coinspect

You build, we defend.

flare

Source Code Audit

FDC Client

August 2024

FDCv1
Source Code Audit

Version: v241004 | Prepared for: Flare | October 2024

Security Assessment

1. Executive Summary

In **August 20224**, **Flare** engaged Coinspect to perform a Source Code Audit of its FDC Client and changes related to it on other projects. The objective of the review was to evaluate the security of the application.

The FDC Client and related changes aim to provide the Flare chain with accurate data about external events and provide users with a way to prove to smart contracts running on the chain that these events happened.

✓ Solved	Caution Advised	✗ Resolution Pending
High 0	High 0	High 0
Medium 2	Medium 0	Medium 0
Low 0	Low 0	Low 0
No Risk 3	No Risk 0	No Risk 0
Total **5**	Total **0**	Total **0**

The review uncovered a race condition that might cause different clients to reach different bit vote consensus, described in `FDC-005` and how the system would fail to react to a malfunctioning verifier server, detailed in `FDC-003`.

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

2.1 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
FDC-003	Clients cannot defend against malicious verifier server	Medium
FDC-005	Clients show local non-determinism in consensus code	Medium
FDC-001	No double-check against faking bit-vote weight	None
FDC-002	Existing attestation index is duplicated	None
FDC-004	Client cannot aggregate responses from different verifier servers	None

3. Scope

The scope comprised four main repositories. Of these repositories, `fdc-client` was marked as the most relevant for this review and the only one to be reviewed almost fully. For the other three repositories, Flare indicated that only the changes that are related to the integration of the FDC system into the protocol should be reviewed.

- https://gitlab.com/flarenetwork/fdc/fdc-client at commit `2f3fefbb1a7c3947f1c5e664404a8d45a1dc2079`
- https://gitlab.com/flarenetwork/flare-smart-contracts-v2 at commit `d1ced1a8e8d511cd78e55add281dcbbcc63b83d4`
- https://gitlab.com/flarenetwork/flare-system-c-chain-indexer at commit `e960399001cb8216632eec235a745acf7070f1cf`
- https://github.com/flare-foundation/flare-system-client at commit `b0bc847ae246a3d9b74dec446c8c12ecc37d47b2`

One repository, the `evm-verifier`, was provided as an example verifier server to facilitate the review. While this repository was in scope, it was not a priority: the system should work while supporting arbitrary attestation types as long as they are decidable.

The `emv-verifier` scope was set to:

- https://gitlab.com/flarenetwork/fdc/evm-verifier at commit `9c648a5e445b6b360dbb9c9fa9d93cd90f85792f`

It is important to stress that Flare stressed that no logic related to HTTP communications part of the systems was to be reviewed. This made the bit-voting logic the principal part of the review.

On September 30, the fixes review was carried out on the `fdc-client` repository at commit `725c1ed89cbaaccfc69e4731b79e0e640c9e2348` of the `fixes` branch. On October 2, an additional fix-commit for `FDC-005` was reviewed. More information is at the `Status` section of the issue.

4. Assessment

The main system under review is the client-side of the `Flare Data Connector Protocol` or `FDC` for short. This protocol is a subprotocol of a bigger `Flare Systems Protocol` or `FSP` for short. The `FSP` is in charge of coordinating the work of each subprotocol and voting on meta-protocol information, such as who the voters for each reward epoch are. The client for the `FSP` is the `Flare System Client` or `FSC` for short.

The `FDC` is integrated into the bigger system and its responsibilities are constrained to providing attestations about external events. These external events are arbitrary, and they need only be decidable: an attester needs to be able to clearly and unequivocally assert whether certain event happened or not.

An implemented attestation is the `EVMTransaction` attestation, which asserts that a transaction with certain properties exists in the Ethereum mainchain.

