Zellic

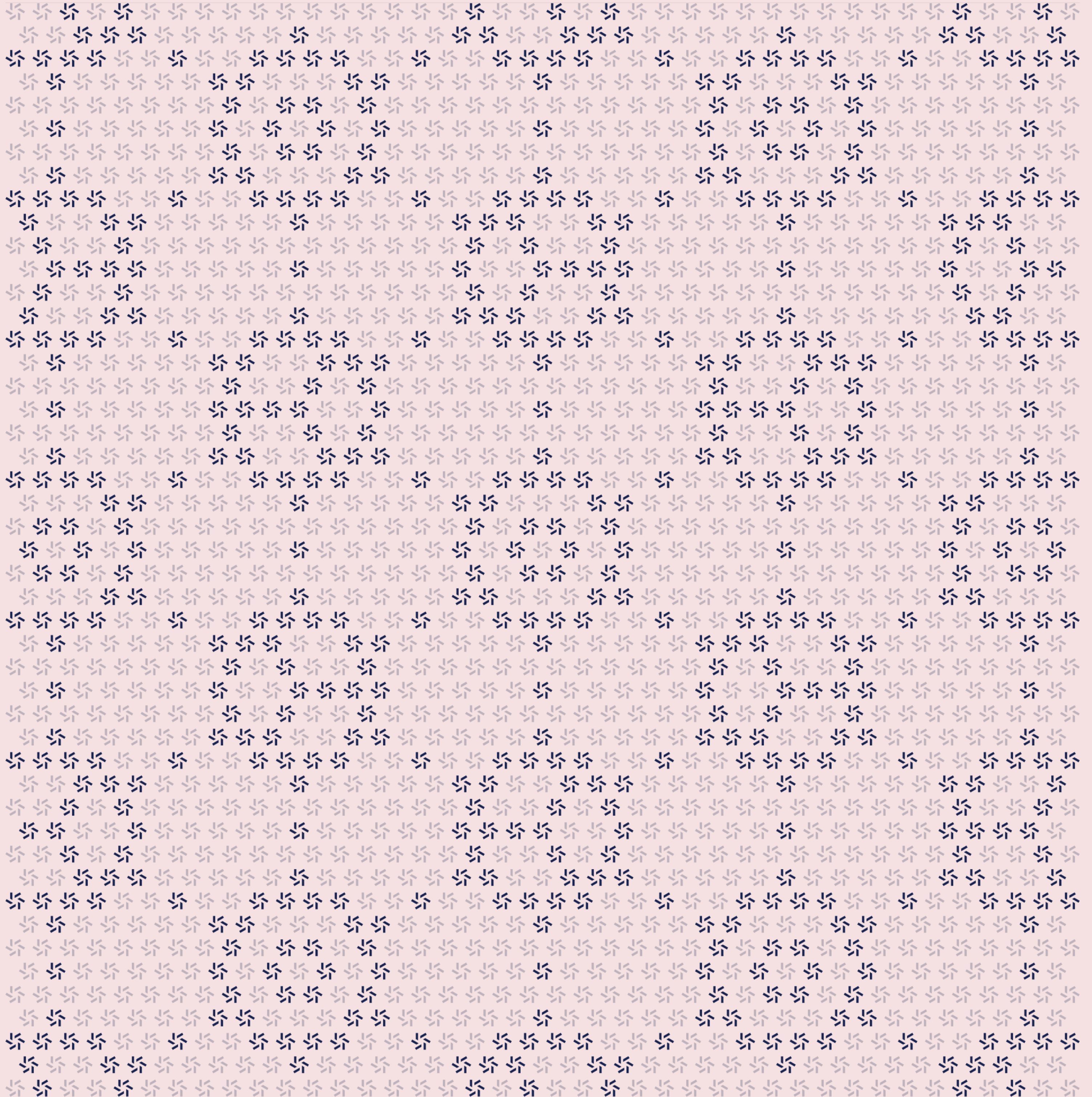
Prepared for
Dr. Elias
TON

Prepared by
William Bowling
Ulrich Myhre
Junyi Wang
Zellic

November 21, 2023

Tonlib

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Executive Summary **4**

1.1. Goals of the Assessment 5

1.2. Non-goals and Limitations 5

1.3. Results 5

2. Introduction **6**

2.1. About Tonlib 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Missing proof check for `blocks.getShards` 11

3.2. Missing proof check for Emulator transactions 13

3.3. The handler for `blocks_lookupBlock` does not check proof 15

3.4. Returned block header not checked 17

3.5. Blank block header returned on error 19

3.6. Directory traversal in `KeyValueDir` 21

3.7. Creating a raw query does not set the destinations 23

3.8. The `smc_getLibraries` handler lacks check for all libraries received 25

3.9. The `buffer_to_hex` utility function reverses nibbles 26

3.10. Hardcoded constants used without a name 27

3.11. The `query.forget` handler does not forget the query 29

3.12. Missing `check` call in `RemoteRunSmcMethod` 30

4. Discussion 31

4.1. Uncompiled and unused files 32

4.2. Test coverage 32

5. Threat Model 32

5.1. Messaging 33

5.2. Module: SimpleEncryption.cpp 33

5.3. Module: TonlibClient.cpp 34

5.4. Module: tonlib-cli.cpp 37

6. Assessment Results 37

6.1. Disclaimer 38

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for TON from October 16th to November 17th, 2023. During this engagement, Zellic reviewed Tonlib's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a malicious message from a lite server cause a stack overflow or memory leak?
- Is every message received from a lite server validated?
- Are the proofs received from a lite server checked correctly?

1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3. Results

During our assessment on the scoped Tonlib classes, we discovered 12 findings. No critical issues were found. One finding was of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for TON's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	5
Informational	4

During our assessment on the scoped Tonlib classes, we discovered 12 findings. No critical issues were found. One finding was of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature. TON acknowledged and addressed all identified findings, with the exception of finding 3.7. ↗, which has been acknowledged.

2. Introduction

2.1. About Tonlib

Tonlib is a key library for TON (The Open Network) network communications used by clients, lite servers, validators, and more.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the classes.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an “Informational” finding higher than a “Low” finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients’ threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped classes itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Tonlib Classes

Repository	https://github.com/ton-blockchain/ton ↗
