Emin Mahrt
Aeternity Establishment
Landstr. 123
FL-9495 Triesen

Security Review: Aeternity Blockchain

Confidential

Version	Changes	Date
0.1	Initial Version	2018-09-12
0.2	Protocol recommendations	2018-09-18
0.3	Source code review, tests on node partially completed	2018-10-24
0.4	Further tests on node completed	2018-11-13
0.5	Review of cryptography code in base app	2018-11-19

1 Management Summary

Aeternity is a project with the goal of creating a new blockchain. It aims to provide a number of features which are not available in existing popular blockchains like Bitcoin and Ethereum.

The development of the Aeternity protocol as well as the reference implementations for nodes and wallets have reached a pre-launch stage. Before the launch, Aeternity wants to perform a security review executed by independent security researchers. The scope of this security review includes the protocol design as well as the design and implementation of reference client software.

cnlab security ag (cnlab) is performing this security review. In order to cover the different aspects of the Aeternity blockchain, the review has been divided into several steps:

- Aeternity protocol design
- Node engineering and implementation (Epoch)
- Wallet engineering and implementation

At this point in time, the first step of the security review and most tests on the reference node (Epoch) have been concluded. Tests for the wallet are still ongoing.

1.1 Protocol Design

We arrive at the conclusion that the overall protocol design of the Aeternity protocol provides a solid basis for a blockchain network.

During our review, **we spotted a problem with the resolution of state channels**. We think that the issue can be addressed in a way that does not endanger the overall protocol.

Furthermore, we formulated a few recommendations in order to improve the network security even further. Some of these recommendations are guidelines for the implementation of the protocol. They have to be followed during the development of both the reference node and the third-party clients.

1.2 “Epoch” Node Reference Implementation

Most planned technical tests and a source code reviews have been concluded. No issues of “high” risk have been found. We found several issues of “medium” and “low” risk which are listed at the end of this report. A final conclusive statement will be provided after the pending tests on the node have been completed.

We arrive at the conclusion that the reference implementation “Epoch” for an Aeternity node provides a good basis for the network.

1.3 Wallet Implementation

During the security review, cnlab has audited the cryptography-related section of the source code in the base app. The base app is designed as an easy-to-use smartphone app, implemented in “progressive web app” technology. Note that this app is meant for managing small amounts of money for daily use. The project recommends more sophisticated solutions for the management of large assets.

During our review, we found and reported **a problem with the storage of wallet keys**. This issue has been fixed by Aeternity in the meantime.

We arrive at the conclusion that the fixed version of the base app provides adequate security for the purpose of an easy-to-use everyday wallet.

1.4 About this report

This report will be updated and completed according to the progress of the review process.

In the sequel, we provide details on our investigation and on our results.

Contents

2 Introduction

2.1 Goal

Aeternity is a project with the goal of creating a new blockchain. This blockchain is inspired by already existing blockchain projects, most notably Bitcoin and Ethereum.

For the scope of this document, we assume that basic terminology related to blockchain technology is known to the user. Terms that do not exist in the area of Bitcoin or Ethereum (or are used differently) will be defined in this document.

The motivation of creating a new own blockchain is the implementation of new features and concepts that are not available in existing blockchains. Features announced by the Aeternity project include:

- Mining based on Cuckoo Cycles
- Bitcoin-NG consensus protocol
- Built-in naming system
- Smart Contracts
- State Channels
- Oracles

The goal of this security review is to analyze and assess the security properties of the Aeternity protocol and the reference implementations which are developed by the Aeternity project.

2.2 System Architecture

The Aeternity project distinguishes between three different types of peers in the network. The three types of peers address different tasks in the network. There is no limitation that prevents any user from operating any type of peer.

The three types are:

(Client) Node

A *node* – sometimes referred to as “full node” or “client node” – is the core of the Aeternity network. This type of node downloads the whole blockchain and copies of all transactions, contracts etc. from its peers and checks them for validity. In addition, it provides this information to wallets. Nodes communicate with other nodes in a peer-to-peer manner using the Noise protocol.

This type of node is run by Aeternity users who are not willing to trust others, but are not themselves interested in mining.

Mining Node

A *mining node* is similar to a client node with additional support for mining. Like the client node, the mining node has to retrieve all data from its peers and check them for validity in order to be able to know the current valid chain to mine on.

Wallet

The *wallet* is not a peer of the network in the classic sense. A wallet is used by Aeternity users to manage their private keys and sign transactions, contract calls etc. While a full node requires a somewhat powerful server with a reliable Internet connection and large storage, a wallet is meant to be the lightweight counterpart for mobile devices. The idea is that users can use a smartphone app as a wallet managing their keys. This enables them to perform payments from anywhere, while at the same time having their private keys in a controlled environment rather than on a node server which is often hosted by a provider and which is always connected to the network.

We consider the separation of node functionality and wallet functionality to be a good choice.

3 Project

3.1 Project Steps and Scope

In the beginning, the scope of the review was defined by Aeternity and cnlab. Because of the large size of the project and of limited time, not all aspects of the initial scope definition could be covered in the first review. During the project, Aeternity and cnlab discussed and adjusted priorities in the scope.
As the security review is covering both concept and technical implementations, it has originally been grouped into three phases.