To understand the threat model, it is important to take into account how each piece of the system is supposed to communicate. Most of the interaction happens through the blockchain itself and is mediated by the `CChain Indexer` project, which is responsible for timely and correctly picking up data from the chain where the relevant smart contracts are deployed. The interactions are as follows:

1. End-users interact with the `FdcHub` smart contract to create an `Attestation Request` (AR)
2. End-users interact with the `FDC` HTTP server to requests proofs for their attestation requests.
3. End-users *might* interact with a `Verifier Server` to create an `Attestation Request` to send on chain.
4. The `FSP` interacts with the `FDC` via the `FDC's` HTTP server. This interaction is between trusted parties: it is expected that participating entities run both an `FDC` and `FSP` client.
5. `FSP` interacts with a `Verifier Server` to carry out verifications of ARs. The `Verifier Server` can be local or third-party.

The interactions most relevant here are the ones between the `FDC` and the underlying blockchain, mediated by the `CChain Indexer`. These interactions comprise reading from the attestation request events (provided by end users) and bit voting, commit and signature events collected in the `Sumission.sol` contract.

As mentioned in the **Scope** section, the HTTP logic was not in scope. This was clarified after Coinspect described an issue by which users were not able to get proofs for their attestations, making the system not usable. Flare answered that these sections of the code were still a work in progress at this point of the review, and clarified that the review should be made assuming that users could get proofs (see the **Security Assumptions** section).

Due to the scope constrains, the threat model for the project was quite narrow: the main threats were related to the processing of Attestations Requests (AR) and bit votes on the FDC client. Coinspect considered two main threat actors: an evil user sending ARs collected via the C-Chain indexer and an evil voter sending specially crafted bit votes. While Coinspect identified the possibility of an evil or malfunctioning verifier (see `FDC-003` and `FDC-004`), these were deemphasized in this review (see `Security Assumptions`).

After the engagement, Coinspect found no ways for an attacker to cause disruption to the FDC either via bit votes or ARs. Nevertheless, a race condition described in `FDC-005` can cause voters to not reach agreement on a consensus bitvote, thus causing a voter to not send their vote on a round or, if it happens in enough voters, causing generalized service disruption. The race condition cannot be triggered arbitrarily by an attacker.

4.1 Security assumptions

Because the system under review depends on several subsystems that all work in unison, and because the scope of the project includes many partial reviews (focused on only changes relevant to the `FDC`), Coinspect made several security assumptions when performing the review.

The main assumptions where:

1. Future `Attestation Requests` are specified correctly and they need to be decidable.
2. Underlying chains work timely and correctly
3. The JSON-RPCs from which the `C Chain Indexer` feeds work correctly
4. The majority of attestators (by weight) are honest and respond timely to requests
5. `FDC` and `FSP` systems are both run from the same entity
6. The HTTP routes are registered and work by providing users with the attestation proofs and correctly deal with the communications between systems

While not directly related to the FDC's and thus not the priority of this review, it was also assumed that the FSC operators protected their private keys correctly and kept only the necessary keys in hot storage. It is worth pointing out that the FSC now has a `insecurePrivateKeys` option which Coinspect assumed voters would not use.

Furthermore, by Flare's request, Coinspect assumed that the verifier server chosen by the FDC operator was honest and would always provide the correct answer to

FDC's queries. Nevertheless, a buggy or malfunctioning verifier server was considered a risk (see `FDC-004`), as the implementation provided by Flare is just a reference and it is expected that different implementations will exist.

4.2 Testing

Flare took a two-pronged approach to testing: there are unit-tests and integration tests.