Version	ton: `469fb08c49f5f526c0cb65c939171bb0f7e5a53e`
Program	tonlib/*
Type	CPP
Platform	TON

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of 11 person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

William Bowling
Engineer
vakzz@zellic.io ↗

Ulrich Myhre
Engineer
unblvr@zellic.io ↗

Junyi Wang
Engineer
Junyi@zellic.io ↗

2.5. Project Timeline

October 16, 2023	Start of primary review period
October 18, 2023	Kick-off call
November 16, 2023	End of primary review period

3. Detailed Findings

3.1. Missing proof check for `blocks.getShards`

Target	**tonlib/TonlibClient.cpp**		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When requesting block shards using the `blocks.getShards` request, the lite server returns a `proof_` as well as `data_` containing the shard configuration:

```
td::Status TonlibClient::do_request(const tonlib_api::blocks_getShards&
    request,

    td::Promise<object_ptr<tonlib_api::blocks_shards>>&& promise)
    {
  TRY_RESULT(block, to_lite_api(*request.id_))
  client_.send_query(
    ton::lite_api::liteServer_getAllShardsInfo(std::move(block)),

    promise.wrap([](lite_api_ptr<ton::lite_api::liteServer_allShardsInfo>&&
    all_shards_info)
                    -> td::Result<object_ptr<tonlib_api::blocks_shards>> {
    td::BufferSlice proof = std::move((*all_shards_info).proof_);
    td::BufferSlice data = std::move((*all_shards_info).data_);
    if (data.empty()) {
        return td::Status::Error("shard configuration is empty");
    } else {
        auto R = vm::std_boc_deserialize(data.clone());
        if (R.is_error()) {
        return R.move_as_error_prefix("cannot deserialize shard configuration:
    ");
        }
        auto root = R.move_as_ok();
        block::ShardConfig sh_conf;
        if (!sh_conf.unpack(vm::load_cell_slice_ref(root))) {
        return td::Status::Error("cannot extract shard block list from shard
    configuration");
        } else {
        auto ids = sh_conf.get_shard_hash_ids(true);
        tonlib_api::blocks_shards shards;
```

```
            for (auto id : ids) {
                auto ref = sh_conf.get_shard_hash(ton::ShardIdFull(id));
                if (ref.not_null()) {
                shards.shards_.push_back(to_tonlib_api(ref->top_block_id()));
                }
            }
            return
        tonlib_api::make_object<tonlib_api::blocks_shards>(std::move(shards));
            }
        }
        }));
      return td::Status::OK();
  }
```

The issue is that the `td::BufferSlice proof` variable is never used to check that the proof is valid and the `data` is correct.

Impact

A lite server can return an empty `proof_` and whatever shard information they wish in `data_`; the returned data does not have to match the block requested by the client.

Recommendations

The `proof` should be checked to ensure that it is the value for the `data` returned and that it matches the block requested by the client.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit `6cd9bfdc` ↗.

