CARNET

Sigurnost HTTP REST API-ja

CERT.hr-PUBDOC-2020-3-398

CERT.hr

Sadržaj

Ovaj dokument izradio je Laboratorij za sustave i signale Zavoda za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

1 Uvod

Potreba za razvojem sve većeg broja *web* aplikacija u što kraćem vremenskom roku rezultirala je promjenom u arhitekturnom pristupu. Za razliku od tradicionalnih monolitnih aplikacija koje su izgrađene kao jedna cjelina, moderne aplikacije nastoje se sve više modularizirati, tj. rastaviti na komponente (mikrousluge) od kojih svaka ima svoj zadatak, a zajedno surađuju kako bi izvršile neki određeni posao. Takva arhitektura omogućava brži i lakši razvoj *web* i mobilnih aplikacija iz nekoliko razloga:

- Programeri mogu istovremeno i neovisno raditi na različitim komponentama, pri čemu promjene ili pogreške na jednoj komponenti neće utjecati na ostale.
- Komponente se mogu koristiti za više aplikacija.
- Programeri jedne komponente ne moraju se baviti detaljima implementacije druge komponente, bitno im je samo znati kako pozvati uslugu i kakav će odgovor dobiti.
- Bitno je lakše testiranje i dijagnostika kod pogrešaka cjelokupnog sustava.
- Svaka komponenta može biti izrađena potpuno drugačijim programskim jezikom i okolinom.

Kako bi mogle surađivati i razmjenjivati usluge i podatke, te komponente komuniciraju preko API-ja (engl. *Application Programming Interface*), tj. programskog sučelja aplikacije. API-jem aplikacije mogu tražiti usluge i podatke operacijskog sustava ili neke druge aplikacije, a na korištenje API-ja se oslanjaju i mobilne aplikacije, kao i bogate *JavaScript* razvojne okoline i biblioteke poput *Angulara* i *Reacta*. Aplikacija koja poziva neki API mora voditi računa samo o tome kako sastaviti i poslati zahtjev (engl. *request*) i kakav će odgovor dobiti (engl. *response*), bez vođenja računa o implementacijskim detaljima funkcije koja će mu pružiti odgovor.

Web API-ji su u osnovi jednostavne okoline tipa klijent-poslužitelj koje komuniciraju putem HTTP protokola. Kako bi ta komunikacija bila jednostavnija i univerzalnija, razvijeno je nekoliko protokola kojih se API-ji pridržavaju za obradu zahtjeva od kojih su najpoznatiji REST i SOAP.

REST je prevladavajući arhitekturni stil koji se u praksi uvijek koristi kroz HTTP aplikacijski protokol. Jednostavan je za implementaciju, razumijevanje i rukovanje pa je zbog toga široko prihvaćen među programerima.

Osim REST-a, za komunikaciju među API-jima može se koristiti i protokol SOAP zasnovan na XML-u, ali u novije vrijeme sve više se preferira REST upravo zbog jednostavnosti korištenja i manjeg zahtjeva na propusnost mreže (engl. *bandwidth*) za slanje zahtjeva i primanje odgovora.

Korištenje (REST) API-ja je olakšalo, organiziralo i ubrzalo razvoj aplikacija, a komunikacija među njima odvija se putem mreže (i to uglavnom preko HTTP protokola). Zahvaljujući takvoj modularnoj arhitekturi čije komponente međusobno komuniciraju mrežom, povećao se i prostor za napad na aplikaciju jer sad napadač ima više krajnjih točaka (engl. *endpoints*) koje može pokušati napasti. Što je više komponenti, veća je šansa

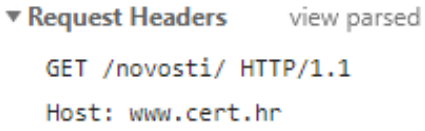
da postoji neka pogreška u konfiguraciji ili programskom kôdu neke od njih. Također, napadač može pokušati prisluškivati ili presretati mrežni promet među komponentama.

Tema ovog dokumenta upravo je sigurnost HTTP REST API-ja i objasnit će česte uobičajene ranjivosti koje se u njima javljaju, kao i mjere zaštite.

2 Sigurnost REST API-ja

REST je kratica za *Representational State Transfer*, a odnosi se na arhitekturni stil koji je najzastupljeniji u izradi *web* (internetskih) aplikacija. REST se oslanja na HTTP aplikacijski protokol koji se standardno koristi prilikom pregledavanja *weba*. Zapravo, cijeli *web* prostor na internetu koristi REST arhitekturu.

Uzmimo za primjer pregledavanje *web* stranice Nacionalnog CERT-a s novostima na adresi https://www.cert.hr/novosti/. Kako bi posjetitelju prikazao stranicu, *web* preglednik prvo mora uspostaviti komunikaciju s poslužiteljem na kojem se nalazi stranica i poslati GET zahtjev za dohvat resursa na adresi https://www.cert.hr/novosti/. Zaglavlje (engl. *header*) poslanog zahtjeva izgledalo bi kao na slici 1.



