are actually on the blockchain.
- *in3-nodes* register with a central `NodeRegistry` smart contract on the Ethereum blockchain.
- Node registration requires a minimum amount of deposit to secure the system.
- This deposit is what is at stake for a node that provides wrong signed blockhash proofs and therefore makes sure that nodes act honestly. Any participant in the system or Ethereum account may attempt to convict a node for providing wrong signed blockhashes in order to be awarded half of the nodes deposit. The other half is burned to not incentivize dishonest nodes to convict themselves in order to withdraw their deposit early.
- The Ethereum account that registers a node is assigned the `owner` role for the node's registry entry.
- The `owner` can update node properties, unregister it, withdraw the deposit after the defined timeout when calling unregister, and transfer the ownership to other accounts.
- The `signer`'s account that is actually signing proofs can be different from the `owner`'s account.

- The `signer`'s account is initially in control of properties during the node registration process (properties must be signed for owner in order for the node to be registered if `signer` is not `owner`), however the `signer` now has control over funds stored in the contract or the node's status. The node can be unregistered and the deposit withdrawn by the `owner` without the `signer`'s consent. The `signer` therefore is in a strong trust relationship with `owner`.

- The `NodeRegistry` is centralized up until after one year of deployment of the `NodeRegistry` contract. An admin `unregisterKey` is allowed to remove active nodes from the registry. However, the admin cannot withdraw the node's funds, this can only be done by the node's owner.

- The `NodeRegistry` provides a directory service for nodes participating in the *in3-network*.

- *in3-nodes* are connected to an Ethereum blockchain node (e.g. Geth or Parity) via an Ethereum conform RPC protocol. This means, they are directly connected to the blockchain and do not have to request proofs themselves as they do not operate in any sort of light-mode.

- *in3-nodes* may ask other nodes to provide proofs in order to verify that they are honest and not providing any signed false information, are on the same chain and working correctly.

- *in3-clients*, based on the documentation, are shipped with a list of boot-nodes that connect them to the `NodeRegistry` to learn what other nodes are active in the system. *in3-client*'s may also have the capabilities to directly interact with the `NodeRegistry` contract in order to learn the network topology.

- *in3-clients* may interact with the Ethereum blockchain via *in3-nodes* that transparently proxy calls to their connected blockchain node.

- *in3-clients* do not have to register at the `NodeRegistry`, they are only consumers in the network.

- *in3-nodes* may unregister from the `NodeRegistry` in which case they are immediately removed from the registry. However, their deposit is locked up until the timeout specified upon node registration by the owner (minimum 1 hour after calling unlock; maximum 1 year).

- *in3-nodes* provide signed proofs to *in3-clients* for lightweight block verification.

- If an *in3-node* (or *watchdog*) detects that another node provides a signed wrong proof, they can attempt to convict the node in the `NodeRegistry`.

- The conviction process follows a commit-reveal scheme in an attempt to prevent front-running. The conviction can only be revealed after waiting for two blocks after the commitment (`convict()`).

- A node can only be convicted if the `NodeRegisty` can verify that the malicious node provided a wrong blockhash. In order to do so the blockhash must be available to the smart contract. This is the case when:
    1. the blockNumber is one of the most recent 256 blocks (except current)
    2. the block is available in the `BlockhashRegistry`
- The `BlockhashRegistry` is only needed to convict rogue nodes for blockhashes of blocks older than 256.
- Blocks are not automatically added to the `BlockhashRegistry`. They can manually be added to the `BlockhashRegistry` by calling one of the functions `snapshot()` or `saveBlockNumber()` to store one of the most recent 256 blocks blockhash (minimal gas consumption). They can also manually be added to the `BlockhashRegistry` by providing an anchoring blockhash that is already in the `BlockhashRegistry` as a starting point, and an array of rlp-encoded blockheaders that hash and chain to that specific blocknumber. The blockhash is needed in order to convict a misbehaving signer.
- The `BlockhashRegistry` is decentralized. There is no admin functionality.
- With the current design blockhashes in the `BlockhashRegistry` can be overwritten. However, ideally only with the one correct blockhash unless a flaw in the blockhash recreation mechanism is found and exploited.

5.3 Threat Model

This section is designed to discuss threat model prepared by Slock.it team, Thread Model for Incubed(src). This threat model is not complete without considering many other concerns mentioned in other sections such as, Key Observations/Recommendations, Trust Model, and Important Security Properties.

Component	Title	Details	Status	Notes
Nodes	Long Time Attack	see doc	open	Using bootstrap nodes mitigates this issue. However, if an old node comes back online, it might not have access to new bootstrap nodes to connect to. Also the system is vulnerable to compromised bootstrap nodes.

Component	Title	Details	Status	Notes
Registry	Inactive Server Spam Attack	see doc	partially addressed	Solutions such as *Dynamic Min Deposit* and *Voting* that are proposed will result in a more secure implementation.
Registry	Self-Convict Attack	see doc	closed	The solution used is to burn 50% of the deposit. However depending on the use case, the attacked client could result in more profit for attacker than 50% of the deposit, hence still profitable to sign a wrong block hash and self-convict. Also if someone convicts your node, you can self convict and get at least 50% back.
Registry	Convict Frontrunner Attack	see doc	closed	The implemented solution, obfuscates the convict work flow. In order to convict: • if the blockNumber is in last 256: the solution almost covers most cases, however the blocknumber and msg.sender will be known in the commit time, this information leakage can result in different front-running attacks. • if blockNumber is older than 256 blocks, user

Component	Title	Details	Status	Notes
				must first update `blockHashRegistry` which signals to everyone in the network about a possible conviction, considering the only incentive for external parties to update blocks is to get half of the convicted deposits, this could be more profitable for other network participants.
Network	Blacklist Attack	see doc	partially addressed	
Network	DDoS Attacks	see doc	closed	No DoS protection is implemented in in3-server. It is unrealistic to expect each node to buy DoS protection plans.
Network	None Verifying Data Provider	see doc	closed	Unclear threat description. Adding a chain of signatures can mitigate possible issues, however assurance of client's proper checking of the signatures is a crucial part of this implementation.
Privacy	Private Keys as API Keys	see doc	closed	Unclear threat description
Privacy	Filtering of Nodes	see doc	partially addressed	Not implemented yet. (Nodes can be filtered in the bootstrapping layer)

Component	Title	Details	Status	Notes
Privacy	Inspecting Data in Relays or Proxies	see doc	open	Not Implemented. Enforce HTTPS and Proper implementation and verification of PKI is required on both ends.
Risk	Risk Calculations	see doc	open	

5.4 Important Security Considerations

The following is a non-exhaustive list of security properties, concerns and threats that were verified in this audit:

- The most important security property of the system is that nodes are required to provide a minimum deposit upon registration. There is a chance that the deposit will be lost in case the node is misbehaving AND other nodes detect and convict the misbehaving node in the `NodeRegistry`.
  - Other nodes that do not observe the wrong proof are not able to convict the node.
  - Nodes may run out of funds and not be able to actually convict misbehaving nodes (not enough to cover gas fees), specially in the case that they are required to submit older blockheaders to `BlockhashRegistry` and follow up with the commit-reveal scheme to convict a node.
  - Nodes may choose not to convict other nodes (collusion).
  - Nodes may choose not to convict if it is not profitable for them (block older than 256 most recent blocks; a lot of effort to recreate blockhashes in the registry for older blocks as there might not be a lot of snapshots available to anchor the recreation).
  - Nodes may choose not to convict under a certain threshold of deposit.
  - Only signed blockheaders provided with proofs can be convicted, however as the client should check the signature before continuing the process, this assumption should hold.
  - It is cheaper to convict within the most recent 256 blocks.
  - It can be very expensive to convict for blocks older than the most recent 256.
  - Trust in the conviction lies in the ability for the smart contract to validate that a blockhash is indeed wrong.

- Blockheader structure is not validated when adding blockhashes for blocks that are not available to the EVM.
- A malicious actor that can add wrong blockhashes to the BlockhashRegistry is able to convict honest nodes.
- Clients may request proofs without a signed blockheader, weakening the security of the system.
- Nodes may attempt to provide false information on fields in the response that are not verifiable by proofs.
- Transport security is not enforced. A fallback to `HTTP` is hinted with property `0x08` in the node registry signaling that the node allows interaction over insecure `http` even though the url is specified as `https`.
  - For plaintext protocols (HTTP) intermediary network hops will be able to observe requests and responses and even tamper with them.
  - Nodes might not be able to validate self-signed certificates. Self-signed certificates should not be accepted anyway.
  - It is unclear if it is feasible to assume that every node will be able to provide a valid certificate signed from a trusted 3rd party.
  - Node's trustStore must be kept up-to-date in order to validate https certificates. Nodes that fail to update the trustStore may reject communications with other nodes.
- Nodes expose the Ethereum RPC interface of the connected Node.
  - This might expose a special trust relationship that can exist with the local blockchain node that might provide privileged functionality for clients connecting from localhost. It is suggested that incubed provides a guide that outlines best practices for security when connecting incubed nodes to a blockchain node.
  - This has an impact on the audit trail for blockchain nodes as their logs will always show the incubed node as the origin for a request. In order to create an accurate audit trail and be able to investigate and collect evidence for potential security relevant the incubed node must make sure to produce a consistent audit trail (log) for requests that are proxied to the blockchain node.
- Nodes may listen to `NodeRegistry` events and perform actions. E.g. `watch.ts` subscribes to `LogNodeUnregisterRequested` (Note: this Event is invalid) and performs a call if the event is emitted. This might be used by a malicious attacker to create a lot of events to force the node to spend gas on a call.
- There is a designed imbalance between the work the client and the incubed node has to perform:
  - Clients can be lightweight, resource constraint
  - Nodes may be more powerful

- However, this imbalance can be exploited by malicious clients to force nodes into performing a lot of work with minimum effort on the client side favoring DoS scenarios (See Issue 6.1 and Issue 6.7).
- Nodes might decide to only provide responses to clients insecurely configured (only proofs no signatures) clients in an attempt to avoid conviction.
- The network starts out centralized with most of the nodes being operated by Slock.it.
  - Lack of adoption is a threat to the system. The more diverse active signing nodes the incubed network is comprised of, the more secure it will be.
  - The chance to get part of the deposit of dishonest nodes is what incentivizes nodes to offer their services in the network. However, in a perfect world all nodes are honest and thus there is no incentive to participate and provide services to the network.
  - The number of nodes in the network is not a metric for the security of it. Multiple nodes may just relay to `infura` or even the same back-end node. Nodes might be dysfunctional, not updated, or in any other way not capable of providing signed blockhashes.

Smart Contracts

- Slock.it `unregisterKey`:

  - Secure key management is paramount and should be audited.
  - Only works until one-year after deployment. However, Slock.it can redeploy and release new software that allows them to extend this period.
  - Allows Slock.it to remove nodes from the registry. The deposit can be withdrawn by the original owner after the timeout that was specified upon registration. However, nodes that are removed from the registry can be re-added, therefore it is questionable whether the existence of this admin functionality is contributing to the security of the system, especially as this functionality will not be available after one year of deployment.
  - There is no functionality for the admin team to renounce admin access via the `unregisterKey` before one year in case this functionality is not needed anymore.

- Registry addresses are hard-coded in node and client software and Slock.it are in control of them passively via software updates. Centralization.

  - The registry can be overriden in the node configuration file or via command-line.

- Minimum deposit of 10 finney is hard-coded. It might become expensive or really cheap to DoS the registry, depending on ETH price in future.

- Commit-reveal scheme could be avoided.

  - One suggested method is to include a salt (or requester's identifier) in each response, to make sure the signed blockhash is only valid for the requester. This results in the signed blockhash to be only valid for the receiver, hence only receiver is able to convict using the blockhash they hold. The proper checks also should be implemented in the smart contract conviction flow.

- Incentivisation to convict someone highly depends on the reward (node.deposit/2).

  - Nodes might detect that someone is signing wrong block-hashes but will not act because it is not profitable for them (`handleRecreation()`).
  - Nodes might not have enough funds to actually convict a misbehaving node.
  - Nodes will spend gas on certain calls (`convict`, `revealConvict`, `recreateBlockheaders`, `registerNode`) decreasing their balance over time.
  - Nodes can currently only earn tokens by convicting nodes.
  - Incentivisation to not convict yourself highly depends on the deposit/2 the malicious node burns
  - A malicious node might still choose to convict themselves if they detect that they are to be convicted soon to at least recover half of the deposit. This impacts the original convictor that will not get any deposit from the conviction process.
  - Adding new blocks might consume a lot of gas, more than deposit/2 - disincentive to convict a node
  - Any node can observe that someone is about to convict a node by monitoring events, the internal convict mapping in storage or calls to commit/reveal or recreate blockhashes.

- Having too many blocks in the system might be problematic if storage rents is implemented in Layer 1 (Storage Rent)

- Specification mentions that nodes should be picked at random for security. However, nodes can define a weight in *In3Node* when registering which might create a bias (malicious nodes might draw more requests by suggesting they have high bandwidth).

in3-node layer

- *in3-node* proxy requests to Ethereum blockchain nodes.
  - Blacklist implementation is not considered safe. It is preferred to implement whitelists for allowed RPC calls.

- As an example `w3.parity.personal.newAccount("password")` is able to create new accounts on `https://in3.slock.it/kovan/nd-2`
    - Blacklist may quickly become outdated when nodes are updated. New functionality and calls may be exposed before the incubed node is updated. Node operators may even forget to update incubed nodes with updated blacklists.
    - Trust assumption between `<blockchain_node>::localhost` can be different with `<blockchain_node>::public` (*in3-node* might break these assumptions). Some nodes might expose special functionality for API listening on localhost.
    - The blockchain node's log will only show requests from incubed nodes. It is important to be able to reconstruct an audit trail in case of security events.
    - Slock.it should maintain conformance with Ethereum RPC protocol changes.
- *in3-node* retrieve RPC urls of other *in3-node*'s from the NodeRegistry. However as there are no sanity checks on these URLs before registration, they can:
    - Be invalid
    - Be used to DDoS other sites/servers/nodes
    - Be re-used to draw more traffic to already existing nodes
        - Even though signer would be wrong. Currently there is no method to convict the wrong node or remove it (unless admin key removes them within the first year)
        - Also each node can have multiple accounts, which all have the ability to sign. This can result in multiple node registration which connect to one in3-node.
- Client force *in3-node*'s to do more work that the requester (signing, asking other nodes, asking geth/parity). This could result in a DoS attack.
- For performance reasons a new binary protocol for JSON-RPC communication is invented.
    - We would like to note that it is highly discouraged to invent a new TLV-based format for this purpose.
    - Decoding of such values is not trivial and dramatically increases the attack surface for attackers. Especially for nodes that may be implemented in non memory-managed languages such as `c`.
    - It is recommended to fall back to an established standard and re-evaluate if a binary protocol is really needed.
- It is not obvious how *in3-node*'s can handle forks.
    - Especially upgrade-forks like the upcoming Istanbul fork on Ethereum puts the system at risk. Depending on when and how the backend blockchain node is updated the incubed node may hand out signatures for wrong hashes. This will cause the loss of funds if conviction is profitable for other nodes.

