APPLICATION AUDIT REPORT

for

STARKWARE

Prepared By: Shuxiao Wang

August 5, 2020

Document Properties

Client	StarkWare
Title	Application Audit Report
Target	ethSTARK
Version	1.0
Author	Jeff Liu
Auditors	Edward Lo, Xudong Shao, Jeff Liu
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	August 5, 2020	Jeff Liu	Final Release Version
1.0-rc1	August 1, 2020	Jeff Liu	Release Candidate #1
0.1	July 25, 2020	Jeff Liu	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **ethSTARK** design document and related application source code, we in the report outline our systematic approach to evaluate potential security issues in the implementation, expose possible semantic inconsistencies between implementation code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the application can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About ethSTARK

STARKs (Scalable Transparent ARguments of Knowledge) are a family of proof-systems characterized by scalability and transparency. ethSTARK implements a STARK protocol as a non-interactive protocol between a prover and a verifier. The prover sends a proof in order to convince the verifier that a certain statement is true. Usually the proven statement indicates that a desired computation on some input was executed correctly. The verifier reads the given proof in order to test the integrity of the proven statement. For an honest prover and a valid computation the verifier is guaranteed to accept the proof. Otherwise, if the prover is dishonest or the computation is compromised, it would require an infeasible amount of computation on the prover's part in order to produce a proof that the verifier will not reject.

The basic information of ethSTARK is as follows:

Table 1.1: Basic Information of ethSTARK

Item	Description
Issuer	StarkWare
Website	https://starkware.co/
Type	Crypto Related Application
Platform	C++
Audit Method	Whitebox
Latest Audit Report	August 5, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/starkware-libs/ethSTARK **(032eda9f83d419eb2eaeef79d446fb77ecc3f019)**

After fixing the issues found in this report, the final commit hash is:

- https://github.com/starkware-libs/ethSTARK **(98c3df50e124bd00124315a693bb0fae76331eb3)**

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the application is considered safe regarding the check item. For any discovered issue, we might run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given application with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented code and compare with the description in the white paper.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of applications from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant to crypto related applications, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given application, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the application. Last but not least, this security audit should not be used as an investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the ethSTARK implementation. During the first phase of our audit, we studied the application source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	5	■■■■■
Medium	2	■■
Low	0	
Informational	4	■■■■
Total	11	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple modules. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, the application is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 7 medium-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Integer Overflow in AddData	Args and Parameters	Fixed
PVE-002	Medium	Infinite Loop in the Verifier	Input Validation Issues	Fixed
PVE-003	High	OOM Vulnerability in the Verifier - #1	Input Validation Issues	Fixed
PVE-004	High	OOM Vulnerability in the Verifier - #2	Input Validation Issues	Fixed
PVE-005	High	OOM Vulnerability in the Verifier - #3	Input Validation Issues	Fixed
PVE-006	Medium	OOM Vulnerability in the Verifier - #4	Input Validation Issues	Fixed
PVE-007	Info.	Integer Overflow in GetFriExpectedDegreeBound	Input Validation Issues	Fixed
PVE-008	Info.	Missing Sanity Check while Accessing FRI Parameters	Input Validation Issues	Fixed
PVE-009	High	OOM Vulnerability in the Verifier - #5	Input Validation Issues	Fixed
PVE-010	High	OOM Vulnerability in the Verifier - #6	Input Validation Issues	Fixed
PVE-011	Info.	Enhencement to the Construction of Zero-Knowledge Proofs	Bussiness Logic Issues	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Integer Overflow in AddData

- ID: PVE-001
- Severity: Informational
- Likelihood: None
- Impact: Medium
- Target: `src/starkware/commitment_scheme/merkle/merkle.cc`
- Category: Args and Parameters [7]
- CWE subcategory: CWE-628 [5]

Description

There is a vulnerability in the commitment scheme of ethSTARK, which could be exploited by attackers to perform DoS attack.

Prover builds Merkle Trees over the series of field elements and sends the Merkle roots to the verifier.