All the repositories contain unit tests. The `fdc-client` has a strong coverage of all the critical packages, such as `bitvote` or `attestation`.

```
ok      local/fdc/client/attestation    1.017s  coverage: 70.7% of
statements
ok      local/fdc/client/attestation/bitVotes   13.915s coverage: 82.6% of
statements
ok      local/fdc/client/collector      2.020s  coverage: 34.6% of
statements
ok      local/fdc/client/config 0.008s  coverage: 74.4% of statements
ok      local/fdc/client/manager        6.030s  coverage: 74.0% of
statements
ok      local/fdc/client/round  0.009s  coverage: 12.2% of statements
ok      local/fdc/client/timing 0.007s  coverage: 51.5% of statements
ok      local/fdc/server        0.118s  coverage: 65.5% of statements
```

The indexer uses a mock-chain for its unit tests and has a coverage of only 50%:

```
go test ./indexer -cover
ok      flare-ftso-indexer/indexer      3.645s  coverage: 47.1% of
statements
```

The `flare-system-client` repository has a strong coverage on the `voters` and `merkle` packages, but only 13% of coverage for its `epoch` package and 50% for crytical components `finalizer` and `protocol`, as well as several packages with no coverage:

```
?       flare-tlc/client/cronjob        [no test files]
ok      flare-tlc/client/epoch  0.009s  coverage: 13.3% of statements
ok      flare-tlc/client/finalizer      6.414s  coverage: 59.9% of
statements
ok      flare-tlc/client/protocol       2.486s  coverage: 51.5% of
statements
ok      flare-tlc/client/shared/voters  0.009s  coverage: 80.3% of
statements
ok      flare-tlc/utils 0.008s  coverage: 8.1% of statements
ok      flare-tlc/utils/merkle  0.008s  coverage: 93.9% of statements
```

On the `flare-smart-contracts-v2` repository at the `fdc/` directory, only the `Verification.sol` contract has any testing done, but it has a 100% coverage.

On the other hand, the integration tests were provided in a separate repository called `fsp-e2e-testing`. Unfortunately, these tests were not fully working with the commit-under-review. While reckoning the system Coinspect ran them on a different commit provided by Flare, but they were not utilized afterwards as they would have yielded results that differed from the set scope for this review.

The testing strategy is sound, with a healthy mix of unit testing and integration tests. Nevertheless, work should be done to improve the coverage and specially tests that exercise corner cases. Issues such as a `FDC-005` would have shown up in testing in the affected functions were covered and the tests ran with the `-race` detector.

Fuzz tests should also be added. Coinspect stressed the `BranchAndBound` methods with fuzz tests to detect potentially harmful inputs. Unfortunately, a 1 second non-modifiable timeout in Go fuzzer limited the utility of the fuzz. Nevertheless, Coinspect found inputs that made the `BranchAndBound` methods run for about 5 seconds. According to Flare, this is an acceptable run time for the method. More work can be done to implement fuzzing, both for stress testing and comparing results with an alternative implementation.

Note that the quoted coverage snippets are modified to remove some non-relevant packages.

5. Detailed Findings

FDC-001

No double-check against faking bit-vote weight

Status	Risk
Solved ✓	**None**
	Impact **Recommendation**
Resolution **Fixed**	Likelihood –

Location

```
fdc-client/client/manager/manager.go
```

Description

The FDC client assumes that the current protection implemented in the smart contracts against two entities registering the same singing policy address will continue to be present in the future and working as intended. While the invariant is currently correctly held by the smart contracts, it would be best to implement defense-in-depth mechanism in the critical path to make sure that voter weight is not faked by evil voters.

If the invariant was not held by the smart contracts, an attacker could fake bit-vote weight by controlling two voters entities and having both use the same signing policy address. This would allow them to completely halt the system by voting for the empty bit vote.

The problem lies in the `ProcessBitVote` method, which is called every time a voter sends a bit vote vector. Consider the following snippet:

```
	signingAddress, exists :=
r.voterSet.SubmitToSigningAddress[message.From]
	if !exists {
		return fmt.Errorf("no signing address")
	}

	voter, exists := r.voterSet.VoterDataMap[signingAddress]
	if !exists {
		return fmt.Errorf("invalid voter")
	}
	weight := voter.Weight
	if weight <= 0 {
		return fmt.Errorf("zero weight voter")
	}
	// check if a bitVote was already submitted by the sender
	weightedBitVote, exists := r.bitVoteCheckList[message.From]
```

Note the the `signingAddress` is used to retrieve the `weight` of this voter, while the `message.From`, which is the sender of the original transaction (see `ExtractPayloads` in `fdc-client/flare-common/payload/payload.go`), is used to check for duplicates.