Slika 1 ***Zaglavlje zahtjeva za dohvat CERT-ove web stranice s novostima***

Takav zahtjev govori poslužitelju na adresi www.cert.hr (*host*) da klijentov *web* preglednik želi dohvatiti resurs/e koji se nalazi u mapi `novosti` pohranjenoj na poslužitelju inačicom protokola HTTP 1.1. Svaki upit i odgovor na *webu* sastoji se od HTTP zaglavlja (engl. *header*) i tijela (engl. *body*). Najvažniji elementi zaglavlja su:

- *Request URL* – puna adresa na koju se šalje upit (u ovom slučaju https://www.cert.hr/novosti/)
- *Request Method* – jedna od HTTP metoda (u ovom slučaju GET)
- *Status Code* – HTTP status kôd (u ovom slučaju 200 OK)
- *Accept* – vrsta podatka koju server ili klijent prihvaća kao odgovor (u ovom slučaju text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9)
- *Host* – naziv računala i TCP port na kojem računalo "sluša" i prima zahtjeve (u ovom slučaju www.cert.hr, i, budući da nije naveden, standardni port za HTTPS prokol 443)

Web poslužitelj odgovara *web* pregledniku i, ako je zahtjev bio ispravan i korisnik smije pregledavati navedenu stranicu, šalje mu traženu *web* stranicu kao odgovor. Stranica je, u pravilu, kodirana u skladu s HTML standardom. Ako stranica koristi slike, programski kôd i druge elemente, oni se najčešće nalaze u odvojenim datotekama, pa će za pregledavanje neke prosječne *web* stranice *web* preglednik poslati na desetke zahtjeva za CSS i *JavaScript* datoteke, slike, itd.

Navedeni scenarij je naizgled nepovezan s REST API-jima jer je opisano što se događa prilikom uobičajenog pregledavanja *weba*, ali REST API-ji rade istu stvar prilikom dohvata stranice, uz razliku da klijentu (onom tko je poslao zahtjev, a to može biti *web* preglednik,

neka aplikacija, drugi API itd.) ne mora slati datoteke kodirane/formatirane prema HTML standardu, već mogu koristiti neki drugi ili posebno dogovoreni format odgovora kojeg će drugi program znati lako pročitati i napraviti dalje s njim što je potrebno. Ti formati su uobičajeno JSON ili XML. REST ne omogućava nikakve dodatne funkcionalnosti koje već nema protokol HTTP, već je REST arhitekturni stil koji koristi HTTP protokol kako bi dohvatio i slao informacije na način da je drugim aplikacijama (a ne čovjeku) njima jednostavno rukovati.

Zahtjev koji bi poslao REST API ne bi navodio put do resursa, nego glagol iz kojeg je razumljivo koji je cilj zahtjeva, npr.:

`GET /dohvatiNovosti` ili

`GET /getNews`

A odgovor bi mogao stići u npr. jednom od formata koji su prikazani na slici **Error! R eference source not found.**:

```
{
   "username" : "my_username",
   "password" : "my_password",
   "validation-factors" : {
      "validationFactors" : [
         {
            "name" : "remote_address",
            "value" : "127.0.0.1"
         }
      ]
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<authentication-context>
  <username>my_username</username>
  <password>my_password</password>
  <validation-factors>
    <validation-factor>
      <name>remote_address</name>
      <value>127.0.0.1</value>
    </validation-factor>
  </validation-factors>
</authentication-context>
```

Slika 2 *JSON (lijevo) i XML (desno) format* (1)

Iako su najčešći, JSON i XML formati razmjene informacija među RESTful API-jima nisu standard i programer aplikacije se isto tako može odlučiti i za korištenje nekog drugog ili svog, npr. binarnog formata.

Vidjeli smo da REST API-ji funkcioniraju dosta slično kao i uobičajene *web* aplikacije. Sukladno tome je i napad na REST API vrlo sličan napadu na aplikaciju i većina ranjivosti koja se javlja u API-jima se u pravilu javlja i u uobičajenim *web* aplikacijama. Ipak, postoje i neke posebnosti kojih ćemo se dotaknuti u određenim poglavljima dokumenta.

Kako je uopće moguće napasti API ili aplikaciju? Prvi korak napada je pronaći ranjivost – pogrešku u programskom kôdu ili konfiguraciji API-ja/aplikacije. Jednom kad pronađe ranjivost, napadač je može zloupotrijebiti i preko nje napasti stranicu (doći do osjetljivih podataka, narušiti integritet/dostupnost usluge...) ili njene korisnike. Neovisno o broju provjera i sigurnosnom testiranju, vrlo je vjerojatno da će svaka aplikacija ili API imati barem nekakvu pogrešku. No, nije svaka pogreška nužno ranjivost i nije svaka ranjivost jednako rizična. Bitno je razumjeti namjenu API-ja i kako i gdje se koristi kako bi se mogla procijeniti rizičnost, a rizičnost je bitna kako bi se mogli predvidjeti najgori scenariji

napada i prioritizirati ispravljanje pogrešaka. Uzmimo za primjer XSS – iako je XSS ozbiljna i vrlo rizična ranjivost koja može dovesti do krađe osjetljivih podataka posjetitelja određene *web* stranice ili izvršavanja određenih aktivnosti u njegovo ime, XSS ranjivost u API-ju koji samo pohranjuje korisničke unose u bazu i nikad ih ne izvršava unutar nekog preglednika nije rizična.