- Forks may have a stalling effect on the incubed network. Nodes might just refuse to sign during a fork until chains are settled.
  - Forks may result in two valid chains (ETC/ETH).
- Security Best Practices for Incubed nodes configuration, deployment, maintenance, development and operations should be visibly linked to the website.
- Nodes may learn who the watchdogs are and avoid providing proofs to them.
- Nodes might not provide a proof or signature even though the client requested it.
- The network is not protected from 51% attacks. If network is comprised of 51% nodes that all resolve to same RPC and dishonest node, network can be misused.
- Malicious node may avoid giving out proofs to any other node that is in the registry. only providing proofs to clients.
- *in3-node* must check NodeRegistry for signer key for any RPC url.
- It's not possible after the first year to get bogus, erroneous, DoS-ing registered nodes off the network.
  - The unregister account is reactive and cannot prevent malicious nodes from re-registering with the registry.
- in3-clients and nodes can force other nodes to reach out to specific potentially malicious nodes in an attempt to exploit a security vulnerability. The general issue here is that nodes can instruct other nodes to request certain information from other nodes in the system. That other node might then respond with a malicious payload e.g. to exploit a buffer overflow in a C implementation of the node. It might also just keep the tcp connection open in an attempt to exhaust the callers resources (file descriptors)
  - A node may request another node to even reach out to itself (request is low effort but building the response is a lot of effort and will even use more resources because the node attempts to connect to itself)
- Nodes may decide to lie only for blockhashes that are expensive to add to the registry because they will not be convicted for them.
- The software will try to auto-convict a node. However, the node does not monitor the tx-pool and does not know that someone else already convicts the node. This will lead to a race where multiple nodes spend gas on `recreateBlockheaders` and convict (ideally all nodes that observed the wrong blockheader; this can be controlled by an attacker). Only one node will receive the deposits. Others may lose up to what is barely profitable for them in the race.
- Nodes might be booted of the network by exploiting error conditions that result in the node software to fail with an error condition (exceptions). We have not investigated such cases and want to note that it is important that a node does not exit with an error condition as a result of another client/node's request or response. This would

allow someone to control the amount of nodes on the network, even reduce it and perform attacks on the reduced number of nodes.

- Local blockchain nodes can be out of sync or stall and therefore available in the registry and network stats, but actually useless as they are not able to provide proofs. The number of nodes online might therefore not be a correct metric for the security of the network.

6 Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 in3-server - amplified DDoS on incubed requests on proof with signature Critical ✓ Fixed

Resolution
Mitigated by adding `maxBlocksSigned` and `maxSignatures` for requests of any client. "The Numbers of signatures a client can ask to fetch is now limited to maxSignatures which defaults to 5" in merge_requests/101. The full extent of this fix is outside the scope of this audit.

Description

It is possible for a client to send a request to each node of the network to request a signature with proof for every other node in the network. This can result in DDoSing the network as there are no costs for the client to request this and client can send the same request to all the nodes in the network, resulting in `n^2` requests.

Examples

1. Client asks each node for `in3_nodeList` to get all the signer addresses, this could also be done using `NodeRegistry` contract

2. Client asks each node for a proof with signature, e.g.:

```
{
    "jsonrpc": "2.0",
    "id": 2,
    "method": "eth_getTransactionByHash",
    "params": ["0xf84cfb78971ebd940d7e4375b077244e93db2c3f88443bb93c561812cfed055c"],
    "in3": {
        "chainId": "0x1",
        "verification": "proofWithSignature",
        "signatures":["0x784bfa9eb182C3a02DbeB5285e3dBa92d717E07a", ALL OTHER SIGNERS
    }
}
```

All the nodes are now sending requests to each other with signature required which is an expensive computation. This can go on for more transactions (or blocks, or other Eth_ requests) and can result in DDoS of the network.

Recommendation

Limit the number of signers in proof with signature requests. Also exclude self.signer from the list. This combined with the remediation of issue 6.6 can partially mitigate the attack vector.

6.2 BlockProof - Node conviction race condition may trick all but one node into losing funds Critical ✓ Fixed

Resolution

Mitigated by:

- 80bb6ecf by checking if blockhash exists and prevent an overwrite, saving gas
- Client will blacklist the server if the signature is missing, has a wrong signer or is invalid.
- 6cc0dbc0 Removing nodes from local available nodes list when the server detects wrong responses

- Other commits to mitigate the mentioned vulnerable scenarios

  With the new handling, the client will not call convict immediately (as this could be exploited again). Instead, the client will do the calculation whether it's worth convicting the server before even calling convict.

  It should be noted that the changes are scattered and modified in the final source code, and this behaviour of IN3-server code is outside the scope of this audit.

Description

TLDR; One node can force all other nodes to convict a specific malicious signer controlled by the attacker and spend gas on something they are not going to be rewarded for. The attacker loses `deposit` but all other nodes that try to convict and recreate in the same block will lose the fees less or equal to `deposit/2` . Another variant forces the same node to recreate the blockheaders multiple times within the same block as the node does not check if it is already convicting/recreating blockheaders.

Nodes can request various types of proofs from other nodes. For example, if a node requests a proof when calling one of the `eth_getBlock*` methods, the `in3-server` 's method `handleBlock` will be called. The request should contain a list of addresses registered to the NodeRegistry that are requested to sign the blockhash.

code/in3-server/src/modules/eth/EthHandler.ts:L105-L112

```
// handle special jspn-rpc
if (request.in3.verification.startsWith('proof'))
  switch (request.method) {
    case 'eth_getBlockByNumber':
    case 'eth_getBlockByHash':
    case 'eth_getBlockTransactionCountByHash':
    case 'eth_getBlockTransactionCountByNumber':
      return handleBlock(this, request)
```

`in3-server` will subsequently reach out to it's connected blockchain node execute the `eth_getBlock*` call to get the block data. If the block data is available the `in3-server` , it will try to collect signatures from the nodes that signature was requested from (`request.in3.signatures` , `collectSignatures()`)

code/in3-server/src/modules/eth/proof.ts:L237-L243

```
// create the proof
response.in3 = {
  proof: {
    type: 'blockProof',
    signatures: await collectSignatures(handler, request.in3.signatures, [{ blockNumb
  }
}
```

If the node does not find the address it will throw an exception. Note that if this exception is not caught it will actually allow someone to boot nodes off the network - which is critical.

code/in3-server/src/chains/signatures.ts:L58-L60

```
const config = nodes.nodes.find(_ => _.address.toLowerCase() === adr.toLowerCase())
if (!config) // TODO do we need to throw here or is it ok to simply not deliver the si
  throw new Error('The ' + adr + ' does not exist within the current registered activ
```

If the address is valid and existent in the `NodeRegistry` the `in3-node` will ask the node to sign the blockhash of the requested blocknumber:

code/in3-server/src/chains/signatures.ts:L69-L84

```
// send the sign-request
let response: RPCResponse
try {
  response = (blocksToRequest.length
    ? await handler.transport.handle(config.url, { id: handler.counter++ || 1, jsonrp
    : { result: [] }) as RPCResponse
  if (response.error) {
    //throw new Error('Could not get the signature from ' + adr + ' for blocks ' + bloc
    logger.error('Could not get the signature from ' + adr + ' for blocks ' + blocks.
    return null
  }
} catch (error) {
  logger.error(error.toString())
  return null
}
```

For all the signed blockhashes that have been returned the `in3-server` will subsequently check if one of the nodes provided a wrong blockhash.

We note that nodes might:

- decided to not follow the `in_3sign` request and just not provide a signed response
- a node might sign with a different key
- a node might sign a different blockheader
- a node might sign a previous blocknumber

In all these cases, the node will not be convicted, even though it was able to request other nodes to perform work.

If another node signed a wrong blockhash the `in3-server` will automatically try to convict it. If the block is within the most recent 255 it will directly call `convict()` on the NodeRegistry (takes less gas). if it is an older block, it will try to recreate the blockchain in the `RlockhashRegistry` (takes more gas).

code/in3-server/src/chains/signatures.ts:L128-L152

```
const convictSignature: Buffer = keccak(Buffer.concat([bytes32(s.blockHash), address(

if (diffBlocks < 255) {

  await callContract(handler.config.rpcUrl, nodes.contract, 'convict(uint,bytes32)',
    privateKey: handler.config.privateKey,
    gas: 500000,
    value: 0,
    confirm: true                      //  we are not waiting for confirmation, since
  })

  handler.watcher.futureConvicts.push({
    convictBlockNumber: latestBlockNumber,
    signer: singingNode.address,
    wrongBlockHash: s.blockHash,
    wrongBlockNumber: s.block,
    v: s.v,
    r: s.r,
    s: s.s,
    recreationDone: true
  })
}
else {
  await handleRecreation(handler, nodes, singingNode, s, diffBlocks)
}
```

The recreation and convict is only done if it is profitable for the node. (Note the issue mentioned in issue 6.13)

code/in3-server/src/chains/signatures.ts:L209-L213

```
const costPerBlock = 86412400000000
const blocksMissing = latestSS - s.block
const costs = blocksMissing * costPerBlock * 1.25

if (costs > (deposit / 2)) {
```

A malicious node can exploit the hardcoded profit economics and the fact that `in3-server` implementation will try to auto-convict nodes in the following scenario:

- malicious node requests a blockproof with an `eth_getBlock*` call from the victim node (`in3-server`) for a block that is not in the most recent 256 blocks (to maximize effort for the node). This equals to spending more gas in order to convict the node (`costs <= (deposit / 2)`).
- the malicious node prepares the `BlockhashRegistry` to contain a blockhash that would maximize the gas needed to convict the malicious node (can be calculated offline; must fulfill `costs <= (deposit /2)` .
- with the blockproof request the malicious node asks the `in3-server` to get the signature from a specific signer. The signer will also be malicious and is going to sign a wrong blockhash with a valid signature.
- the malicious signer is going to lose it's deposit but the deposit also incentivizes other nodes to spend gas on the conviction process. The higher the deposit, the more an `in3-server` is willing to spend on the conviction.

In this scenario one malicious node tries to trick another node into convicting a malicious signer while having to spend the maximum amount of gas to make it profitable for the node.

The problem is, that the malicious node can ask multiple (or even all other nodes in the registry) to provide a blockproof and ask the malicious signer for a signed blockhash. All nodes will come to the conclusion that the signer returned an invalid hash and will try to convict the node. They will try to recreate the blockchain in the `BlockhashRegistry` for a barely profitable scenario. Since `in3-nodes` do not monitor the tx-pool they will not know that other nodes are already trying to convict the node. All nodes are going to spend gas on recreating the same blockchain in the `BlockhashRegistry` leading to all but the first transaction in the block to lose funds (up to `deposit/2` based on the hardcoded `costPerBlock`)

Another variant of the same issue is that nodes do not check if they already convicted another node (or recreated blockheaders). An attacker can therefore force a specific node

to convict a malicious node multiple times before the nodes transactions are actually in a block as the nodes does not check if it is already convicting that node. The node might lose gas on the recreation/conviction process multiple times.

Recommendation

To reduce the impact of multiple nodes trying to update the blockhashRegistry at the same time and avoid nodes losing gas by recreating the same blocks over and over again, the `BlockhashRegistry` should **require** that the target blockhash for the blocknumber does not yet exist in the registry (similar to the issue mentioned in https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/24).

6.3 `NodeRegistry` Front-running attack on `convict()` `Critical`

`✓ Fixed`

Resolution

Blocknumber is removed from `convict` function, which removes any signal for an attacker in the scenario provided. However, the order of the transactions to convict a wrong signed hash is necessary to prevent any front-running attacks:

1. Convict(_Blockhash)
2. recreate Blockheaders
3. RevealConvict (minimum 2 blocks after `convict` but as soon as recreateBlockheaders is confirmed)

The fixes were introduced in ecf2c6a6 and f4250c9a, although later on NodeRegistry contract was split in two other contracts `NodeRegistryLogic` and `NodeRegistryData` and further changes were done in the conviction flow in different commits.

Description

`convict(uint _blockNumber, bytes32 _hash)` and `revealConvict()` are designed to prevent front-running and they do so for the purpose they are designed for. However, if the malicious node, is still sending out the wrong blockhash for the convicted block, anyone seeing the initial convict transaction, can check the convicted blocknumber with the nodes and send his own `revealConvict` before the original sender.

The original sender will be the one updating the block headers
`recreateBlockheaders(_blockNumber, _blockheaders)` , and the attacker can just watch for the update headers to perform this attack.

Recommendation

For the first attack vector, remove the blocknumber from the
`convict(uint _blockNumber, bytes32 _hash)` inputs and just use the hash.

6.4 NodeRegistry - URL can be arbitrary dns resolvable names, IP's and even localhost or private subnets Major ✓ Fixed

Resolution

This issue has been addressed with the following commits:

- 4c93a10f adding 48 hours delay in the server code before they communicate with the newly registered nodes.
- merge_requests/111 adding a whole new smart contract to the IN3 system, `IN3WhiteList.sol` , and supporting code in the server.
- issues/94 To prevent attacker to use nodes as a DoS network, a DNS record verification is discussed to be implemented.

It is a design decision to base the Node registry on URLs (DNS resolvable names). This has the implications outlined in this issue and they cannot easily be mitigated. Adding a delay until nodes can be used after registration only delays the problem. Assuming that an entity curates the registry or a whitelist is in place centralizes the system. Adding DNS record verification still allows an owner of a DNS entry to point its name to any IP address they would like it to point to. It certainly makes it harder to add RPC URLs with DNS names that are not in control of the attacker but it also adds a whole lot more complexity to the system (including manual steps performed by the node operator). In the end, the system allows IP based URLs in the registry which cannot be used for DNS validation.

Note that the server code changes, and the new smart contract `IN3WhiteList.sol` are outside the scope of the original audit. We strongly recommend to reduce complexity and audit the final codebase before mainnet deployment.