This would allow an attacker to:

1. Register a voter `A` and set their `signingAddress` to `C_signing`
2. Register a voter `B` and set their `signingAddress` to `C_signing`
3. Send a bitvote with `A`, which will use the weight of `C_signing`
4. Send another bitvote with `B`, which will use the weight of `C_sigining` again and not be considered a duplicate

Recommendation

Double-check for duplicated usages of the address from which the weight is fetched to make this safety-critical logic not depend on a single component.

Status

Fixed. The invariant is held by the smart contracts. As an added safety measures, checks that make sure that the signing policy is consistent have been added in commit `725c1ed89cbaaccfc69e4731b79e0e640c9e2348`.

Proof of concept

The test is intended for `fdc-client/client/manager/manager_test.go`. The system should err out on the second attempt, but it will not do so.

```
func TestManagerMethodsEvil(t *testing.T) {
        cfg, err := config.ReadUserRaw(USER_FILE)
        require.NoError(t, err)

        sharedDataPipes := shared.NewDataPipes()
        mngr, err := New(&cfg, sharedDataPipes)
        require.NoError(t, err)

        signingPolicyParsed, err :=
policy.ParseSigningPolicyInitializedEvent(policyLog)

        require.NoError(t, err)

        submitToSigning := make(map[common.Address]common.Address)

        for i := range signingPolicyParsed.Voters {
                submitToSigning[signingPolicyParsed.Voters[i]] =
signingPolicyParsed.Voters[i]
        }
        // ! Replace so that submit[0]  = siging[1]
        // ! Now we have 0 --> 1
        // ! And                 1 --> 1
        submitToSigning[signingPolicyParsed.Voters[0]] =
signingPolicyParsed.Voters[1]
        // the submitter is 0x8fe15e1048f90bc028a60007c7d5b55d9d20de66
        // this is the signing address of the submitter:
0xCCb478bBA9c76AE21e13906A06aeb210ad3593cf
        fmt.Printf("\033[0;31m[COIN]\033[0m submitToSigning: %+v\n",

submitToSigning[common.HexToAddress("0x8fe15e1048f90bc028a60007c7d5b55d
9d20de66")])

        votersData := shared.VotersData{Policy: signingPolicyParsed,
SubmitToSigningAddress: submitToSigning}
        err = mngr.OnSigningPolicy(votersData)
        require.NoError(t, err)

        fmt.Println("---- first bitvote ---")

        bitVoteMessageCorrect := bitVoteMessage
        bitVoteMessageCorrect.Payload = []byte{664111 % 256, 0, 0}
        bitVoteMessageCorrect.From =
common.HexToAddress("0x8fe15e1048f90bc028a60007c7d5b55d9d20de66")
        fmt.Printf("[TEST] from: %s\n", bitVoteMessageCorrect.From)
        err = mngr.OnBitVote(bitVoteMessageCorrect)
        require.NoError(t, err)

        r, ok := mngr.Rounds.Get(664111)
        require.True(t, ok)
        r.ComputeConsensusBitVote()
        fmt.Println("---- second bitvote ---")

        // Let us now send a bitvote from another sender, but who also
has a signing address set to
        // 0xCCb478bBA9c76AE21e13906A06aeb210ad3593cf
        // That is: 0xCCb478bBA9c76AE21e13906A06aeb210ad3593cf
themselves!
        bitVoteMessageRepeat := bitVoteMessage
        bitVoteMessageRepeat.From =
common.HexToAddress("0xCCb478bBA9c76AE21e13906A06aeb210ad3593cf")
        bitVoteMessageRepeat.Payload = []byte{664111 % 256, 0, 0}
        err = mngr.OnBitVote(bitVoteMessageRepeat)
        require.NoError(t, err)

        r, ok = mngr.Rounds.Get(664111)
        require.True(t, ok)
}
```

