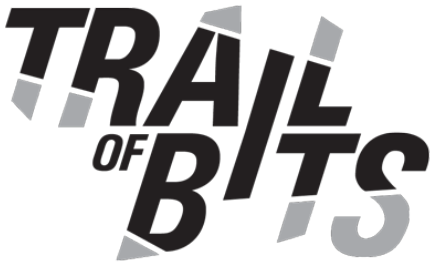

Teller

Security Assessment

December 1, 2020

Prepared For:
Ryan Berkun | *Teller Finance*
ryan@teller.finance

Prepared By:
Alex Useche | *Trail of Bits*
alex.useche@trailfobits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Kristin Mayo | *Trail of Bits*
kristin.mayo@trailofbits.com

Executive Summary

Teller engaged Trail of Bits September 28–October 6 and November 23-25, 2020, to review the security of the off-chain components of The Teller Protocol. We conducted this assessment over four person-weeks with two engineers working from the commits shown in the Coverage section of this document.

During the first week of the assessment we performed manual and automated review of provided documentation and code. The initial focus was to identify components of the Teller protocol that are directly reachable by users. We also conducted dynamic tests by running the above code bases locally whenever possible. The second week was focused on testing input and error handling issues. We prioritized the validator code base, as it contains JSON-RPC endpoints available to users for issuing and requested loans.

Our review resulted in 10 findings ranging from medium to informational severity. The medium-severity issues included overly permissive cross-origin resource sharing (CORS) settings (TOB-TEL-002), use of environment variables for storing RSA keys (TOB-TEL-003), insufficient logging (TOB-TEL-006), and unbound function recursions (TOB-TEL-008).

Additionally, verbose errors were returned by the validator application (TOB-TEL-005). Low-severity issues include Insufficient testing coverage (TOB-TEL-010), and an insecure function used to generate signer nonce values (TOB-TEL-009). Two informational issues—insufficient user input validation (TOB-TEL-007) and the use of RSA signatures (TOB-TEL-004)—were also discovered.

Additional notes on code quality and static tooling findings appear in Appendix B.

While no high-risk findings were discovered, the Teller protocol is still being actively developed. New vulnerabilities may surface as the protocol grows, so the biggest areas of improvement concern maturity in terms of security controls around input validation, logging and monitoring, and testing for invalid and malicious input. As the codebase matures, it will be important to focus on these areas to minimize the possibility of introducing new vulnerabilities that could be exploited by attackers.

We recommend centralizing logging, monitoring, and error handling. Moreover, we suggest a security-in-depth approach to input validation, configuration of services, and use of cryptography.

Project Dashboard

Application Summary

Name	Teller
Version	See Coverage
Type	TypeScript
Platforms	Web

Engagement Summary

Dates	September 28–October 6, 2020 November 23–November 25, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total Medium-Severity Issues	4	■ ■ ■ ■
Total Low-Severity Issues	4	■ ■ ■ ■
Total Informational-Severity Issues	2	■ ■
Total	10	

Category Breakdown

Patching	1	■
Access Controls	1	■
Cryptography	3	■ ■ ■
Error Reporting	2	■ ■
Auditing and Logging	1	■
Data Validation	1	■
Denial of Service	1	■
Total	10	

Engagement Goals

The engagement was scoped to provide a security assessment of the off-chain components of the Teller Protocol, including the `validator`, `contract-event-istener`, `data-service`, and `subgraph` repositories.

Specifically, we sought to answer the following questions:

- Are the components that make up the overall architecture prone to generic data validation errors that may lead to injection vulnerabilities?
- Do any of components leak sensitive information through errors or server responses to clients?
- Do the components handle sensitive data in a safe and cryptographically secure manner?
- Can attackers influence the way interest values are calculated?
- Is it possible for an attacker to reduce the system's availability?
- Can an attacker perform unauthorized operations?

Coverage

This section highlights some of the analysis coverage we achieved based on our high-level engagement goals.