Phase	Topics	Results
Concept review of the protocol	• Mining • Cryptography • State channels • Oracles • Smart contracts • Naming system	Report including: • Concept description • Potential problems • Recommendations for improvement
Review of node implementations	• AEVM • Noise protocol • External interfaces • Code review of critical sections • Used frameworks and libraries	Report including: • System description • Findings documenting discovered weaknesses • Recommendations for improvement
Review of wallet implementations	• Cryptography • Storage of secrets • Security of interfaces • Punctual code review • Used frameworks and libraries	Report including: • System description • Findings documenting discovered weaknesses • Recommendations for improvement

Table 1: Review phases and scope

The current coverage of the listed aspects is documented in the following chapters. Areas which are currently not covered in this document may be covered in future versions of this report or in future reviews.

3.2 Sources of Information

The information for our assessment has been gathered from project documentation, source code and interviews with project staff.

The following people have been involved in the project:

Person	Company	Areas
Emin Mahrt	Aeternity	Project
Sasha Hanse	Aeternity	Protocol design
Thomas Arts, PhD	Aeternity	Node development, protocol, threat model
Philipp Piwowarsky	Aeternity	Node development
Erik Stenman, PhD	Aeternity	Node development
Hans Svensson, PhD	Aeternity	Node development
Denis Davidyuk	Aeternity	Wallet development
Emil Wagner	Aeternity	Wallet development
Stephan Verbücheln	cnlab security ag	Project; protocol, node, wallet reviews
Zuzana Trubini, PhD	cnlab security ag	Protocol review
Paul Schöbi, PhD	cnlab security ag	Protocol review
Dominic Peisker	cnlab security ag	Node web API zests
Stefan Kunz	cnlab security ag	Noise interface tests

Table 2: List of people involved in the security review

Project documentation and source code for the reference implementations have been obtained from the Aeternity Git repositories which are hosted on Github:

Repository	Description
https://github.com/aeternity/protocol	Protocol specification and documentation
https://github.com/aeternity/aetmodel	Threat model discussing threats and countermeasures
https://github.com/aeternity/epoch	“Epoch” reference implementation of an Aeternity node
https://github.com/aeternity/aepp-base	“Base App” wallet implementation

Table 3: Aeternity repositories

Note that some of those components depend on additional repositories which are controlled by Aeternity (e.g. the “enoise” implementation of the Noise protocol in Erlang). The project also depends on external standard libraries and frameworks (e.g. Erlang runtime, NaCl/libsodium, implementation of Cuckoo proof-of-work).

All used components are open source and available for public review.

4 Protocol

This section describes different aspects of the Aeternity protocol on a conceptual level. The focus of this assessment are areas where Aeternity differs from the Bitcoin blockchain. We consider Bitcoin suitable as a point of reference since it is in live operation for several years and has been subject to discussions and reviews in the security community for some time now.

4.1 Proof-of-work with Cuckoo Cycles

As a blockchain-based protocol, Aeternity employs a mining algorithm. The project has chosen to use proof-of-work based on Cuckoo Cycles. This proof-of-work algorithm was proposed by John Tromp[1]. In contrast to pure hash-based proof-of-work as known from Bitcoin, Cuckoo mining consists of several steps:

1. Generate the block header and compute its hash value.
2. Generate a random graph from the block hash (and a random nonce) using a deterministic method.
3. During graph generation, find a cycle of a defined minimum length in the graph.
4. Hash the graph and the resulting cycle and check whether the resulting hash output meets the difficulty target.

There are several motivations for selecting this algorithm rather than classic brute-force SHA256 hashing. The representation of the graph is expected to require a large amount of memory. As searching for the cycle requires random access to many different nodes of the graph, the performance of a solver for this problem is bound by memory size and latency rather than CPU resources.

The algorithm was designed with the goal that it will not be possible to efficiently implement this algorithm in an application-specific integrated circuit (ASIC). Other proof-of-work algorithms have been used by previous blockchain projects with the same motivation, e.g. “scrypt” as proof-of-work in Litecoin. However, just as for all previously proposed proof-of-work algorithms, ASIC implementations for Cuckoo Cycle proof-of-work have already been announced as well.

Another motivation for using Cuckoo Cycles is the claim that it will consume less energy than pure SHA256-based mining. We consider this claim questionable. It can be argued that since the energy price is the main constraint for mining, lower power consumption will result in additional mining equipment. The only limitation is that there might be a different ration between equipment cost and energy cost, as compared to traditional purely hash-based proof-of-work.

[1] https://github.com/tromp/cuckoo

4.1.1 Hardness of the Cuckoo Cycle Problem

The hardness of the Cuckoo Cycle problem has not been proven by John Tromp in his whitepaper. This does not seem to be a problem, since it can be argued that the algorithm implicitly falls back to SHA256-based mining. If the generation of the graph and the search for a Cuckoo Cycle turn out to be efficiently computable, miners still have to test a large number of nonces until the final SHA256 hash meets the difficulty target. However, this assumption holds only as long as the algorithmic improvement on solving the Cuckoo-Cycle problem is publicly known. A more complex proof-of-work scheme always increases the number of potentially weak spots, possibly enabling a single party to find a shortcut. If only a single party knew a way to accelerate its proof-of-work computation significantly, it would be able to perform different kinds of attacks employing secret forks of the blockchain.

We consider Cuckoo-Cycle-based proof of work to be a suitable choice.