3.2. Missing proof check for Emulator transactions

Target	**tonlib/TonlibClient.cpp**		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `RunEmulator` class has a method for fetching the transactions that will later be run by the emulator:

```
td::Status get_transactions(std::int64_t lt) {
  TRY_RESULT(lite_block, to_lite_api(*to_tonlib_api(block_id_.id)));
  auto after
    = ton::lite_api::make_object<ton::lite_api::liteServer_transactionId3>(
    request_.address.addr, lt);
  auto query =
    ton::lite_api::liteServer_listBlockTransactions(std::move(lite_block),
    0b10100111, 256, std::move(after), false, false);

  client_.send_query(std::move(query), [self =
    this](lite_api_ptr<ton::lite_api::liteServer_blockTransactions>&&
    bTxes)
    {
    if (!bTxes) {
      self->check(td::Status::Error("liteServer.blockTransactions is
    null"));
      return;
    }

    std::int64_t last_lt = 0;
    for (auto& id : bTxes->ids_) {
      last_lt = id->lt_;
      if (id->account_ != self->request_.address.addr) {
        continue;
      }

      if (id->lt_ == self->request_.lt && id->hash_ == self->request_.hash)
    {
        self->incomplete_ = false;
      }
```

```
        self->transactions_.push_back({});
        self->get_transaction(id->lt_, id->hash_, [self,
      i = self->transactions_.size() - 1](auto transaction) {
      self->set_transaction(i, std::move(transaction)); });

        if (!self->incomplete_) {
          return;
        }
      }

      if (bTxes->incomplete_) {
        self->check(self->get_transactions(last_lt));
      }
    });
    return td::Status::OK();
  }
```

The issue is that the proof for the returned `liteServer_blockTransactions` object is never checked, allowing a malicious lite server to return any transactions they wish.

Impact

A malicious lite server could return invalid transactions, causing the result of the emulator to be incorrect.

Recommendations

The proof for the returned `liteServer_blockTransactions` object should be checked to ensure that it is valid and matches the expected block ID.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit 23f8e448 ↗.

3.3. The handler for `blocks_lookupBlock` does not check proof

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Medium
Likelihood	Low	Impact	Medium

Description

The function below, handling the `blocks_lookupBlock` request, does not check the proof of the returned block header.

```
td::Status TonlibClient::do_request(const tonlib_api::blocks_lookupBlock&
    request,

    td::Promise<object_ptr<tonlib_api::ton_blockIdExt>>&& promise)
    {
  client_.send_query(ton::lite_api::liteServer_lookupBlock(
                     request.mode_,

    ton::lite_api::make_object<ton::lite_api::tonNode_blockId>(
                     (*request.id_).workchain_, (*request.id_).shard_,
    (*request.id_).seqno_),
                     (td::uint64)(request.lt_),
                     (td::uint32)(request.utime_)),

    promise.wrap([](lite_api_ptr<ton::lite_api::liteServer_blockHeader>&&
    header)
    {
                     const auto& id = header->id_;
                     return to_tonlib_api(*id);

    //tonlib_api::make_object<tonlib_api::ton_blockIdExt>(
                     // ton::tonlib_api::ton_blockIdExt(id->workchain_,
    id->)
                     //);
                 }));
  return td::Status::OK();
}
```

Impact

This allows the lite server to forge block headers and return them.

Recommendations

Check the header proof and reject the data if the proof is incorrect.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit `9fea6235` ↗.

3.4. Returned block header not checked

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Low
Likelihood	Medium	Impact	Low

Description

When requesting a block header using the `blocks.getBlockHeader` request, the lite server returns a `header_proof_`, which is used to generate the block header:

```
td::Status TonlibClient::do_request(const
    tonlib_api::blocks_getBlockHeader& request,

    td::Promise<object_ptr<tonlib_api::blocks_header>>&& promise)
    {
  TRY_RESULT(block, to_lite_api(*request.id_))
  client_.send_query(ton::lite_api::liteServer_getBlockHeader(
    std::move(block),
    0xffff),
    promise.wrap([](lite_api_ptr<ton::lite_api::liteServer_blockHeader>&&
    hdr)
    {
    auto blk_id = ton::create_block_id(hdr->id_);
    auto R = vm::std_boc_deserialize(std::move(hdr->header_proof_));
    tonlib_api::blocks_header header;
    if (R.is_error()) {
        LOG(WARNING) << "R.is_error() ";
    } else {
        auto root = R.move_as_ok();
        try {
        ton::RootHash vhash{root->get_hash().bits()};
        auto virt_root = vm::MerkleProof::virtualize(root, 1);
        if (virt_root.is_null()) {
            LOG(WARNING) << "virt root is null";
        } else {
            std::vector<ton::BlockIdExt> prev;
            ton::BlockIdExt mc_blkid;
            bool after_split;
            auto res = block::unpack_block_prev_blk_ext(virt_root, blk_id,
    prev, mc_blkid, after_split);
```

```
            if (res.is_error()) {
            LOG(WARNING) << "res.is_error() ";
            } else {
            block::gen::Block::Record blk;
            block::gen::BlockInfo::Record info;
            if (!(tlb::unpack_cell(virt_root, blk)
    && tlb::unpack_cell(blk.info, info))) {
                LOG(WARNING) << "unpack failed";
            } else {
                header.id_ = to_tonlib_api(blk_id);
                header.global_id_ = blk.global_id;
                header.version_ = info.version;
                header.flags_ = info.flags;
// [...]
```