The engagement was scoped to provide a security assessment of the following component that make up the Teller solution:

- https://github.com/teller-protocol/validator: `c4cf8ff`.
- https://github.com/teller-protocol/contract-event-listener: `3124bd0.`
- https://github.com/teller-protocol/data-service: `616bc14`.
- https://github.com/teller-protocol/subgraph: `1caa05d`.

Note that the commit shown for the validator project corresponds to the code pulled on November 23, 2020, as the validor project was actively developed during the first week and half of the audit.

The primary method used to review the `contract-event-listener`, `data-service`, and `subgraph` repos was static code analysis. We reviewed data validation logic, as well as error handling, logging and monitoring, and cryptographic controls. We also leveraged static analysis tools to aid our process.

After mapping out the edges of the protocol most likely to be attacked, we focused on the methods exposed by the `validator` project. Teller also requested that we focus on those two methods for the last week of the audit. In addition to static code analysis, we conducted dynamic tests by running the validator project locally. We fuzzed the `arrowheadCRA` and `accruedInterest` methods dynamically to observe how the application behaved in response to invalid and malicious input. This also allowed us to test for potential race condition issues and rate limiting issues that could cause denial-of-service conditions.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Ensure dependencies are up to date.** Several node modules have been reported and documented as malicious. Keep modules up to date and verify their integrity after installation. TOB-TEL-001

❑ **Create a whitelist of endpoints from which requests are expected**. This list can be stored in configuration files, and can include `localhost` for development. Then, when a request is received, validate the origin of the request with the whitelist, and set the `Access-Control-Allow-Origin` header accordingly. TOB-TEL-002

❑ **Consider letting the user authenticate the private key manually.** TOB-TEL-003

❑ **Switch `"rsa"` to `"ed25519"` when generating key pairs.** TOB-TEL-004

❑ **Make sure that JSON parsing errors are caught.** TOB-TEL-005

❑ **Establish alerts on the aggregated application logs.** Log exceptions, particularly those that could result from users submitting malicious input. TOB-TEL-006

❑ **Validate all user requests against malicious input.** TOB-TEL-007

❑ **Establish a maximum number of retries.** Check against that value whenever the function in question gets called, and return an error after that number is reached. TOB-TEL-008

❑ **Replace the use of `math.random` with a CSPRNG.** TOB-TEL-009

❑ **Add thorough unit and integration tests for the codebase using a development framework to facilitate more complicated testing scenarios.** TOB-TEL-010

Long term

❑ **Track and audit dependencies on a regular basis.** It's not uncommon for dependencies to be added during development but not removed when no longer needed.

Consider integrating automated dependency auditing in the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure the Teller codebase does not use, and is not affected by, the vulnerable functionality of the dependency. TOB-TEL-001

❑ **Use Hashicorp Vault and one of its `Node.js` clients.** TOB-TEL-003

❑ **Add unit tests that check for different forms of invalid input, and consider fuzzing the application to determine whether additional errors result in verbose errors returned to users.** TOB-TEL-005

❑ **Create a centralized logging strategy that aggregates and centralizes application logs.** Ideally, logs could be submitted to cloud solutions such as Graphana and ElasticSearch. Additionally, establish alerts on abnormal system and application behaviors. This should include a baseline of operation, and alert the core response team if those values exceed their normal range. TOB-TEL-006

❑ **Devise a centralized strategy for validation of common variables submitted by users, such as wallet addresses and tokens.** TOB-TEL-007

❑ **Whenever generating random values for cryptographic use, use a CSPRNG.** TOB-TEL-009

❑ **Use a development framework to facilitate complex testing scenarios and integrate it into a CI/CD pipeline.** TOB-TEL-010

Findings Summary

#	Title	Type	Severity
1	Outdated and vulnerable dependencies	Patching	Low
2	Overly permissive cross-origin resource sharing (CORS) settings	Access Controls	Medium
3	Consider using Vault	Cryptography	Medium
4	Consider using EdDSA signatures	Cryptography	Informational
5	Verbose errors	Error Reporting	Low
6	Insufficient logging	Auditing and Logging	Medium
7	Insufficient user input validation	Data Validation	Informational
8	Unbound retries could lead to denial of service	Denial of Service	Medium
9	Insecure function used to generate signer nonce	Cryptography	Low
10	Insufficient testing coverage	Error Reporting	Low

1. Outdated and vulnerable dependencies

Severity: Low
Type: Patching
Target: `package.json`

Difficulty: Low
Finding ID: TOB-TEL-001

Description
Although dependency scans did not yield a direct threat to the codebase, `npm audit` has identified dependencies with known vulnerabilities. Because of the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the Teller system as a whole. The output detailing the identified issues has been included in Appendix C.

