Programowanie Defensywne

Podatności w dokumentach JSON

Arkadiusz Chrobot

Katedra Systemów Informatycznych

11 kwietnia 2024

Plan

1. JavaScript Object Notation
2. Powielone klucze
3. Interpretacja niedopuszczalnych znaków
4. Reprezentacja liczb
5. Niepoprawny dokument JSON
6. Inne podatności
7. Przeciwdziałanie

JavaScript Object Notation

Fragment rozmowy Szefa (SZ) z Kierownikiem Programistów (KP)

...
SZ: Siema! Łącz mnie z Jasonem.
KP: (niesłyszalna kwestia)
SZ: Z Jasonem... Z tym, co nie działają tabelki przez niego na stronie.
KP: (niesłyszalna kwestia)
SZ: Jak to nie pracuje? Kto go zwolnił?
KP: (niesłyszalna kwestia)
SZ: Jak to nie jest człowiekiem? To kogo wyście zatrudnili?
...

Autorzy: HRejterzy, Źródło: https://www.youtube.com/watch?v=FtLiS0V8hkg

JavaScript Object Notation

JavaScript Object Notation (JSON) jest stosunkowo prostym (ang. *lightweight*) formatem dokumentów tekstowych. Zapis danych w tym formacie bazuje na składni języka JavaScript, w szczególności na konstrukcie obiektu, składającym się z par klucz-wartość, oraz tablicy. Jego autorem jest Douglas Crockford. JSON jest powszechnie stosowany w aplikacjach internetowych do wymiany danych. Istnieje kilka dokumentów standaryzujących ten format:

1. RFC 8259 [1],
2. ECMA-262 [2],
3. JSON5 [3],
4. HJSON [4],

Niestety, powstały one w czasach, kiedy już JSON był w użyciu i istniały parsery interpretujące ten język [5]. W związku z tym dokumenty te nie precyzują jak przetwarzać przypadki brzegowe (wymieniony dokument RFC jest jednym z najmniej precyzyjnych), a czasem dodają nowe elementy formatu, które są dosyć kontrowersyjne.

JavaScript Object Notation

Skutek jest taki, że wiele interpreterów JSON, dostępnych w różnych, a czasem w tym samym języku programowania, generuje różne wyniki dla tego samego dokumentu [5, 6]. To może prowadzić do powstania podatności, np. w aplikacji opartej na architekturze mikrousług (ang. *microservices*), gdzie każdy komponent może być zaimplementowany przy użyciu innego języka programowania i przy wykorzystaniu innych bibliotek.

Powielone klucze

Standard JSON zdefiniowany przez RFC 8259 *zaleca*, aby klucze były unikatowe, ale nie określa jak powinien zachować się parser, jeśli w dokumencie znajdą się co najmniej dwa takie same klucze. Podobnie ta kwestia jest traktowana w innych standardach. W istniejących interpreterach JSON ten problem został rozwiązany różnie: część z nich zgłasza błąd po napotkaniu duplikatu klucza i przerywa działanie, część uznaje za prawidłowy tylko pierwszy znaleziony klucz (i związaną z nim wartość), a pozostałe jako taki przyjmują ostatni napotkany klucz (najczęstsze podejście). Przyjrzyjmy się kilku takim parserom i sprawdźmy jakie mogą być konsekwencje takiej niespójności.

Powielone klucze

Załóżmy, że format JSON jest używany w pewnej aplikacji internetowej do tworzenia bazy danych użytkowników lub w komunikatach przesyłanych do komponentu odpowiedzialnego za autoryzację. Przyjmijmy, że struktura typowego wpisu JSON jest następująca:

```
{
        "login":"alice",
        "role":"user"
}
```

Co się stanie, jeśli użytkownikowi o loginie `alice` uda się dodać do tego obiektu dodatkowy klucz? Przykładowo, tak:

```
{
        "login":"alice",
        "role":"user",
        "role":"admin"
}
```

Odpowiedź zależy od użytego parsera. Przyjrzyjmy się zatem kilku z nich.

Powielone klucze

Python

```python
#!/usr/bin/env python3
import json
#import simdjson as json
#import rapidjson as json
#import simplejson as json
import sys

def display(file_name):
    with open(file_name) as file:
        text = json.load(file)
        print(text)

if __name__ == "__main__":
    if len(sys.argv) == 2:
        display(sys.argv[1])
```

Powielone klucze

Python

Program z poprzedniego slajdu pozwala sprawdzić działanie kilku parserów JSON dostępnych w języku Python. Domyślnie używany jest standardowy, z pakietu `json`. Aby uruchomić jeden z pozostałych należy usnąć znak komentarza (`#`) przed odpowiednim wierszem i dodać go przed instrukcją `import json`. Wspomniane parsery to `pysmidjson`, `rapidjson` i `simplejson`. Niestety, wszystkie one uznają za prawidłowy ostatni powielony klucz, zatem wynik ich interpretacji jest następujący:

```
{'login': 'alice', 'role': 'admin'}
```

Zatem użytkownik o loginie `alice` zostanie administratorem systemu.