The issue is that the `vhash` of the returned proof is created but never checked to ensure that it matches the requested block root hash.

Impact

A lite server can return any block header they wish so long as they generate a valid proof for it; it does not have to match the block requested by the client.

Recommendations

The `vhash` of the returned proof should be checked to ensure that it matches the requested block root hash, similar to other places in the codebase:

```
 ton::RootHash vhash{virt_root->get_hash().bits()};
if (ton::RootHash{virt_root->get_hash().bits()} != block.root_hash_) {
    return td::Status::Error("block header has incorrect root hash");
}
```

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit `fabeeb07` ↗.

3.5. Blank block header returned on error

Target	**tonlib/TonlibClient.cpp**		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When requesting a block header using the `blocks.getBlockHeader` request, if there are any errors decoding the response from the lite server, an uninitialized header object is still returned instead of an error:

```
td::Status TonlibClient::do_request(const
    tonlib_api::blocks_getBlockHeader& request,

    td::Promise<object_ptr<tonlib_api::blocks_header>>&& promise)
    {
  TRY_RESULT(block, to_lite_api(*request.id_))
  client_.send_query(ton::lite_api::liteServer_getBlockHeader(
    std::move(block),
    0xffff),
    promise.wrap([](lite_api_ptr<ton::lite_api::liteServer_blockHeader>&&
    hdr)
    {
    auto blk_id = ton::create_block_id(hdr->id_);
    auto R = vm::std_boc_deserialize(std::move(hdr->header_proof_));
    tonlib_api::blocks_header header;
    if (R.is_error()) {
        LOG(WARNING) << "R.is_error() ";
    } else {
        auto root = R.move_as_ok();
        try {
        // [...]
        } catch (vm::VmError& err) {
        auto E = err.as_status(PSLICE() << "error processing header for
    " << blk_id.to_str() << " :");
        LOG(ERROR) << std::move(E);
        } catch (vm::VmVirtError& err) {
        auto E = err.as_status(PSLICE() << "error processing header for
    " << blk_id.to_str() << " :");
        LOG(ERROR) << std::move(E);
```

```
        } catch (...) {
        LOG(WARNING) << "exception catched ";
        }
      }
      return
      tonlib_api::make_object<tonlib_api::blocks_header>(std::move(header));
      }));
    return td::Status::OK();
  }
```

Impact

A client requesting a block header could receive a blank header object instead and not be aware that there was an error processing the response from the lite server.

Recommendations

If there is an error processing the response from the lite server, an appropriate `td::Status::Error` should be returned instead of an uninitialized header object.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit 28051ce1 ↗.

3.6. Directory traversal in `KeyValueDir`

Target	tonlib/KeyValue.cpp		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

The `KeyValueDir` class implements a key-value store backed by a directory on disk. When adding or setting a value, the key is used as the file name and the path generated using the `to_file_path` helper:

```
  td::Status add(td::Slice key, td::Slice value) override {
    auto path = to_file_path(key.str());
    if (td::stat(path).is_ok()) {
      return td::Status::Error(PSLICE() << "File " << path << "already
    exists");
    }
    return td::atomic_write_file(path, value);
  }

  td::Status set(td::Slice key, td::Slice value) override {
    return td::atomic_write_file(to_file_path(key.str()), value);
  }

// [...]

 private:
  std::string directory_;

  std::string to_file_path(std::string key) {
    return directory_ + TD_DIR_SLASH + key;
  }
```

The issue is that the `key` may contain `../`, causing the file to be written outside the intended directory.

Impact

If the `key` is ever controllable by a malicious actor, it could be used to write files to arbitrary locations on the file system.

Currently, all calls to `set` or `add` that have controllable keys are part of the configuration provided by the client using TonLib and not influenced by a remote actor.

Recommendations

The final path should be checked to ensure that it is within the intended directory.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit `4df4fa20` ↗.

