Flare
Attestation Suite Fixes Review

coinspect

flare

Attestation Suite

Security Assessment

Version: v240220 Prepared for: Flare January 2024

Security Assessment

1. Executive Summary

In **December 2023**, **Flare** engaged Coinspect to perform a source code review of the Attestation Suite. The objective of the project was to evaluate the security of the off-chain components of the system, critical for the correct working of the attestation protocol.

The **Attestation Suite** as a whole is a set of off-chain programs, contracts and consensus code which aim to feed the Flare blockchain with facts about external chains. Coinspect analyzed risks arising from the codebase itself and also from the design, documentation and expected usage of the system. Operational risks and other concerns not directly related to the codebase, but relevant to the safety of the system, are outlined in the Assessment section of this report.

	✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High	2	0	0
Medium	3	1	0
Low	1	1	0
No Risk	1	0	0
Total	7	2	0

During the review, 2 high-risk issues (`ATC-22` and `ATC-28`) were found in the highest priority target: the attestations' specifications. These issues could impact users by facilitating scams. Another issue, `ATC-29`, describes how the testing approach is insufficient to ensure the correct implementation of current and future features.

Several other issues were discovered: `ATC-26` describes how consensus might never be reached in certain conditions, `ATC-30` demonstrates how the verifier servers will all eventually halt due to an ever-growing cache. `ATC-25`, `ATC-27` and `ATC-31` are low-severity issues that describe improvements in password handling and attestations responses.

2. Summary of Findings

2.2 Findings where caution is advised

These issues have been addressed, but their risks have not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating the issues or increasing their probability.

Findings with a risk of `None` pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
ATC-29	Limited tests for MCC's transaction attestation	Medium
ATC-25	Attackers can perform exhaustive search on passwords	Low

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
ATC-22	EVMTransaction specification allows rugpulls to appear legitimate	High
ATC-28	Scammers can trick users into sending payments that they can denounce as non-existent	High
ATC-23	Legacy transactions result in wasted gas	Medium

ATC-26	Consensus might never be reached	Medium
ATC-30	Proof server will eventually halt due to ever-growing cache	Medium
ATC-31	Users are led to errors by different responses in testnet and mainnet	Low
ATC-27	Missing members in the EVMTransaction attestation response	None

3. Scope

The scope of the review was limited to the off-chain components of the following five repositories at a fixed commit:

- State Connector Protocol at commit `f51f0610e3ee824647dd36ad8948ff80d41c454a`
- Multichain Client Library at commit `ca83b7555f77c6d00eebf1a43b2ff0e4f82101c2`
- Attestation client at commit `44d7c631dcbe16e79568a8f6872169450a80a098`
- Doge indexer at commit `0233408613efc362a0a6926bb805eeb4f689bad7`
- EVM verifier at commit `9a283fcb8b08034609ed82d81fc4cffa25b536d6`
- Verifier Server Template at commit `d760374f9856eb0842e77f56a2021e5f946c2628`

During the audit, Flare introduced an update to the **State Connector Protocol** and the **Attestation Client** at commits `0f6f5f19aa09e2536db6031803457cedcc8129d9` and `1f9f31552514bbf23250cd171af3529eca6c900b` respectively. These new commits were reviewed only in regard with the changes made to the `AddressValidity` attestation type. This led to the `ATC-30` finding.

Flare stated that top-most priority for the review is the attestation definitions themselves: their specification in markdown, the associated `.sol` type definition, the `.json` file from which verifier code will be generated, and the verifier code itself, are all critical to the correct working of the system.

On the other hand, the **Verifier Server Template** was established as a low priority component; requiring only a cursory review to recommend best practices. The same requirement was made referring to the CLI tool that generates code based on attestation definitions.

While researching the project, Coinspect found a **HIGH** severity issue which allowed attestors to copy other's vote without doing any work. The issue is in out-of-scope components as it arises from the interactions between the Flare consensus code and the `StateConnector.sol` contract. The issue is nevertheless included due to its impact in the **Attestation Protocol**. These out-of-scope components were not reviewed in depth.

On February 20, Flare asked Coinspect to update the Attestation client commit hash, due to a repository cleanup. Coinspect verified that the content remains the same, and the new commit is: `2e626430278919839c84ec172e9dc05d9aff26e9`.