Recommendation 1:	The project should carefully assess whether using a new proof-of-work scheme (i.e. the Cuckoo Cycle) is worth taking the related risks, given that the expected advantages are not guaranteed.

4.2 Bitcoin-NG

Bitcoin-NG is a variation of the Bitcoin consensus protocol proposed by Emin Gün Sirer et al at the Usenix conference[2].

Bitcoin-NG reduces the on-chain confirmation latency by decoupling the block generation into two different parts: *leader election* and *transaction serialization*. The key ideas of this variation can be summarized as follows:

- There are two types of blockchain blocks: Keyblocks (for leader election) and microblocks (for transaction serialization).
- Keyblocks are generated just like Bitcoin blocks, using a mining scheme that adjusts its difficulty such that a keyblock is generated every three minutes (on average).
- Keyblocks contain the static mining reward, but no transactions and therefore no fees.
- The time between the generation of two keyblocks is called epoch.
- The miner of a keyblock is called leader of the epoch, he is identified by a public key that he stored in the keyblock.
- During the epoch, the leader generates a chain of microblocks containing new transactions. Those blocks are signed with the leader key. No proof-of-work is performed on microblocks.
- The miner of the next keyblock includes the last block hash from the microblock chain created during the previous epoch.
- The transaction fees from the microblocks are split between the two epoch leaders.
- Miners have the ability to include “proof of frauds” in order to punish dishonest leaders from previous epochs. A proof of fraud consists of a set of contradicting microblocks generated by the same epoch leader.

The main motivation for Bitcoin-NG is that microblocks can be generated much more frequently than proof-of-work blocks. This results in much shorter confirmation times for transactions.

We consider Bitcoin-NG do be a good choice providing low-latency transaction confirmation while keeping full proof-of-work ledger security as known from Bitcoin.

Care has to be taken that the private keys for key blocks are not leaked. If an attacker gets access to a keyblock key, he can use it for double-spending attempts by signing contradicting microblocks. Note that the penalty for incorrect microblocks has to be paid by the miner, not by the attacker.

Recommendation 2:	Since the private keys used for the signing of microblocks are on a server that is exposed to the Internet, we recommend that the node implementation will generate a new key for each keyblock that is mined.

[2] https://www.usenix.org/node/194907

4.3 Smart Contracts

One of the central features in Aeternity are smart contracts, developed first during the Ethereum project as a generalization and extension of Bitcoin transactions.

On the implementation level, Bitcoin transactions is in fact a small program in a computer language that encodes the conditions that have to be met in order to spend its value. This computer language is limited in a way that keeps these scripts simple to validate. It does not contain loops and is therefore not Turing-complete.

Smart contracts are an extension of the idea by using a Turing complete language. This results in the problem that the computation time for validating a transaction (now also called contract) is not bound. A user could create a transaction with loops and conditional jumps, which means that it is impossible to validate whether the program will terminate for a certain input, known in computer science as the Halting problem. And even when the program will halt, it is possible that one has to execute a large number of operations in order to validate it. In the context of blockchains, this results in a high validation cost for nodes.

In order to solve this problem, users who call the contract have to pay for computation time. The computation time for the contract call is measured in “gas”.

The following operations exist with regards to contracts:

Creating a contract

Users can post a new contract to the blockchain. For this operation, a fee based on the contract size is required, just as for any other kind of transaction. In addition, an initial call to the contract has to be performed.

Calling a contract

Users can call a contract. For this operation, a fee based on the size of the call transaction is required. In addition, the user has to provide enough gas for the execution. For every instruction in the contract language, a predefined amount of gas is required.

4.3.1 AEVM

On the Aeternity blockchain, smart contracts are not stored in form of text programs. Instead, they are compiled into bytecode programs, which are then executed in the Aeternity Virtual Machine (AEVM). The bytecode program is interpreted by the AEVM in an imperative manner, it consists of opcodes and data which are executed in order.

When the contract is called by a user, a certain amount of “gas” is required for each operation. This limits the cost that nodes and miners have for validating contracts and computing the corresponding results.

Computation time is paid for using gas instead of AE tokens. This has the advantage that miners can require a gas price reflecting the capacity of the network. The gas price is adjustable because the cost for the validation of contracts is independent from the mining difficulty.

In order to limit the computation time for the nodes that have to validate the blocks, every microblock has a “gas” limit for all calls.

4.3.2 Sofia

Aeternity has defined a high-level programming language for the purpose of writing smart contracts. This language is called Sofia.

Sofia is designed as a functional programming language. Thanks to that, it is less prone to some common programming mistakes for imperative languages. In addition, it is easier to analyze the semantics of a contract and to prove correctness.

We consider the choice of Sofia being a functional language to be an appropriate choice.

In order to generate the bytecode contract from the Sofia program, a compilation process is required. It is important that users can comprehend what a contract is doing. Since the Sofia representation of a contract is much simpler to read and analyze, we consider it desirable that users have a way to check whether a particular contract on the blockchain was built from a particular Sofia program.

Recommendation 3:	The compilation process should be designed in a way that allows for reproduceable builds.