Ovaj dokument prvenstveno je namijenjen programerima kojima će dati širu sliku u sigurnosnu stranu aplikacije, predstaviti im uobičajene sigurnosne ranjivosti i napade i time ih upoznati sa sigurnosnim mjerama na koje moraju obratiti pažnju prilikom razvoja REST API-ja (ali i *web* aplikacija općenito). Ranjivosti koje će se opisati i prokomentirati kroz dokument su 10 najčešće viđenih ranjivosti u 2017. godini koje je popisala i poredala po rizičnosti organizacija OWASP, globalni autoritet kad je u pitanju *web* sigurnost (2).

OWASP Top 10 - 2017
A1:2017-Injection
A2:2017-Broken Authentication
A3:2017-Sensitive Data Exposure
A4:2017-XML External Entities (XXE)
A5:2017-Broken Access Control
A6:2017-Security Misconfiguration
A7:2017-Cross-Site Scripting (XSS)
A8:2017-Insecure Deserialization
A9:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

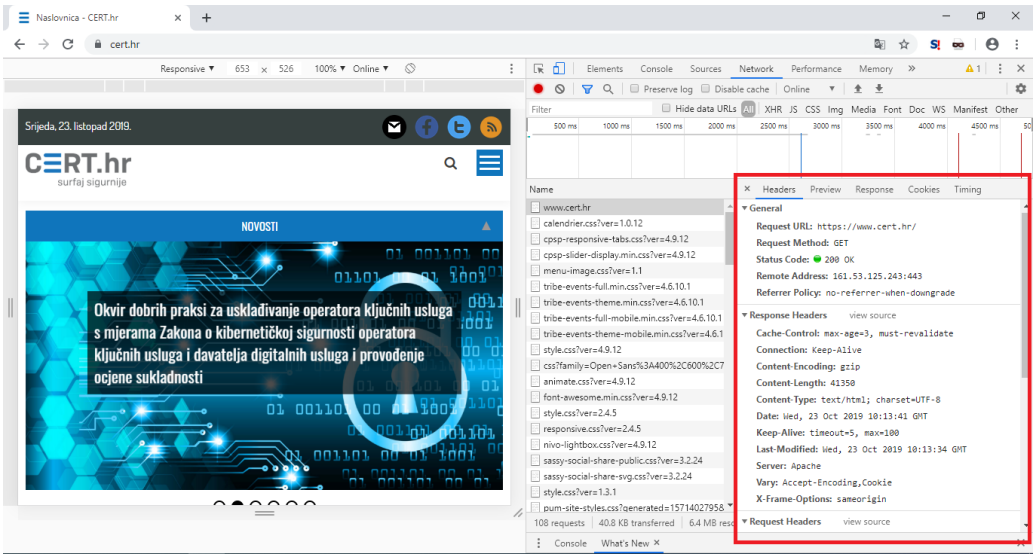
Slika 3 *10 najčešćih i najrizičnijih ranjivosti u 2017. godini prema organizaciji OWASP* (2)

Za demonstraciju napada na ranjivosti koristit će se *Vulnerable API* - ranjivi API koji je razvijen upravo kao primjer API-ja s nekim čestim ranjivostima na kojem se korisnici mogu upoznati s određenim propustima i pogreškama i pokušati napasti API preko tih ranjivosti. Njegov izvorni kôd dostupan je u repozitoriju.

Vulnerable API od korisnika traži autentifikaciju korisničkim imenom i lozinkom, nakon čega mu dodjeljuje autentifikacijski token. U bazi postoji više korisničkih zapisa, ali svaki korisnik smije pregledavati samo svoje vlastite zapise slanjem zahtjeva `GET /user/USER_ID` i autentifikacijskog tokena u polju zaglavlja *X-Auth-Token*. Ako je token ispravan i pripada korisniku čiji se zapis pokušava pregledati, korisniku će API odgovoriti zapisom.

Općenito, za slanje zahtjeva (i testiranje) API-ja mogu se koristiti:

- *Developer Tools* unutar preglednika (desni klik -> *Inspect Element* -> *Network*)



Slika 4 *Developer Tools unutar preglednika Mozzila Firefox*

- presretajući *web* posrednik (eng. *intercepting proxy*) poput OWASP ZAP-a ili *Burp Suitea*
- programski alati za slanje HTTP zahtjeva poput *cURL*-a
- napredniji alati za slanje HTTP zahtjeva s dodatnim funkcionalnostima kao što su *Postman*, *Insomnia*, *SoapUI*...
- kombinacije programskih jezika i biblioteka (npr. *Python* + biblioteka *requests*)

Kroz ovaj dokument za slanje zahtjeva *Vulnerable API*-ju koristit će se *Postman* - komercijalna *desktop* aplikacija, dostupna za preuzimanje sa službene stranice. *Postman* se može koristiti sa ili bez registracije, ali za sve funkcionalnosti koje će se koristiti u ovom dokumentu nije potrebna registracija.