3.1 Fixes review

On **January 22, 2024**, the Flare team provided a document and a new commit (`e6cb1df2c400a5531cfd5de8789158335fce0c80`) outlining how they addressed each of the issues described in this report.

Coinspect reviewed the changes to the code and provided feedback on how to better approach the fixes for `ATC-23` and `ATC-25`. Coinspect also found `ATC-31` was not fixed in the provided commit. On the other hand, issues `ATC-22`, `ATC-26`, `ATC-28` and `ATC-30` were correctly addressed. Issue `ATC-27` was acknowledged by Flare, but it has been reassessed as a **Recommendation**-level issue, as it poses no risk to the platform at the moment. Issue `ATC-29` has also been acknowledged. After feedback from Flare, an issue originally reported under ID `ATC-24` has been retracted.

`STC-01`, an issue originally reported in this document in an out-of-scope component, has been moved to a separate document to keep this report only about the off-chain system.

After the feedback from Coinspect, Flare provided a new commit for the attestation client (`175fa082caf32ff1999cfdda6e2ebb9a0784c5eb`) which addressed `ATC-31` and improved on the fix for `ATC-23`.

On February 20, Flare asked Coinspect to update the commit hashes aforementioned, due to a repository cleanup. Coinspect verified that the content remains the same, and the commit hashes (`original`, `new`) are:

- (`e6cb1df2c400a5531cfd5de8789158335fce0c80`, `98eef3b9afbcec07b493af0d664b551c9d8704df`)
- (`175fa082caf32ff1999cfdda6e2ebb9a0784c5eb`, `aeaf96bfae5251bb38c52dfd46397670c914db0b`)

4 Assessment

The **Attestation Suite** as a whole is a set off-chain programs, contracts and consensus code that aim to feed the Flare blockchain with facts about external chains. These facts are attested to by a set of *attestors* who are hand-picked by the Flare organization. The *attestors* vote in rounds with a commit/reveal scheme for a merkle root which condenses the set of *facts* attestors are willing to confirm happened. A merkle root is accepted if a simple majority of *attestors* commit to it. Users are responsible for then finding an *attestor* willing to provide a *proof* for the fact they are interested in, and they can then use this proof on smart contracts on the Flare network. Ultimately, being able to verify it reduces to the accepted merkle root using the State Connector contract.

The system is expected to be open: anyone can be a user. Users are expected to be other developers which create smart contracts and off-chain systems that depend on the correct verification of facts held by attestation providers. Coinspect considered this open-ended design when considering the risk and likelihood of issues: because the design-space for applications that use the oracle is near-infinite and the system is open, it is expected that even issues arising only in a specific set of circumstances are realistic and worth fixing. When possible, Coinspect has provided example of how things might go wrong; but it is important to note that the actual implications for users might be even worse than presented due to design decision that are impossible to foresee.

The system contains an indexer in charge of ingesting confirmed blocks. This is done by indexing each block after a configured amount of confirmations passed (this value depends on each chain). This design adds a delay into the system as each block will be indexed after the configured amount of confirmations multiplied by the average chain's block time. Because of this architecture, the indexer has no cleanup or delete feature as it will only process blocks after a configured amount of confirmations.

4.1 Security assumptions

Small honest majority

One of the most critical assumption the project makes is that the majority of attestors are honest and well behaved. Coinspect has already highlighted this fact in previous reports, but it bears repeating as the amount of attestors is only 9. An attacker with access to only 5 attestors' private key or a collusion of only 5 entities can provide fake information to the system. This risk was mentioned in previous reports, referencing a past exploit to a bridge based on compromising 5 out of 9 signers.

While Flare has acknowledged and somewhat mitigated this risk by the existence of *private attestation sets* (which are outside the scope of this audit); *private attestation sets* are only useful for the validators running them. What is more, the collusion or attacker can still *censor* attestations indefinitely.

Stable and consistent JSON-RPC responses from other blockchains

The system assumes that JSON-RPC responses from blockchains outside the direct control of Flare are going to be consistent in time and work correctly for a majority of attestors through time.

No incentive schema

Flare is still developing the incentive system to make it rational for attestation providers to run. This is specially important due to the costs of running an attestation provider.