Exploit Scenario
A malicious actor identifies an outdated node module with an existing proof of concept for a known security vulnerability and uses this exploit against an unwitting user.

Recommendations
Short term, ensure dependencies are up to date. Several node modules have been reported and documented as malicious. Keep modules up to date and verify their integrity after installation.

Long term, track and audit dependencies on a regular basis. It is not uncommon for dependencies to be added during development but not removed when no longer needed. Consider integrating automated dependency auditing in the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure the Teller codebase does not use, and is not affected by, the vulnerable functionality of the dependency.

2. Overly permissive cross-origin resource sharing (CORS) settings

Severity: Medium
Type: Access Controls
Target: `validator, data-service`

Difficulty: Medium
Finding ID: TOB-TEL-002

Description
The validator and data-service applications configure cross-origin resource sharing (CORS) headers that are overly permissive. Since CORS settings are not specified, the default asterisk wild flag (*) is set for the `Access-Control-Allow-Origin` header. As a result, the application can receive and respond to requests originating from unintended endpoints.

```
expressServer.use(cors({ methods: ["POST"] }));
```

Figure 2.1: validator/src/app.ts#L21.

```
export const wrap = (main: Main): Main => {
 return async function (event: ApiGatewayEvent) {
   const result = await main(event);
   result.headers["Access-Control-Allow-Origin"] = "*";
   console.log(result);
   return result;
  };
};
```

Figure 2.2: data-service/packages/backend/src/utils/wrap.ts#L3-12.

```
app.use(cors());
```

Figure 2.3: data-service/packages/backend/src/dev-server.ts#L12.

Exploit Scenario
An attacker with access to the running validator service sets up a malicious website that interacts with the service. Users interact with the site believing they are using a legitimate client for the validator service. The attacker can then provide the same functionality as the service while simultaneously logging data submitted by users, including transaction data.

Recommendation
Short term, create a whitelist of endpoints from which requests are expected. This list can be stored in configuration files, and can include `localhost` for development. Then, when a request is received, validate the origin of the request with the whitelist, and set the `Access-Control-Allow-Origin` header accordingly.

3. Consider using Vault

Severity: Medium
Type: Cryptography
Target: `data-service`

Difficulty: Medium
Finding ID: TOB-TEL-003

Description
Hashicorp Vault is an all-in-one key and secrets management solution with cache functionality. Rather than keeping the RSA passphrase in memory as an environment variable—which is readable by anyone who has local access with sufficient permissions—we can request the passphrase from Vault after the user authenticates, thereby keeping the passphrase slightly more secret.

```
const RSA_KEY_PASSPHRASE = process.env.RSA_KEY_PASSPHRASE;
if (!RSA_KEY_PASSPHRASE) throw new Error("Bad environment: Missing RSA_KEY_PASSPHRASE");
```

Figure 3.1: Key passphrase is passed as an environment variable.
(data-service/packages/backend/scripts/generate-key-pair.ts)

Exploit Scenario
A malicious local user is able to read both the environment variable and the key pair, and creates a fake service pretending to be the authentic `data-service`.

Recommendations
Short term, consider letting the user authenticate the private key manually.

Long term, use Hashicorp Vault and one of its `Node.js` clients.