Alati popust *Postmana* nisu korisni samo za jednokratna testiranja, već se mogu koristiti i za sastavljanje automatiziranih (i funkcionalnih i sigurnosnih) testova koji se mogu uključiti u sam proces razvoja aplikacije. Na taj način programeri prilikom nadograđivanja API-ja, objavljivanja nove inačice ili izmjene/ispravljanja postojećih funkcionalnosti mogu u *Postmanu* pokrenuti unaprijed pripremljene testove i provjeriti podudaraju li se odgovori s ispravnima i očekivanima ili je došlo do nekakve pogreške. Ne može se očekivati da takvi automatizirani testovi provjere baš sve ranjivosti, ali mogu provjeriti veliki dio, što uvelike olakšava sigurnosno testiranje i testiranje općenito.

2.1 A1 – Umetanje (engl. *Injection*)

Umetanje se odnosi na podmetanje napadačevog kôda ili naredbe koje će API zatim na prevaru, smatrajući da je riječ o legitimnom kôdu, izvršiti. Umetanjem kôda napadač može pristupiti podacima i operacijama na koje nema pravo. Umetnuti se može JavaScript kôd, naredba ljuski operacijskog sustava, SQL, LDAP ili NoSQL upit itd. Npr. ako API očekuje da mu se pošalju korisničko ime i lozinka prema kojima će zatim pretražiti bazu i vidjeti postoji li takav korisnik, napadač može umjesto očekivanih podataka unijeti i neočekivani

kôd. Jednom kad API prihvati podmetnut kôd i proslijedi ga na izvršavanje bazi, operacijskom sustavu ili kome je već namijenjen, ne postoji mehanizam kojim bi npr. baza razlikovala legitiman kôd od zlonamjernog već će izvršiti svaki proslijeđeni upit.

Iz tog razloga vrlo je bitno ograničiti korisnikov unos –ne možemo vjerovati da će korisnik uistinu unijeti samo ono što od njega očekujemo i zato ga moramo spriječiti da uspije umetnuti nešto zlonamjerno. Drugim riječima, svaki korisnički unos mora se validirati, filtrirati i sanitizirati i tek tada prihvatiti i proslijediti dalje na izvršavanje. Iako na prvi pogled djeluje jednostavno, a postoje i već gotove funkcije u raznim programskim jezicima koje validiraju korisnički unos, OWASP je uočio da je ova ranjivost česta među *web* aplikacijama i API-jima i kategorizirao ju kao najrizičniju ranjivost.

Promotrimo važnost ograničenja korisničkog unosa na naizgled trivijalnom primjeru: zamislimo da postoji *online* trgovina čiji programski kôd ne provjerava je li iznos plaćanja ili količina kupljenih artikala broj veći od 0. Tad bi korisnik mogao unijeti negativnu vrijednost za količinu kupljenih artikala i umjesto platiti određeni iznos, dobiti te novce na svoj račun.

Vratimo se na primjer ranjivog REST API-ja - *Vulnerable* API. Korisnik može uputiti zahtjev *Vulnerable* API-ju s naredbom *uptime* operacijskom sustavu kako bi saznao koje je vrijeme na poslužitelju na kojem je *Vulnerable* API pohranjen. Osim naredbe *uptime*, korisnik može unijeti i zastavicu (engl. *flag*) koja će se pridodati naredbi, a njom korisnik može dobiti vrijeme u određenom formatu. Detalji o ovoj naredbi i njenim zastavicama mogu se pogledati na poveznici.

Bitno je za primijetiti da je u dokumentaciji *Vulnerable API*-ja navedeno je da se API-ju mora poslati GET zahtjev sljedećeg oblika kako bi se saznalo vrijeme na poslužitelju:

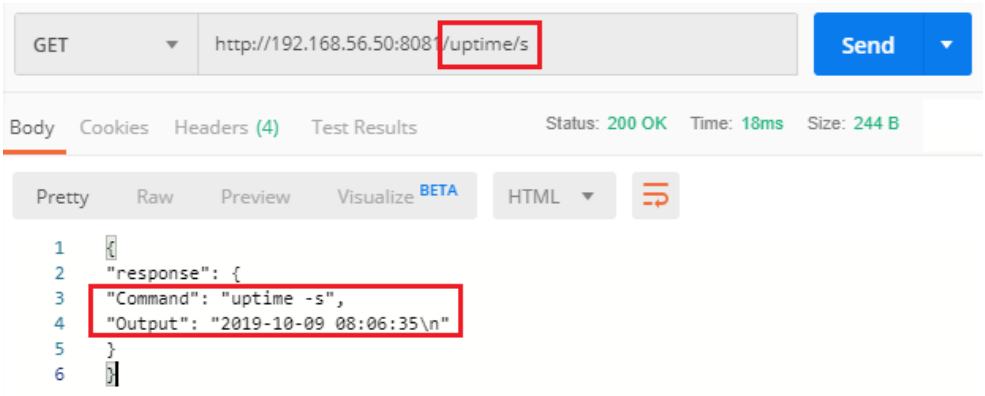
```
GET /uptime/FLAG
```

Takav će zahtjev tražiti od operacijskog sustava da izvrši naredbu:

```
uptime -FLAG
```

Već smo napomenuli da API-ji moraju pratiti dogovoreni standard i poslati zahtjev u točno određenom obliku kako bi dobili odgovor. Ako ne bismo pratili pravila iz dokumentacije, već poslali npr. zahtjev `POST /uptime/FLAG` ili `GET /time/FLAG`, ne bismo dobili očekivani odgovor.