4.2 Decentralization

The system is only somewhat decentralized; with only 9 attestors intended to exist at the moment. These attestors are also hand-picked by a single organization.

Due to the cost of running an attestation provider service, it is possible to envision a market opportunity for a service that provides attestation providers with information about the chains. This would severely degrade the already small number of independent information providers.

4.3 Code quality

As in previous reviews, Coinspect noted that the code has several comments with marks of `TODO` and `DANGER`, as well as several commented out and empty files. This may indicate that certain parts of the implementations still have not reached a definitive state of stability.

Some examples of these observations follow:

```
  // DANGER: How to handle this if there are a lot of transactions with
same payment reference in the interval?
```

```
    // todo: this causes async growing - this should be queued and run from
main async
```

```
        if (typeof blockHashOrHeight === "string") {
            blockHash = blockHashOrHeight as string;
            if (PREFIXED_STD_BLOCK_HASH_REGEX.test(blockHash)) {
                blockHash = unPrefix0x(blockHash);
            }
            // TODO match with some regex
        }
```

```
    public get transactionIds(): string[] {
        // TODO update block type
        // eslint-disable-next-line @typescript-eslint/ban-ts-comment
        // @ts-ignore
        // eslint-disable-next-line @typescript-eslint/no-non-null-
assertion
        return this.data.tx!.map((tx) => prefix0x(tx));
    }
```

```
  // TODO: Ready
  public async getLastConfirmedBlockNumber(): Promise<number> {
    try {
      const tipState = await this._getTipStateObject();
      return tipState.latestIndexedHeight;
    } catch {
      // TODO: Print or at least log this
      return 0;
    }
  }

// TODO: Ready
  public async getLatestBlockTimestamp(): Promise<BlockHeightSample | null>
{
    try {
      const tipState = await this._getTipStateObject();
      return {
        height: tipState.latestTipHeight,
        timestamp: tipState.timestamp,
      };
    } catch {
      // TODO: Print or at least log this
      return null;
    }
  }
```

There are also snippets with comments that contradict the implementation, leaving the desired behavior unclear:

```
              // Outputs without address do not break one-to-one condition
              if (oneToOne && !voutAmount.address && voutAmount.amount >=
BigInt(0)) {
                  oneToOne = false;
              }
```

In addition, reviewers found outdated documentation, such as the following, which mentions a non-existing `getTransactionFromCache` method:

```
// Usage:
// 1) External service should initialize relevant MCC client through
CachedMccClient wrapper class
// 2) Services should use the following call sequence
//   (a) try calling `getTransactionFromCache` or `getBlockFromCache`
//   (b) if response is null, then check `canAccept`.
//        (i) If `true` is returned, call `getTransaction` (`getBlock`)
//        (ii) if `false` is returned, sleep for a while and retry
`canAccept`. Repeat this until `true` is
//             eventually returned and then proceed with (i)
```

Also, Coinspect identified that some files are present in the codebase but are all commented out, this is the case of `verifier-config.ts`.

The code also has linter warnings and empty files, for example in:

- `src/servers/verifier-server/src/verification/address-validity.ts`

And also in the `DogeIndexerVerifier` repository, inside the `afauth/` directory:

- `serializers`
- `tests`
- `urls`
- `views`

4.5 Testing

While the code has a reasonable coverage, the testing scenarios are not enough for such a critical component of the Flare ecosystem. Most tests only exercise basic success/failure conditions. Depending on the precise repository involved, tests take the data from either real blockchain data via indexers or with a `test-data` database. Both of these data sources present problems, as adding new scenarios involve either making actual public transactions to a blockchain and having an indexer or at least node available; or modifying a `sqlite` database by hand. Important data in the database is held in a `binary blob` of compressed data, further complicating the task.

These approaches are not bad, but a clear path to add new test cases to quickly generate border scenarios is a must on these kind of projects, aiding not only in security reviews but also to make sure that bugs are not reintroduced.

Coinspect **strongly** recommends that a system to provide raw JSON data mimicking a node response as input data to tests. This way, a new test can be added simply by creating JSON data. This would work in complement with the other tests present in the project.

This risk of not being able to quickly and effectively test transactions without them being available on a public chain is highlighted in `ATC-29`. Coinspect had already highlighted deficiencies in the test system before. See `ATC-7`.