4. Consider using EdDSA signatures

Severity: Informational
Type: Cryptography
Target: `data-service`

Difficulty: Low
Finding ID: TOB-TEL-004

Description

A lot can silently go wrong when using RSA signatures. While the default settings by `crypto.generateKeyPairSync` should be secure, elliptic curve signatures offer the same signature API, at the same security levels, at greater signing speeds and smaller key sizes. ECDSA by itself is also fragile, but EdDSA is slightly more resistant to the subtle elliptic curve attacks that are being discovered lately.

```
const keyPair = crypto.generateKeyPairSync("rsa", {
 modulusLength: 4096,
 publicKeyEncoding: {
   type: "spki",
   format: "pem",
 },
 privateKeyEncoding: {
   type: "pkcs8",
   format: "pem",
   cipher: "aes-256-cbc",
   passphrase: RSA_KEY_PASSPHRASE,
 },
});
```

Figure 4.1: RSA signatures used when generating key pairs.
(data-service/packages/backend/scripts/generate-key-pair.ts)

Recommendation

Short term, switch `"rsa"` to `"ed25519"` when generating key pairs.

5. Verbose errors

Severity: Low
Type: Error Reporting
Target: `validator`

Difficulty: Low
Finding ID: TOB-TEL-005

Description
The `validator` application returns verbose error messages when sending a request with a malformed JSON body. The returned message includes internal paths for node modules responsible for JSON parsing. To replicate, send a POST request to the validator service with an invalid JSON.

```
SyntaxError: Unexpected string in JSON at position 158<br>    at JSON.parse
(&lt;anonymous&gt;)<br>    at parse (/Users/au/Assessments/Teller/Internal
Repo/audit-teller/client-code/validator/node_modules/body-parser/lib/types/json.js:89:19)<br
>    at /Users/au/Assessments/Teller/Internal
Repo/audit-teller/client-code/validator/node_modules/body-parser/lib/read.js:121:18<br>
   at invokeCallback (/Users/au/Assessments/Teller/Internal
Repo/audit-teller/client-code/validator/node_modules/raw-body/index.js:224:16)<br>  
 at done (/Users/au/Assessments/Teller/Internal
Repo/audit-teller/client-code/validator/node_modules/raw-body/index.js:213:7)<br>  
 at IncomingMessage.onEnd (/Users/au/Assessments/Teller/Internal
Repo/audit-teller/client-code/validator/node_modules/raw-body/index.js:273:7)<br>  
 at IncomingMessage.emit (events.js:203:15)<br>    at
IncomingMessage.EventEmitter.emit (domain.js:448:20)<br>    at endReadableNT
(_stream_readable.js:1145:12)<br>    at process._tickCallback
(internal/process/next_tick.js:63:19)
```

Figure 5.1: Error returned in response to invalid JSON.

Exploit Scenario
An attacker is able to learn about internal paths and libraries used by the application.

Recommendations
Short term, make sure that JSON parsing errors are caught.

Long term, add unit tests that check for different forms of invalid input, and consider fuzzing the application to determine whether additional errors result in verbose errors returned to users.

6. Insufficient logging

Severity: Medium
Type: Auditing and Logging
Target: `validator`, `contract-event-listener`

Difficulty: Medium
Finding ID: TOB-TEL-006

Description
There was no logging strategy for any of the services audited. Logging was only done with `console.log` statements, which are insufficient. This could make it very difficult to triage potential bugs, vulnerabilities, and intrusions. As a result, security-relevant logging information must be reconstructed from the blockchain, which is cumbersome to perform during a regulatory audit or to detect when an intrusion or attack has occurred. Attacks on the validator application may be very difficult to detect and reconstruct without sufficient logging.

Exploit Scenario
An attacker is able to take advantage of a vulnerability in any of the affected applications as a result of code changes or vulnerabilities in third-party libraries. Due to insufficient log aggregation and centralization, the compromise remains unnoticed.

Recommendations
Short term, establish alerting on the aggregated application logs. Log exceptions, particularly those that could result from users submitting malicious input.

Long term, create a centralized logging strategy that aggregates and centralizes application logs. Ideally, logs could be submitted to cloud solutions such as Graphana and ElasticSearch. Additionally, establish alerts on abnormal system and application behaviors. This should include a baseline of operation, and alert the core response team should those values exceed their normal range.

7. Insufficient user input validation

Severity: Informational
Type: Data Validation
Target: `validator`

Difficulty: Undetermined
Finding ID: TOB-TEL-007

Description
The `validator` application exposes a JSON-RPC interface to users over the internet. However, it lacks sufficient validation of user-submitted data. The main two methods exposed by this application are `arrowheadCRA` and `accruedInterest`. Both methods validate datetime values and borrow assets against a whitelist of accepted wallet assets.