Prover would add the elements to the merkle tree by calling `AddSegmentForCommitment`.

```
void MerkleCommitmentSchemeProver::AddSegmentForCommitment(
    gsl::span<const std::byte> segment_data, size_t segment_index) {
  ASSERT_RELEASE(
      segment_data.size() == SegmentLengthInElements() * kSizeOfElement,
      "Segment size is " + std::to_string(segment_data.size()) + " instead of the
          expected " +
          std::to_string(kSizeOfElement * SegmentLengthInElements()) + ".");
  tree_.AddData(
      segment_data.as_span<const Blake2s160>(), segment_index * SegmentLengthInElements
          ());
}
```

Listing 3.1: src/starkware/commitment_scheme/merkle/merkle_commitment_scheme.cc

`AddData` takes in the data and the index of the data, then copy the data to `nodes_` which contains the tree elements.

```
void MerkleTree::AddData(gsl::span<const Blake2s160> data, uint64_t start_index) {
  ASSERT_RELEASE(
      start_index + data.size() <= data_length_,
      "Data of length " + std::to_string(data.size()) + ", starting at " +
          std::to_string(start_index) + " exceeds the data length declared at tree
              construction, " +
          std::to_string(data_length_) + ".");
  // Copy given data to the leaves of the tree.
  VLOG(5) << "Adding data at start_index = " << start_index << ", of size " << data.size
      ();
  std::copy(data.begin(), data.end(), nodes_.begin() + data_length_ + start_index);
  // Hash to compute all internal nodes that can be derived solely from the given data.
  uint64_t cur = (data_length_ + start_index) / 2;
  // Based on the given data, we compute its parent nodes' hashes (referred to here as "
      sub_layer").
  for (size_t sub_layer_length = data.size() / 2; sub_layer_length > 0;
       sub_layer_length /= 2, cur /= 2) {
    for (size_t i = cur; i < cur + sub_layer_length; i++) {
      // Compute next sub-layer.
      nodes_[i] = Blake2s160::Hash(nodes_[i * 2], nodes_[i * 2 + 1]);
      VLOG(6) << "Wrote to inner node #" << i;
    }
  }
}
```

Listing 3.2: src/starkware/commitment_scheme/merkle/merkle.cc

However, if the `start_index` is very large, there could be an integer overflow in `start_index + data.size()`. So the first assertion check would be bypassed and there would be an Out-Of-Bounds write on the stack when calling the `std::copy`.

Specifically, if the node exports the AddSegmentForCommitment as an api, the caller could use this integer overflow issue to crash the node.

Recommendation Check whether there would be an integer overflow in the addition `start_index + data.size()`.

3.2 Infinite Loop in Verifier

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `src/starkware/commitment_scheme/merkle/merkle.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the verifier code of ethSTARK, which could be exploited by attackers to perform DoS attack.

The STARK protocol includes commitment phase and query phase. During the query phase, the verifier first computes `query_indices_` from the parameters, then verify the fri layers.

```
void FriVerifier::VerifyFri() {
  Init();
  // Commitment phase.
  {
    AnnotationScope scope(channel_.get(), "Commitment");
    CommitmentPhase();
    ReadLastLayerCoefficients();
  }

  // Query phase.
  query_indices_ = fri::details::ChooseQueryIndices(
      channel_.get(), params_->GetLayerDomainSize(params_->fri_step_list.at(0)), params_
          ->n_queries,
      params_->proof_of_work_bits);
  // Verifier cannot send randomness to the prover after the following line.
  channel_->BeginQueryPhase();

  // Decommitment phase.
  AnnotationScope scope(channel_.get(), "Decommitment");

  VerifyFirstLayer();

  // Inner layers.
  VerifyInnerLayers();

  // Last layer.
  VerifyLastLayer();
}

}
```

Listing 3.3: src/starkware/fri/fri_verifier.cc

The function `first_layer_queries_callback_` which is defined in `stark.cc` is called to verify the first layer.

```cpp
void StarkVerifier::PerformLowDegreeTest(const CompositionOracleVerifier& oracle) {
  AnnotationScope scope(channel_.get(), "FRI");

  // Check that the fri_step_list and last_layer_degree_bound parameters are consistent with the
  // oracle degree bound.
  const uint64_t expected_fri_degree_bound = GetFriExpectedDegreeBound(*params_->fri_params);
  const uint64_t oracle_degree_bound = oracle.ConstraintsDegreeBound() * params_->TraceLength();
  ASSERT_RELEASE(
      expected_fri_degree_bound == oracle_degree_bound,
      "FRI parameters do not match oracle degree. Expected FRI degree from "
      "FriParameters: " +
          std::to_string(expected_fri_degree_bound) +
          ". STARK: " + std::to_string(oracle_degree_bound) + ".");

  // Prepare FRI.
  FriVerifier::FirstLayerCallback first_layer_queries_callback =
      [this, &oracle](const std::vector<uint64_t>& fri_queries) {
        AnnotationScope scope(channel_.get(), "Virtual Oracle");
        const auto queries =
            FriQueriesToEvaluationDomainQueries(fri_queries, params_->TraceLength());
        return oracle.VerifyDecommitment(queries);
      };
  FriVerifier fri_verifier(
      UseOwned(channel_), UseOwned(extension_table_verifier_factory_),
      UseOwned(params_->fri_params), UseOwned(&first_layer_queries_callback));
  fri_verifier.VerifyFri();
}
```