5. Detailed Findings

ATC-22

EVMTransaction specification allows rugpulls to appear legitimate

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`state-connector-protocol/contracts/interface/types/EVMTransaction.sol`

A scammer doing a rugpull can use the `EVMTransaction` specification to make their scam appear legitimate. Innocent users are misled by the attestation spec assertion that an event's first topic is always the event signature. Both the specification in markdown, its associated definition and all the code generated from it make the same mistake:

```
| `topics`          | `bytes32[]`    | An array of up to 4 32-byte
strings of indexed log arguments. The first string is the signature of
```

```
the event. |
```

```
    /**
     * @notice Event log record
     * @custom:above An `Event` is a struct with the following fields:
     * @custom:below The fields are in line with [EVM event logs]
(https://ethereum.org/en/developers/docs/apis/json-
rpc/#eth_getfilterchanges).
     * @param logIndex The consecutive number of the event in block.
     * @param emitterAddress The address of the contract that emitted
the event.
     * @param topics An array of up to four 32-byte strings of indexed
log arguments. The first string is the signature of the event.
     * @param data Concatenated 32-byte strings of non-indexed log
arguments. At least 32 bytes long.
     * @param removed It is `true` if the log was removed due to a
chain reorganization and `false` if it is a valid log.
     */
    struct Event {
        uint32 logIndex;
        address emitterAddress;
        bytes32[] topics;
        bytes data;
        bool removed;
    }
```

The `EVM` has no rule about what the first topic of an event is. Solidity programs *generally* include the signature in the first topic; but programmers can use the `anonymous` keyword on an event to set an arbitrary first topic. Programmers might also be using raw assembly or even other languages where the default behavior is different.

An attacker can abuse this mistaken assertion to make a seemingly innocent bridge contract that has a backdoor. Consider a cross-chain mint: the attacker can create a method that appears to mint *only* if a certain event has been emitted, but by creating a backdoor with an `anonymous` method they are able to `mint` an unlimited supply for themselves, breaking the supposed peg.

It is worth noting that this specification can also lead to mistakes by non-malicious developers who simply use the `anonymous` keyword in their contracts that depend on the attestation.

Proof of Concept

The following case is a minimal example of a bridge that relies on the **Attestation Client** to determine if a transaction was made in the underlying chain.

For simplicity, the code is Solidity based pseudo-code and simplifies a lot of interactions and checks.

```
// Flare Side
contract FlareReceiver {
    address ethSide;

function verifyPaymentInETHAndMintWithRewards(ETHTx ethTx, byte[]
proof) {
          // assume state connector reverts if proof is not ok
          stateConnector.verifyETHTransaction(ethTx, proof)
          // starting from now we can assume tx actually happened in
eth

// the flare side will now check that the transaction was to
          // the bridge on the ethSide and that the first
          // element in the signature is the signature of the
          // event "PaymentMade(address)"
          require(ethTx.events.length > 0);
          bytes32[] topics = ethTx.events[0].topics
          require(topics.length == 2)
          require(topics[0] == keccack256("PaymentMade(address)")[0:4],
            "not correct event");
          require(ethTx.events[0].emmiter == ethSide,
            "not to our bridge");
          address to = address(topics[1]);
          mint(to, ethTx.value);
    }
}

// ETH Side
contract ETHBridge {
    event PaymentMade(address indexed to);
    event Commitment(bytes32 indexed commit, address indexed from)
anonymous;

address private owner;

constructor() {
        owner = msg.sender;
    }

function transferToFlare(to address) payable {
        emit PaymentMade(to);
    }

// debugging event function, return my money in case i mess up
    function commit(bytes32 commitment) {
        require(msg.sender == owner, "not the owner");
        emit Commitment(commitment, msg.sender);
        msg.sender.call(msg.value);
    }
}
```

The `commit` method is now a subtle backdoor for the attacker, who can simply call it a commitment that is `keccack256("PaymentMade(address))[0:4]` to `mint` and break the peg.

The users that want to check the correct functioning of the bridge would see the attestation specification and believe the contract is well behaved.

```
* @param topics An array of up to four 32-byte strings of indexed log
arguments. The first string is the signature of the event.
```

Recommendation

Change the spec to consider also the case where an anonymous event is attested.