Primjer legitimnog i očekivanog zahtjeva te odgovora na zahtjev prikazan je na slici 5.

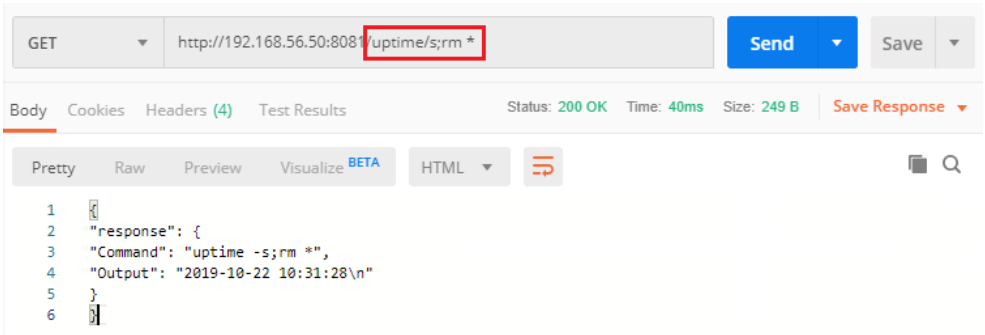


Slika 5 ***Pozivanje legitimne naredbe „uptime –s“***

No, isprobavanjem različitih kombinacija zastavica zaključujemo da *Vulnerable* API u stvari uopće ne provjerava što je korisnik unio na mjestu gdje se očekuje zastavica, već njegov unos, kakav god da je, proslijedi na izvršavanje naredbenom retku operacijskog sustava. Zbog toga napadač može umjesto zastavice unijeti i neku svoju zlonamjernu naredbu, pri čemu bi jedan primjer napada bio:

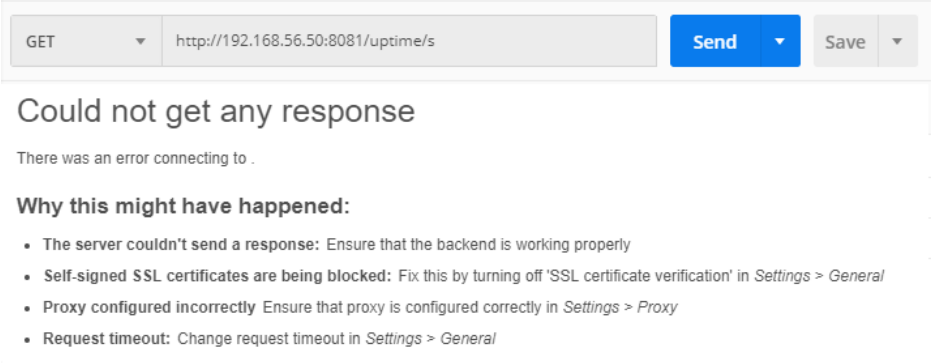
1) Napadač šalje GET zahtjev oblika: `GET /uptime/s;rm *`

2) API će proslijediti na izvršavanje i prikazati rezultat naredbe `uptime -s`, ali znak „;“ kojeg je napadač podmetnuo govori ljuski operacijskog sustava da treba izvršiti i još jednu naredbu, a to je naredba `rm *` koja briše sve datoteke iz trenutnog direktorija



Slika 6 ***Umetanje naredbe operacijskom sustavu (engl. Command Injection)***

3) API je postao nedostupan (jer su obrisane sve datoteke i skripte koje su mu potrebne za rad)



Slika 7 ***Aplikacija je nedostupna jer je napadač umetanjem naredbe obrisao sve datoteke***

Okvirne smjernice za sprječavanje napada umetanjem temelje se na tome da **nikad ne treba vjerovati korisnikovom unosu**, već se taj unos uvijek treba ograničiti, filtrirati, validirati, sanitizirati i odbiti nedozvoljene znakove (npr. u demonstriranom slučaju ne bi moglo doći do izvršavanja zlonamjerne naredbe ako bi se odbio znak „;“ ili sastavila lista zastavica koje se smiju unijeti uz naredbu *uptime*). Osim toga, dobra je praksa i ograničiti unos znakovnih nizova regularnim izrazima (engl. *regex*) i (ako to ima smisla i primjenjivo je u aplikaciji) ograničiti veličinu zahtjeva te odbiti svaki zahtjev koji je veći od dozvoljenog.