It is to be expected that the Sofia compiler will change over time. Since it is desired to minimize the size of a contract as well as the amount of gas consumed by a call, advanced optimization techniques will play a big role in compiler development. This will likely result in the fact that different compiler versions will generate different bytecode representations from the same Sofia program. If a user wants to validate whether a contract on the blockchain was built from a particular Sofia program, he will need to use the correct compiler version in order to do this.

Recommendation 4:	The compiler should be versioned in a way that enables users to reproduce contracts that have been compiled with an older compiler version.

4.4 State Channels

State Channels are held between two nodes and enable them to exchange value off-chain, with the blockchain acting as arbiter.

Only the *channel_create* and *channel_close* transactions are required to be published to the blockchain (and therefore have to be paid for). Thus, the state channels enable the parties to transact in a fast, cheap and private way.

Each party keeps a channel state tree consisting of all channel data: accounts, contracts and contract calls.

The state of the channel is represented by the *channel_id*, root hash of the channel state tree (*state_hash*) and a *round* (strictly increasing on every update, starting at 0 on channel initialization).

The channel state is co-signed on every mutually agreed off-chain update (as a part of the off-chain transaction). This way the newest channel state can be proven and conflicts can be resolved by publishing the last state on the blockchain.

The on-chain conflict resolution is neither fast, nor cheap, nor private, however the possibility to resolve conflicts on-chain should work preventively and discourage off-chain misbehavior.

4.4.1 Channel Opening

- **To open a state channel** initiator and respondent establish an off-chain communication (e.g. TCP), exchange off-chain messages to agree on the conditions (*initiator_amount, responder_amount, lock_period, channel_reserve,* ...) and both sign the *channel_create* on-chain transaction which is then posted on-chain.
- **The channel is considered open** after the co-signed *channel_create* transaction was included in a block (and achieved minimal depth). This transaction specifies how much tokens initiator and responder do deposit in the channel (taken from their on-chain accounts) and how to handle closing disputes (*lock_period, channel_reserve* ← to be lost in case of malicious behavior).

4.4.2 Channel Updates

a) **Off-chain** (2-phase protocol):

1) One party proposes a change by sending a single-singed off-chain transaction consisting of the list of the operations to be performed on the previous state together with a new channel state (reflecting the expected outcome of these operations).
2) The receiver verifies that the new state is the valid outcome of the proposed operations, updates his local channel state tree and co-signs and returns the proposed transaction.

Each update must have a strictly increasing *round* (starting at 0 on channel initialization).

Parties participating in a state channel must be in a position to locally evaluate all transactions (including contracts via AEVM) within a state channel.

b) **On-chain:**

- **By mutual agreement** (*channel_deposit, channel_withdraw*):

Mutually agreed on-chain updates are performed by posting a co-signed on-chain transaction containing the new channel state (*channel_id, state_hash* and *round*).

- **Without agreement** (*channel_snapshot_solo, force_progress*):
    - The *channel_snapshot_solo* is a single-signed on-chain transaction containing the (supposedly) last co-signed off-chain transaction and thus reflecting the (supposedly) newest channel state (represented by *state_hash* and *round*). The *round* has to be greater than the one currently present on chain. After *channel_snapshot_solo* was included in a block, the channel cannot be closed with an older state.
    - The *channel_force_progress* is a single-signed on-chain transaction, forcing the other party to "acknowledge" the outcome of a valid contract call to an off-chain contract (applied to the supposedly last channel state):

      1. The forcing party posts a *channel_force_progress* transaction containing enough information (including the last co-signed state and a corresponding Proof of Inclusion PoI) such that an off-chain contract can be executed on-chain.

      The *channel_force_progress* transaction contains further the contract call with gas limit and gas price for the on-chain execution as well as the expected new *state_hash*.

      2. When the transaction is included in a block, the contract call is executed (a call object is added on-chain) and the gas is consumed (from the channel account of the forcing party).

      If the outcome of the contract call is consistent with the expected *state_hash* the channel state tree is updated accordingly (and is considered to be the new on-chain state of the channel) and the channel round is incremented by one.

      Otherwise the on-chain channel object is not modified (except for subtracting the gas fee from the account of the forcing party?).

      3. The other party can object by posting a newer co-signed state (e.g. via *channel_snapshot_solo*). The co-signed state with the same or greater round (posted after the *channel_force_progress*) will replace the on-chain produced one.

      As the channel round is increased by one with every *channel_force_progress* transaction it has to be prevented that a malicious party can post numerous *channel_force_progress* transactions in sequence, starting from an old state but eventually overtaking the newest off-chain round. To achieve this a *channel_force_progress* transaction can be included in a block if either the last on-chain transaction is not *channel_force_progress* or if the *channel_force_progress* transaction achieved depth specified by *lock_period*.

The proposed rules are not sufficient to prevent all potential cases of abuse of *channel_force_progress*. Consider the following example:

- Two parties Alice and Bob have operated a state channel up to a state **s** with round number 100.

- A attempts to cheat B by posting a *channel_force_progress* based on state round 99, resulting in a state **s'** with round number 100.
- The current protocol prevents A from posting another *channel_force_progress* operation. However, A can now perform a *channel_close_solo* operation on state **s'**.
- As a result, A can now post another *channel_force_progress* operation based on the state **s'**, resulting in a state **t** with round number 101.
- B is now no longer allowed to post the correct last state **s** signed by both A and B, because its round number is only 100.