Status

Fixed. The specification has been updated so as not mislead users.

ATC-28

Scammers can trick users into sending payments that they can denounce as non-existent

Status

Solved

✓

Resolution

Fixed

Risk

High

🟥🟥🟥

Impact

High

Likelihood

High

Location

```
src/servers/verifier-server/src/verification/generic-chain-
verifications.ts
```

Description

The `PaymentNonExistent` attestation is supposed to be valid when a payment has not been made. Nevertheless, due to a hardcoded value in the code, the attestation only considers payments made from the first input on an UTXO transaction. This does not correspond with the specification and makes honest payers liable to be denounced by dishonest participants.

A scam would work roughly as follows: an attacker deploys Flare-side contract that requires depositing a collateral in exchange for a payment in an UTXO-chain like **BTC**. If the user makes a payment, the collateral plus a reward is transferred to them. If the user did not make the payment, the collateral is kept by the contract.

There would also be a frontend the victim interacts with. As an incentive to participate, the frontend would offer to subsidize the payment, even 100%. The frontend would craft a **BTC** transaction that made the payment with **the first input** unlocking a UTXO with a non-standard script controlled by the attacker.

The user can analyze all the facts and, if following all the documentation, conclude that this would result in free money. Nevertheless, due to a quirk in how `ReferencedPaymentNonexistence` is implemented, the attacker would be able to denounce the victim as non-paying, even when they payed, only by virtue of controlling the first input of the transaction. The scammer needs to make sure that input unlocks an UTXO that is not-standard and thus does not have an `address` associated.

Due to:

1. The first input being controlled by the attacker,
2. and the UTXO unlocked by that input having a non-standard script and thus no address.

The attestation would result in success, as the `verifier` checks only the first input to request the `paymentSummary`.

```
      // TODO: standard address hash
      const destinationAddressHashTmp =
Web3.utils.soliditySha3(address);
      if (destinationAddressHashTmp ===
request.requestBody.destinationAddressHash) {
        const paymentSummary = fullTxData.paymentSummary({ inUtxo: 0,
outUtxo });
```

While this is OK in most scenarios, as the inputs are not important when verifying if a payment has been made, *if* the input does not hve an address, an error is returned from the `mcc` library:

```
        const spentAmount = this.spentAmounts[inUtxo];

if (!spentAmount.address) {
            return { status: PaymentSummaryStatus.NoSpentAmountAddress
};
        }
```

This `address` is simply the address of the `scriptPubKey` of the `prevout` of the `input`:

```
                  return {
                      address: mapper?.prevout?.scriptPubKey?.address,
                      amount: amount,
                      utxo: mapper?.vout,
                  } as AddressAmount;
              });
```

As the status is `NoSpentAmountAddress`, the verifier will break the for-loop:

```
       // TODO: standard address hash
       const destinationAddressHashTmp =
Web3.utils.soliditySha3(address);
       if (destinationAddressHashTmp ===
request.requestBody.destinationAddressHash) {
         const paymentSummary = fullTxData.paymentSummary({ inUtxo: 0,
outUtxo });

if (paymentSummary.status !== PaymentSummaryStatus.Success) {
           // Payment summary for each output matching the destination
address is equal, so the destination address has been processed.
           break;
         }
```

This would ultimately end with the attestation being `OK`, and the scammer successfully denouncing a payment that was effectively made, as if it were non-existent.

Keep in mind that the `ReferencedPaymentNonExistence` specification makes no mention of only the first input being considered.

Recommendation

Iterate over the `inputs` of a transaction instead of checking only the first one.

Status

Fixed. The attestation now ignores the inputs of the transaction and only considers outputs, making it more straightforward and less error prone.

ATC-29

Limited tests for MCC's transaction attestation

Status
Caution Advised

Resolution
Deferred

Risk
Medium

Impact
Medium

Likelihood
High

Description

The core functionality of MCC is attest the inclusion of transactions. The test system relies on transactions being available on public chains, making it impossible to verify the attestation mechanism at the required extension.

This is specially important for UTXO chains that allow transactions to have arbitrarily complex scripts, whereas the handling procedures must be tested exhaustively against adversarial scenarios.