FDC-002

Existing attestation index is duplicated

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

`fdc-client/client/round/round.go`

Description

The `Index` of the new attestation is added to the existing-attestation twice when the method `AddAttestation` is called.

```
func (r *Round) AddAttestation(attToAdd *attestation.Attestation) bool
{
        identifier := crypto.Keccak256Hash(attToAdd.Request)
        att, exists := r.attestationMap[identifier]
        if exists {
                att.Fee.Add(att.Fee, attToAdd.Fee)
                if attestation.EarlierLog(attToAdd.Index(),
att.Index()) {
                        att.Indexes = utils.Prepend(att.Indexes,
attToAdd.Index())
                }
                att.Indexes = append(att.Indexes, attToAdd.Index())
                return false
        }
```

Note the scenario where the innermost if-condition is true and the `attToAdd` index is smaller than the existing attestation. In that case, the `attToAdd` index is prepended to the list of indexes of the attestation. Immediately afterwards, it is appended.

The result is that the previous value is sandwiched between two values of the new attestation to add.

This has no impact in the code under-review as only the head of the `Indexes` slice is accessed. The `getRequestController` uses the whole slice, but it is out of scope for this review as it is under construction.

Recommendation

Add an `else` branch to avoid appending the indexes when they were prepended.

Status

Fixed in commit `ea4bf7f6ef5b282d2deffd96056dc812814e11e5`. /--

FDC-003

Clients cannot defend against malicious verifier server

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

Description

A malicious or malfunctioning verifier server can delay their responses or send requests that are too big for the client to handle, which would cause the client to be delayed and potentially miss rounds. Notice the similarity with `FDC-005`.

The likelihood of this issue is `low` because it requires a malfunctioning implementation of a verifier server *or* a verifier server that is malicious in its own right. It is unlikely for a downright malicious verifier server to carry out this attack, as they could simply provide erroneous responses.

Nevertheless, a bug in the implementation of verifier servers is still possible.

Recommendation

Implement measures to limit the impact of a malfunctioning verifier server such as timeouts and limiting how much data from the request is read.

Status

Fixed in commit `725c1ed89cbaaccfc69e4731b79e0e640c9e2348`. The FDC client now implements a timeout and reads only 10MB of data from the response of the verifier server.

FDC-004

Client cannot aggregate responses from different verifier servers

Status
Solved

✓

Resolution
Deferred

Risk
None

Impact
Recommendation

Likelihood
–

Location

Description

FDC clients should be able to configure redundant verifier servers to be able to protect themselves from a potentially malicious verifier server.

While verifier servers were assumed to be honest for this review, the clients can relax this assumptions by adding the possibility of querying N different verifiers from the client. Client operators could then decide a threshold M where N out of M verifiers need to agree on the response for the client to accept it as the canonical response.

This would allow clients to not have to absolutely trust a single verifier server. Instead, N out of the M chosen servers must be evil for the verifier to vote with a wrong attestation.

Recommendation

Implement the logic to support N-out-of-M attestation verification.

Status

The Flare team has acknowledged that this would allow clients to be more certain of the correctness of the responses from the verifiers servers, but that nevertheless at this stage of the product clients need to communicate with verifiers that return the correct response. This improvement will be considered in future updates.

FDC-005

Clients show local non-determinism in consensus code

Status

Solved

✓

Resolution

Fixed

Risk

Medium

🟥🟥⬜

Impact

High

Likelihood

Low

Location

`fdc-client/client/attestation/bitVotes/branchAndBoundVotes.go`

Description

Clients can trigger a race-condition on the algorithm that decides the canonical bitvote. While the global-result is assured to be deterministic due to how the the `BranchAndBound` algorithm works; a locally-observable race condition is triggered. If subtleties about the underlying algorithm change, this could lead to voters not being able to reach consensus on the bit vote.