Recommendation 5:	In order to avoid the described situation, the protocol should be defined in a way that a state signed by both parties always overrides a single-party operation. This should be restricted only by the state channel's *lock_period*.

4.4.3 Channel Closing

- **By mutual agreement**: One of the parties posts a co-signed on-chain transaction *channel_close_mutual* specifying how to distribute channel amount.
- **Without agreement – by on-chain enforcement**:
    1. One party posts a single-signed *channel_close_solo* transaction. This transaction has to contain enough information (including the last co-signed state and a corresponding PoI) such that the channel can be closed.
    2. The other party has some time (specified as *lock_period*) to object by posting a *channel_slash* transaction containing a newer state (as a part of a co-signed off-chain transaction) and a corresponding PoI.
    3. After *channel_close_solo* and *channel_slash* expired: One of the two parties posts *channel_settle* - specifying the distribution of the channel amount corresponding to the newest on-chain state.

  The channel is considered closed after a *channel_close_mutual* or a *channel_settle* transaction was included in a block (and achieved minimal depth). The channel amount is distributed accordingly to the on-chain accounts of initiator and respondent.

4.4.4 Channel state tree

Each block in the blockchain commits to a Patricia Merkle tree of open channels (by including its root hash), where *channel_id* specifies the path. Leaf nodes store information reflecting the current state of the given channel: *channel_id, initiator_pubkey, responder_pubkey, channel_amount, initiator_amount, responder_amount, channel_reserve, state_hash* (last published state_hash), *round* (last published round), *lock_period, closes_at* (on-chain channel closing height)*, force_blocked_until* (on-chain channel height after which a new force progress can be included in a block)*.*

4.4.5 Additional Considerations

We consider state channels to be a useful extension of the blockchain protocol. If it proves in practice that users are effectively incentivized to not act dishonestly, this extension has the potential to make the blockchain more scalable since a lot of transactions do not have to be stored in the blockchain.

One problem that comes to mind is the fact that with the current protocol design, one of the two participants has to sign a transaction first. This leaves an advantage to the second participant, since he can now agree or refuse to sign the transaction, or even decide to sign it later.

Since the state channels are meant to be opened by two parties who are communicating directly, it would be a big improvement to allow the parties to sign the transactions using an interactive scheme that avoids this asymmetric situation.

Recommendation 6:	We recommend to add an interactive signing process to the protocol in order to avoid a disadvantage for any of the parties.

4.5 Oracles

Oracles provide a possibility to include real-world data (e.g. election results) into the blockchain. This is of particular interest for creating contracts with an outcome depending on a real-world event.

Any node can create an oracle by posting an *oracle_register* transaction on the chain. Any user can query an oracle by posting an *oracle_query* transaction on the chain. The oracle operator answers the query by posting an *oracle_response* transaction, which can be used in smart contracts.

Each block commits to a Patricia Merkle tree of oracles: the oracle state tree (consisting of oracle objects and query objects). Thus, the existence of an oracle and of a query/response can be proven.

The Oracle operation is completely based on trust and reputation. The blockchain does not provide any means to prevent the oracles from posting incorrect answers.

4.5.1 Oracle Transactions

The *oracle_register* transaction specifies the address of the oracle, the query and response format definitions, the query fee (to be paid to the oracle operator), the TTL (time to live) and the transaction fee for the miner (containing a component proportional to the TTL).

The oracle operator can extend the TTL of an existing oracle by posting an *oracle_extend* transaction. At this point in time, there seems to be no operation allowing the oracle owner to change other oracle properties, e.g. the query fee.

Recommendation 7:	It should be discussed whether to replace the *oracle_extend* operation by an *oracle_update* operation which allows to change other parameters as well.

The *oracle_query* transaction contains the query in a binary format and specifies the sender and the oracle addresses, the query TTL, the response TTL, the query fee (intended for the oracle operator and locked up until the oracle responds or the TTL expires) and the transaction fee for the miner (with a component proportional to the query TTL).

The oracle operator responds to a query by posting an *oracle_response* transaction signed with the oracle account's private key. The mining fee for the response transaction is paid by the oracle, although it depends on the response TTL defined by the corresponding query.

4.5.2 Oracle State Tree

The oracle state tree contains oracle objects and oracle query objects. Oracle queries form a subtree of the corresponding oracle (as the first part of the query id consists of the oracle id).

The oracle object is created by an *oracle_register* transaction, updated with a new TTL by an *oracle_extend* transaction and deleted when its TTL expires.

The oracle query object is created by an *oracle_query* transaction and closed by an *oracle_response* transaction. Once the oracle query object is closed, it is immutable. Expired oracle query objects are deleted.

4.6 Naming System

This aspect of Aeternity will be assessed during the engineering and implementation review.

4.7 Selection of Cryptographic Primitives

The Aeternity protocol makes use of a number of different cryptographic primitives. This includes:

- Elliptic-curve-based digital signatures
- Cryptographic hash functions
- Ciphers for transport security
- Patricia Merkle trees as data structures with proof-of-inclusion capabilities

In all these areas, Aeternity is relying on published cryptographic algorithms. Some of the algorithms are government and industry standards. In addition, some algorithms designed by the researchers Dan Bernstein, Tanja Lange et al. are used. All of the algorithms have been reviewed and discussed by the global cryptographic research community.

The exact usage of different algorithms for the different use cases will be in scope of the engineering and implementation review.