For example, consider the tests in `MCC` that test that checks for correct parsing of a payment summary:

```
describe.skip(`TESTNET: BTC payment summary with op return,
${getTestFile(__filename)}`, function () {
    let MccClient: MCC.BTC;
    let transaction: BtcTransaction;
    const txid =
"67926749297f9ef450071585526fc2c0d0f1b9e40a8ac50d124c2e6d53c2c3b3";

before(async function () {
        MccClient = new MCC.BTC(BtcMccConnection);
        transaction = await MccClient.getTransaction(txid);
    });

//  it("Should return status", async function () {
    //     const status = await MccClient.getNodeStatus();

//     console.log("status", status.version);

//     console.dir(status, { depth: 10 });
    //  });

it("Should be full transaction", async function () {
        expect(transaction.type).to.eq("payment");
    });

it("Should get payment summary", async function () {
        const ps = await transaction.paymentSummary({ inUtxo: 0,
outUtxo: 1 });
        console.dir(ps, { depth: 10 });
    });
});
```

The test:

1. Depends on a single transaction found on testnet
2. Does not run because it is marked as `skip`
3. If if was not marked as `skip`, one of the scenarios just uses `console.log` and will thus never fail.

Recommendation

Implement a system that tests transactions without them necessarily being on chain. This should *complement* more integrated tests that interact with actual nodes running testchains.

Review and improve tests to make sure negative scenarios are covered. For example, there should tests that checks what is *not* a transaction with a payment reference.

Status

Deferred. Flare is in the process of decoupling components of the attestation system, which will make testing more flexible.

ATC-23

Legacy transactions result in wasted gas

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

Low

Likelihood

High

Location

```
attestation-client/src/utils/helpers/Web3Functions.ts
```

Description

Attestors use legacy transactions instead of the new `Type-2` transactions introduced by EIP1559.

By using legacy transactions, attestors are wasting gas by blindly specifying a `gasPrice` instead of taking advantage of the predictable `baseFee` to potentially pay less for transactions.

Research supports the claim that adopting new-style transactions lead to lower gas usage.

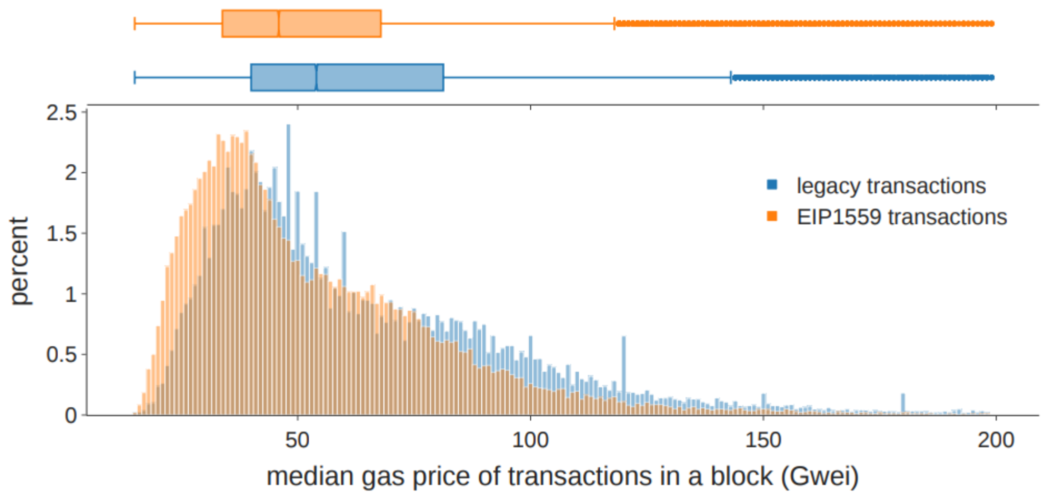

Figure 6: Distributions of median gas prices of a block for legacy transactions and EIP-1559 transactions. The distribution of EIP-1559 transactions is overall to the left of that of legacy transactions, which means that users who adopt EIP-1559 pay a lower gas price.