Listing 3.4: src/starkware/stark/stark.cc

The table verifier stores the elements in a map and verifies the decommitment.

```cpp
template <typename FieldElementT>
bool TableVerifierImpl<FieldElementT>::VerifyDecommitment(
    const std::map<RowCol, FieldElementT>& all_rows_data) {
  // We gather the elements of each row in sequence, as bytes, and store them in a map, with the row
  // number as key.
  std::map<uint64_t, std::vector<std::byte>> integrity_map{};
  // We rely on the fact that std::map is sorted by key, and our keys are compared row-first, to
  // assume that iterating over all_rows_data is iterating over cells and rows in the natural order
  // one reads numbers in a table: top to bottom, left to right.
  const size_t element_size = FieldElementT::SizeInBytes();
  for (auto all_rows_it = all_rows_data.begin(); all_rows_it != all_rows_data.end();) {
    size_t cur_row = all_rows_it->first.GetRow();
```

```
    auto iter_bool =
        integrity_map.insert({cur_row, std::vector<std::byte>(n_columns_ * element_size)});
    ASSERT_RELEASE(iter_bool.second, "Row already exists in the map.");
    for (size_t col = 0, pos = 0; col < n_columns_; ++col, ++all_rows_it, pos += element_size) {
      ASSERT_RELEASE(all_rows_it != all_rows_data.end(), "Not enough columns in the map.");
      ASSERT_RELEASE(
          all_rows_it->first.GetRow() == cur_row,
          "Data skips to next row before finishing the current.");
      all_rows_it->second.ToBytes(
          gsl::make_span(iter_bool.first->second).subspan(pos, element_size));
    }
  }

  return commitment_scheme_->VerifyIntegrity(integrity_map);
}

}
```

Listing 3.5: src/starkware/commitment_scheme/table_verifier_impl.inl

Finally, the function `VerifyDecommitment` defined in merkle tree is called to verify the decommitment by computing the merkle root.

```
template <typename FieldElementT>
bool MerkleTree::VerifyDecommitment(
    const std::map<uint64_t, Blake2s160>& data_to_verify, uint64_t total_data_length,
    const Blake2s160& merkle_root, VerifierChannel* channel) {
  ASSERT_RELEASE(
      total_data_length > 0, "Data length has to be at least 1 (i.e. tree cannot be empty).");

  std::queue<std::pair<uint64_t, Blake2s160>> queue;
  // Fix offset of query enumeration.
  for (const auto& to_verify : data_to_verify) {
    queue.emplace(to_verify.first + total_data_length, to_verify.second);
  }

  // We iterate over the known nodes, i.e. the ones given within data_to_verify or computed from
  // known nodes, and using the decommitment nodes - we add more 'known nodes' to the pool, until
  // either we have no more known nodes, or we can compute the hash of the root.
  std::array<Blake2s160, 2> siblings = {};

  uint64_t node_index;
  Blake2s160 node_hash;
  std::tie(node_index, node_hash) = queue.front();
  while (node_index != uint64_t(1)) {
    queue.pop();
    gsl::at(siblings, node_index & 1) = node_hash;
```

```
    Blake2s160 sibling_node_hash;
    uint64_t sibling_node_index = node_index ^ 1;
    if (!queue.empty() && queue.front().first == sibling_node_index) {
      // Node's sibling is already known. Take it from known_nodes.
      VLOG(7) << "Node " << node_index << "'s sibling is already known.";
      sibling_node_hash = queue.front().second;
      queue.pop();
    } else {
      // This node's sibling is part of the authentication nodes. Read it from the channel.
      const Blake2s160 decommitment_node =
          channel->ReceiveDecommitmentNode("For node " + std::to_string(
              sibling_node_index));
      VLOG(7) << "Fetching node " << sibling_node_index << " from channel.";
      sibling_node_hash = decommitment_node;
    }
    gsl::at(siblings, sibling_node_index & 1) = sibling_node_hash;
    VLOG(7) << "Adding hash for " << node_index;
    VLOG(7) << "Hashing " << siblings[0] << " and " << siblings[1];
    queue.emplace(node_index / 2, Blake2s160::Hash(siblings[0], siblings[1]));

    std::tie(node_index, node_hash) = queue.front();
  }

  return queue.front().second == merkle_root;
}

}
```