Description

As outlined in issue 6.9 the `NodeRegistry` allows anyone to register nodes with arbitrary URLs. The `url` is then used by `in3-server` or clients to connect to other nodes in the system. Signers can only be convicted if they sign wrong blockhashes. However, if they never provide any signatures they can stay in the registry for as long as they want and sabotage the network. The Registry implements an admin functionality that is available for the first year to remove misbehaving nodes (or spam entries) from the Registry. However, this is insufficient as an attacker might just re-register nodes after the minimum timeout they specify or spend some more finneys on registering more nodes. Depending on the eth-price this will be more or less profitable.

From an attackers perspective the `NodeRegistry` is a good source of information for reconnaissance, allows to de-anonymize and profile nodes based on dns entries or netblocks or responses to `in3_stats` (https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/49), makes a good list of target for DoS attacks on the system or makes it easy to exploit nodes for certain yet unknown security vulnerabilities.

Since nodes and potentially clients (not in scope) do not validate the rpc URL received from the `NodeRegistry` they will try to connect to whatever is stored in a nodes `url` entry.

code/in3-server/src/chains/signatures.ts:L58-L75

```
const config = nodes.nodes.find(_ => _.address.toLowerCase() === adr.toLowerCase())
if (!config) // TODO do we need to throw here or is it ok to simply not deliver the sig
  throw new Error('The ' + adr + ' does not exist within the current registered activ

// get cache signatures and remaining blocks that have no signatures
const cachedSignatures: Signature[] = []
const blocksToRequest = blocks.filter(b => {
  const s = signatureCaches.get(b.hash) && false
  return s ? cachedSignatures.push(s) * 0 : true
})

// send the sign-request
let response: RPCResponse
try {
  response = (blocksToRequest.length
    ? await handler.transport.handle(config.url, { id: handler.counter++ || 1, jsonrp
    : { result: [] }) as RPCResponse
  if (response.error) {
```

This allows for a wide range of attacks not limited to:

- An attacker might register a node with an empty or invalid URL. The `in3-server` does not validate the URL and therefore will attempt to connect to the invalid URL, spending resources (cpu, file-descriptors, ..) to find out that it is invalid.
- An attacker might register a node with a URL that is pointing to another node's rpc endpoint and specify weights that suggest that it is capable of service a lot of requests to draw more traffic towards that node in an attempt to cause a DoS situation.
- An attacker might register a node for a http/https website at any port in an extortion attempt directed to website owners. The incubed network nodes will have to learn themselves that the URL is invalid and they will at least attempt to connect the website once.
- An attacker might update the node information in the `NodeRegistry` for a specific node every block, providing a new `url` (or a slightly different URLs issue 6.9) to avoid client/node URL blacklists.
- An attacker might provide IP addresses instead of DNS resolvable names with the `url` in an attempt to draw traffic to targets, avoiding canonicalization and blacklisting features.
- An attacker might provide a URL that points to private IP netblocks for IPv4 or IPv6 in various formats. Combined with the ability to ask another node to connect to an attacker defined `url` (via blockproof, `signatures[] -> signer_address -> signer.url`) this might allow an attacker to enumerate services in the LAN of node operators.
- An attacker might provide the loopback IPv4, IPv6 or resolvable name as the URL in an attempt to make the node connect to local loopback services (service discovery, bypassing authentication for some local running services - however this is very limited to the requests nodes may execute).
- URLs may be provided in various formats: resolvable dns names, IPv4, IPv6 and depending on the http handler implementation even in Decimal, Hex or Octal form (i.e. http://2130706433/)
- A valid DNS resolvable name might point to a localhost or private IP netblock.

Since none of the rpc endpoints provide signatures they cannot be convicted or removed (unless the `unregisterKey` does it within the first year. However, that will not solve the problem that someone can re-register the same URLs over and over again)

Recommendation

It is a fundamental design decision of the system architecture to allow rpc urls in the Node Registry, therefore this issue can only be partially mitigated unless the system design is reworked. It is therefore suggested to add checks to both the registry contract (coarse

validation to avoid adding invalid urls) and node implementations (rigorous validation of URL's and resolved IP addresses) and filter out any potentially harmful destinations.

6.5 Malicious clients can use forks or reorgs to convict honest nodes Major Won't Fix

Resolution

Default value for past signed blocks is changed to 10 blocks. Slockit plans to use their off-chain channels to notify clients for planned forks. They also looking into using fork oracles in the future releases to detect planned hardforks to mitigate risks.

Description

In case of reorgs it is possible to have more than 6 blocks in a node that gets replaced by a new longer chain. Also for forks, such as upcoming Istanbul fork, it's common to have some nodes taking some time to update and they will be in the wrong chain for the time being. In both cases, in3-nodes are prone to sign blocks that are considered invalid in the main chain. Malicious nodes can catch these instances and convict the honest users in the main chain to get 50% of their deposits.

Recommendation

No perfect solution comes to mind at this time. One possible mitigation method for forks could be to disable the network on the time of the fork but this is most certainly going to be a threat to the system itself.

6.6 in3-server - should protect itself from abusive clients Major
✓ Fixed

Resolution

Slockit implemented their own DOS protection for incubed server in merge_requests/99. The variant of this implementation adds more complexity to the code base. The benchmark and testing of the new DOS protection is not in scope for this audit.

> The incubed server has now an additional DOS-Protection build in. Here we first estimate a Weight of such a request and add them together for all incoming requests per IP of the client per Minute. Since we estimate the execution, we can prevent a client running DOS-Attacks from the same IP with heavy requests (such as eth_getLogs)

Description

The in3-node implementation should provide features for client request throttling to avoid that a client can consume most of the nodes resources by causing a lot of resource intensive requests.

This is a general problem to the system which is designed to make sure that low resource clients can verify blockchain properties. What this means is that almost all of the client requests are very lightweight. Clients can request nodes to sign data for them. A sign request involves cryptographic operations and a http-rpc request to a back-end blockchain node. The imbalance is clearly visible in the case of blockProofs where a client may request another node to interact with a smart contract (NodeRegistry) and ask other nodes to sign blockhashes. All other nodes will have to get the requested block data from their local blockchain nodes and the incubed node requesting the signatures will have to wait for all responses. The client instead only has to send out that request once and may just leave that tcp connection open. It might even consume more resources from a specific node by requesting the same signatures again and again not even waiting for a response but causing a lot of work on the node that has to collect all the signatures. This combined with unbound requests for signatures or other properties can easily be exploited by a powerful client implementation with a mission to stall the whole incubed network.

Recommendation

According to the threat model outlines a general DDoS scenario specific to rpcUrls. It discusses that the nodes are themselves responsible for DDoS protection. However, DDoS protection is a multi-layer approach and it is highly unlikely that every node-operator will hide their nodes behind a DDoS CDN like cloudflare. We therefore suggest to also build in strict limitations for clients that can be checked in code. Similar to `checkPerformanceLimits` which is just checking for some specific it is suggested to implement a multi-layer throttling mechanism that prevents nodes from being abused by single clients. Methods must be designed with (D)DoS scenarios in mind to avoid that third parties are abusing the network for DDoS campaigns or trying to DoS the incubed network.

code/in3-server/src/modules/eth/EthHandler.ts:L74-L91

```
private checkPerformanceLimits(request: RPCRequest) {
  if (request.method === 'eth_call') {
    if (!request.params || request.params.length < 2) throw new Error('eth_call must
    const tx = request.params as TxRequest
    if (!tx || (tx.gas && toNumber(tx.gas) > 10000000)) throw new Error('eth_call wit
  }
  else if (request.method === 'eth_getLogs') {
    if (!request.params || request.params.length < 1) throw new Error('eth_getLogs mu
    const filter: LogFilter = request.params[0]
    let toB = filter && filter.toBlock
    if (toB === 'latest' || toB === 'pending' || !toB) toB = this.watcher && this.wat
    let fromB = toB && filter && filter.fromBlock
    if (fromB === 'earliest') fromB = 1;
    const range = fromB && (toNumber(toB) - toNumber(fromB))
    if (range > (request.in3.verification.startsWith('proof') ? 1000 : 10000))
      throw new Error('eth_getLogs for a range of ' + range + ' blocks is not allowed
  }
}
```

- implement request throttling per client
- implement caching mechanism for similar requests if it is expected that the same response is to be delivered multiple times
- implement general performance limits and reject further requests if the node is close to exhausting its resources (soft DoS)
- make sure the node does not exhaust the systems resources
- implement throttling per request method
- design methods to prevent (D)DoS in the first place. Methods that allow a client to send one request that causes a node to perform multiple client controlled requests must be avoided or at least bound and throttled (issue 6.7, https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/50).

6.7 in3-server - DoS on `in3.sign` and other requests `Major` `✓ Fixed`

> ### Resolution
>
> Similar to issue 6.1, Mitigated by adding `maxBlocksSigned` and `maxSignatures` for requests of any client. "The Numbers of signatures a client can ask to fetch is now

limited to maxSignatures which defaults to 5" in merge_requests/101. The full extent of this fix is outside the scope of this audit.

> We have limited the number of block you can ask to sign in the in3_sign-request. The default is 10, because this function is also used for eth_getLogs to provide proof for all events. This limit will also limit the result of logs returned to include only max 10 different blocks.

Description

It is free for the client to ask the nodes to sign block hashes (and also other requests). `in3.sign([{"blockNumber": 123}])` Takes an array of objects that will result in multiple requests in the node. This sample request has (at least) two internal requests, one `eth_getBlockByNumber` and signing the block hash.

These requests can be continuously sent out to clients and result in using computation power of the nodes without any expense from the client.

Examples

Request to get and sign the first 200 blocks:

```
web3.manager.request_blocking("in3_sign", [{'blockNumber':i} for i in range(200)])
```

Recommendation

Limit the number of blocks (input), or do not accept arrays for input.

6.8 in3-server - key management Major Pending

Resolution

The breakdown of the fixes addressed with git.slock.it/PR/13 are as follows:

- Keys should never be stored or accepted in plaintext format Keys should only be accepted in an encrypted and protected format

The private key in `code/in3-server/config.json` has been removed. The repository still contains private keys at least in the following locations:

- `package.json`
- `vscode/launch.json`
- `example_docker-compose.yml`

Note that private keys indexed by a git repository can be restored from the repository history.

The following statement has been provided to address this issue:

> We have removed all examples and usage of plain private keys and replaced them with json-keystore files. Also in the documentation we added warnings on how to deal with keys, especially with hints to the bash history or enviroment

---

- A single key should be used for only one purpose. Keys should not be shared.

The following statement has been provided to address this issue:

> This is why we seperated the owner and signer-key. This way you can use a multisig to securly protect the owner-key. The signer-key is used to sign blocks (and convict) and is not able to do anything else (not even changing its own url)

- The application should support developers in understanding where cryptographic keys are stored within the application as well as in which memory regions they might be accessible for other applications

Addressed by wrapping the private key in an object that stores the key in encrypted form and only decrypts it when signing. The key is cleared after usage. The IN3-server still allows raw private keys to be configured. A warning is printed if that is the case. The loaded raw private key is temporarily assigned to a local variable and not explicitly cleared by the method.

> While we used to keep the unlocked key as part of the config, we have now removed the key from the config and store them in a special signer-function.
> https://git.slock.it/in3/ts/in3-server/merge_requests/113

- Keys should be protected in memory and only decrypted for the duration of time they are actively used. Keys should not be stored with the applications source-code repository

see previous remediation note.

> After unlocking the signer key, we encrypt it again and keep it encrypted only decrypting it when signing. This way the raw private key only exist for a very short time in memory and will be filled with 0 right after. (https://git.slock.it/in3/ts/in3-server/merge_requests/113/diffs#653b04fa41e35b55181776b9f14620b661cff64c_54_73)

- Use standard libraries for cryptographic operations

The following statement has been provided to address this issue

> We are using ethereumjs-libs.

- Use the system keystore and API to sign and avoid to store key material at all

The following statement has been provided to address this issue

> We are looking into using different signer-apis, even supporting hardware-modules like HSMs. But this may happen in future releases.

- The application should store the keys eth-address (util.getAddress()) instead of re-calculating it multiple times from the private key.

Fixed by generating the address for a private key once and storing it in a private key wrapper object.

- Do not leak credentials and key material in debug-mode, to local log-output or external log aggregators.

`txArgs` still contains a field `privateKey` as outlined in the issue description. However, this `privateKey` now represents the wrapper object noted in a previous comment which only provides access to the ETH address generated from the raw private key.

> The following statement has been provided to address this issue:
>
>> since the private key and the passphrase are actually deleted from the
>> config, logoutputs or even debug will not be able to leak this information.

Description

Secure and efficient key management is a challenge for any cryptographic system. Incubed nodes for example require an account on the ethereum blockchain to actively participate in the incubed network. The account and therefore a private-key is used to sign transactions on the ethereum blockchain and to provide signed proofs to other in3-nodes.

This means that an attacker that is able to discover the keys used by an `in3-server` by any mechanism may be able to impersonate that node, steal the nodes funds or sign wrong data on behalf of the node which might also lead to a loss of funds.

The private key for the `in3-server` can be specified in a configuration file called `config.json` residing in the program working dir. Settings from the `config.json` can be overridden via command-line options. The application keeps configuration parameters available internally in an `IN3RPCConfig` object and passes this object as an initialization parameter to other objects.

The key can either be provided in plaintext as a hex-string starting with `0x` or within an ethereum keystore format compatible protected keystore file. Either way it is provided it will be held in plaintext in the object.