Transactions are signed and sent with `Web3Functions._signAndFinalize3()`, which specifies a `gasPrice` directly, meaning they are legacy.

```
private async _signAndFinalize3(label: string, toAddress: string,
fnToEncode: any): Promise<any> {
    try {
        const nonce = await this.getNonce();
        const gasPrice = await this.gasPrice();
        const tx = {
        from: this.account.address,
        to: toAddress,
        gas: this.gasLimit,
        gasPrice: gasPrice,
        data: fnToEncode.encodeABI(),
        nonce,
        };

{...}
}
```

Recommendation

Add `EIP1559` compatibility by using using the `max_priority_fee_per_gas` and `max_fee_per_gas` fields when signing and sending transactions.

Status

Fixed. The change originally set the `maxPriorityFeePerGas` to zero, which is not recommended. Flare updated the fix with a new default value of 20Gwei which can be overriden by operators.

ATC-26

Consensus might never be reached

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Medium

Location

```
attestation-client/src/attester/FlareDataCollector.ts
```

Description

Attestors might not reach consensus on bit voting or merkle roots because as the `eventComparator` used to sort events fails to consider the case when `a.logIndex > b.logIndex`, presumably because of a typo which makes it read `a.logIndex > a.logIndex`:

```
  private eventComparator(a: any, b: any): number {
    if (a.blockNumber < b.blockNumber) return -1;
    if (a.blockNumber > b.blockNumber) return 1;

if (a.logIndex > a.logIndex) return -1;
    if (a.logIndex < b.logIndex) return 1;

return 0;
  }
```

This makes the `eventComparator` a non-well-defined comparator, in particular because it does not comply by the anti-symmetry property: `eventComparator(2,3) == 1` but `eventComparator(3,2) == 0`. As the property is not upheld, the behavior is not well defined:

```
If a comparing function does not satisfy all of purity, stability,
reflexivity, anti-symmetry, and transitivity rules, as explained in the
description, the program's behavior is not well-defined.
```

Attestors will presumably run the code with `NodeJS` and thus use the same `V8` engine, minimizing the risk fact that the undefined behavior shows in practice.

Nevertheless, in Coinspect tests the behavior when using the `eventCompartor` is simply to *not* sort the array and keep it as-is.

This makes the `sort`ing of events useless, and in turn introduces the risk that different RPC providers return events in different order. Attestation providers will thus not reach agreement when bit voting or committing to a merkle root.

Recommendation

Fix the third rule of `eventComparator` so it compares both events' indexes.

Status

Fixed. The comparison is now correct.

ATC-30

Proof server will eventually halt due to ever-growing cache

Status
Solved

✓

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

`src/servers/web-server/src/services/proof-engine.service.ts`

Description

Users will be unable to request proofs from attestors because, eventually, the server will stop due to caches getting to big.

```
  // never expiring cache. Once round data are finalized, they do not
change.
  // cache expires only on process restart.
  private cache = {};
  private requestCache = {};
```

While the `cache` map is fairly contained in size, the `requestCache` contains an array of data for each round.

While the issue is considered of **LOW** likelihood due to the time needed to experience degradation considering the amount of data being stored; its impact is **HIGH** because users will be **unable** to request their attestation proof at some point in time, creating a time window for attackers.

Consider also that instances of this same vulnerability have already been reported: **ATC-3** and **VER-1**.

Recommendation

Make sure caches are cleared periodically.

Status

Fixed. Caches are now cleaned.

ATC-25

Attackers can perform exhaustive search on passwords

Status
Caution Advised

Resolution
Partially Fixed

Risk
Low

Impact
Low

Likelihood
Low

Location

`src/utils/security/encrypt.ts`

Description

Attackers can try to perform exhaustive search on the encryption keys because the system does not use a PKDF to derive the encrpytion key from a `password`, instead it uses `SHA256`.

```
export function encryptString(password: string, text: string): string {
  const passwordHash = crypto.createHash("sha256").update(password,
"ascii").digest();
  const initVector = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv("aes-256-gcm", passwordHash,
initVector);
  const encbuf = cipher.update(text, "utf-8");
  return Buffer.concat([initVector, encbuf]).toString("base64");
}
```

PKDF are the recommended way to store passwords, as they are slower to brute-force.

Recommendation

Use a **PKDF**, such as `scrypt`. **PKDF** generally have some parameters that need to be adjusted for the desired security level.