Listing 3.6: src/starkware/commitment_scheme/merkle/merkle.cc

However, if the parameter `n_queries` provided by the prover is set to 0, the `data_to_verify` passed to `VerifyDecommitment` would be empty and queue stores the nodes would also be empty. In this case, the verfier will fall into an infinite loop since `node_index` is always 0.

Specifically, an attacker could send the input with `n_queries` as 0 to the verifier to perform a DOS attack.

Recommendation Add a check that the `n_queries` should be greater than 0.

3.3 OOM Vulnerability in Verifier - #1

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `src/starkware/fri/fri_verifier.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the verifier codes of ethSTARK, which could be exploited by attackers to perform DoS attack.

The STARK protocol includes commitment phase and query phase. At the beginning of the protocol, the verifier initilize with the provided parameters.

```
class FriVerifier {
 public:
  using FirstLayerCallback =
      std::function<std::vector<ExtensionFieldElement>(const std::vector<uint64_t>&
          queries)>;

  FriVerifier(
      MaybeOwnedPtr<VerifierChannel> channel,
      MaybeOwnedPtr<const TableVerifierFactory<ExtensionFieldElement>>
          table_verifier_factory,
      MaybeOwnedPtr<const FriParameters> params,
      MaybeOwnedPtr<FirstLayerCallback> first_layer_queries_callback)
      : channel_(UseOwned(channel)),
        table_verifier_factory_(UseOwned(table_verifier_factory)),
        params_(UseOwned(params)),
        first_layer_queries_callback_(UseOwned(first_layer_queries_callback)),
        n_layers_(params_->fri_step_list.size()) {}
    .....
}
```

Listing 3.7: src/starkware/fri/fri_verifier.h

Parameter `n_layers_` is the size of `fri_step_list`.

```
void FriVerifier::Init() {
  eval_points_.reserve(n_layers_ - 1);
  table_verifiers_.reserve(n_layers_ - 1);
  query_results_.reserve(params_->n_queries);
}
```

Listing 3.8: src/starkware/fri/fri_verifier.cc

The verifier reserse the memory for `eval_points_` and `table_verifiers_` with the size `n_layers_ - 1`. However, if the `n_layers` is set to 0 by the prover. The size `n_layers_ - 1` will be a huge number. And this will leads to Out-Of-Memory problem in the verifier.

Specifically, an attacker could send the input with `n_layers_` as 0 to the verifier to perform a DOS attack.

Recommendation Add a check that the `n_layers_` should be greater than 0.

3.4 OOM Vulnerability in Verifier - #2

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: src/starkware/fri/fri_verifier .cc
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the verifier code of ethSTARK, which could be exploited by attackers to perform DoS attack.

The STARK protocol includes commitment phase and query phase. At the beginning of the protocol, the verifier initilize with the provided parameters and reserse the memory for query_results_ with the size params_->n_queries.

```
void FriVerifier::Init() {
  eval_points_.reserve(n_layers_ - 1);
  table_verifiers_.reserve(n_layers_ - 1);
  query_results_.reserve(params_->n_queries);
}
```

Listing 3.9: src/starkware/fri/fri_verifier.cc

However, if the params_->n_queries is set to a very big number by the prover, this would lead to Out-Of-Memory problem in the verifier.

Specifically, an attacker could send the input with params_->n_queries as a very big number to the verifier to perform a DOS attack.

Recommendation Add a check on params_->n_queries.

3.5 OOM Vulnerability in Verifier - #3

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `src/starkware/fri/fri_verifier.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the verifier code of ethSTARK, which could be exploited by attackers to perform DoS attack.

The STARK protocol includes commitment phase and query phase. At the beginning of the protocol, the verifier reads the json input and initilizes with the provided parameters.

It gets the size of proof and initialize the proof vector.

```
VerifierInput GetVerifierInput() {
  JsonValue input = JsonValue::FromFile(FLAGS_in_file);
  std::string proof_hex = input["proof_hex"].AsString();
  std::vector<std::byte> proof((proof_hex.size() - 1) / 2);
  starkware::HexStringToBytes(proof_hex, proof);
  return {input["public_input"], input["proof_parameters"], proof};
}
```

Listing 3.10: src/starkware/main/rescue/rescue_verifier_main.cc

However, if the proof is empty, `proof_hex.size()- 1` would be a huge number. This would lead to Out-Of-Memory problem in the verifier since proof vector consumes lots of memory.