To understand the issue it is important to understand that the set of voters of a round must reach a single canonical bitvector that represents the attestations they are willing to provide a resolution for in a given round. The FDC implements a Branch and Bound strategy for deciding which attestations are included after prefiltering some that will always be included in the round and some that will never be.

The FDC will attempt to either use BranchAndBoundVotes or BranchAndBoundBits, depending on the length of the slices of votes and bits, respectively. In any case, a similar process is fired up: two go routines with slightly different strategies are started and waited for in the main thread.

What follows is a highly simplified snippet of the relevant code, in a Go-like pseudocode:

```
solutions := make(Solution, 2)
done1 := make(chan bool, 1)
done2 := make(chan bool, 1)
go func() {
    solution := BranchAndBound(...)
    solutions[0] = solution
    done1 <- true
    if solution.Optimal {
        solutions[1] = nil
        done2 <- true
    }
}
go func() {
    solution := BranchAndBound(...)
    solutions[1] = solution
    done2 <- true
}

<-done1
<-done2
if solutions[0] == nil { return solutions[1] }
if solutions[1] == nil { return solutions[0] }
if solutions[0] < solutions[1] { return solutions[1] }
return solutions[0]
```

Note that while both gorutines write to the same array, the main write (that of the solution) is always in different indexes, making the concurrent write safe.

Nevertheless, there is one instance when a concurrent write happens to the same variable: when the first solution is Optimal, solution[1] is marked as nil.

The problem then arises when:

1. A client fires up the two gorutines, first and second.
2. first and second send to done1 and done2 respectively at approximately the same time
3. Now there's a race condition between the main thread to read from solutions[1] and first to write nil to it

While a race conditions does happen, under the current BranchAndBound algorithm the end result of the method is deterministic. This happens because if the solution is Optimal, then solutions[0] >= solutions[1], making the last if condition false, and thus returning solutions[0]; the same result that would be observed *without* the race condition.

Nevertheless, this behavior is *not* due to an invariant held in the `BranchAndBoundDouble` method and is instead dependent on the underlying `BranchAndBound` algorithm, making it risky as consensus code. Consider what would happen if something changed in the underlying implementation:

On some clients, `solutions[1]` will be marked as `nil` before they reach the if-switch at the bottom of the method. For those clients, they will always returns `solutions[0]`.

On other clients, `solutions[1]` will not be `nil`. If `solutions[1]` is bigger than `solutions[0]`, `solutions[1]` will be returned. Note that this is equivalent to the process when the first solution is not optimal.

Recommendation

Do not write to the same variable on two different gorutines and always wait for the whole gorutine process to finish before unlocking the main thread.

In practice, this can be achieved by the introduction of a third variable that indicates that the `solution[1]` should be ignored and by unlocking the main thread only after the if condition that checks if the solution is optimal.

```
...
go func() {
    solution := BranchAndBoundVotes(...)
    solutions[0] = solution
    if solution.Optimal {
        ignoreSecondSolution = true
        done2 <- true
    }
    done1 <- true
}
```

Note that more changes would be necessary: the if conditions below the gorutines should be modified to check the `ignoreSecondSolution` flag. The `done2` channel should be waited on first to avoid a potential deadlock where `done2` is ready first on the second gorutine, but the `done2` on the first gorutine cannot be sent becaues the channel is full, thus never sending on `done1`.

Status

Flare has fixed the race condition by adding the `ignoreSecondSolution` variable. It is worth noting that the fix implemented does not exactly match what was described by Coinspect, although it avoids writing to the same variable on two different gorutines.

There's still a possible race-condition where *reading* from the `ignoreSecondSolution` variable.

On an additional fix commit `0c9fcdd5993d097f48cdc0a21d35b61fbd01441b`, Flare has totally removed the race condition by slightly changing the solution describe above so that the `firstDone` signal is only sent when `ignoreSecondSolution` has been either marked `true` or `false`. Tests with the Golang race detector do not trigger a race and reviewers found no way one could be triggered.

6. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.