The application accepts plaintext private keys and the keys are stored unprotected in the applications memory in JavaScript objects. The `in3-server` might even re-use the nodes private key which may weaken the security provided by the node. The repository leaks a series of presumably 'test private keys' and the default config file already comes with a private key set that might be shared across unvary users that fail to override it.

code/in3-server/config.json:L1-L4

```
{
  "privateKey": "0xc858a0f49ce12df65031ba0eb0b353abc74f93f8ccd43df9682fd2e2293a4db3",
  "rpcUrl": "http://rpc-kovan.slock.it"
}
```

code/in3-server/package.json:L20-L31

```
    "docker-run": "docker run -p 8500:8500 docker.slock.it/slockit/in3-server:latest --pr
    "docker-setup": "docker run -p 8500:8500 slockit/in3-server:latest --privateKey=0x385
    "local": "export NODE_ENV=0 && npm run build && node ./js/src/server/server.js --priv
    "ipfs": "docker run -d -p 5001:5001 jbenet/go-ipfs  daemon --offline",
    "linkIn3": "cd node_modules; rm -rf in3; ln -s ../../in3 in3; cd ..",
    "lint:solium": "node node_modules/ethlint/bin/solium.js -d contracts/",
    "lint:solium:fix": "node node_modules/ethlint/bin/solium.js -d contracts/ --fix",
    "lint:solhint": "node node_modules/solhint/solhint.js \"contracts/**/*.sol\" -w 0",
    "local-env": "export NODE_ENV=0 && npm run build && node ./js/src/server/server.js --
    "local-env2": "export NODE_ENV=0 && npm run build && node ./js/src/server/server.js -
    "local-env3": "export NODE_ENV=0 && npm run build && node ./js/src/server/server.js -
    "local-env4": "export NODE_ENV=0 && npm run build && node ./js/src/server/server.js -
```

The private key is also passed as arguments to other functions. In error cases these may
leak the private key to log interfaces or remote log aggregation instances (sentry). See
`txargs.privateKey` in the example below:

code/in3-server/src/util/tx.ts:L100-L100

```
const key = toBuffer(txargs.privateKey)
```

code/in3-server/src/util/tx.ts:L134-L140

```
const txHash = await transport.handle(url, {
  jsonrpc: '2.0',
  id: idCount++,
  method: 'eth_sendRawTransaction',
  params: [toHex(tx.serialize())]
}).then((_: RPCResponse) => _.error ? Promise.reject(new SentryError('Error sending t
```

Recommendation

- Keys should never be stored or accepted in plaintext format.
  - Keys should not be stored in plaintext on the file-system as they might easily be
    exposed to other users. Credentials on the file-system must be tightly restricted
    by access control.
  - Keys should not be provided as plaintext via environment variables as this might
    make them available to other processes sharing the same environment (child-
    processes, e.g. same shell session)
  - Keys should not be provided as plaintext via command-line arguments as they
    might persist in the shell's command history or might be available to privileged

system accounts that can query other processes startup parameters.

- Keys should only be accepted in an encrypted and protected format.
- A single key should be used for only one purpose. Keys should not be shared.
  - The use of the same key for two different cryptographic processes may weaken the security provided by one or both of the processes.
  - The use of the same key for two different applications may weaken the security provided by one or both of the applications.
  - Limiting the use of a key limits the damage that could be done if the key is compromised.
  - Node owners keys should not be re-used as signer keys.
- The application should support developers in understanding where cryptographic keys are stored within the application as well as in which memory regions they might be accessible for other applications.
- Keys should be protected in memory and only decrypted for the duration of time they are actively used.
- Keys should not be stored with the applications source-code repository.
- Use standard libraries for cryptographic operations.
- Use the system keystore and API to sign and avoid to store key material at all.
- The application should store the keys eth-address (`util.getAddress()`) instead of re-calculating it multiple times from the private key.
- Do not leak credentials and key material in debug-mode, to local log-output or external log aggregators.

6.9 NodeRegistry - Multiple `nodes` can share slightly different RPC URL `Major` `✓ Fixed`

Resolution
Same mitigation as issue 6.4.

Description

One of the requirements for Node registration is to have a unique URL which is not already used by a different owner. The uniqueness check is done by hashing the provided `_url` and checking if someone already registered with that hash of `_url` .

However, byte-equality checks (via hashing in this case) to enforce uniqueness will not work for URLs. For example, while the following URLs are not equal and will result in different `urlHashes` they can logically be the same end-point:

- `https://some-server.com/in3-rpc`
- `https://some-server.com:443/in3-rpc`
- `https://some-server.com/in3-rpc/`
- `https://some-server.com/in3-rpc///`
- `https://some-server.com/in3-rpc?something`
- `https://some-server.com/in3-rpc?something&something`
- `https://www.some-server.com/in3-rpc?something` (if www resolves to the same ip)

code/in3-contracts/contracts/NodeRegistry.sol:L547-L553

```solidity
bytes32 urlHash = keccak256(bytes(_url));

// make sure this url and also this owner was not registered before.
// solium-disable-next-line
require(!urlIndex[urlHash].used && signerIndex[_signer].stage == Stages.NotInUse,
    "a node with the same url or signer is already registered");
```

This leads to the following attack vectors:

- A user signs up multiple nodes that resolve to the same end-point (URL). A minimum deposit of `0.01 ether` is required for each registration. Registering multiple nodes for the same end-point might allow an attacker to increase their chance of being picked to provide proofs. Registering multiple nodes requires unique signer addresses per node.

- Also one node can have multiple accounts, hence one node can have slightly different URL and different accounts as the `signer` s.

- DoS - A user might register nodes for URLs that do not serve in3-clients in an attempt to DDoS e.g. in an attempt to extort web-site operators. This is kind of a reflection attack where nodes will request other nodes from the contract and try to contact them over RPC. Since it is http-rpc it will consume resources on the receiving end.

- DoS - A user might register Nodes with RPC URLs of other nodes, manipulating weights to cause more traffic than the node can actually handle. Nodes will try to communicate with that node. If no proof is requested the node will not even know that someone else signed up other nodes with their RPC URL to cause problems. If they

request proof the original signer will return a signed proof and the node will fail due to a signature mismatch. However, *the node cannot be convicted* and therefore forced to lose the deposit as conviction is bound the signer and the block was not signed by the rogue node entry. There will be no way to remove the node from the registry other than the admin functionality.

Recommendation

Canonicalize URLs, but that will not completely prevent someone from registering nodes for other end-points or websites. Nodes can be removed by an admin in the first year but not after that. Rogue owners cannot be prevented from registering random nodes with high weights and minimum deposit. They cannot be convicted as they do not serve proofs. Rogue owners can still unregister to receive their deposit after messing with the system.

6.10 in3-server - should enforce safe settings for minBlockHeight

`Medium` `Won't Fix`

Resolution

The default block is changed to 10 and `minBlockHeight` is added to the registry (as part of the properties) in 8c72633e, but allow the user to define a `minBlockHeight` lower than this number. The client is responsible to review the settings depending on how secure they want their nodes to be.

Client response:

We have discussed this, but decided to keep it flexible. This means:

1. We have put the minBlockHeight into the registry (as part of the properties). Because these properties indicate the limit and capabilities of the node and give the client a chance to filter out nodes if they don't match the requirements. So each client is able to filter out node who are not willing to take the risk and sign for example latest-6. Of course these nodes will most likely only store a low deposit (you can not have a signature of a young block and a high deposit), but if you need a high security the nodes with a deposit will propably wait at least 10 or more blocks. In order to protect the owner of a node of using insecure settings, we will use our wizard to check the deposit and minBlockHeights and warn or educate the user. The reason why this

flexibility is important, is because there use cases where dapps will not accept the let user wait 10 blocks before confirming a transaction. If the dapp developer needs a signature of a younger block, he will need to live with the fact, that he won't be able to find a high deposit to secure it.

2. We also changed the default to 10 blocks, but allow the user to define a minBlockHeight lower than this number. In this case the node would write a warning in the logfile, but still accepts the user configuration. This allows to use incubed also on different chains other than the mainnet.

3. The `safeMinBlockHeight` is now dependend on different chains, which is one single function, so we don't have hardcoded values in different places anymore.

Description

A node that is signing wrong blockhashes might get their deposit slashed from the registry. The entity that is convicting a node that signs a wrong blockhash is awarded half of the deposit.

A threat to this kind of system is that blocks might constantly be reorganized in the chain, especially with the latest block. Allowing a node to sign the latest block will definitely put the node's deposit at stake with every signature they provide.

A node can configure the `minBlockHeight` it is about to sign with a configurative option. The option defaults to a `minBlockHeight` of `6` in the default config:

code/in3-server/src/server/config.ts:L32-L32

```
minBlockHeight: 6,
```

And again in the signing function for blockheaders:

code/in3-server/src/chains/signatures.ts:L189-L189

```
const blockHeight = handler.config.minBlockHeight === undefined ? 6 : handler.config.
```

`handleSign` will refuse to sign any block that is within the last 5 blocks. The `6` th block will be signed.

code/in3-server/src/chains/signatures.ts:L190-L193

```
const tooYoungBlock = blockData.find(block => toNumber(blockNumber) - toNumber(block.
if (tooYoungBlock)
  throw new Error(' cannot sign for block ' + tooYoungBlock.number + ', because the b
```

However, a user is not prevented from configuring an insecure `minBlockHeight` (e.g. `0`) which will very likely lead to the loss of funds because the node will be signing the latest block.

The current default of `6` blocks leads to an approximate lag of `14 (avg blocktime) *6 (blocks) = 84 seconds`. While this is a favorable setting because it allows nodes to provide signatures for blocks that are at least older than 6 blocks it might still not be secure. For example, CryptoExchange Kraken requires at least 30 confirmation (abt. 6 minutes) until a transaction is confirmed. For Bitcoin it is said to be safe to wait more than 6 blocks (abt. 1 hr) for a transaction to be confirmed. ETC even underwent a deep chain reorg that could have caused many nodes to lose their deposits. The ethereum whitepaper defines an uncle that can be referenced in a block to have the following property: `It must be a direct child of the k-th generation ancestor of B, where 2 <= k <= 7.` This suggests that `k=7` 'th block can at least still be an uncle. Bitfinex requires a minimum of 10 confirmations. Some blockchain explorers and analytics tools also require a minimum of 10 confirmations. Scraped data from https://etherscan.io/blocks_forked?ps=100 shows 3 forks of depth 3 since they started keeping records 115 days ago, and no forks deeper than 3. So some applications might legitimately pick a number somewhere between 5 and 20, trading some security for better UX. However, it should be re-evaluated whether the current default provides enough security to protect the nodes funds with a trade-off of lag to the network.

Given these values it is suggested to revalidate the default of a `minBlockHeight` of 6 in favor of a more secure depth to make sure that - with a default setting - nodes will not lose funds in case of re-orgs.

Recommendation

- `config.minBlockHeight` should always be set to a sane value when loading the configuration. There should be no need to reset it to a hardcoded default value of 6 in `handleSign`. Do not hardcode the values in various places in the config.

- normalize and sanitize the settings to make sure that after loading they are always valid and within reasonable bounds. the application should refuse to run with a `minBlockHeader` set to `0` as this is a guarantee for losing funds. Other nodes can enumerate nodes that are misconfigured (e.g. with `minBlockHeight` being `0`) to request signatures just to convict them on micro-forks.
- assume a secure default setting for every chain (note that this might be different for every chain). allow to override the value by the user. warn the user of less secure settings and do not allow to set settings that are obviously leading to the loss of funds.
- re-evaluate the `minBlockHeight` of 6 for the ethereum blockchain and choose a koservative secure default.

6.11 in3-server - rpc proof handler specification inconsistency

`Medium` `✓ Fixed`

Resolution

Addressed with https://git.slock.it/in3/ts/in3-server/issues/100. Checks for `proof` and `proofWithSignature` are more strict now, `never` is not checked and assumed to be the default. Falling back to no security as a default is not considered best practice. `signatures` has been renamed to the more accurate name `signers`. The client now allows both `signers` and `signatures` and we suggest already to start planning to phase-out this ambiguity, strictly enforce the specified protocol and reduce special cases and complexity in future iterations.

Description

According to the specification incubed requests must specify whether they want to have a proof or not. There are three variants of proofs that can be requested:

- `never` - no proof appended
- `proof` - proof but no signed blockhashes
- `proofWithSignature` - proof and a request to sign blockhashes from the list of addresses provided in `signatures`.

Note that the name `signatures` for the array of signers a blockhash signature is requested from is misleading. It is actually signer addresses as listed in the `NodeRegistry` and not signatures.

Following the `in3-server` we found at least one inconsistency (and suspect more) with the proof requested by a client. The graceful check for the existence of something starting with `proof` will pass `proof` and `proofWithSignature` but also any other `proofXYZ` to the blockproof handler.

code/in3-server/src/modules/eth/EthHandler.ts:L106-L112

```
if (request.in3.verification.startsWith('proof'))
  switch (request.method) {
    case 'eth_getBlockByNumber':
    case 'eth_getBlockByHash':
    case 'eth_getBlockTransactionCountByHash':
    case 'eth_getBlockTransactionCountByNumber':
      return handleBlock(this, request)
```

Following through `handleBlock` we cannot find any check for `proofWithSignature`. The string is not found in the whole codebase which also suggests it is not tested. However, the code assumes that because `request.in3.signatures` is not empty, signatures were requested. This is inconsistent with the specification and a protocol violation.

code/in3-server/src/modules/eth/proof.ts:L237-L244

```
// create the proof
response.in3 = {
  proof: {
    type: 'blockProof',
    signatures: await collectSignatures(handler, request.in3.signatures, [{ blockNumb
  }
}
```

The same is valid for all other types of proofs. `proofWithSignature` is never checked and it is assumed that `proofWithSignature` was requested just because `request.in3.signatures` is present non-empty.

The same is true for 'never' which is actually never handled in code.

Recommendation

The protocol should be strictly enforced without allowing any ambiguities and unsharpness. Ambiguities and gracefulness in the protocol can lead to severe inconsistencies and encourage client authors to not strictly adhere to the protocol. This makes it hard to update and maintain the protocol in the future and may allow potential attackers enough freedom to exploit the protocol. Furthermore the specification must be

kept up-to-date at all times. The specification is to lead development and code must always be verified against the specification.

6.12 in3-server - hardcoded gas limit could result in failed transactions/requests `Medium` `✓ Fixed`

> ### Resolution
>
> Fixed by using web3 `eth_estimateGas` in [merge_requests/109](#) to dynamically price the gas according to the network state.

Description

There are many instances of hardcoded gas limit in `in3-server` that depending on the complexity of the transaction or gas cost changes in Ethereum could result in failed transactions.

Examples

`convict()` :

code/in3-server/src/chains/signatures.ts:L132-L137

```
await callContract(handler.config.rpcUrl, nodes.contract, 'convict(uint,bytes32)', [s
  privateKey: handler.config.privateKey,
  gas: 500000,
  value: 0,
  confirm: true                        //  we are not waiting for confirmation, since w
})
```

`recreateBlockheaders()` :

code/in3-server/src/chains/signatures.ts:L275-L280

```
await callContract(handler.config.rpcUrl, blockHashRegistry, 'recreateBlockheaders(ui
  privateKey: handler.config.privateKey,
  gas: 8000000,
  value: 0,
  confirm: true                        //  we are not waiting for confirmation, since w
})
```

Other instances of hard coded gasLimit or gasPrice:

code/in3-server/src/modules/eth/EthHandler.ts:L78-L79

```
  if (!tx || (tx.gas && toNumber(tx.gas) > 10000000)) throw new Error('eth_call with
}
```

Recommendation

Use web3 gas estimate instead. To be sure, there can be an additional gas added to the estimated value or `max(HARDCODED_GAS, estimated_amount)`

6.13 in3-server - handleRecreation tries to recreate blockchain if no block is available to recreate it from Medium ✓ Fixed

> ### Resolution
>
> Mitigated by 502b5528 by falling back to using the current block in case `searchForAvailableBlock` returns `0`. Costs can be zero, but cannot be negative anymore.
>
> The behaviour of the IN3-server code is outside the scope of this audit. However, while verifying the fixes for this specific issue it was observed that the `watch.ts:handleConvict()` relies on a static hardcoded cost calculation. We further note that the cost calculation formula has an error and is missing parentheses to avoid that costs can be zero. We did not see a reason for the costs not to be allowed to be zero. Furthermore, costs are calculated based on the difference of the conviction block to the latest block. Actual recreation costs can be less if there is an available block in blockhashRegistry to recreate it from that is other than the latest block.