Specifically, an attacker could send the input with empty proof to the verifier to perform a DOS attack.

Recommendation Add a check on the proof size.

3.6 OOM Vulnerability in the Verifier - #4

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `starkware/stark/stark.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the rescue-verifier module while parsing the parameter file, which could be exploited by the attackers to perform an Out-of-Memory attack against the verifier.

The parameter file contains parameters that configure the STARK protocol. This affects the way the proof is generated by the prover and interpreted by the verifier.

```
StarkParameters::StarkParameters(
    size_t n_evaluation_domain_cosets, size_t trace_length, MaybeOwnedPtr<const Air> air,
    MaybeOwnedPtr<FriParameters> fri_params)
    : evaluation_domain(trace_length, n_evaluation_domain_cosets),
      composition_eval_domain(GenerateCompositionDomain(*air)),
      air(std::move(air)),
      fri_params(std::move(fri_params)) {
  ASSERT_RELEASE(
      IsPowerOfTwo(n_evaluation_domain_cosets), "The number of cosets must be a power of 2.");

  // Check that the fri_step_list and last_layer_degree_bound parameters are consistent with the
  // trace length. This is the expected degree in the out of domain sampling stage.
  const uint64_t expected_fri_degree_bound = GetFriExpectedDegreeBound(*this->fri_params);
  const uint64_t stark_degree_bound = trace_length;
  ASSERT_RELEASE(
      expected_fri_degree_bound == stark_degree_bound,
      "FRI parameters do not match stark degree bound. Expected FRI degree from "
      "FriParameters: " +
          std::to_string(expected_fri_degree_bound) +
          ". STARK: " + std::to_string(stark_degree_bound) + ".");
}

StarkParameters StarkParameters::FromJson(const JsonValue& json, MaybeOwnedPtr<const Air> air) {
  const uint64_t trace_length = air->TraceLength();
  const size_t log_trace_length = SafeLog2(trace_length);
  const size_t log_n_cosets = json["log_n_cosets"].AsSizeT();
  const size_t n_cosets = Pow2(log_n_cosets);
```

```
106   const Coset evaluation_domain(Pow2(log_trace_length + log_n_cosets), BaseFieldElement
          ::One());

108   FriParameters fri_params = FriParameters::FromJson(json["fri"], evaluation_domain);

110   return StarkParameters(
111       n_cosets, trace_length, std::move(air), UseMovedValue(std::move(fri_params)));
112 }
```

Listing 3.11: starkware/stark/stark.cc

The rescue-verifier module would try to initialize some configurations from the parameter json file when startup, which contains parameters for STARK / FRI protocol.

However, there is a lack of sanity check while parsing the parameter json file.

The variable `log_n_cosetes` defines the log number of cosets (line 103). rescue-verifier will compute the number of cosets (line 104) and pass it to the process to finish constructing STARK parameter (line 110).

```
std::vector<BaseFieldElement> GetCosetsOffsets(
    const size_t n_cosets, const BaseFieldElement& domain_generator,
    const BaseFieldElement& common_offset) {
  // Define result vector.
  std::vector<BaseFieldElement> result;
  result.reserve(n_cosets);

  // Compute the offsets vector.
  BaseFieldElement offset = common_offset;
  result.emplace_back(offset);
  for (size_t i = 1; i < n_cosets; ++i) {
    offset *= domain_generator;
    result.emplace_back(offset);
  }

  return result;
}

}  // namespace

EvaluationDomain::EvaluationDomain(size_t trace_size, size_t n_cosets)
    : trace_group_(trace_size, BaseFieldElement::One()) {
  ASSERT_RELEASE(trace_size > 1, "trace_size must be > 1.");
  ASSERT_RELEASE(IsPowerOfTwo(trace_size), "trace_size must be a power of 2.");
  ASSERT_RELEASE(IsPowerOfTwo(n_cosets), "n_cosets must be a power of 2.");
  cosets_offsets_ = GetCosetsOffsets(
      n_cosets, GetSubGroupGenerator(trace_size * n_cosets), BaseFieldElement::Generator
          ());
}
```

Listing 3.12: starkware/algebra/domains/evaluation_domain.cc

From the above code snippets, we know rescue-verifier will try to reserve enough room for the number of cosets (line 15). Although there are some assertions, e.g., `n_cosets` must be a power of 2 (line 34) and `n_cosets` * `trace_size` must also be a power of 2 and can devide the field size (2**61 + 20 * 2**32 + 1), a malicious attacker could still craft a large `n_cosets` to bypass these assertions, and in the end lead to a Out-of-Memory attack.