Powielone klucze

JavaScript

```
'use strict';

const file_stream = require('fs');

if(process.argv.length === 3) {
        let data =
 ↪  file_stream.readFileSync(process.argv[2]);
        let json = JSON.parse(data);
        console.log(json);
}
```

Powielone klucze

JavaScript

Co ciekawe parser JSON dla języka JavaScript (środowisko Node.js) zadziała bardzo podobnie, o czym można się przekonać uruchamiając skrypt z poprzedniego slajdu:

```
node parse.js test2.json
```

```
{ login: 'alice', role: 'admin' }
```

I tym razem za prawidłowy został przyjęty ostatni powielony klucz.

Powielone klucze

Java

Sytuacja ulegnie jednak zmianie, jeśli użyty zostanie parser `jsoniter` (JSON iteratror), który dostępny jest w językach Java i Go.

Powielone klucze

Java

```
package pl.kielce.tu;

import com.jsoniter.JsonIterator;
import com.jsoniter.any.Any;
import java.io.*;
```

Powielone klucze

Java

```
1  public class JSONParser {
2      public static void main(String[] args) {
3          if(args.length==1) {
4              try(BufferedReader reader = new BufferedReader(new
   ↪  FileReader(args[0]))) {
5                  StringBuilder builder = new StringBuilder();
6                  reader.lines().forEach(builder::append);
7                  String text = builder.toString();
8                  Any json = JsonIterator.deserialize(text);
9                  System.out.println(json);
10                 System.out.println(json.get("role"));
11                 System.out.println(json.toString());
12             } catch(IOException exception) {
13                 exception.printStackTrace();
14             }
15         }
16     }
17 }
```

Powielone klucze

Java

Wynik działania tego programu będzie następujący:

```
{ "login": "alice", "role": "user", "role": "admin"}
user
{"login": "alice","role": "admin"}
```

Zwróćmy uwagę, że interpreter zaakceptował zdublowany klucz (wiersz nr 9), ale zapytany o jego wartość zwraca tę, która jest skojarzona z jego pierwszym wystąpieniem (wiersz nr 10). Co interesujące, metoda `toString()` obiektu klasy `Any` (wiersz nr 11) zwraca dokument JSON, który ma tylko jeden klucz `role` z wartością odpowiadającą ostatniemu jego wystąpieniu w oryginalnym dokumencie JSON. To z kolei jest przykład problemu z *serializacją* do formatu JSON. Znane są interpretery JSON dostępne w języku C++ (`rapidjson`), które zachowują oryginalną strukturę dokumentu.

Powielone klucze

JSON Schema

Podobnie jak w przypadku języka XML, dla formatu JSON także opracowano język schematów, który pozwala określić jak dany dokument interpretować. Sprawdźmy, czy pomoże on uniknąć problemów ze spreparowanym wpisem JSON.

Powielone klucze

JSON Schema

```
1  #!/usr/bin/env python3
2  import json
3  import sys
4  from jsonschema import validate
5
6  schema = {
7      "type": "object",
8      "properties": {"login": {"type": "string"}, "role":
       ↪  {"enum": ["user"]}},
9  }
10
11 def display(file_name):
12     with open(file_name) as file:
13         text = json.load(file)
14         validate(instance=text, schema=schema)
15         print(text)
```

Powielone klucze

JSON Schema

```
1  if __name__ == "__main__":
2      if len(sys.argv) == 2:
3          display(sys.argv[1])
```

Powielone klucze

JSON Schema

W przedstawionym skrypcie określono schemat, który zezwala, aby klucz `"role"` przyjmował jedynie wartość `"user"` (wiersz nr 8). Faktycznie, jeśli ten program będzie przetwarzał następujący dokument JSON:

```
{
        "login":"alice",
        "role":"user",
        "role":"admin"
}
```

to uzna go za błędny, ale wystarczy zamienić miejscami dwa wiersze:

```
{
        "login":"alice",
        "role":"admin",
        "role":"user"
}
```

aby uznał go za prawidłowy.