3.7. Creating a raw query does not set the destinations

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Low
Likelihood	N/A	Impact	Low

Description

The `raw.createQuery` handler creates and registers a new raw query based on the request:

```
td::Status TonlibClient::do_request(const tonlib_api::raw_createQuery&
    request,

    td::Promise<object_ptr<tonlib_api::query_info>>&& promise)
    {
  if (!request.destination_) {
    return TonlibError::EmptyField("destination");
  }
  TRY_RESULT(account_address,
    get_account_address(request.destination_->account_address_));

  td::optional<ton::SmartContract::State> smc_state;
  if (!request.init_code_.empty()) {
    TRY_RESULT_PREFIX(code, vm::std_boc_deserialize(request.init_code_),
    TonlibError::InvalidBagOfCells("init_code"));
    TRY_RESULT_PREFIX(data, vm::std_boc_deserialize(request.init_data_),
    TonlibError::InvalidBagOfCells("init_data"));
    smc_state = ton::SmartContract::State{std::move(code),
    std::move(data)};
  }
  TRY_RESULT_PREFIX(body, vm::std_boc_deserialize(request.body_),
    TonlibError::InvalidBagOfCells("body"));

  td::Promise<td::unique_ptr<Query>> new_promise =
      promise.send_closure(actor_id(this),
    &TonlibClient::finish_create_query);

  make_request(int_api::GetAccountState{account_address,
    query_context_.block_id.copy(), {}},
    new_promise.wrap([smc_state = std::move(smc_state),
    body = std::move(body)](auto&& source) mutable {
```

```
        Query::Raw raw;
        if (smc_state) {
          source->set_new_state(smc_state.unwrap());
        }
        raw.new_state = source->get_new_state();
        raw.message_body = std::move(body);
        raw.message =
            ton::GenericAccount::create_ext_message(source->get_address(),
      raw.new_state, raw.message_body);
        raw.source = std::move(source);
        return td::make_unique<Query>(std::move(raw));
      }));
    return td::Status::OK();
  }
```

This issue is that the `destinations` field of the raw query is never set, even though the `destination` is part of the request. The `destinations` of the raw query are used when estimating the fee for the query, so if the `destinations` are not set, then the fee will be incorrect.

Impact

If a client used `raw.createQuery` and then tries to get an estimate of the fee for it using `query.estimateFees`, the returned estimate will not include any `destination_fees`.

Recommendations

Set the `destinations` field of the raw query when creating it to the `destination` from the request.

Remediation

This issue has been acknowledged by TON.

> Currently raw.createQuery+query.estimateFees calculates fee of external message only on source account (which is destination field of request). It is not supposed to emulate transaction on destination account.

3.8. The `smc_getLibraries` handler lacks check for all libraries received

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Low
Likelihood	N/A	Impact	Low

Description

In the following function, there is no check that all the requested libraries are received. The lite server can return an incomplete list, and the request would complete successfully.

```
td::Status TonlibClient::do_request(const tonlib_api::smc_getLibraries&
    request,

    td::Promise<object_ptr<tonlib_api::smc_libraryResult>>&& promise)
    {
```

Impact

Potentially, there can be unintuitive error messages. However, as no code uses this method at this moment, it does not have any impact. If this method is used, then it may result in an unintuitive error message when the requested library is used but missing.

Recommendations

Fail or succeed this request depending on whether all requested libraries are returned.

Remediation

This issue has been acknowledged by TON, and fixes were implemented in the following commits:

- `1d955971` ↗
- `7a228d65` ↗

3.9. The `buffer_to_hex` utility function reverses nibbles

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The `buffer_to_hex` function in tdutils/td/utils/misc.cpp reverses the nibbles of the byte buffer passed in.

```
string buffer_to_hex(Slice buffer) {
  const char *hex = "0123456789ABCDEF";
  string res(2 * buffer.size(), '\0');
  for (std::size_t i = 0; i < buffer.size(); i++) {
    auto c = buffer.ubegin()[i];
    res[2 * i] = hex[c & 15];
    res[2 * i + 1] = hex[c >> 4];
  }
  return res;
}
```

Impact

Exporting keys from the CLI will request the password for a public key but print the public key scrambled in the manner described above. There are also a number of log messages using the incorrect print.

Recommendations

Fix the above function by reversing the order of the nibbles.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit b37a7613 ↗.

3.10. Hardcoded constants used without a name

Target	tonlib/TontlibClient.cpp		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

In the following code, around line 2876, the type ID is directly hardcoded.

```
if (type == 0 || type == 0x2167da4b) {
  td::Status status;
```

In the following code, around line 5288, the mode bit flags are hardcoded.

```
bool check_proof = request.mode_ & 32;
bool reverse_mode = request.mode_ & 64;
bool has_starting_tx = request.mode_ & 128;
```

```
if (mode & 4 && !tvalue->get_hash().bits().equals(bTxes->ids_[count]-
    >hash_.bits(), 256))
    {
    return td::Status::Error("Couldn't verify proof (hash)");
}
if (mode & 2 && cur_trans != td::BitArray<64>(bTxes->ids_[count]->lt_)) {
    return td::Status::Error("Couldn't verify proof (lt)");
}
if (mode & 1 && cur_addr != bTxes->ids_[count]->account_) {
    return td::Status::Error("Couldn't verify proof (account)");
}
```

Impact

The constants could be considered unintuitive to developers unfamiliar with the code.