Recommendation Add sanity check on the number of cosets in the parameter file.

3.7 Integer Overflow in GetFriExpectedDegreeBound

- ID: PVE-007
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Targets: `starkware/stark/stark.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a vulnerability in the rescue-verifier module while parsing the parameter file, which could be exploited by the attackers to bypass some sanity checks and might lead to other exploitations.

The parameter file contains parameters that configure the STARK protocol. This affects the way the proof is generated by the prover and interpreted by the verifier.

```
uint64_t GetFriExpectedDegreeBound(const FriParameters& fri_params) {
  uint64_t expected_bound = fri_params.last_layer_degree_bound;
  for (const size_t fri_step : fri_params.fri_step_list) {
    expected_bound *= Pow2(fri_step);
  }
  return expected_bound;
}

}  // namespace

// ---------------------------------------------------------------------------------------------

//  StarkParameters
// ---------------------------------------------------------------------------------------------


static Coset GenerateCompositionDomain(const Air& air) {
  const size_t size = air.GetCompositionPolynomialDegreeBound();
  return Coset(size, BaseFieldElement::Generator());
}

StarkParameters::StarkParameters(
    size_t n_evaluation_domain_cosets, size_t trace_length, MaybeOwnedPtr<const Air> air,
    MaybeOwnedPtr<FriParameters> fri_params)
    : evaluation_domain(trace_length, n_evaluation_domain_cosets),
      composition_eval_domain(GenerateCompositionDomain(*air)),
      air(std::move(air)),
      fri_params(std::move(fri_params)) {
  ASSERT_RELEASE(
      IsPowerOfTwo(n_evaluation_domain_cosets), "The number of cosets must be a power of 2.");
```

```
88    // Check that the fri_step_list and last_layer_degree_bound parameters are consistent with the
89    // trace length. This is the expected degree in the out of domain sampling stage.
90    const uint64_t expected_fri_degree_bound = GetFriExpectedDegreeBound(*this->fri_params);
91    const uint64_t stark_degree_bound = trace_length;
92    ASSERT_RELEASE(
93        expected_fri_degree_bound == stark_degree_bound,
94        "FRI parameters do not match stark degree bound. Expected FRI degree from "
95        "FriParameters: " +
96            std::to_string(expected_fri_degree_bound) +
97            ". STARK: " + std::to_string(stark_degree_bound) + ".");
98  }
```

Listing 3.13: starkware/stark/stark.cc

rescue-verifier will try to initialize some configurations from the parameter json file when startup, which contains parameters for STARK / FRI protocol.

However, there is a lack of sanity check while parsing the parameter json file.

rescue-verifier would make sure the degree bound of FRI protocol specified in the parameter file will equal to trace length (line 90-97). It first calculate the FRI protocol degree bound by calling GetFriExpectedDegreeBound, which utilizes a for loop to multiply related elements in the parameter file. However, the result might be overflowed if we have following situations:

1. big number of last_layer_degree_bound
2. big number of large fri_step_list[i]

An attacker can maliciously crafts such numbers in the parameter file and bypass the assertion.

Recommendation Add sanity check while calculating the expected FRI degree bound or set an upper limit for each elements.

3.8 Missing Sanity Check while Accessing FRI Parameters

- ID: PVE-008
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `starkware/fri/fri_verifier.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a lack of sanity check in the rescue-verifier module while parsing the parameter file, which could be exploited by the attackers to cause an exception during the verifying process.

The parameter file contains parameters that configure the STARK protocol. This affects the way the proof is generated by the prover and interpreted by the verifier.

```
void FriVerifier::VerifyFri() {
  Init();
  // Commitment phase.
  {
    AnnotationScope scope(channel_.get(), "Commitment");
    CommitmentPhase();
    ReadLastLayerCoefficients();
  }

  // Query phase.
  query_indices_ = fri::details::ChooseQueryIndices(
      channel_.get(), params_->GetLayerDomainSize(params_->fri_step_list.at(0)), params_
          ->n_queries,
      params_->proof_of_work_bits);
  // Verifier cannot send randomness to the prover after the following line.
  channel_->BeginQueryPhase();

  // Decommitment phase.
  AnnotationScope scope(channel_.get(), "Decommitment");

  VerifyFirstLayer();

  // Inner layers.
  VerifyInnerLayers();

  // Last layer.
  VerifyLastLayer();
}
```

Listing 3.14: starkware/fri/fri_verifier.cc

rescue-verifier would try to initialize some configurations from the parameter json file when startup, which contains parameters for STARK / FRI protocol.