Interpretacja niedopuszczalnych znaków

Parsery JSON różnią się także w zakresie interpretacji niedopuszczalnych znaków, które mogą się pojawić w dokumencie. Przykładowo, znaki o punktach kodowych (ang. *code points*) od `U+D800` do `U+DFFF` są w kodzie UTF-8 uznawane za nieprawidłowe, choć występują w UTF-16. Sprawdźmy do jakich problemów może to prowadzić na przykładzie dwóch parserów dostępnych w języku Python. Załóżmy, że interpretowany dokument będzie ma następującą postać:

```
{
        "login":"alice",
        "role":"user",
        "role\ud888":"admin"
}
```

Interpretacja niedopuszczalnych znaków

Python

```
#!/usr/bin/env python3
import json
import sys

def display(file_name):
    with open(file_name) as file:
        text = json.load(file)
        print(text)

if __name__ == "__main__":
    if len(sys.argv) == 2:
        display(sys.argv[1])
```

Interpretacja niedopuszczalnych znaków

Python

Program posługujący się standardowym parserem JSON zinterpretuje trzeci klucz jako prawidłowy i różny od drugiego, choć powinien uznać go za niepoprawny:

```
{'login': 'alice', 'role': 'user', 'role\ud888': 'admin'}
```

Interpretacja niedopuszczalnych znaków

Python

```python
#!/usr/bin/env python2
import ujson as json
import sys

def display(file_name):
    with open(file_name) as file:
        text = json.load(file)
        print(text)

if __name__ == "__main__":
    if len(sys.argv) == 2:
        display(sys.argv[1])
```

Interpretacja niedopuszczalnych znaków

Python

Jeśli jednak do interpretacji tego dokumentu zostanie użyty język Python w wersji 2.7 i parser z pakietu `ujson`, to wynik będzie inny. Program uzna ostatni klucz za duplikat drugiego i przyjmie jego wartość za prawidłową:

```
{u'login': u'alice', u'role': u'admin'}
```

Reprezentacja liczb

Standard RFC 8259 stanowi, że liczby zmiennoprzecinkowe użyte w dokumencie JSON *powinny* być zgodne ze standardem IEEE *754 binary 64* (podwójna precyzja) i nie powinny być większe. Z kolei liczby całkowite muszą mieścić się w przedziale $[-2^{53}+1, 2^{53}-1]$. A co się stanie, jeśli nie będą spełniały tych warunków? Odpowiedź znowu zależy od użytego interpretera JSON. W przypadku zbyt dużych liczb całkowitych część parserów zinterpretuje ich wartość taką, jaka jest oryginalne. Inne obetną część tej wartości. Jeszcze inne zamienią ją na liczbę zmiennoprzecinkową. W końcu znajdą się i takie parsery, które przyjmą, że jest to liczba 0. Te rozbieżności mogą prowadzić do wielu problemów. Sprawdźmy jak interpretowane są duże liczby zmiennoprzecinkowe na przykładzie 10^{1024} zapisanej w dokumencie:

```
{"number":1E1024}
```

Reprezentacja liczb

Python i JavaScript

Prezentowany wcześniej standardowy parser dostępny w języku Python zinterpretuje tę liczbę jako plus nieskończoność:

```
{'number': inf}
```

To samo zrobi parser JSON w również wcześniej zaprezentowanym skrypcie JavaScript przeznaczonym dla środowiska Node.js:

```
{ number: Infinity }
```

Inaczej zachowa się jednak parser ▸ jansson dostępny w języku C.

Reprezentacja liczb

c

```c
#include<stdio.h>
#include<string.h>
#include<jansson.h>
```

Reprezentacja liczb

C

```c
void read_json_file(char file_name[])
{
    FILE *file = fopen(file_name,"r");
    if(file) {
        char buffer[500];
        char json[5000]={'\0'};
        while(fgets(buffer,sizeof(buffer),file)) {
            //To nie jest bezpieczne rozwiązanie.
            strncat(json,buffer,strlen(buffer));
        }
        printf("%s",json);
        json_error_t exception;
        json_t *root = json_loads(json,0,&exception);
```

Reprezentacja liczb

c

```
        if(root) {
            if(json_typeof(root)==JSON_OBJECT) {
                const char *key;
                json_t *value;
                json_object_foreach(root,key,value) {
                    printf("klucz: %s\n",key);
                    printf("wartość:
↪   %f\n",json_real_value(value));
                }
            }
            json_decref(root);

        } else {
            fprintf(stderr,"Błąd przetwarzania JSON w
↪   wierszu nr %d, tekst:
↪   %s\n",exception.line,exception.text);
```

Reprezentacja liczb

C

```
        if(fclose(file))
            perror("fclose()");
    } else
        perror("fopen()");
}
```