Description

A node that wants to convict another node for false proof must update the `BlockhashRegistry` for signatures provided in blocks older than the most recent 256 blocks. Only when the smart contract is able to verify that the signed blockhash is wrong the convicting node will be able to receive half of its deposit.

The `in3-server` implements an automated mechanism to recreate blockhashes. It first searches for an existing blockhash within a range of blocks. If one is found and it is profitable (gas spend vs. amount awarded) the node will try to recreate the blockchain updating the registry.

- The call to `searchForAvailableBlock` might return `0` (default) because no block is actually found within the range, this will cause `costs` to be negative and the code will proceed trying to convict the node even though it cannot work.

- The call to `searchForAvailableBlock` might also return the convict block number (`latestSS==s.block`) in which case costs will be `0` and the code will still proceed trying to recreate the blockheaders and convict the node.

code/in3-server/src/chains/signatures.ts:L207-L231

```
  const [, deposit, , , , , , ,] = await callContract(handler.config.rpcUrl, nodes.cont
  const latestSS = toNumber((await callContract(handler.config.rpcUrl, blockHashRegistr
  const costPerBlock = 86412400000000
  const blocksMissing = latestSS - s.block
  const costs = blocksMissing * costPerBlock * 1.25

  if (costs > (deposit / 2)) {

    console.log("not worth it")
    //it's not worth it
    return
  }
  else {

    // it's worth convicting the server
    const blockrequest = []
    for (let i = 0; i < blocksMissing; i++) {
      blockrequest.push({
        jsonrpc: '2.0',
        id: i + 1,
        method: 'eth_getBlockByNumber', params: [
          toHex(latestSS - i), false
        ]
      })
    }
  }
```

Please note that certain parts of the code rely on hardcoded gas values. Gas economics might change with future versions of the evm and have to be re-validated with every version. It is also good practice to provide inline comments about how and on what base certain values were selected.

Recommendation

Verify that the call succeeds and returns valid values. Check if the block already exists in the `BlockhashRegistry` and avoid recreation. Also note that `searchForAvailableBlock` can wrap with values close to `uint_max` even though that is unlikely to happen. In general, return values for external calls should be validated more rigorously.

6.14 Impossible to remove malicious nodes after the initial period

Medium   ✓ Fixed

Resolution

This issue has been addressed with a large change-set that splits the NodeRegistry into two contracts, which results in a code flow that mitigates this issue by making the logic contract upgradable (after 47 days of notice). The resolution adds more complexity to the system, and this complexity is not covered by the original audit. Splitting up the contracts has the side-effect of events being emitted by two different contracts, requiring nodes to subscribe to both contracts' events.

The need for removing malicious nodes from the registry, arises from the design decision to allow anyone to register any URL. These URLs might not actually belong to the registrar of the URL and might not be IN3 nodes. This is partially mitigated by a centralization feature introduced in the mitigation phase that implements whitelist functionality for adding nodes.

We generally advocate against adding complexity, centralization and upgrading mechanisms that can allow one party to misuse functionalities of the contract system for their benefit (e.g. `adminSetNodeDeposit` is only used to reset the deposit but allows the Logic contract to set any deposit; the logic contract is set by the owner and there is a 47 day timelock).

We believe the solution to this issue, should have not been this complex. The trust model of the system is changed with this solution, now the logic contract can allow the admin a wide range of control over the system state and data.

The following statement has been provided with the change-set:

> During the 1st year, we will keep the current mechanic even though it's a centralized approach. However, we changed the structure of the smart contracts and separated the NodeRegistry into two different smart contracts: NodeRegistryLogic and NodeRegistryData. After a successful deployment only the NodeRegistryLogic-contract is able to write data into the NodeRegistryData-contract. This way, we can keep the stored data (e.g. the nodeList) in the NodeRegistryData-contract while changing the way the data gets added/updated/removed is handled in the NodeRegistryLogic-contract. We also provided a function to update the NodeRegistryLogic-contract, so that we are able to change to a better solution for removing nodes in an updated contract.

Description

The system has centralized power structure for the first year after deployment. An `unregisterKey` (creator of the contract) is allowed to remove Nodes that are in state `Stages.Active` from the registry, only in 1st year.

However, there is no possibility to remove malicious nodes from the registry after that.

code/in3-contracts/contracts/NodeRegistry.sol:L249-L264

```
/// @dev only callable in the 1st year after deployment
function removeNodeFromRegistry(address _signer)
    external
    onlyActiveState(_signer)
{

    // solium-disable-next-line security/no-block-members
    require(block.timestamp < (blockTimeStampDeployment + YEAR_DEFINITION), "only in
    require(msg.sender == unregisterKey, "only unregisterKey is allowed to remove noc

    SignerInformation storage si = signerIndex[_signer];
    In3Node memory n = nodes[si.index];

    unregisterNodeInternal(si, n);

}
```

Recommendation

Provide a solution for the network to remove fraudulent node entries. This could be done by voting mechanism (with staking, etc).

6.15 `NodeRegistry.registerNodeFor()` no replay protection and expiration Medium   Won't Fix

Resolution

This issue was addressed with the following statement:

> In our understanding of the relationship between node-owner and signer the owner both are controlled by the very same entity, thus the owner should always know the privateKey of the signer. With this in mind a replay-protection would be useless, as the owner could always sign the necessary

message. The reason why we separated the signer from the owner was to enable the possibility of owning an in3-node as with a multisig-account, as due to the nature of the exposal of the signer-key the possibility of it being leaked somehow is given (e.g. someone "hacks" the server), making the signer-key more unsecure. In addition, even though it's possible to replay the register as an owner it would unfeasable, as the owner would have to pay for the deposit anyway thus rendering the attack useless as there would be no benefit for an owner to do it.

Description

An owner can register a node with the signer not being the owner by calling `registerNodeFor`. The owner submits a message signed for the owner including the properties of the node including the url.

- The signed data does not include the `registryID` nor the `NodeRegistry`'s address and can therefore be used by the owner to submit the same node to multiple registries or chains without the signers consent.

- The signed data does not expire and can be re-used by the owner indefinitely to submit the same node again to future contracts or the same contract after the node has been removed.

- Arguments are not validated in the external function (also see issue 6.17)

code/in3-contracts/contracts/NodeRegistry.sol:L215-L223

```solidity
bytes32 tempHash = keccak256(
    abi.encodePacked(
        _url,
        _props,
        _timeout,
        _weight,
        msg.sender
    )
);
```

Recommendation

Include `registryID` and an expiration timestamp that is checked in the contract with the signed data. Validate function arguments.

6.16 BlockhashRegistry - Structure of provided blockheaders should be validated `Medium` `✓ Fixed`

Resolution

Mitigated by:

- 99f35fce - validating the block number in the provided RLP encoded input data
- 79e5a302 - fixes the potential out of bounds access for `parentHash` by requiring the the input to contain at least data up until including the `parentHash` in the user provided RLP blob. However, this check does not enforce that the minimum amount of data is available to extract the `blockNumber`

Additionally we would like to note the following:

- While the code decodes the RLPLongList structure that contains the blockheader fields it does not decode the RLPLongString `parentHash` and just assumes one length-byte for it.
- The length of the RLPLongString `parentHash` is never used but skipped instead.
- The decoding is incomplete and fragile. The method does not attempt to decode other fields in the struct to verify that they are indeed valid RLP data. For the blockNumber extraction a fixed offset of `444` is assumed to access the `difficulty` RLP field (this might through as the minimum input length up to this field is not enforced). `difficulty` is then skipped and the `blockNumber` is accessed.
- The minimum input data length enforced is shorter than a typical blockheader.
- The code relies on implicit exceptions for out of bounds array access instead of verifying early on that enough input bytes are available to extract the required data.

We would also like to note that the commit referenced as mitigation does not appear to be based on the audit code.

Description

`getParentAndBlockhash` takes an rlp-encoded blockheader blob, extracts the parent parent hash and returns both the parent hash and the calculated blockhash of the provided data.

The method is used to add blockhashes to the registry that are older than 256 blocks as they are not available to the evm directly. This is done by establishing a trust-chain from a blockhash that is already in the registry up to an older block

1. The method assumes that valid rlp encoded data is provided but the structure is not verified (rlp decodes completely; block number is correct; timestamp is younger than prevs, …), giving a wide range of freedom to an attacker with enough hashing power (or exploiting potential future issues with keccak) to forge blocks that would never be accepted by clients, but may be accepted by this smart contract. (threat: mining pool forging arbitrary non-conformant blocks to exploit the BlockhashRegistry)

2. It is not checked that input was actually provided. However, accessing an array at an invalid index will raise an exception in the EVM. Providing a single byte > `0xf7` will yield a result and succeed even though it would have never been accepted by a real node.

3. It is assumed that the first byte is the rlp encoded length byte and an offset into the provided `_blockheader` bytes-array is calculated. Memory is subsequently accessed via a low-level `mload` at this calculated offset. However, it is never validated that the offset actually lies within the provided range of bytes `_blockheader` leading to an out-of-bounds memory read access.

4. The rlp encoded data is only partially decoded. For the first rlp list the number of length bytes is extracted. For the rlp encoded long string a length byte of 1 is assumed. The inline comment appears to be inaccurate or might be misleading.
   `// we also have to add "2" = 1 byte to it to skip the length-information`

5. Invalid intermediary blocks (e.g. with parent hash `0x00`) will be accepted potentially allowing an attacker to optimize the effort needed to forge invalid blocks skipping to the desired blocknumber overwriting a certain blockhash (see issue 6.18)

6. With one collisions (very unlikely) an attacker can add arbitrary or even random values to the BlockchainRegistry. The parent-hash of the starting blockheader cannot be verified by the contract (
   ```
   [target_block_random]<--parent_hash--[rnd]<--parent_hash--[rnd]<--parent_hash--...<--
   parent_hash--[collision]<--parent_hash_collission--[anchor_block]
   ```
). While nodes can verify block structure and bail on invalid structure and check the first blocks hash and make sure the chain is in-tact the contract can't. Therefore one cannot assume the same trust in the blockchain registry when recreating blocks compared to running a full node.

code/in3-contracts/contracts/BlockhashRegistry.sol:L98-L126

```
function getParentAndBlockhash(bytes memory _blockheader) public pure returns (bytes3

    /// we need the 1st byte of the blockheader to calculate the position of the parent
    uint8 first = uint8(_blockheader[0]);

    /// calculates the offset
    /// by using the 1st byte (usually f9) and substracting f7 to get the start point o
    /// we also have to add "2" = 1 byte to it to skip the length-information
    require(first > 0xf7, "invalid offset");
    uint8 offset = first - 0xf7 + 2;

    /// we are using assembly because it's the most efficent way to access the parent b
    // solium-disable-next-line security/no-inline-assembly
    assembly { // solhint-disable-line no-inline-assembly
        // mstore to get the memory pointer of the blockheader to 0x20
        mstore(0x20, _blockheader)

        // we load the pointer we just stored
        // then we add 0x20 (32 bytes) to get to the start of the blockheader
        // then we add the offset we calculated
        // and load it to the parentHash variable
        parentHash :=mload(
            add(
                add(
                    mload(0x20), 0x20
                ), offset)
        )
    }
    bhash = keccak256(_blockheader);
```

Recommendation

1. Validate that the provided data is within a sane range of bytes that is expected (min/max blockheader sizes).
2. Validate that the provided data is actually an rlp encoded blockheader.
3. Validate that the offset for the parent Hash is within the provided data.
4. Validate that the parent Hash is non zero.
5. Validate that blockhashes do not repeat.

6.17 Registries - Incomplete input validation and inconsistent order of validations Medium Pending

Resolution

This issue describes general inconsistencies of the smart contract code base. The inconsistencies have been addressed with multiple change-sets:

Issues that have been addressed by the development team:

- `BlockhashRegistry.reCalculateBlockheaders` - bhash can be zero; blockheaders can be empty

  Fixed in 8d2bfa40 by adding the missing checks.

- `BlockhashRegistry.recreateBlockheaders` - blockheaders can be empty; Arguments should be validated before calculating values that depend on them.

  Fixed in 8d2bfa40 by adding the missing checks.

- `NodeRegistry.removeNode` - should check `require(_nodeIndex < nodes.length)` first before any other action.

  Fixed in 47255587 by adding the missing checks.

- `NodeRegistry.registerNodeFor` - Signature version `v` should be checked to be either `27 || 28` before verifying it.

  The fix in 47255587 introduced a serious typo (`v != _v`) that has been fixed with 4a0377c5 .

- `NodeRegistry.revealConvict` - unchecked `signer`

  Addressed with the comment that `signer gets checked by ecrecover` (slock.it/issue/10).

- `NodeRegistry.revealConvict` - signer status can be checked earlier. Addressed with the following comment (slock.it/issue/10):

  Due to the seperation of the contracts we will now check check the signatures and whether the blockhash is right. Only after this steps we will call into the NodeRegistryData contracts, thus potentially saving gas

- `NodeRegistry.updateNode` - the check if the `newURL` is registered can be done earlier

  Fixed in 4786a966.

- `BlockhashRegistry.getParentAndBlockhash` - blockheader structure can be random as long as parenthash can be extracted

  This issue has been reviewed as part of issue 6.16 (99f35fce).

Issues that have **not been addressed** by the development team and still persist:

- `BlockhashRegistry.searchForAvailableBlock` - `_startNumber + _numBlocks` can be > `block.number; _startNumber + _numBlocks` can overflow.

  This issue has not been addressed.

General Notes:

- Ideally commits directly reference issues that were raised during the audit. During the review of the mitigations provided with the change-sets for the listed issues we observed that change-sets contain changes that are not directly related to the issues. (e.g. 79e5a302)

Description

Methods and Functions usually live in one of two worlds:

- public API - methods declared with visibility `public` or `external` exposed for interaction by other parties
- internal API - methods declared with visibility `internal`, `private` that are not exposed for interaction by other parties

While it is good practice to visually distinguish internal from public API by following commonly accepted naming convention e.g. by prefixing internal functions with an underscore (`_doSomething` vs. `doSomething`) or adding the keyword `unsafe` to unsafe functions that are not performing checks and may have a dramatic effect to the system (`_unsafePayout` vs. `RequestPayout`), it is important to properly verify that inputs to methods are within expected ranges for the implementation.