Recommendations

Use enumerations or name constants instead.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit b65b3294 ↗.

3.11. The `query.forget` handler does not forget the query

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The `query.forget` request handler in TonlibClient checks to ensure that the requested query exists, but it does not remove it from the map:

```
td::Status TonlibClient::do_request(tonlib_api::query_forget& request,

    td::Promise<object_ptr<tonlib_api::ok>>&& promise)
    {
  auto it = queries_.find(request.id_);
  if (it == queries_.end()) {
    return TonlibError::InvalidQueryId();
  }
  promise.set_value(tonlib_api::make_object<tonlib_api::ok>());
  return td::Status::OK();
}
```

Instead, it leaves the query in the map and returns an `OK` response.

Impact

Long-running clients may wish to remove queries after they have been completed to reduce memory usage, but even after calling `query.forget`, the query will remain in the map.

Recommendations

Remove the query from the map if `query.forget` is called.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit 48eb64ad ↗.

3.12. Missing `check` call in `RemoteRunSmcMethod`

Target	tonlib/TonlibClient.cpp		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

When `RemoteRunSmcMethod` is used without a specific block, the latest block is fetched and used:

```
void with_last_block(td::Result<LastBlockState> r_last_block) {
  check(do_with_last_block(std::move(r_last_block)));
}

td::Status with_block_id() {
  TRY_RESULT(method_id, query_.args.get_method_id());
  TRY_RESULT(serialized_stack, query_.args.get_serialized_stack());
  client_.send_query(
      //liteServer.runSmcMethod mode:# id:tonNode.blockIdExt
    account:liteServer.accountId method_id:long params:bytes =
    liteServer.RunMethodResult;
      ton::lite_api::liteServer_runSmcMethod(
          0x1f, ton::create_tl_lite_block_id(query_.block_id.value()),
          ton::create_tl_object<ton::lite_api::liteServer_accountId>(
            query_.address.workchain, query_.address.addr),
          method_id, std::move(serialized_stack)),
      [self = this](auto r_state) {
    self->with_run_method_result(std::move(r_state)); },
      query_.block_id.value().id.seqno);
  return td::Status::OK();
}

td::Status do_with_last_block(td::Result<LastBlockState> r_last_block) {
  TRY_RESULT(last_block, std::move(r_last_block));
  query_.block_id = std::move(last_block.last_block_id);
  with_block_id();
  return td::Status::OK();
}

void start_up() override {
  if (query_.block_id) {
```

```
      check(with_block_id());
    } else {
      client_.with_last_block(
          [self = this](td::Result<LastBlockState> r_last_block) {
      self->with_last_block(std::move(r_last_block)); });
    }
  }
```

The issue is that the call to `with_block_id` from `do_with_last_block` is missing a surrounding `check()`. If `with_block_id` returns an error (for example, if no method ID is supplied), then the error will be ignored and not progress any further as the promise will never be fulfilled or `stop` will never be called.

Impact

If `RemoteRunSmcMethod` is used without a specific block and an error occurs in `with_block_id`, the client will not receive the appropriate error and will have to wait for the actor to time-out without knowing why.

Recommendations

The call to `with_block_id` should be wrapped in a `check` to automatically propagate any errors.

Remediation

This issue has been acknowledged by TON, and a fix was implemented in commit 09dae07f ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Uncompiled and unused files

The following files are not included in the CMake build and are never compiled or used:

- ./tonlib/CellString.cpp
- ./tonlib/CellString.h
- ./tonlib/ClientActor.cpp
- ./tonlib/ClientActor.h
- ./tonlib/GenericAccount.cpp
- ./tonlib/GenericAccount.h
- ./tonlib/TestGiver.cpp
- ./tonlib/TestGiver.h
- ./tonlib/TestWallet.cpp
- ./tonlib/TestWallet.h
- ./tonlib/Wallet.cpp
- ./tonlib/Wallet.h

If they are intended to be used, they should be added to the CMake build so that they are built and tested; otherwise, they should be removed.

This issue has been acknowledged by TON, and a fix was implemented in commit `9e56af60` ↗.