Reprezentacja liczb

C

```c
int main(int argc, char **argv)
{
    if(argc==2) {
        read_json_file(argv[1]);
    }
    return 0;
}
```

Reprezentacja liczb

C

Ten interpreter uzna, że wartość dla klucza `"number"` jest błędna i przerwie interpretację dokumentu:

```
./parser: /usr/lib/x86_64-linux-gnu/libjansson.so.4: no
 ↪  version information available (required by
 ↪  ./parser)
{"number":1E1024}
Błąd przetwarzania JSON w wierszu nr 1, tekst: real
 ↪  number overflow near '1E1024'
```

Niepoprawny dokument JSON

Żaden ze standardów nie precyzuje również jak powinien się zachować parser JSON, jeśli będzie interpretował niepoprawny dokument, np. taki:

```
{"key":"value"}=
```

Nadmiarowy znak = użyty w żądaniach HTTP może dostarczyć potencjalnemu intruzowi możliwości przeprowadzenia ataku typ CSRF (ang. *Cross-Site Request Forgery*). Wszystko zależy od tego, jak potraktuje go parser JSON. Źle skonstruowane dokumenty JSON mogą być również użyte do przeprowadzania ataków typu DoS.

Inne podatności

Podobne wady posiadają również binarne wersje JSON (BSON, CBOR i inne) [5]. Postać tekstowa JSON może być wykorzystana w atakach polegających na wstrzyknięciu złośliwego kodu (JSON Injection) [7]. Ich celem jest zmiana DOM strony internetowej, tak aby np. pojawił się na niej link, który kliknięty przez jej użytkownika spowoduje wykonanie niepożądanej czynności. Potencjalnie niebezpieczne są metody JavaScript, które jako argument przyjmują dane z niezaufanego źródła, np.:

```
JSON.parse()
jQuery.parseJSON()
$.parseJSON()
```

Dokumenty JSON mogą być także wykorzystane w atakach przeciwko stosowanej w przeglądarkach polityce tego samego pochodzenia (ang. *Same Origin Policy*). Może ona nie obejmować kodu JavaScript, w związku z tym obiekty JSON mogą być użyte do kradzieży danych [8]. Tego typu ataki są znane jako „przejęcie JSON” (ang. *JSON Hijacking*).

Inne podatności

Odmiana formatu JSON — JSONP (ang. *JSON with Padding*) może być użyta do obejścia polityki CSP w przeglądarce i wstrzyknięcia kodu JavaScript (atak XSS).

Przeciwdziałanie

Jak się przekonaliśmy wcześniej, walidatory bazujące na schematach przetwarzania JSON nie są skuteczne w ochronie przed atakami wykorzystującymi podatności związane z przetwarzaniem takich dokumentów. Mogą być jedynie w pewnym zakresie pomocne. Skuteczną ochronę przed takim zagrożeniem zapewniają jedynie decyzje podjęte na etapie projektowania i implementacji aplikacji.

1. Jeśli jest to możliwe, to należy użyć tylko jednego języka programowania i tylko jednej implementacji parsera JSON.
2. Jeśli konieczne jest użycie wielu parserów lub języków programowania, to należy wybrać tylko te, które interpretują ten format w ten sam sposób.
3. Jeśli użyte parsery JSON mają różne opcje konfiguracji, to należy użyć we wszystkich komponentach takich, które zapewnią tę samą interpretację formatu JSON.
4. Jeśli komunikaty lub pliki w formacie JSON mogą być przechowywane lub przesyłane poza zaufanymi komponentami, to należy zastosować do ich ochrony podpisy kryptograficzne.

- *The JavaScript Object Notation (JSON) Data Interchange Format — RFC 8259*. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259.
- *ECMA-262, 13th edition, June 2022 ECMAScript® 2022 Language Specification*. 2022. URL: https://262.ecma-international.org/.
- *JSON5 — JSON for Humans*. 2023. URL: https://json5.org/.
- *Hjson, a user interface for JSON*. 2023. URL: https://hjson.github.io/.
- Jake Miller. *An Exploration of JSON Interoperability Vulnerabilities*. 2021. URL: https://bishopfox.com/blog/json-interoperability-vulnerabilities.
- Nicolas Seriot. *Parsing JSON is a Minefield*. 2018. URL: https://seriot.ch/projects/parsing_json.html.

- *DOM-based client-side JSON injection*. 2023. URL: https://portswigger.net/web-security/dom-based/client-side-json-injection.
- Gareth Heyes. *JSON hijacking For the modern web*. 2023. URL: https://owasp.org/www-pdf-archive/OWASPLondon20161124_JSON_Hijacking_Gareth_Heyes.pdf.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!