Input validation checks should be explicit and well documented as part of the code's documentation. This is to make sure that smart-contracts are robust against erroneous inputs and reduce the potential attack surface for exploitation.

It is good practice to verify the methods input as early as possible and only perform further actions if the validation succeeds. Methods can be split into an external or public API that

performs initial checks and subsequently calls an internal method that performs the action.

The following lists some public API methods that are not properly checking the provided data:

- `BlockhashRegistry.reCalculateBlockheaders` - bhash can be zero; blockheaders can be empty
- `BlockhashRegistry.getParentAndBlockhash` - blockheader structure can be random as long as parenthash can be extracted
- `BlockhashRegistry.recreateBlockheaders` - blockheaders can be empty; Arguments should be validated before calculating values that depend on them:

code/in3-contracts/contracts/BlockhashRegistry.sol:L70-L70

```
assert(_blockNumber > _blockheaders.length);
```

- `BlockhashRegistry.searchForAvailableBlock` - `_startNumber + _numBlocks` can be > `block.number; _startNumber + _numBlocks` can overflow.

- `NodeRegistry.removeNode` - should check `require(_nodeIndex < nodes.length)` first before any other action.

code/in3-contracts/contracts/NodeRegistry.sol:L602-L609

```
function removeNode(uint _nodeIndex) internal {
    // trigger event
    emit LogNodeRemoved(nodes[_nodeIndex].url, nodes[_nodeIndex].signer);
    // deleting the old entry
    delete urlIndex[keccak256(bytes(nodes[_nodeIndex].url))];
    uint length = nodes.length;

    assert(length > 0);
```

- `NodeRegistry.registerNodeFor` - Signature version `v` should be checked to be either `27 || 28` before verifying it.

code/in3-contracts/contracts/NodeRegistry.sol:L200-L212

```
function registerNodeFor(
    string calldata _url,
    uint64 _props,
    uint64 _timeout,
    address _signer,
    uint64 _weight,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
)
    external
    payable
{
```

- `NodeRegistry.revealConvict` - unchecked `signer`

code/in3-contracts/contracts/NodeRegistry.sol:L321-L321

```
SignerInformation storage si = signerIndex[_signer];
```

- `NodeRegistry.revealConvict` - signer status can be checked earlier.

code/in3-contracts/contracts/NodeRegistry.sol:L344-L344

```
require(si.stage != Stages.Convicted, "node already convicted");
```

- `NodeRegistry.updateNode` - the check if the `newURL` is registered can be done earlier

code/in3-contracts/contracts/NodeRegistry.sol:L444-L444

```
require(!urlIndex[newURl].used, "url is already in use");
```

Recommendation

Use Checks-Effects-Interactions pattern for all functions.

6.18 BlockhashRegistry - `recreateBlockheaders` allows invalid parent hashes for intermediary blocks Medium ✓ Fixed

Resolution

Fixed by requiring valid parent hashes for blockheaders.

Description

It is assumed that a blockhash of `0x00` is invalid, but the method accepts intermediary parent hashes extracted from blockheaders that are zero when establishing the trust chain.

`recreateBlockheaders` relies on `reCalculateBlockheaders` to correctly establish a chain of trust from the provided list of `_blockheaders` to a valid blockhash stored in the contract. However, `reCalculateBlockheaders` fails to raise an exception in case `getParentAndBlockhash` returns a blockhash of `0x00`. Subsequently it will skip over invalid blockhashes and continue to establish the trust chain without raising an error.

This may allow an attacker with enough hashing power to store a blockheader hash that is actually invalid on the real chain but accepted within this smart contract. This may even only be done temporarily to overwrite an existing hash for a short period of time (see https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/24).

code/in3-contracts/contracts/BlockhashRegistry.sol:L141-L147

```
for (uint i = 0; i < _blockheaders.length; i++) {
    (calcParent, calcBlockhash) = getParentAndBlockhash(_blockheaders[i]);
    if (calcBlockhash != currentBlockhash) {
        return 0x0;
    }
    currentBlockhash = calcParent;
}
```

Recommendation

Stop processing the array of `_blockheaders` immediately if a blockheader is invalid.

6.19 BlockhashRegistry - `recreateBlockheaders` succeeds and emits an event even though no blockheaders have been provided

Medium   ✓ Fixed

Resolution

Fixed the vulnerable scenarios by adding proper checks to:

- Prevent passing empty `_blockheaders` in 8d2bfa40
- Prevent storing the same blockhash twice in 80bb6ecf

Description

The method is used to re-create blockhashes from a list of rlp-encoded `_blockheaders`. However, the method never checks if `_blockheaders` actually contains items. The result is, that the method will unnecessarily store the same value that is already in the `blockhashMapping` at the same location and wrongly log `LogBlockhashAdded` even though nothing has been added nor changed.

- 1. assume `_blockheaders` is empty and the registry already knows the blockhash of `_blockNumber`

code/in3-contracts/contracts/BlockhashRegistry.sol:L61-L67

```
function recreateBlockheaders(uint _blockNumber, bytes[] memory _blockheaders) public

    bytes32 currentBlockhash = blockhashMapping[_blockNumber];
    require(currentBlockhash != 0x0, "parentBlock is not available");

    bytes32 calculatedHash = reCalculateBlockheaders(_blockheaders, currentBlockhash)
    require(calculatedHash != 0x0, "invalid headers");
```

- 2. An attempt is made to re-calculate the hash of an empty `_blockheaders` array (also passing the `currentBlockhash` from the registry)

code/in3-contracts/contracts/BlockhashRegistry.sol:L66-L66

```
bytes32 calculatedHash = reCalculateBlockheaders(_blockheaders, currentBlockhash);
```

- 3. The following loop in `reCalculateBlockheaders` is skipped and the `currentBlockhash` is returned.

code/in3-contracts/contracts/BlockhashRegistry.sol:L134-L149

```
function reCalculateBlockheaders(bytes[] memory _blockheaders, bytes32 _bHash) public

    bytes32 currentBlockhash = _bHash;
    bytes32 calcParent = 0x0;
    bytes32 calcBlockhash = 0x0;

    /// save to use for up to 200 blocks, exponential increase of gas-usage afterwards
    for (uint i = 0; i < _blockheaders.length; i++) {
        (calcParent, calcBlockhash) = getParentAndBlockhash(_blockheaders[i]);
        if (calcBlockhash != currentBlockhash) {
            return 0x0;
        }
        currentBlockhash = calcParent;
    }

    return currentBlockhash;
```

- 4. The assertion does not fire, the `bnr` to store the `calculatedHash` is the same as the one initially provided to the method as an argument.. Nothing has changed but an event is emitted.

code/in3-contracts/contracts/BlockhashRegistry.sol:L69-L74

```
    /// we should never fail this assert, as this would mean that we were able to recre
    assert(_blockNumber > _blockheaders.length);
    uint bnr = _blockNumber - _blockheaders.length;
    blockhashMapping[bnr] = calculatedHash;
    emit LogBlockhashAdded(bnr, calculatedHash);
}
```

Recommendation

The method is crucial for the system to work correctly and must be tightly controlled by input validation. It should not be allowed to overwrite an existing value in the contract (issue 6.29) or emit an event even though nothing has happened. Therefore validate that user provided input is within safe bounds. In this case, that at least one `_blockheader` has been provided. Validate that `_blockNumber` is less than `block.number` and do not expect that parts of the code will throw and safe the contract from exploitation.

6.20 `NodeRegistry.updateNode` replaces `signer` with owner and emits inconsistent events Medium ✓ Fixed

Reviewed merged changes at in3-contracts/5cb54165.

- The method now emits a distinct event *twice* when node properties are updated.
- The event correctly emits the signer.
- When updating a node URL, the new URLInformation now correctly sets the signer.

However, there is a discrepancy between the process of registering a node and updating node's properties. When registering a node the owner has to provide a signed message containing the registration properties from the signer. Once the node is registered it can be unilaterally updated by the owner without requiring the signers permission to do so. According to slock.it it is assumed that the node owner and the signer are in control of the same entity and therefore this is not a concern.

Description

When the `owner` calls `updateNode()` function providing a new `url` for the node, the `signer` of the url is replaced by `msg.sender` which in this case is the owner of the node. Note that new URL can resolve to the same URL as before (See https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/36).

code/in3-contracts/contracts/NodeRegistry.sol:L438-L452

```
if (newURl != keccak256(bytes(node.url))) {

    // deleting the old entry
    delete urlIndex[keccak256(bytes(node.url))];

    // make sure the new url is not already in use
    require(!urlIndex[newURl].used, "url is already in use");

    UrlInformation memory ui;
    ui.used = true;
    ui.signer = msg.sender;
    urlIndex[newURl] = ui;
    node.url = _url;
}
```

Furthermore, the method emits a `LogNodeRegistered` event when the node structure is updated. However, the event will always emit `msg.sender` as the signer even though that might not be true. For example, if the `url` does not change, the signer can still be another account that was previously registered with `registerNodeFor` and is not necessarily the `owner`.

code/in3-contracts/contracts/NodeRegistry.sol:L473-L478

```
emit LogNodeRegistered(
    node.url,
    _props,
    msg.sender,
    node.deposit
);
```

code/in3-contracts/contracts/NodeRegistry.sol:L30-L30

```
event LogNodeRegistered(string url, uint props, address signer, uint deposit);
```

Recommendation

- The `updateNode()` function gets the signer as an input used to reference the node structure and this `signer` should be set for the `UrlInformation`.

```
function updateNode(
        address _signer,
        string calldata _url,
        uint64 _props,
        uint64 _timeout,
        uint64 _weight
    )
```

- The method should actually only allow to change node properties when `owner==signer` otherwise `updateNode` is bypassing the strict requirements enforced with `registerNodeFor` where e.g. the `url` needs to be signed by the signer in order to register it.
- The emitted event should always emit `node.signer` instead of `msg.signer` which can be wrong.
- The method should emit its own distinct event `LogNodeUpdated` for audit purposes and to be able to distinguish new node registrations from node structure updates. This

might also require software changes to client/node implementations to listen for node updates.

6.21 NodeRegistry - `In3Node memory n` is never used Minor ✓ Fixed

> ### Resolution
>
> Fixed by removing the modifier and move the node-signer check to functions in
> f1fd7943

Description

NodeRegistry `In3Node memory n` is never used inside the modifier `onlyActiveState`.

code/in3-contracts/contracts/NodeRegistry.sol:L125-L133

```solidity
modifier onlyActiveState(address _signer) {

    SignerInformation memory si = signerIndex[_signer];
    require(si.stage == Stages.Active, "address is not an in3-signer");

    In3Node memory n = nodes[si.index];
    assert(nodes[si.index].signer == _signer);
    _;
}
```

Recommendation

Use `n` in the assertion to access the node signer `assert(n.signer == _signer);` or directly access it from storage and avoid copying the struct.

6.22 NodeRegistry - `returnDeposit` and `transferOwnership` should emit an event Minor ✓ Fixed

> ### Resolution
>
> Fixed in f1fd7943 by adding new events (`LogDepositReturned`, `LogNodeRemoved`,
> `LogNodeConvicted`, `LogOwnershipChanged`, `LogNodeUpdated`, `LogNodeRegistered`)

Description

Important state changing functions should emit an event for the purpose of having an audit trail and being able to monitor the smart contract usage and performance.

Recommendation

Emit events for `returnDeposit` and `transferOwnership`.

6.23 in3-server - `in3_stats` leaks information Minor Won't Fix

Resolution
There is a config-option `-profile.noStats` to deactivate stats, which defaults to `True` in the current release.

Description

`in3_stat` shows information from node activities in the currentMonth, currentDay, currentHour which can result in leaking information about the functionality that node is being used for. This information might be valuable when an attacker wants to find out how utilized a node is and if any reflection attacks are successful (https://github.com/ConsenSys/slockit-in3-audit-2019-09/issues/50).

Examples

```
{
    'profile': AttributeDict({
        'name': 'Slockit2',
        'icon': 'https://slock.it/assets/slock_logo.png',
        'url': 'https://slock.it'
    }),
    'stats': Attr
ibuteDict({
        'upSince': 1568400626355,
        'currentMonth': AttributeDict({
            'requests': 47618,
            'lastRequest': 1569422025347,
            'methods': AttributeDict({
                'nd-2': 2,
                'eth_call': 940,
                'eth_blockNumber': 25,
                'eth_getBlockByNumber': 45395,
                'web3_clientVersion': 386,
                'admin_datadir': 7,
                'admin_peers': 11,
                'shh_version': 7,
                'shh_info': 14,
                'admin_nodeInfo ': 9, '
                txpool_status ': 9, '
                personal_listAccounts ': 3, '
                eth_chainId ': 12, '
                eth_protocolVersion ': 6, '
                net_listening ': 6, '
                net_peerCount ': 6, '
                eth_syncing ': 6,
                'eth_mining': 6,
                'eth_hashrate': 6,
                'eth_gasPrice': 18,
                'eth_coinbase': 44,
                'eth_accounts': 54,
                'eth_getBalance': 321,
                'personal_unlockAccount': 61,
                'personal_
                importRawKey ': 5, '
                personal_newAccount ': 8, '
                eth_estimateGas ': 16, '
                eth_sendRawTransaction ': 9, '
                eth_getTransactionReceipt ': 49, '
                in3_sign ': 59, '
                eth_getCode ': 33,
                'eth_getTransactionCount': 15,
                'eth_getLogs': 8,
                'in3_stats': 16,
                'in3_validatorlist': 15,
                'in3_nodeList': 15,
                'in3_call': 15,
                'proof in3 sign': 1
```

```
                    proof_in3_sign : 1
                })
            }),
        'currentDay': AttributeDict({
            'requests': 144,
            'lastRequest': 1569422025347,
            'methods': AttributeDict({
                'eth_getBlockByNumber': 135,
                'web3_clientVersion': 6,
                'eth_coinbase': 1,
                'eth_accounts': 1,
                'in3_stats': 1
            })
        }),
        'currentHour': AttributeDict({
            'requests': 144,
            'lastRequest': 1569422025346,
            'methods': AttributeDict({
                'eth_getBlockByNumber': 135,
                'web3_clientVersion': 6,
                'eth_coinbase': 1,
                'eth_accounts': 1,
                'in3_stats': 1
            })
        })
    })
}
```

Recommendation

Make sure if this information is needed, if not enable it just for debugging purposes.

6.24 NodeRegistry - removeNode unnecessarily casts the nodeIndex to uint64 potentially truncating its value [Minor] [✓ Fixed]

> ### Resolution
>
> Fixed as per recommendation https://git.slock.it/in3/in3-contracts/commit/6c35dd422e27eec1b1d2f70e328268014cadb515.