4.2. Test coverage

In our assessment of the test suite, we observed that while it provides adequate coverage for the files under `tonlib/keys`, the remaining paths appear to be undertested or not covered at all.

For example, the TonlibClient is a critical component of the system used as a gateway between the client SDKs and the lite servers. While there are unit tests to check some basic requests and operations, the majority of the classes and requests have no test coverage. It is essential to include tests for these to ensure the functions behave predictably under all conditions.

In our assessment, we found that enhancing test coverage for these specific areas would further bolster the reliability and resilience of the project. We recommend expanding the test suite to include test cases addressing the above points.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the classes and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Messaging

The messaging structure is defined in the documentation ↗, and the encryption algorithm is implemented exactly as the documentation states.

The TON blockchain puts no actual restrictions on the message body of internal messages, but it is recommended to follow something similar to the reference implementation. One important rationale for this is that some messages can expect a response, and the contracts will need to understand what is being responded to.

Internal messages are have an `op` field, which either identifies the method to invoke in the smart contract or one of the reserved identifiers. The `op` field should have the highest bit (MSB out of 32 bits) set to 1 if it is a response or 0 otherwise. After `op`, there is `query_id`, which is used to match queries and responses. This field can be omitted if there is no expectations of a response.

Setting `op` to the number 0 means "simple transfer message with comment" and is unencrypted. These messages can still be binary data, but then it is required to start with `0xff` as the first byte. Very long messages can be split into multiple, chained cells where the root cell has 123 bytes of the message and every subsequent cell has up to 127 additional bytes.

When `op` is `0x2167da4b`, it refers to "transfer message with encrypted comment". These messages can be chained within multiple cells, where the first cell can contain up to 35 bytes and the rest 127 bytes. There is an upper limit on 16 such chained cells and a string length of 1,024. Each cell, except for the last one, has a reference to the next cell.

There is also two reserved `op` responses, `0xfffffffe` and `0xffffffff`. A response with `op = 0xffffffff` signals the error "operation not supported", which is the response a smart contract would send if a function call does not exist. The `0xfffffffe` should be translated to "operation not allowed". Unknown responses are to be ignored to avoid creating a messaging loop.

5.2. Module: SimpleEncryption.cpp

The module SimpleEncryption implements utility functions for encrypting and decrypting data and is optionally used when smart contracts interact through so-called **internal messages**. The encryption algorithm is defined in the documentation ↗, and the current implementation is mostly the same as the documentation states.

The base cryptographic primitives used in SimpleEncryption rely on libraries like Ed25519.cpp and crypto.cpp for the heavy lifting. These are out of scope for the audit and assumed safe. Most

of them are mainly thin wrappers around OpenSSL primitives.

The currently documented and implemented algorithm is as follows:

1. A shared secret is calculated using the sender's private key and the receiver's public key.
2. Assume that the sender wallet address has `isBouncable=1` and `isTestnetOnly=0` and set `salt` to the user-friendly address ↗ representation of it.
3. Create a cryptographically secure and random prefix such that the message length is divisible by 16 (i.e., the AES block length). The prefix is at least 16, and at most 31 bytes long.
4. Calculate an HMAC over the prefixed data, using `salt` as the secret. Then calculate another HMAC over the result, using the shared secret as the next secret (using SHA-512 both times). The first 32 bytes of this becomes the AES-CBC `key`, and the next 16 bytes becomes the initialization vector (`IV`).

The data is encrypted using the `key` and `IV`, and this is sent along with the first calculated HMAC. The receiver will be able to recalculate the rest based on the shared secret. As an optimization, the sender and receiver key is XORed together and sent along with the message, so it is not necessary to look up the public key of the other.

Another mode in SimpleEncryption is for encrypting and decrypting data directly (e.g., `td::SecureString SimpleEncryption::encrypt_data(td::Slice data, td::Slice secret)`). For this function, it is crucial that the `secret` comes from a secure and secret source. It is used together with the SHA-256 hash of the prefixed data to derive the encryption key, and this hash is prefixed onto the encrypted message. It is also crucial that the message hash is verified before any attempts are done to unpad, deserialize, or otherwise parse the decrypted data, as this could leak secret data through the propagated error messages. The reference implementation does this well enough but also opts to use the message hash as the HMAC secret, which is nonstandard and would be dangerous if that data were not randomly prefixed before encryption.

5.3. Module: TonlibClient.cpp

Class: `AccountState`

The `AccountState` class contains various helper functions for dealing with a `RawAccountState`, allowing it to be easily used by other classes. On construction, it calls `guess_type` to try and determine the wallet type based on the code hash of the account state. After determining the type, the class can be used to convert the data and code cells into the appropriate Tonlib classes.