However, some values are not sufficiently validated against malicious input, including tokens and wallet addresses used in GraphQL queries. Additionally, `loanTermLength` is not validated against negative values.

While it was not possible to exploit this finding, accepting input from users without prior sanitization and validation could lead to a number of attacks, including GraphQL injection. Moreover, as the project grows in size and complexity, new vulnerabilities may be introduced as a result of insufficient input validation, such as Cross-Site Scripting (if malicious input is reflected on a front-end client).

Recommendations
Short term, validate all user requests against malicious input.

Long term, devise a centralized strategy for validation of common variables submitted by users, such as wallet addresses and tokens.

8. Unbound retries could lead to denial of service

Severity: Medium
Type: Denial of Service
Target: `data-service`

Difficulty: High
Finding ID: TOB-TEL-008

Description

The `data-service` source code contains a function, `getAssetReportRetry`, used to pull reports from a Plaid client. This function is called recursively if the request to Plaid fails. Each time the function is called, the request to Plaid is delayed by an increasing amount of time. However, there are no measures in place to prevent the function from being called indefinitely if there's an issue with the Plaid client or the values submitted in the request to Plaid.

```
export async function getAssetReportRetry(assetReportToken: string, retries = 0):
Promise<plaid.AssetReport> {
 try {
   return (await plaidClient.getAssetReport(assetReportToken, false)).report;
 } catch (e) {
   retries++;
   return await new Promise((resolve) => {
     setTimeout(async () => {
       resolve(await getAssetReportRetry(assetReportToken, retries));
     }, (1 + retries) * 2000);
   });
 }
}
```

Figure 8.1: Key passphrase is passed as an environment variable. (/data-service/packages/backend/src/lambdas/bank-info/get-asset-report.ts#L23-34)

Exploit Scenario

An issue with Plaid causes the function to be continuously called for one or more requests. This could result in exhaustion of resources and denial-of-service conditions.

Recommendation

Short term, establish a maximum number of retries. Check against that value whenever the function in question gets called, and return an error after that number is reached.

9. Insecure function used to generate signer nonce

Severity: Low
Type: Cryptography
Target: `validator`

Difficulty: High
Finding ID: TOB-TEL-009

Description

The `validator` application generates a signer nonce using the JavaScript function `Math.Random()`, which is not a cryptographically secure pseudo-random number generator (CSPRNG). In this particular application, using a CSPRNG is not essential, but it is considered best practice to use them whenever generating random values in cryptography.

```
const signerNonce = (Math.random() * 20000).toFixed(0);
const signature = await web3.eth.sign(
 hashResponse({
   consensusAddress,
   chainId,
   interest,
   requestHash,
   responseTime,
   signerNonce,
 }),
 signer,
);
```

Figure 9.1: Math.random used for nonce generation.
(/validator/methods/accruedInterest/index.ts#L194-206)

```
 const signerNonce = (Math.random() * 20000).toFixed(0);


 const response = {
   chainId,
   collateralRatio,
   consensusAddress,
   interestRate: interestRate.toString(),
   maxLoanAmount: loanSize.toString(),
   signerNonce,
   requestHash,
   responseTime: Big(requestTime).add(5).toString(),
```

```
};
```

Figure 9.2: `Math.random` used for nonce generation.
(validator/methods/arrowheadCRA/index.ts#L169-180)

Exploit Scenario
An attacker is able to repeatedly generate and observe nonce values in a sequential manner and may be able to deduce the pseudo-random number generator's internal state. With this knowledge, an attacker can perform an offline calculation and predict future nonce values generated to confirm transactions. As a result, the attacker may be able to spoof responses, provided they know the other values used to generate such responses.

Recommendations
Short term, replace the use of `math.random` with a CSPRNG.

Long term, whenever generating random values for cryptographic use, use a CSPRNG.