5 Node

Nodes are the core components of the Aeternity network. In this section, we will assess the reference implementation for an Aeternity node called “Epoch”.

Note that even before the production launch of the Aeternity network, another node (“elixir-node”) has been implemented by a third party. elixir-node is out of scope of this assessment.

5.1 Design

The reference implementation for an Aeternity node is written in the Erlang programming language. Erlang is a functional language designed by Ericsson of Sweden as a language for scalable distributed systems. A core feature of Erlang is the distribution of tasks into a very large number (thousands and more) lightweight processes that run independently on a processor. Thus, Erlang provides a solid process segregation by design.

We consider Erlang to be a good choice for a scalable, distributed system.

The node implementation consists of different components:

- Node core
- HTTP server (Cowboy)
- Noise protocol implementation (Erlang)
- eNaCl binding to libsodium (elliptic-curve cryptography)

5.2 Miner Nodes

Miner nodes are identical to regular nodes from an implementation point of view. The mining is implemented in a separate process on the Unix server and is running outside of the Erlang runtime environment. This process is based on the Cuckoo C-language implementation by John Tromp[3].

[3] https://github.com/tromp/cuckoo

5.3 Tests on the Epoch Implementation

The reference node implemented by Aeternity is called Epoch. Epoch has two interfaces exposed to the network: The node is connected to peer nodes using the Noise protocol. The user who is operating the node can access a REST API via HTTP. Using this REST API, he can ask the node for information about blocks, transactions, etc. This API can be used by the wallet in order to gain information about the current state of the network.

At the moment, the API does not implement any advanced functionality (e.g. proof of existence for transactions). In the current setup, the user has to set up his own node or work with a node whom he fully trusts.

API Interface (HTTP/REST)

The HTTP/REST API is implemented using the Cowboy web server which is written in Erlang. There are two separate TCP ports (typically 3113 and 3013) for internal and external API functions, respectively.

The external API is available via the network and provides information about the blockchain state which is public, e.g. information about the most recent block or about a particular transaction. In the current implementation, HTTP without TLS is enabled by default. The internal API is accessible only from localhost.

The following tests have been performed in this area:

- Automatic scans and tests using web-security tools
- Manual tests on the HTTP interface
- Manual tests on the API
    - Incorrect data types and format
    - Implausible data values
    - SQL injection
    - Cross-site scripting

Strengths

- The API is not vulnerable to common attacks (SQL injection, XSS).
- Internal and external API calls are strictly separated (different TCP ports).
- The internal API is only available to “localhost”.

Weaknesses

- Server reacts to non-numeric text input with "400 - Bad Request". This response is not defined in the swagger.json file. (Finding 1)
- Server reacts to a negative number with "500 - Internal Server Error". This indicates that not all error conditions are appropriately handled. This can lead to unpredictable server behavior. (Finding 2)
- When sending a get-request to test for SQL injection vulnerabilities, the server usually replies with "400 - Bad Request". In the server response, the server sends information back to the client (e.g. "reason":"wrong public key" for HTTP GET request on /v2/accounts/). For a few requests, the server does not include any information in the response. (Finding 3)
- When sending a HTTP GET request with a string as parameter in the URL including " \" in the string, the server response is unexpected. (Finding 4)
- In the default configuration, the REST API is available through the unencrypted HTTP protocol. If wallets connect to a node using this protocol, privacy might be impacted. It is visible for attackers on the network (e.g. a public WiFi), which lookups the wallet is performing. There is also a danger that attackers modify HTTP responses, e.g. for reporting wrong balances. (Finding 5)

P2P Interface (Noise)

The node communicates with other nodes with the (encrypted) Noise protocol. Inside the Noise channel, a comparatively simple custom gossip protocol is used. Nodes exchange transactions, block headers, etc. via gossip messages. Every node collects all the objects and locally builds a blockchain representation by matching objects into the hash trees. Nodes can ask other nodes for a particular object (e.g. a transaction) by reference to the transaction ID (hash of the transaction). No complex protocol for synchronizing a full local blockchain representation is used at this point.

The Noise interface is of particular interest for two reasons: First of all, for a node to work it always has to be exposed to the network (the API could be limited in a way that restricts access to the owner's own wallet). Secondly, it will have connections with unknown and untrusted peers.

As cipher spec, "Noise_XK_25519_ChaChaPoly_BLAKE2b" has been selected. We consider this an appropriate choice.

It is important to note that the Noise implementation for Erlang was created by the Aeternity project, since no implementation in Erlang existed. All cryptographic operations are performed in the "libsodium" implementation, accessed through the "enacl" ("Erlang NaCL") wrapper. The "libsodium" library, created by Daniel Bernstein et al., is known for its design goal of providing simple APIs. The motivation is to avoid common errors which are usually caused by missing checks or by choice of inappropriate parameters.

The following tests have been performed in this area:

- Basic fuzzing of the Noise interface

Additional tests are pending in this area:

- More advanced fuzzing of the noise interface
- Fuzzing of the application protocol inside the encrypted Noise tunnel

Strengths

- The well-known Noise protocol (rather than a custom solution) is used for transport security.
- An appropriate set of ciphers has been selected.
- A renowned library is used for cryptographic operations.

Weaknesses

No weaknesses have been found in this area so far. Please note that additional tests will be performed.