Class: `GetMasterchainBlockSignatures`

The `GetMasterchainBlockSignatures` class is used to fetch the signatures of a block. It first retrieves the latest block ID, then looks up the previous block by querying the lite server. The returned block is checked to ensure that the `seqno` is one less than the latest block, and then a proof for it is requested and validated. If valid, a proof for the latest block is requested and validated before returning the signatures from the last link of the proof chain.

Class: `GetRawAccountState`

The `GetRawAccountState` class is used to fetch the account state for an address using either a provided block ID or using the latest block. The returned account state is then validated against the requested block ID and address. It is used by various different classes for fetching the account state and exposed to clients via the `raw.getAccountState` request.

Class: `GetShardBlockProof`

The `GetShardBlockProof` class is used to get and retrieve a Merkle proof for a shardchain block by calling `liteServer.getShardBlockProof` on the lite server. All the returned links from the lite server are validated to ensure the proof is correct, then the proof for the masterchain is requested and validated. The proof is then returned to the caller.

Class: `GetTransactionHistory`

The `GetTransactionHistory` class is used by the `tonlib_api::raw_getTransactions` to fetch transactions for an account starting from an existing `transactionId`. It queries the lite server using `liteServer.getTransactions`, then validates the returned proof to ensure that the returned transactions are legitimate for the requested transaction hash.

Class: `Query`

The `Query` class is used by the TonlibClient in the `queries_` map to keep track of which query IDs have been registered, and it contains numerous helper functions for working with the raw query and to calculate the gas cost and fee estimate of the query.

Class: `RemoteRunSmcMethod`

The `RemoteRunSmcMethod` class runs an SMC method on the lite server either using the supplied block ID or the latest block if none is provided. It validates the account state from the returned run results against the requested block ID and address. It is used by the TonlibClient for performing various DNS queries.

Class: `RunEmulator`

The `RunEmulator` class is used for fetching the account state for an address just after the supplied transaction has been run. It does this by first requesting the block header for the specified address shard and verifying the proof. It then fetches the masterchain state root from the lite server using the `liteServer_getConfigAll` request and validates it against the block ID from the previous step. It then requests the account state using `GetRawAccountState`, the block transaction IDs, and then the transaction history for each of the returned IDs. Once all the transactions have been fetched, it uses the `emulator::TransactionEmulator` to emulate them to determine the state of the account and returns it.

Class: `TonlibClient`

The `TonlibClient` class is the main entry point for clients wishing to make requests to lite servers, to interact with local wallets, or to configure the Tonlib library itself. It contains all the handlers for requests generated with from the `tonlib_api.tl` and is the main class that ties together all the other classes in the library.

A client initiates a request using the `TonlibClient::request` function along with an ID and a `tonlib_api::Function`:

```
TonlibClient::request(td::uint64 id,
    tonlib_api::object_ptr<tonlib_api::Function> function)
```

If the request is a static one (for example, `tonlib_api::setLogVerbosityLevel`), then it is immediately run and the `on_result` function is called, passing the response to the `TonlibCallback` specified when the `TonlibClient` was created. For nonstatic requests, the state is checked to ensure that it is not closed or uninitialized (the `tonlib_api::init` must be the first request made to set up the library). The appropriate `do_request` is then called to handle the request, which when complete will call the `on_result` function.

Class: `TonlibQueryActor`

The `TonlibQueryActor` class is a base class that implements the `send_query`, allowing other classes to send queries to the client without needing to reimplement the `send_query` method. It is used by the following classes:

- `GuessRevisions` — Used by the `tonlib_api::guessAccountRevision` handler to fetch the account state for the provided targets and determines the different revisions based on the code hash.
- `GenericCreateSendGrams` — Used by the `tonlib_api::createQuery` handler to perform various actions such as managing DNS, restricted wallets, payment channels, and sending messages/grams.

5.4. Module: tonlib-cli.cpp

Class: `TonlibCli`

The `TonlibCli` class is the main class for the CLI that invokes the Tonlib for the user and includes some convenience features such as key management. Some of the features include the following:

- Get basic information about a lite server such as server timestamp and capabilities
- Management of lite server instances in a configuration file
- Query information about the state of the blockchain
- Call smart contract methods
- Make transactions for accounts with a saved private key
- Save, encrypt, and decrypt client keys

The attack surface of the client CLI can come from copied configuration files from untrusted sources, being invoked on behalf of an untrusted party with arbitrary input, and in an unlikely scenario, malicious lite servers attacking not Tonlib itself but the CLI.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the TON mainnet.

During our assessment on the scoped Tonlib classes, we discovered 12 findings. No critical issues were found. One finding was of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature. TON acknowledged and addressed all identified findings, with the exception of finding 3.7. ↗, which has been acknowledged.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.