Naravno, kako bi se provjerilo da je umetanje kôda spriječeno ili barem minimizirano, potrebno je testirati sve parametre, zaglavlja, URL-ove, kolačiće i ulazne podatke kako bi se provjerilo da se putem njih ne može unijeti zlonamjerni sadržaj. Za ovakvo testiranje iznimno je korisno tzv. *fuzzanje* parametara koje je ugrađeno u naprednije alate za slanje zahtjeva, a time i u *Postman*. *Fuzzanje* parametara je automatizirano, uzastopno isprobavanje različitih vrijednosti određenog parametra. Drugim riječima, *fuzzanje* parametara će za svaki testirani parametar (koji može biti dio URL-a na koji se šalje zahtjev, dio zaglavlja ili tijela zahtjeva) isprobati različite vrijednosti od kojih su neke legitimne, a neke nisu, i provjeriti odbija li API nelegitimne vrijednosti parametara.

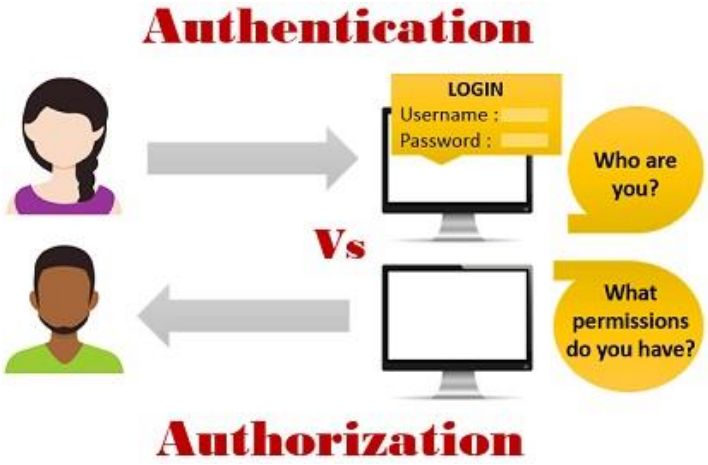
2.2 A2 – Neučinkovita autentifikacija (engl. *Broken Authentication*)

Kako bi se koristio neki API (a isto vrijedi i za *web* aplikacije), kako bi se pristupalo njegovim resursima i uspješno mu se slali zahtjevi i dobivali odgovori, često je prvo potrebno potvrditi da smo onaj korisnik koji ima na to pravo, i to u dva koraka:

1. **Autentifikacija**. Korisnik mora dokazati da je stvarno onaj za kojeg se predstavlja, tj. da je stvarno on vlasnik korisničkog računa kojim se pokušava prijaviti. Npr. ako se korisnik pokušava prijaviti adresom e-pošte ja@ja.hr, mora znati lozinku koja pripada tom računu. API-ji mogu implementirati nekoliko načina autentifikacije koji se koriste i kod običnih *web* aplikacija (*Basic* autentifikacija, kolačići, tokeni…). Iznimka je JWT token (*JSON Web Token*) karakterističan za API-je, a koji će se detaljnije objasniti u poglavlju o neučinkovitoj kontroli pristupa.

2. **Autorizacija**: Korisnik mora imati prava na obavljanje određenih radnji i pristup određenim resursima preko API-ja. Postoje različite razine ovlasti za korištenje API-ja i *web* aplikacija (npr. običan korisnik, privilegirani korisnik, administrator…) i nema svatko pravo obavljati sve radnje. Mora postojati valjana

kontrola pristupa kako bi se spriječile određene radnje ili pristup osjetljivim podacima korisnicima bez potrebnih privilegija.



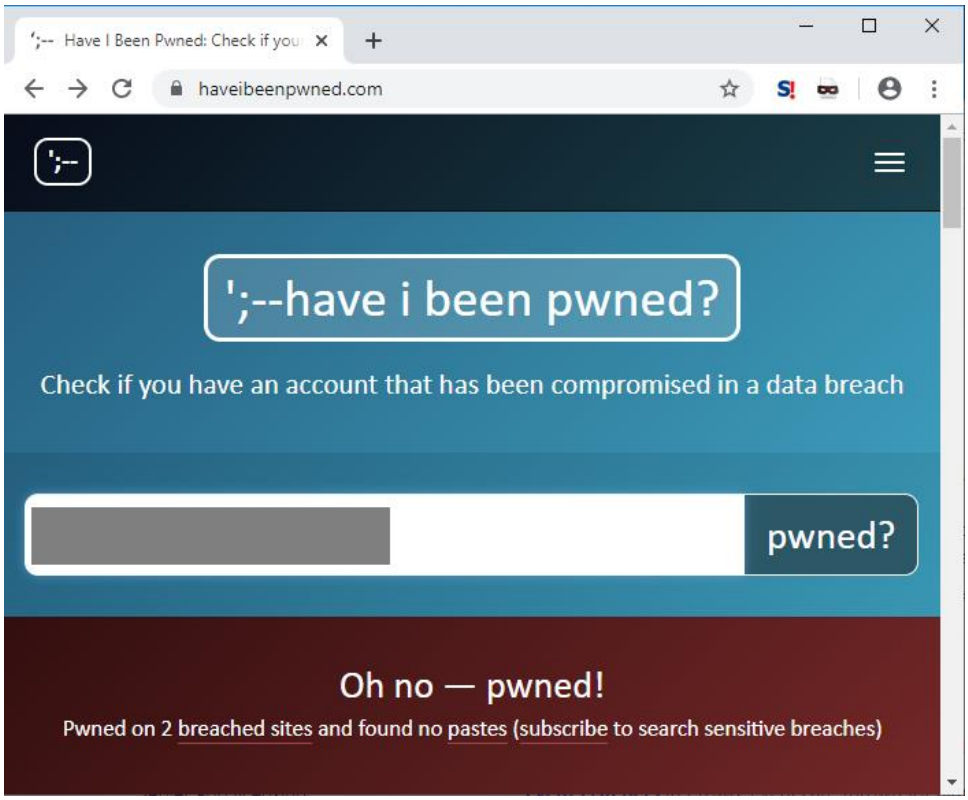
Slika 8 *Autentifikacija i autorizacija* (3)