5.4 Source Code (Epoch)

Cryptography

In a typical setup, an Epoch node is operated in combination with a wallet. The recommended way of operation is that the node does not have any private keys which can be used to create transactions. However, a node uses cryptography (based on libsodium) in a variety of different areas:

- Communication with other nodes (Noise)
- Verification of blocks, transactions, contracts etc. (no private key required)
- Managing and using keys for creating microblocks (mining nodes only)

In order to perform those tasks, a node typically holds two sets of private and public keys, in the source code referred to as “sign” key and “peer” key.

The **peer key** is generated on first startup of the node. It is used for the Noise handshake. The protection of this key is of limited relevance, as nodes are connecting to unknown and untrusted nodes all the time. It might however become of a higher importance if nodes start to track other nodes and their reputation based on their public keys.

The **sign key** is generated for mining. The node adds it to every keyblock candidate. In case that the node successfully mines the keyblock, the key is used to sign microblocks (following the Bitcoin-NG scheme). It is important to note that the scope of this key is limited as the node generates a new sign key for every new round of mining. However, leakage of the sign key can still result in damage. In order to make miners behave honest, other peers will look for contradicting microblocks signed with the same sign key. If such contradicting microblocks are found, the miner will be punished and will not receive his reward. This can happen for a certain period of time (defined by the block height) after the block has been mined. In order to prevent malicious parties from generating contradicting microblocks (which will sabotage a miner), the sign keys have to be kept secret for this period of time.

For management of the keys, a separate Erlang process is used. The data structures are configured in a way that prevents key leakage in stack traces and memory dumps.

The following review steps have been performed in this area:

- Review of source code regarding the use of cryptographic algorithms and parameters
- Review of source code regarding the management of keys
- Review of source code regarding the storage of keys on disk
- Walking through source code with Aeternity developers to clarify remaining questions

Strengths

- Node operation does not require private wallet keys to be stored on the node.
- The renowned “libsodium” library is used for cryptographic operations.
- Leakage of the node’s keys results in limited damage due to the limited relevance and scope of those keys.
- Keys are handled in a separate Erlang process. The data structures are configured in a way that prevents key leakage through stack traces and memory dumps.

Weaknesses

- There is no option for node operators to protect their keys (peer key and sign key) using an individual password. (Finding 6)

6 Wallet

Wallets are the end-user tools that manage user's accounts and addresses as well as all related private keys. In the current project vision, a typical wallet would be a smartphone app for iOS or Android. The current reference implementation for a wallet implements such an app.

In the long term, many kinds of different wallets are conceivable. The Aeternity project itself or third parties might implement wallets as web applications which run in a user's browser, as hardware tokens comparable to Trezor, etc.

6.1 Design

At the current state of the project, only a limited number of the planned wallet features have been implemented in the reference wallet app.

Currently implemented are:

- Management of accounts, public and private keys
- Generation of digital signatures (e.g. for transactions)
- Simple communication with a node's HTTP API

Not yet implemented are:

- Advanced communication with a node
- Processing of proofs of existence provided by nodes
- Smart contract management functionality
- State channel management functionality

It is important to note that the current implementation of a wallet does not validate any information provided by the node. Thus, a wallet requires a node to be fully trusted.

This is acceptable for a setup where an Aeternity user operates his own full node. However, on a long term, we recommend implementing more security features in order to improve the experience for users who do not operate their own nodes.

Recommendation 8:	We recommend that advanced interactive functions between wallets and nodes will be implemented soon.

6.2 Implementation

A source-code review has been performed on the “base wallet”. The review was focused on the usage of cryptographic routines. This includes both the usage of public-key cryptography for the handling of wallet keys and transactions and the usage of ciphers to protect stored keys in the app.

Strengths

- No issues have been found with the usage of public-key cryptography.

Weaknesses

- In the base app, wallet keys are encrypted with a key derived from the user's password. No minimum requirements for the password are enforced. Since the base app is a progressive web app, no additional protection (e.g. hardware keystore) is used. There is the danger that attackers brute force simple passwords. (Finding 101)
- The Base Wallet stores confidential data using AES-CTR mode. This mode has similar properties as a stream cipher. Since no MAC is computed on the data, an attacker can selectively modify the data by just XORing a modification string to the cryptogram (without knowledge of the key). (Finding 103)

7 Findings

7.1 Overview

This section lists the findings of the assessment. Findings are classified according to the related risk:

High-risk findings need to be removed urgently and as soon as possible.

Medium-risk findings need to be handled, but timing is not as critical.

Low-risk findings show room for improvement, but do not need to be fixed for a secure system.

Classification		Statistics
H	High risk	0
M	Medium risk	4
L	Low risk	4
✓	Solved	1

Table 4: Findings statistics

The following table shows the list of findings.

No.	Description	Risk
1	REST API: Undefined Response	L
2	REST API: Internal Server Error	M
3	REST API: Missing data in server reply	L
4	REST API: Inconsistent server reply	L
5	REST API: Unencrypted access to API	M
6	Epoch: Missing key protection on disk	M
101	Base Wallet: Weak key storage	M
102	Base Wallet: Insecure re-use of cipher stream	✓
103	Base Wallet: Unauthenticated encryption mode	L

Table 5: List of findings

7.2 Finding Details

No.	**1**	
Area		REST API: Undefined Response
Observation		15.10.2018 HTTP GET request to /key-blocks/height/{height} Server reacts to non-numeric text input with "400 - Bad Request". This response is not defined in the swagger.json file.
Risk	**L**	Unexpected reaction by the client.
Remark		
Recommendation		Define errors for all cases.