While a common recommendation for `scrypt` is `N=16384, r=8, p=1`; Coinspect recommends a bigger `N` for modern hardware if the performance cost is acceptable. Our recommendations are:

- For interactive, web logins: `N=65536, r=8, p=1` (~85ms)
- For file encryption or critical passwords: `N=2097152, r=8, p=1` (~2.5ms)

The performance was measured on a high-end desktop PC. The stronger parameters are intended to protect critical data or when login is needed only a few times a day. For example, a wallet-protected password with access to an encrypted private key where the user logins only once a day. In this specific case, the stronger parameters would be the recommended ones.

Other **PKDF** algorithms are available: `argon2` and `bcrypt` are also good choices if `scrypt` is not available or desirable for any reason.

Status

Partially fixed. `scrypt` is now used, but the *salt* parameter is the fixed string "Flare". A random string per user can improve the safety of the passwords. Flare has acknowledged this and stated that they expect to redesign this part of the codebase in the future.

ATC-27

Missing members in the EVMTransaction attestation response

Status
Solved

✓

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Location

`state-connector-protocol/contracts/interface/types/EVMTransaction.sol`

Description

The `EVMTransaction` attestation does not include some transaction properties such as the gas spent, gas fees and, gas price, which could be relevant for a third party app.

The response has the following members:

```
    struct ResponseBody {
        uint64 blockNumber;
        uint64 timestamp;
        address sourceAddress;
        bool isDeployment;
        address receivingAddress;
        uint256 value;
        bytes input;
        uint8 status;
        Event[] events;
    }
```

For example, developers using Flare's Attestation Client won't be able to check that a transaction has not return bombed the sender on the underlying EVM, because the gas spent is not part of the response body.

Say that a protocol operates on Flare and Ethereum, and works similarly to `FAssets` but allows agents to deploy contracts on both chains (non-EOA accounts on both sides). With this type of attestation, the project can't check that deposit transactions on the Ethereum side used a reasonable amount of gas (for example, between an estimated range). Because of this, the protocol can't decide, for example, when to slash or challenge a malicious actor that makes a griefing attack on their users if they have stakes in the Flare side.

Recommendation

Evaluate including more relevant transaction parameters into the `EVMTransaction`'s response body.

Status

Acknowledged. Flare has stated they will consider adding these if there is demand for them.

ATC-31

Users are led to errors by different responses in testnet and mainnet

Status

Solved

Resolution

Fixed

Risk

Low

Impact

Low

Likelihood

Low

Location

```
src/servers/verifier-server/src/verification/address-validity/address-
validity-btc.ts
```

Description

The error can be found in the `verifyAddressBTC` method. When a BTC address has the wrong leading byte, `testnet` will answer with a `NOT_CONFIRMED` status.

For example, addresses `3ZutBt4wHMh3ikFJ2HfB1sQYWurg6f3r6U` and `maaj43o2wt34j31ej5pmP6htCHFP8coUHg` for mainnet and testnet are invalid in the same way, but the verifier will give different responses.

Recommendation

Make sure testnet and mainnet responses are identical.

Status

Fixed. The method now behaves the same in testnet and mainnet.

Proof of concept

Test added to `test/addressValidity/btcAddress.test.ts`.

```
  it.only("should give same result for invalid P2SH testnet and
mainnet", function () {
    const address = "3ZutBt4wHMh3ikFJ2HfB1sQYWurg6f3r6U";
    const testnet = "maaj43o2wt34j31ej5pmP6htCHFP8coUHg";

const resp = verifyAddressBTC(address, "");
    const respTestnet = verifyAddressBTC(testnet, "TESTNET");
    console.log("mainnet response");
    console.log(resp);
    console.log("testnet response");
    console.log(respTestnet);

expect(resp.status).to.eq(respTestnet.status);
    expect(resp.response.isValid).to.eq(respTestnet.response.isValid);
  });
```

Output:

```
mainnet response
{
  status: 'OK',
  response: {
    isValid: false,
    standardAddress: '',
    standardAddressHash:
'0x0000000000000000000000000000000000000000000000000000000000000000'
  }
}
testnet response
{ status: 'NOT_CONFIRMED' }
    1) should give same result for invalid P2SH testnet and mainnet

0 passing (6ms)
  1 failing
```

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.