Description

`removeNode` removes a node from the Nodes array. This is done by copying the last node of the array to the `_nodeIndex` of the node that is to be removed. Finally the node array size is decreased.

A Node's index is also referenced in the `SignerInformation` struct. This index needs to be adjusted when removing a node from the array as the last node is copied to the index of the node that is to be removed.

When adjusting the Node's index in the `SignerInformation` struct `removeNode` casts the index to `uint64`. This is both unnecessary as the struct defines the index as `uint` and theoretically dangerous if a node at an index greater than `uint64_max` is removed. The resulting `SignerInformation` index will be truncated to `uint64` leading to an inconsistency in the contract.

code/in3-contracts/contracts/NodeRegistry.sol:L60-L69

```
struct SignerInformation {
    uint64 lockedTime;              /// timestamp until the deposit of an in3-node
    address owner;                  /// the owner of the node

    Stages stage;                   /// state of the address

    uint depositAmount;             /// amount of deposit to be locked, used only

    uint index;                     /// current index-position of the node in the
}
```

code/in3-contracts/contracts/NodeRegistry.sol:L614-L620

```
// move the last entry to the removed one.
In3Node memory m = nodes[length - 1];
nodes[_nodeIndex] = m;

SignerInformation storage si = signerIndex[m.signer];
si.index = uint64(_nodeIndex);
nodes.length--;
```

Recommendation

Do not cast and therefore truncate the index.

6.25 Registries - general inconsistencies Minor Pending

Resolution

The breakdown of the fixes are as follows:

- NodeRegistry - check for addr(0) being passed. This is anyway only done in the constructor and will not require a lot of gas.

The proper checks for registry addresses are added in 4786a966.

- NodeRregistry - unnecessary payable

Removed payable modifier everywhere, as ERC20 support is added to the system. ERC20 support is not part of this audit.

- NodeRegistry - deposit checks can be combined into one function to make the code more readable. The min deposit amount could be exposed as public const to allow other entities to query the contracts minimum deposit similar to the max ether amount. `MAX_ETHER_LIMIT` should make clear that this limit is only applicable in the first year (e.g. `MAX_ETHER_LIMIT_FIRST_YEAR`).

Fixed and variables renamed.

- NodeRegistry - `require(si.owner == msg.sender)` can be checked before accessing the `nodes` array

Added proper checks in c9e75b35.

- NodeRegistry - removeNode resets index to a valid node array index of 0. Even though the code will access the index it is good practice to set this to an invalid value to make sure it raises an error condition if it is wrongly accessed in a future revision of the code. This is mainly a safeguard. Another option is to invalidate the 0 index.

Although the index is not set to 0, this issue is not yet fixed (Follow up here).

- NodeRegistry - implicitly set defaults are hard to maintain. This should be a constant state variable that can be queried to be transparent about minimum and maximum values. Prefer throwing an exception instead of automatically setting the value to a minimum as this might be unexpected by a client and can cover error conditions.

`timeout` has been removed, so this is obsolete as it is not in the new code anymore.

- NodeRegistry - one year startup period: instead of storing the deployment timestamp the contract should store the end-of-admin-timestamp.

Fixed as recommended `timestampAdminKeyActive = block.timestamp + YEAR_DEFINITION;`

- NodeRegistry - inefficient re-calculation of hash

Fixed ([issues/16](#)).

- NodeRegistry - weight should be part of proofHash

Added in [9fa5548d](#).

- NodeRegistry - updateNode if the new timeout is smaller than the current timeout it will silently be ignored. This may be unexpected by the caller and cover error conditions where a client provides wrong inputs. Raising an exception should be preferred in such cases instead of gracefully assuming values.

Fixed by removing timeout variable ([issues/16](#)).

- NodeRegistry - admin functionality should be clearly named as such for transparency reasons (e.g. `adminRemovenodeFromRegistry`).

Renamed all admin function in both contracts with prefix `admin`.

Description

- NodeRegistry - check for addr(0) being passed. This is anyway only done in the constructor and will not require a lot of gas.

code/in3-contracts/contracts/NodeRegistry.sol:L138-L139

```
constructor(BlockhashRegistry _blockRegistry) public {
    blockRegistry = _blockRegistry;
```

- NodeRregistry - unnecessary payable

code/in3-contracts/contracts/NodeRegistry.sol:L535-L535

```
    address payable _owner,
```

- NodeRegistry - deposit checks can be combined into one function to make the code more readable. The min deposit amount could be exposed as public const to allow other entities to query the contracts minimum deposit similar to the max ether amount. `MAX_ETHER_LIMIT` should make clear that this limit is only applicable in the first year (e.g. `MAX_ETHER_LIMIT_FIRST_YEAR`) .

code/in3-contracts/contracts/NodeRegistry.sol:L543-L545

```
require(_deposit >= 10 finney, "not enough deposit");

checkNodeProperties(_deposit, _timeout);
```

code/in3-contracts/contracts/NodeRegistry.sol:L120-L120

```
uint constant internal MAX_ETHER_LIMIT = 50 ether;
```

- NodeRegistry - `require(si.owner == msg.sender)` can be checked before accessing the `nodes` array

code/in3-contracts/contracts/NodeRegistry.sol:L402-L404

```
SignerInformation storage si = signerIndex[_signer];
In3Node memory n = nodes[si.index];
require(si.owner == msg.sender, "only for the in3-node owner");
```

- NodeRegistry - removeNode resets index to a valid node array index of 0. Even though the code will access the index it is good practice to set this to an invalid value to make sure it raises an error condition if it is wrongly accessed in a future revision of the code. This is mainly a safeguard. Another option is to invalidate the 0 index.

code/in3-contracts/contracts/NodeRegistry.sol:L612-L612

```
signerIndex[nodes[_nodeIndex].signer].index = 0;
```

- NodeRegistry - implicitly set defaults are hard to maintain. This should be a constant state variable that can be queried to be transparent about minimum and maximum

values. Prefer throwing an exception instead of automatically setting the value to a minimum as this might be unexpected by a client and can cover error conditions.

code/in3-contracts/contracts/NodeRegistry.sol:L565-L565

```
m.timeout = _timeout > 1 hours ? _timeout : 1 hours;
```

- NodeRegistry - one year startup period: instead of storing the deployment timestamp the contract should store the end-of-admin-timestamp.

code/in3-contracts/contracts/NodeRegistry.sol:L256-L256

```
require(block.timestamp < (blockTimeStampDeployment + YEAR_DEFINITION), "only in 1st
```

- NodeRegistry - inefficient re-calculation of hash

code/in3-contracts/contracts/NodeRegistry.sol:L438-L441

```
if (newURl != keccak256(bytes(node.url))) {

    // deleting the old entry
    delete urlIndex[keccak256(bytes(node.url))];
```

- NodeRegistry - weight should be part of proofHash

code/in3-contracts/contracts/NodeRegistry.sol:L490-L502

```
function calcProofHash(In3Node memory _node) internal pure returns (bytes32) {

    return keccak256(
        abi.encodePacked(
            _node.deposit,
            _node.timeout,
            _node.registerTime,
            _node.props,
            _node.signer,
            _node.url
        )
    );
}
```

- NodeRegistry - updateNode if the new timeout is smaller than the current timeout it will silently be ignored. This may be unexpected by the caller and cover error conditions where a client provides wrong inputs. Raising an exception should be preferred in such cases instead of gracefully assuming values.

code/in3-contracts/contracts/NodeRegistry.sol:L463-L465

```
if (_timeout > node.timeout) {
    node.timeout = _timeout;
}
```

- NodeRegistry - admin functionality should be clearly named as such for transparency reasons (e.g. `adminRemovenodeFromRegistry`).

6.26 BlockhashRegistry- assembly code can be optimized `Minor`

`✓ Fixed`

Resolution
Fixed as per recommendation with https://git.slock.it/in3/in3-contracts/commit/87f02a7c4f5c30d2b4be42f331c1306e85d42ca6.

Description

The following code can be optimized by removing `mload` and `mstore` :

code/in3-contracts/contracts/BlockhashRegistry.sol:L106-L125

```
require(first > 0xf7, "invalid offset");
uint8 offset = first - 0xf7 + 2;

/// we are using assembly because it's the most efficent way to access the parent block
// solium-disable-next-line security/no-inline-assembly
assembly { // solhint-disable-line no-inline-assembly
    // mstore to get the memory pointer of the blockheader to 0x20
    mstore(0x20, _blockheader)

    // we load the pointer we just stored
    // then we add 0x20 (32 bytes) to get to the start of the blockheader
    // then we add the offset we calculated
    // and load it to the parentHash variable
    parentHash :=mload(
        add(
            add(
                mload(0x20), 0x20
            ), offset)
    )
}
```

Recommendation

```
assembly { // solhint-disable-line no-inline-assembly
        // mstore to get the memory pointer of the blockheader to 0x20
        //mstore(0x20, _blockheader)  //@audit should assign 0x20ptr to variable fi

        // we load the pointer we just stored
        // then we add 0x20 (32 bytes) to get to the start of the blockheader
        // then we add the offset we calculated
        // and load it to the parentHash variable
        parentHash :=mload(
            add(
                add(
                    _blockheader, 0x20
                ), offset)
        )
    }
```

6.27 Experimental Compiler features are enabled - ABIEncoderV2 `Minor` `Won't Fix`

Resolution

This issue has been addressed with the following statement:

> In order to pass structs between contracts we need that new ABIEncoder.
> [..] The old NodeRegistry did not require the `ABIEncoderV2`. [..] But due to the
> separation of the contracts in Logic and Data we are passing certain data-
> structures between contracts.

Description

The smart contracts enable experimental compiler features. Please note that these features are experimental for a reason and should be avoided unless explicitly required.

code/in3-contracts/contracts/BlockhashRegistry.sol:L21-L21

```
pragma experimental ABIEncoderV2;
```

Seems that `NodeRegistry` does not require any `ABIEncoderV2` specific functionality.

code/in3-contracts/contracts/NodeRegistry.sol:L21-L21

```
pragma experimental ABIEncoderV2;
```

6.28 BlockhashRegistry - `recreateBlockheaders()` should use the evm provided blockhash when applicable Minor Pending

Resolution

The provided code-change at 79fa3ef1 is not addressing the raised concerns. As noted in the recommendation it is suggested to completely skip the recreation routine if the target blockhash (`_blockNumber.sub(_blockheaders.length)`) is available to the evm. The method should call `saveBlockNumber(_blockNumber)` instead.

The commit attempts to add a verification for extracted blockhashes from the user provided RLP data if the blockhash for the block is available. However, the variable name `currentBlock` is misleading making it hard to follow the authors intent.

Description

There are different levels of trust attached to blockhashes stored in the BlockhashRegistry. On one side there are blockhashes which data-source is the evm (`blockhash(blocknumber)`) and on the other side there are blockhashes that have been fed into the system by recalculating block-headers and establishing a trust chain to an already existing blockhash in the contract. While the contract can trust the result of `blockhash(blocknumber)` for the most recent 256 blocks because the information is coming directly from the evm, blockhashes that are re-created by calling `recreateBlockheaders` are manually verified and trust relies on the proper validation of the chain of block-headers provided.

Side-effect: Also saves gas by avoiding unnecessary calculations within the `recreateBlockheaders()` codepath as blockhash is already available via evm.

Recommendation

`recreateBlockheaders()` should prefer to use `blockhash(number)` by calling `saveBlockNumber()` instead of re-calculating the blockhash from the user provided chain of blockheaders, if the blockhash can easily be accessed by the evm (most recent 256 blocks, except current block). Check if `_blockheaders.length > 0 && _blockNumber.sub(_blockheaders.length) < block.number-256` .

6.29 BlockhashRegistry - Existing blockhashes can be overwritten

`Minor` `✓ Fixed`

> ### Resolution
>
> Addressed with 80bb6ecf and 17d450cf by checking if blockhash exists and changing the `assert` to `require` .

Description

Last 256 blocks, that are available in the EVM environment, are stored in `BlockhashRegistry` by calling `snapshot()` or `saveBlockNumber(uint _blockNumber)` functions. Older blocks are

recreated by calling `recreateBlockheaders` .

The methods will overwrite existing blockhashes.

code/in3-contracts/contracts/BlockhashRegistry.sol:L79-L87

```solidity
function saveBlockNumber(uint _blockNumber) public {

    bytes32 bHash = blockhash(_blockNumber);

    require(bHash != 0x0, "block not available");

    blockhashMapping[_blockNumber] = bHash;
    emit LogBlockhashAdded(_blockNumber, bHash);
}
```

code/in3-contracts/contracts/BlockhashRegistry.sol:L72

```solidity
blockhashMapping[bnr] = calculatedHash;
```

Recommendation

If a block is already saved in the smart contract, it can be checked and a SSTORE can be prevented to save gas. Require that blocknumber hash is not stored.

```solidity
require(blockhashMapping[_blockNumber] == 0x0, "block already saved");
```

7 Tool-Based Analysis

Several tools were used to perform automated analysis of the reviewed contracts. These issues were reviewed by the audit team, and relevant issues are listed in the Issue Details section.