No.	**2**	
Area		REST API: Internal Server Error
Observation		15.10.2018 HTTP GET request to /key-blocks/height/{height} Server reacts to a negative number with "500 - Internal Server Error". This indicates that not all error conditions are appropriately handled. This can lead to unpredictable server behavior.
Risk	**M**	Unexpected node behavior. Undiscovered security-relevant conditions due to lack of meaningful error messages.
Remark		
Recommendation		Implement error handling for all error conditions.

No.	**3**	
Area		REST API: Missing data in server reply
Observation		15.10.2018 When sending a get-request to test for SQL injection vulnerabilities, the server usually replies with "400 - Bad Request". In the server response, the server sends information back to the client (e.g. "reason":"wrong public key" for HTTP GET request on /v2/accounts/). For a few requests, the server does not include any information in the response, e.g.: - /v2/accounts/'%20or%20username%20like%20'% - /v2/key-blocks/height/'%20or%20uid%20like%20'% Tested for the following paths: - /v2/accounts/ - /v2/key-blocks/height/
Risk	**L**	Undiscovered security-relevant errors due to incomplete error messages.
Remark		
Recommendation		Implement consistent error messages.

No.	**4**	
Area		REST API: Inconsistent server reply
Observation		17.10.2018 When sending a HTTP GET request with a string as parameter in the URL, e.g. to - /key-blocks/hash/{hash} - /accounts/{pubkey} and including " \" in the string, the server response looks like the following: > HTTP/1.1 404 Not Found > connection: close > content-length: 0 > date: Wed, 17 Oct 2018 13:32:45 GMT > server: Cowboy In all other cases of invalid strings, the response looks like this: HTTP/1.1 400 Bad Request > connection: close > content-length: 31 > content-type: application/json > date: Wed, 17 Oct 2018 13:34:03 GMT > server: Cowboy > > {"reason":"Invalid public key"}
Risk	**L**	Undiscovered security-relevant issues due to incomplete error handling.
Remark		
Recommendation		Implement consistent error handling.

No.	**5**	
Area		REST API: Unencrypted access to API
Observation		24.10.2018 In the default configuration, the REST API is available through the unencrypted HTTP protocol. If wallets connect to a node using this protocol, privacy might be impacted. It is visible for attackers on the network (e.g. a public WiFi), what lookups the wallet is performing. There is also a danger that attackers modify HTTP responses, e.g. reporting wrong balances.
Risk	**M**	Unauthorized access to private information, unauthorized modification of data.
Remark		The risk is only "medium" as setting up a HTTPS proxy is recommended by the project.
Recommendation		Use HTTPS as default protocol.

No.	**6**	
Area		Epoch: Missing key protection on disk
Observation		24.10.2018 At the moment, there is no option for node operators to protect their keys (peer key and sign key) using an individual password.
Risk	**M**	Unauthorized access to keys.
Remark		The risk is only "medium" as in the current setting, the node keys are of limited importance.
Recommendation		Provide an option to encrypt keys with a password. This could be implemented by prompting for the password on node launch.

No.	**101**	
Area		Base Wallet: Weak key storage
Observation		13.11.2018 In the base app, wallet keys are encrypted with a key derived from the user's password. No minimum requirements for the password are enforced. Since the base app is a progressive web app, no additional protection (e.g. hardware keystore) is used. There is the danger that attackers brute force simple passwords.
Risk	**M**	Unauthorized access to funds.
Remark		The risk is only "medium" as the base wallet is only recommended for small funds and because users can choose a strong password.
Recommendation		Require a password length which is sufficient to protect against offline brute-force attacks.

No.	**102**	
Area		Base Wallet: Insecure re-use of cipher stream
Observation		13.11.2018 The wallet uses a private key and a chaincode in order to deterministically generate keys according to the HD (hierarchical deterministic) wallet specifications. For the storage of the wallet private key and the chaincode, encryption with AES-CTR mode is used. In the current implementation, the same counter value is used for encryption of the two data fields. As a result, the private key can be decrypted by an attacker without knowing the key by computing: privkey = chaincode XOR encPrivkey XOR encChaincode Note that the chaincode is considered public information according to the HD wallet specifications. 16.11.2018 The implementation has been changed to use different counter values. The problem is therefore solved.
Risk	✓	Unauthorized access to funds.
Remark		
Recommendation		Make sure that the implementation uses a different counter value for each data set.

No.	**103**	
Area		Base Wallet: Unauthenticated encryption mode
Observation		19.11.2018 The Base Wallet stores confidential data using AES-CTR mode. This mode has similar properties as a stream cipher. Since no MAC is computed on the data, an attacker can selectively modify the data by just XORing a modification string to the cryptogram (without knowledge of the key).
Risk	**L**	Unauthorized modification of encrypted data.
Remark		The risk is considered only "low" since only random keys are stored with this method. Note that this can become a problem if structured data is encrypted with the same method.
Recommendation		Use a MAC or an authenticated mode of operation for the data encryption in order detect any data modification.