However, there is a lack of sanity check while parsing the parameter json file.

Specifically, while deciding the query indices (line 166-168), rescue-verifier will pass the first element in the `fri_step_list` without checking its length, and it will trigger an exception if the vector is empty.

The same issue could also be found in `VerifyFirstLayer()`, `VerifyInnerLayers()`, and `VerifyLastLayer()`.

Recommendation Add sanity check on the fri_step_list, make sure it's not empty.

3.9 OOM Vulnerability in the Verifier - #5

- ID: PVE-009
- Severity: High
- Likelihood: High
- Impact: Medium
- Targets: `starkware/fri/fri_verifier.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a lack of sanity check in the rescue-verifier module while parsing the parameter file, which could be exploited by the attackers to perform an Out-of-Memory attack against the verifier.

The parameter file contains parameters that configure the STARK protocol. This affects the way the proof is generated by the prover and interpreted by the verifier.

```
void FriVerifier::ReadLastLayerCoefficients() {
  AnnotationScope scope(channel_.get(), "Last Layer");
  const size_t fri_step_sum = Sum(params_->fri_step_list);
  const uint64_t last_layer_size = params_->GetLayerDomainSize(fri_step_sum);

  // Allocate a vector of zeros of size last_layer_size and fill the first last_layer_degree_bound
  // elements.
  std::vector<ExtensionFieldElement> last_layer_coefficients_vector(
      last_layer_size, ExtensionFieldElement::Zero());
  channel_->ReceiveFieldElementSpan(
      gsl::make_span(last_layer_coefficients_vector).subspan(0, params_->last_layer_degree_bound),
      "Coefficients");

  ASSERT_RELEASE(
      params_->last_layer_degree_bound <= last_layer_size,
      "last_layer_degree_bound (" + std::to_string(params_->last_layer_degree_bound) +
          ") must be <= last_layer_size (" + std::to_string(last_layer_size) + ").");

  size_t last_layer_basis_index = Sum(params_->fri_step_list);
  const Coset lde_domain = params_->GetCosetForLayer(last_layer_basis_index);

  std::unique_ptr<LdeManager<ExtensionFieldElement>> last_layer_lde =
      MakeLdeManager<ExtensionFieldElement>(lde_domain, /*eval_in_natural_order=*/false);

  last_layer_lde->AddFromCoefficients(
      gsl::span<const ExtensionFieldElement>(last_layer_coefficients_vector));
  expected_last_layer_ = ExtensionFieldElement::UninitializedVector(last_layer_size);

  last_layer_lde->EvalOnCoset(
      lde_domain.Offset(), std::vector<gsl::span<ExtensionFieldElement>>{*expected_last_layer_});
```

```
76 }
```

Listing 3.15: starkware/fri/fri_verifier.cc

During commitment phase, rescue-verifier would read the coefficients of the last layer from input. It would first compute the size of the last layer and allocate a vector for it (line 48-54).

However, there is a lack of sanity check on the calculated size.

Specifically, if the sum of `fri_step_list` is very small, we might end up with a very big `last_layer_size`
.

If rescue-verifier uses the calculated size to allocate spaces without checking, it would lead to Out-of-Memory. The same issue also applies when rescue-verifier wants to allocate spaces for `expected_last_layer_` (line 72).

Recommendation Add sanity check on the calculated size for any allocations.

3.10 OOM Vulnerability in the Verifier - #6

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Targets: `starkware/fri/fri_details.cc`
- Category: Input Validation Issues [3]
- CWE subcategory: CWE-349 [4]

Description

There is a lack of sanity check in the rescue-verifier module while parsing the parameter file, which could be exploited by the attackers to perform an Out-of-Memory attack against the verifier.