7.1 MythX

MythX is a security analysis API for Ethereum smart contracts. It performs multiple types of analysis, including fuzzing and symbolic execution, to detect many common vulnerability types. The tool was used for automated vulnerability discovery for all audited contracts and libraries. More details on MythX can be found at mythx.io.
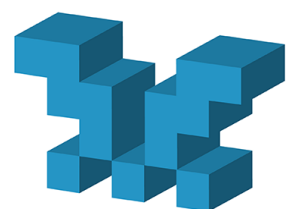
Below is the raw output of the MythX vulnerability scan:

```json
[
    {
        "issues": [
            {
                "swcID": "SWC-127",
                "swcTitle": "",
                "description": {
                    "head": "jump to arbitrary destination",
                    "tail": "A caller can trigger a jump to an arbitrary destination.
                },
                "severity": "High",
                "locations": [
                    {
                        "sourceMap": "20901:1:1",
                        "sourceType": "raw-bytecode",
                        "sourceFormat": "evm-byzantium-bytecode",
                        "sourceList": [
                            "0x63ad5e3d2c8551cf64f6d0425940efdeb79801907fcad157d1c829
                            "0xd8d70c0998b3293c364b1cde922c80081d70d02726e74091c8aae6
                        ]
                    },
                    {
                        "sourceMap": "23263:248:-1",
                        "sourceType": "solidity-file",
                        "sourceFormat": "text",
                        "sourceList": [
                            "NodeRegistry.sol"
                        ]
                    }
                ],
                "extra": {
                    "discoveryTime": 5593089348,
                    "testCases": [
                        {
                            "initialState": {
                                "accounts": {
                                    "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa0": {
                                        "nonce": 0,
                                        "balance": "0x000000000000000000000000000000
                                        "code": "",
                                        "storage": {}
                                    },
                                    "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa1": {
                                        "nonce": 1,
                                        "balance": "0x000000000000000000000000000000
                                        "code": "",
                                        "storage": {}
                                    },
                                    "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa2": {
                                        "nonce": 1,
                                        "balance": "0x000000000000000000000000000000
                                        "code": ""
```

```json
                            "code": "",
                            "storage": {}
                        },
                        "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa3": {
                            "nonce": 1,
                            "balance": "0x00000000000000000000000000000
                            "code": "0x00",
                            "storage": {}
                        },
                        "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa4": {
                            "nonce": 1,
                            "balance": "0x00000000000000000000000000000
                            "code": "0xfd",
                            "storage": {}
                        },
                        "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa5": {
                            "nonce": 1,
                            "balance": "0x00000000000000000000000000000
                            "code": "0x608060405260005600a165627a7a723058
                            "storage": {}
                        },
                        "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa6": {
                            "nonce": 1,
                            "balance": "0x00000000000000000000000000000
                            "code": "0x608060405273aaaaaaaaaaaaaaaaaaaaa
                            "storage": {}
                        },
                        "0xaffeaffeaffeaffeaffeaffeaffeaffeaffeaffe": {
                            "nonce": 0,
                            "balance": "0x00000000000000000000000000000
                            "code": "",
                            "storage": {}
                        }
                    }
                },
                "steps": [
                    {
                        "address": "",
                        "gasLimit": "0xffffff",
                        "gasPrice": "0x773594000",
                        "input": REMOVED,
                        "origin": "0xaffeaffeaffeaffeaffeaffeaffeaffe
                        "value": "0x0",
                        "blockCoinbase": "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                        "blockDifficulty": "0xa7d7343662e26",
                        "blockGasLimit": "0xffffff",
                        "blockNumber": "0x661a55",
                        "blockTime": "0x5be99aa8"
                    },
                    {
                        "address": "0x0901d12ebe1b195e5aa8748e62bd7734ae1
                        "gasLimit": "0x7d00",
```

```
                            "gasPrice": "0x773594000",
                            "input": "0xac4898730000000000000000000000000000000
                            "origin": "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                            "value": "0x9",
                            "blockCoinbase": "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                            "blockDifficulty": "0x52c054bfb494c",
                            "blockGasLimit": "0x7d0000",
                            "blockNumber": "0x661a55",
                            "blockTime": "0x5be99aa8"
                        }
                    ]
                }
            ],
            "toolName": "harvey"
        }
    },
    {
        "swcID": "SWC-120",
        "swcTitle": "Weak Sources of Randomness from Chain Attributes",
        "description": {
            "head": "Potential use of a weak source of randomness \"blockhash
            "tail": "Using past or the current block hashes through \"blockha
        },
        "severity": "Medium",
        "locations": [
            {
                "sourceMap": "6746:25:0",
                "sourceType": "solidity-file",
                "sourceFormat": "text",
                "sourceList": [
                    "NodeRegistry.sol"
                ]
            }
        ],
        "extra": {
            "discoveryTime": 1666071716,
            "toolName": "maru"
        }
    },
    {
        "swcID": "SWC-120",
        "swcTitle": "Weak Sources of Randomness from Chain Attributes",
        "description": {
            "head": "Potential use of a weak source of randomness \"blockhash
            "tail": "Using past or the current block hashes through \"blockha
        },
        "severity": "Medium",
        "locations": [
            {
                "sourceMap": "13158:23:0",
                "sourceType": "solidity-file",
                "sourceFormat": "text".
```

```json
                    "sourceList": [
                        "NodeRegistry.sol"
                    ]
                }
            ],
            "extra": {
                "discoveryTime": 1666159516,
                "toolName": "maru"
            }
        },
        {
            "swcID": "SWC-120",
            "swcTitle": "Weak Sources of Randomness from Chain Attributes",
            "description": {
                "head": "Potential use of a weak source of randomness \"block.num
                "tail": "Using past or the current block hashes through \"block.r
            },
            "severity": "Medium",
            "locations": [
                {
                    "sourceMap": "6756:12:0",
                    "sourceType": "solidity-file",
                    "sourceFormat": "text",
                    "sourceList": [
                        "NodeRegistry.sol"
                    ]
                }
            ],
            "extra": {
                "discoveryTime": 1666984722,
                "toolName": "maru"
            }
        },
        {
            "swcID": "SWC-120",
            "swcTitle": "Weak Sources of Randomness from Chain Attributes",
            "description": {
                "head": "Potential use of a weak source of randomness \"block.num
                "tail": "Using past or the current block hashes through \"block.r
            },
            "severity": "Medium",
            "locations": [
                {
                    "sourceMap": "7532:12:0",
                    "sourceType": "solidity-file",
                    "sourceFormat": "text",
                    "sourceList": [
                        "NodeRegistry.sol"
                    ]
                }
            ],
```
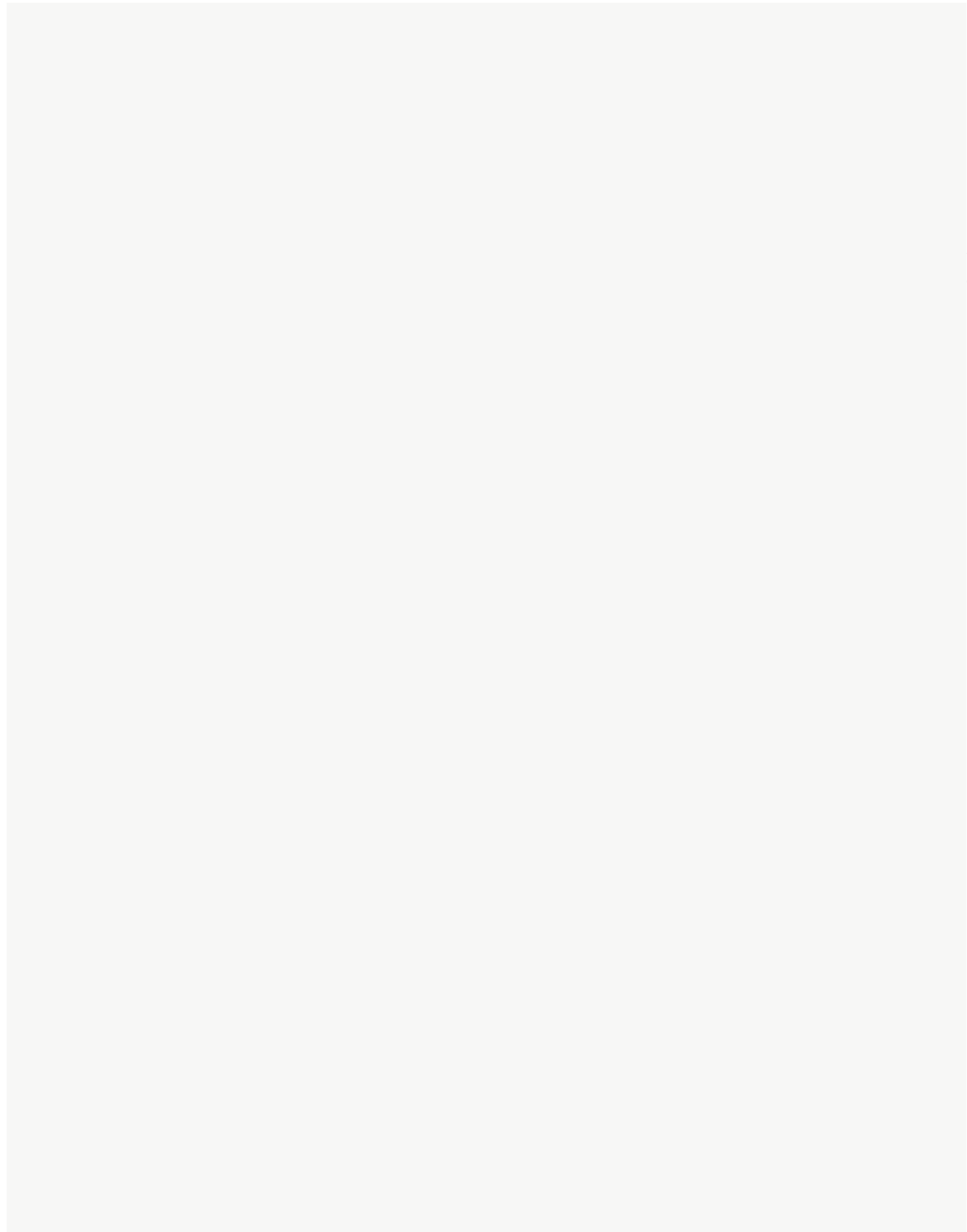
7.2 Ethlint

Ethlint is an open source project for linting Solidity code. Only security-related issues were reviewed by the audit team.

Below is the raw output of the Ethlint vulnerability scan:


```
contracts/BlockhashRegistry.sol
  21:0      warning    Avoid using experimental features in production code    no-expe
  46:12     warning    Line exceeds the limit of 145 characters                max-ler
  61:1      error      Line contains trailing whitespace                       no-trai
  79:4      warning    Line exceeds the limit of 145 characters                max-ler
  81:1      error      Line contains trailing whitespace                       no-trai
  98:4      warning    Line exceeds the limit of 145 characters                max-ler
  134:4     warning    Line exceeds the limit of 145 characters                max-ler
  142:1     error      Line contains trailing whitespace                       no-trai

contracts/NodeRegistry.sol
  21:0      warning    Avoid using experimental features in production code    no-expe
  117:1     error      Line contains trailing whitespace                       no-trai
  123:1     error      Line contains trailing whitespace                       no-trai
  128:8     warning    Line exceeds the limit of 145 characters                max-ler
  143:1     error      Line contains trailing whitespace                       no-trai
  143:8     warning    Line exceeds the limit of 145 characters                max-ler
  152:1     error      Line contains trailing whitespace                       no-trai
  152:4     warning    Line exceeds the limit of 145 characters                max-ler
  197:1     error      Line contains trailing whitespace                       no-trai
  200:1     error      Line contains trailing whitespace                       no-trai
  215:1     error      Line contains trailing whitespace                       no-trai
  224:1     error      Line contains trailing whitespace                       no-trai
  324:1     error      Line contains trailing whitespace                       no-trai
  342:1     error      Line contains trailing whitespace                       no-trai
  448:1     error      Line contains trailing whitespace                       no-trai
  555:2     error      Line contains trailing whitespace                       no-trai
  555:8     warning    Line exceeds the limit of 145 characters                max-ler
  568:1     error      Line contains trailing whitespace                       no-trai
  571:1     error      Line contains trailing whitespace                       no-trai
  602:1     error      Line contains trailing whitespace                       no-trai
  615:1     error      Line contains trailing whitespace                       no-trai

✖ 19 errors, 10 warnings found.
```

7.3 Surya

Surya is an utility tool for smart contract systems. It provides a number of visual outputs and information about structure of smart contracts. It also supports querying the function call graph in multiple ways to aid in the manual inspection and control flow analysis of contracts.

Below is a complete list of functions with their visibility and modifiers:

Contract	Type	Bases		
└	Function Name	Visibility	Mutability	Modifiers
NodeRegistry	Implementation			
└	<Constructor>	Public ❗	🛑	
└	convict	External ❗	🛑	NO❗
└	registerNode	External ❗	💵	NO❗
└	registerNodeFor	External ❗	💵	NO❗
└	removeNodeFromRegistry	External ❗	🛑	onlyActiveState
└	returnDeposit	External ❗	🛑	NO❗
└	revealConvict	External ❗	🛑	NO❗
└	transferOwnership	External ❗	🛑	onlyActiveState
└	unregisteringNode	External ❗	🛑	onlyActiveState
└	updateNode	External ❗	💵	onlyActiveState
└	totalNodes	External ❗		NO❗
└	calcProofHash	Internal 🔒		
└	checkNodeProperties	Internal 🔒		
└	registerNodeInternal	Internal 🔒	🛑	
└	unregisterNodeInternal	Internal 🔒	🛑	
└	removeNode	Internal 🔒	🛑	
BlockhashRegi	Implementation			

Contract	Type	Bases		
stry				
└	<Constructor>	Public ❗	🛑	
└	searchForAvailableBlock	External ❗		NO ❗
└	recreateBlockheaders	Public ❗	🛑	NO ❗
└	saveBlockNumber	Public ❗	🛑	NO ❗
└	snapshot	Public ❗	🛑	NO ❗
└	getParentAndBlockhash	Public ❗		NO ❗
└	reCalculateBlockheaders	Public ❗		NO ❗

Legend

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

7.4 Other Tools

Other security tools such as Slither was also used to identify problems in the smart contract.

7.5 Test Coverage

Code coverage metrics indicate the amount of lines/statements/branches that are covered by the test-suite. It's important to note that "100% test coverage" does not indicate the code has no vulnerabilities. Be aware that code coverage does not provide information about the individual test-cases quality.

A fork of the Solidity-Coverage tool was used to measure the portion of the code base exercised by the test suite, and identify areas with little or no coverage. Specific sections of the code where necessary test coverage is missing are included in the Issue Details section.

The project is using the automated testing framework provided by Truffle. The test-suite is evaluating 62 individual tests and the test-suite passed without errors. The corresponding console output can be found here.

A code coverage report was generated and is provided along other tool output. The test coverage results for `NodeRegistry.sol` can be viewed here. The test coverage results for `BlockhashRegistry.sol` can be viewed here. Please find a summary of the coverage results below.

File		Statements		Branches	
BlockhashRegistry.sol		100%	30/30	92.86%	
NodeRegistry.sol		100%	123/123	95.45%	

Appendix 1 - Disclosure

a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.
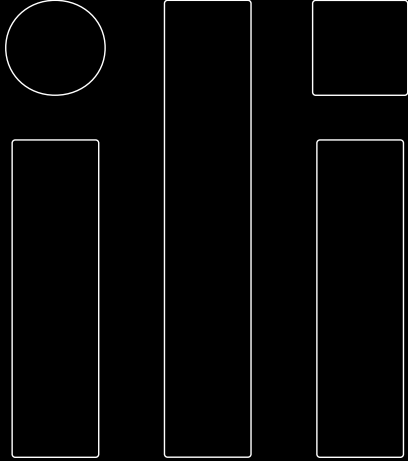
TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US

AUDITS

FUZZING

SCRIBBLE

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

CAREERS

PRIVACY
POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.