Neučinkovita autentifikacija je ranjivost koja omogućuje napadaču da se uspješno prijavi putem korisničkog računa nekog drugog, legitimnog korisnika u čiji je račun „provalio“, tj. preuzeo njegov račun npr. pogađanjem lozinke. Posebno je opasno ako napadač uspije preuzeti račun korisnika s administratorskim ovlastima jer tad može napraviti najviše štete i pristupiti osjetljivim podacima.

Cilj ovakvog napada je preuzeti jedan (idealno administratorski) ili više korisničkih računa i steći sve privilegije koje ima i žrtva. Neke od često viđenih metoda kojima napadači pokušavaju preuzeti račune su:

- ***Brute-force* napad.** Metodom uzastopnih pokušaja napadač isprobava razne kombinacije znakova za korisnička imena i lozinke. O složenosti i duljini lozinke ovisi i vrijeme potrebno da napadač uspješno pogodi lozinku.

- ***Credential stuffing* napad**. Napadač je već prikupio postojeće kombinacije korisničkih imena i lozinki iz neke *online* baze s podacima koji su „iscurili“ s neke druge *web* aplikacije ili API-ja (npr. s *LinkedIna* je 2012. godine procurilo oko 6.5 milijuna *hasheva* korisničkih lozinki (4)). Pokušava primijeniti te kombinacije na veći broj drugih stranica u nadi da korisnik još negdje koristi isto korisničko ime i lozinku. Također, ti podaci mu mogu pomoći pri sastavljanju rječnika za napad jer može primijetiti koje se lozinke najčešće koriste kao i koje su lozinke karakteristične za nekog korisnika. Jedna korisna stranica na kojoj se može provjeriti nalazi li se vaša adresa e-pošte u nekoj bazi podataka koja je kompromitirana je haveibeenpwned.com.



Slika 9 ***Vjerodajnica s upisanom adresom e-pošte iscurila je s dvije web stranice***

- ***Dictionary attack.*** Napadač unaprijed priprema rječnik s najčešćim lozinkama i raznim riječima i redom ih isprobava. Postoje alati poput *THC Hydre* koji pomažu napadaču pri *online* pogađanju lozinke.

- **Krađa autentifikacijskog tokena ili kolačića.** Jedna od karakteristika HTTP protokola je da je tzv. *stateless*, tj. ne pamti prethodna stanja. To znači da poslužitelj svaki put kad stigne zahtjev mora provjeriti korisničko ime i lozinku i utvrditi da je taj korisnik autentificiran i autoriziran. Kako se zahtjev ne bi morao svaki put slati s korisničkim imenom i lozinkom, poslužitelj dodijeli klijentu jedinstveni autentifikacijski token ili kolačić po kojem ga dalje prepoznaje. Napadač može prisluškivanjem mrežnog prometa ili na neke druge načine saznati korisnikov token/kolačić i slati zahtjeve API-ju s njim. API će smatrati da je riječ o legitimnom korisniku i dopustiti napadaču radnje u ime žrtve.

Odgovornost za ovakve napade dijelom leži na korisniku API-ja/aplikacije, a dijelom na njihovim programerima. S korisničke strane nailazimo na probleme jednostavnih i predvidljivih šifri koje napadač lako pogodi, kao i „recikliranju“ lozinki. Lozinke dobrog dijela korisnika interneta su slabe, predvidljive i ponavljaju se na većem broju stranica na koje se registriraju kako bi ih lakše zapamtili. Koliko god da se ljude educira o važnosti složene i sigurne lozinke, nerijetko će ipak odabrati kraću i jednostavniju lozinku kako bi je uspjeli zapamtiti.

Zato je s programerske strane preporučljivo korisnika „prisiliti“ na sigurnu lozinku, tj. postaviti minimalnu duljinu lozinke i tražiti više vrsta alfanumeričkih znakova (brojevi, specijalni znakovi, slova…). Time će se otežati probijanje korisničkog računa. Također, treba izbjegavati slabe procedure za ponovno postavljanje lozinke poput jednostavnih

sigurnosnih pitanja na koje napadač može pogoditi odgovor (npr. Gdje ste rođeni?). U slučaju većeg broja neuspješnih prijava (npr. 10 uzastopnih neuspješnih pokušaja prijave) potrebno je privremeno onemogućiti korisnički račun kako bi se spriječio napad uzastopnim pokušajima (engl. *brute-force*). Račun bi trebao biti onemogućen dovoljno dugo da se spriječe napadi pogađanja vjerodajnica i uzastopnih pokušaja, ali opet ne predugo kako ne bi došlo do uskraćene usluge (engl. *Denial of Service*). Iako to nije svugdje moguće, dobra je praksa koristiti višefaktorsku autentifikaciju.