10. Insufficient testing coverage

Severity: Low
Type: Error Reporting
Target: `validator`, `contract-event-listener`

Difficulty: Low
Finding ID: TOB-TEL-010

Description
The `validator` codebase does not include sufficient test cases. Robust unit and integration tests are critical for catching the introduction of certain bugs and logic errors early in the development process. Projects should strive for thorough coverage against bad inputs as well as "happy path" scenarios. This will greatly increase confidence in the functionality of the code for users and developers alike.

Additionally, unit tests are a beneficial addition for the testing architecture to verify that subsets of the application work locally as expected. These tests allow developers to effectively detect early inconsistencies in code behavior before deployment.

Exploit Scenario
Teller discovers a bug in the application and writes a patch to update it. The new code causes backward incompatibility and deviates from the contract's expected behavior. Due to insufficient testing coverage, this is not caught by the testing suite.

Recommendations
Short term, add thorough unit and integration tests for the codebase using a development framework to facilitate more complicated testing scenarios.

Long term, use a development framework to facilitate complex testing scenarios and integrate it into a CI/CD pipeline.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Quality Recommendations

Insufficient Error Handling
The `validator` catches most errors by using a wrapper function that only checks whether an error has the property `business` before returning a generic error message.

```
export const wrapper = <P, R>(
 fn: (params: P) => Promise<R>,
): ((params: P) => Promise<R | JSONRPCError>) => {
 return async function (params: P): Promise<R | JSONRPCError> {
   try {
     return await fn(params);
   } catch (e) {
     if (e.business) {
       return { message: e.message };
     } else {
       log(e);
       return { message: "Internal Error" };
     }
   }
 };
};
```

Figure B.1: Several error types were not handled.
(/validator/src/methods/wrapper.ts#L5-20)

Additionally, some error-handling logic appeared to be incomplete. For instance, the code snippet below shows several error types in response to requests to endpoints that obtained bank data for a given user. Several error types were not handled appropriately.

```
 if (response.error) {
   console.log(response.error);
   switch (response.error.code) {
     case "BAD_REQUEST":
       break;
     case "INTERNAL_ERROR":
       break;
     case "PROPERTY_FAILS":
```

```
      break;
    case "UNAUTHORIZED":
      break;
  }
  throw "BANK_INFO_FAILED";
} else {
  // TODO: make sure math checks out with possible values here
  // OR fence against unnatural values
  return response.payload;
}
```

Figure B.2: Several error types were not handled.
(/validator/src/methods/arrowheadCRA/helpers/bank-info.ts#L32-49)

Code 200 returned even on failed requests

All failed requests to the `validator` project, not including those containing invalid JSON requests, returned a 200 response. This can be ineffective from a user experience perspective, and can make it difficult to log and monitor potential issues in the codebase.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
content-length: 62
content-type: application/json; charset=utf-8
Date: Tue, 01 Dec 2020 20:27:46 GMT
Connection: close

{"jsonrpc":"2.0","id":1,"result":{"message":"Internal Error"}}
```

Figure B.3: A 200 response is received even when the request fails.

C. Vulnerable Dependencies

contract-event-listener

CVE	Module	Dependency of	Title	Info
CVE-2019-10744	lodash	knex, lodash, rsmq-worker, eslint	Prototype Pollution	https://npmjs.com/advisories/1523

data-service

CVE	Module	Dependency of	Title	Info
CVE-2020-8116	dot-prop	lerna	Prototype Pollution	https://npmjs.com/advisories/1213
CVE-2020-15168	node-fetch	codecov, lerna	Denial of Service	https://npmjs.com/advisories/1556

subgraph

CVE	Module	Dependency of	Title	Info
N/A	yargs-parser	@graphprotocol/graph-cli	Prototype Pollution	https://npmjs.com/advisories/1500
CVE-2020-7720	node-forge	@graphprotocol/graph-cli	Prototype Pollution in node-forge	https://npmjs.com/advisories/1561

web2

CVE	Module	Dependency of	Title	Info
N/A	web3	web3	Insecure Credential Storage	https://npmjs.com/advisories/877
N/A	serialize-javascript	mocha	Remote Code Execution	https://npmjs.com/advisories/1548