The parameter file contains parameters that configure the STARK protocol. This affects the way the proof is generated by the prover and interpreted by the verifier.

```
void FriVerifier::VerifyFirstLayer() {
  AnnotationScope scope(channel_.get(), "Layer 0");
  const size_t first_fri_step = params_->fri_step_list.at(0);
  std::vector<uint64_t> first_layer_queries =
      fri::details::SecondLayerQueriesToFirstLayerQueries(query_indices_, first_fri_step
          );
  const std::vector<ExtensionFieldElement> first_layer_results =
      (*first_layer_queries_callback_)(first_layer_queries);
  ASSERT_RELEASE(
      first_layer_results.size() == first_layer_queries.size(),
      "Returned number of queries does not match the number sent.");
  const size_t first_layer_coset_size = Pow2(first_fri_step);
  for (size_t i = 0; i < first_layer_queries.size(); i += first_layer_coset_size) {
    query_results_.push_back(fri::details::ApplyFriLayers(
        gsl::make_span(first_layer_results).subspan(i, first_layer_coset_size), *
            first_eval_point_,
        *params_, 0, first_layer_queries[i]));
  }
}
```

Listing 3.16: starkware/fri/fri_verifier.cc

The verification process consists of several phases: Commitment / Query / Decommitment, and rescue-verifier would go through each phase with information extracted from input file. After that, rescue-verifier would follow the FRI protocol and verify each layer. For the first layer, rescue-verifier would get the first element from `fri_step_list` vector and pass it to `SecondLayerQueriesToFirstLayerQueries` to get the 2nd layer queries.

```
std::vector<uint64_t> SecondLayerQueriesToFirstLayerQueries(
    const std::vector<uint64_t>& query_indices, size_t first_fri_step) {
  std::vector<uint64_t> first_layer_queries;
  const size_t first_layer_coset_size = Pow2(first_fri_step);
  first_layer_queries.reserve(query_indices.size() * first_layer_coset_size);
```

```
61   for (uint64_t idx : query_indices) {
62     for (uint64_t i = idx * first_layer_coset_size; i < (idx + 1) *
         first_layer_coset_size; ++i) {
63       first_layer_queries.push_back(i);
64     }
65   }
66   return first_layer_queries;
67 }
```

Listing 3.17: starkware/fri/fri_details.cc

However, there is a lack of sanity check on the 1st FRI step retrieved.

Specifically, if the 1st element in `fri_step_list` is a big number, e.g., 2ˆ63, rescue-verifier would have to allocate a lot of memory for `first_layer_queries` (line 60), which would lead to Out-of-Memory.

Recommendation Add sanity check or limit on each element in the fri_step_list.

3.11 Enhencement to the Construction of Zero-Knowledge Proofs

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: N/A
- Category: Business Logics [6]
- CWE subcategory: CWE-1068 [2]

Description

The Fiat-Shamir transformation is the most efficient construction of non-interactive zero-knowledge proofs, which is also used in the construction of ethSTART proofs. There are two variants of the transformation that appear in existing literature, and both variants start with the prover making a commitment. The strong variant then hashes both the commitment and the statement to be proved, whereas the weak variant hashes only the commitment. According to some publications from the academia such as this paper [1], "How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios", the difference between the two variants yields dramatically different security guarantees: in situations where malicious provers can select their statements adaptively, the weak Fiat-Shamir transformation yields unsound/unextractable proofs.

The Fiat-Shamir transformation used in ethSTARK belongs to the weak variant, that only the commitment was hashed into the proof, not the statement itself. To avoid the possible risk of the weak Fiat-Shamir transformation, we suggest to add the statement as part of the proof hash also.

4 Conclusion

In this audit, we thoroughly analyzed the ethSTARK documentation and implementation. In our opinion, the application is well-designed, and the code base is well-organized. During the audit, several medium or lower issues were found, and the quality of the implementation can be improved by resolving these issues. Please note that those identified issues have been promptly confirmed and fixed.

Meanwhile, we need to emphasize that one security audit may not discover all security issues of the audited application. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] David Bernhard etc. How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. https://eprint.iacr.org/2016/771.pdf.

[2] MITRE. CWE-1068: Inconsistency Between Implementation and Documented Design. https://cwe.mitre.org/data/definitions/1068.html.

[3] MITRE. CWE-1215: Data Validation Issues. https://cwe.mitre.org/data/definitions/1215.html.

[4] MITRE. CWE-477: Acceptance of Extraneous Untrusted Data With Trusted Data. https://cwe.mitre.org/data/definitions/349.html.

[5] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE CATEGORY: Often Misused: Arguments and Parameters. https://cwe.mitre.org/data/definitions/559.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.