S druge strane, česte ranjivosti API-ja koje se odnose isključivo na programersku stranu i koje se mogu spriječiti su:

- **Korisničke vjerodajnice se pohranjuju nezaštićene (tj. nešifrirane).** Unutarnji zaposlenik ili napadač koji je uspio doći do baze podataka sada vidi lozinke svih korisnika jer su pohrajene u vidljivom obliku (engl. *clear text*) umjesto da su *hashirane* ili su *hashirane* slabim algoritmom. Napadač može pokušati iskoristiti te kombinacije korisničkih imena/adresa e-pošte i lozinki i na drugim mjestima (što bi mu moglo uspjeti jer korisnici često koriste iste lozinke). Za *hashiranje* lozinki treba se, osim dobrog kriptografskog algoritma, koristiti i *salt* koji mora biti jedinstven za svakog korisnika i ne smije biti prekratak.

- **Predvidljive ili jednostavne vjerodajnice koje napadač lako pogodi.** Iako je odgovornost za „čvrstu" lozinku prvenstveno na korisniku, API bi trebao spriječiti korisnika da, primjerice, za lozinku postavi neku kratku riječ od svega nekoliko znakova iste vrste. Napadač može pokušati pogoditi predvidljive vjerodajnice (npr. admin/admin) ili pronaći takve lozinke *brute-force* ili *dictionary* napadima.

- ***Session* ID korisnika je vidljiv u URL-u (tzv. *URL rewriting*).** Kad se prijavi, korisnik vidi svoj *Session ID* u URL-u. Svaki put kad kopira URL i nekome ga pošalje (npr. želi nešto nekome pokazati), taj netko se može prijaviti s njegovim *Session ID*-jem čime dobiva pristup njegovom računu i može koristiti njegovu kreditnu karticu i ostale podatke.

- ***Session ID* ne ističe nakon određenog vremena ili nakon odjave.** Žrtva koristi javno računalo (*Internet caffee*, knjižnica…) i posjećuje stranice na koje se prijavljuje sa svojim korisničkim podacima. Nakon što žrtva ode, aplikacija neko vrijeme, iako korisnik nije bio aktivan ili je zatvorio preglednik, i dalje čuva sesiju koju može zloupotrijebiti osoba koja na računalo dođe nakon nje.

- **Lozinke, *Session ID* i ostale vjerodajnice šalju se nešifriranim komunikacijskim kanalima.** Napadač presreće i prisluškuje mrežu kojom komuniciraju API-ji i gleda pakete koje razmjenjuju. U paketima pronalazi lozinku, *Session ID,* autentifikacijski token, kolačiće i ostale podatke u *plain text* obliku.

Promotrimo sigurnost autentifikacije na našem primjeru ranjivog API-ja. Ako pogledamo dokumentaciju *Vulnerable* API-ja, primijetit ćemo da većina interakcija s API-jem zahtijeva i autentifikacijski token koji se prosljeđuje prilagođenim HTTP zaglavljem *X-Auth-Token*. Iako većina REST API-ja koristi standardna autorizacijska zaglavlja (engl. *Authorization Headers*) za tokene, ni ovakva prilagođena zaglavlja nisu rijetkost. *Postman* podržava i standardna i prilagođena zaglavlja.

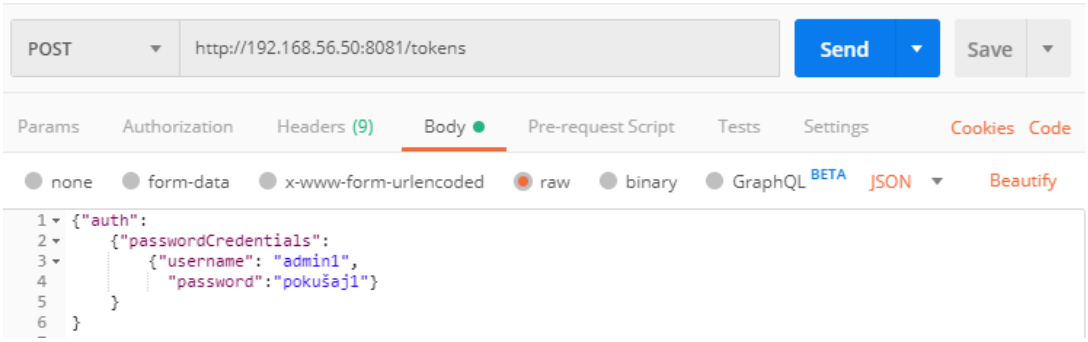
Na primjeru *Vulnerable API*-ja možemo nabrojati nekoliko ključnih autentifikacijskih ranjivosti koje bi napadač mogao iskoristiti za preuzimanje računa nekog drugog korisnika:

- Nema ograničenja vezanih uz duljinu i sadržaj lozinke. Lozinke su redom *pass1, pass2, pass3*... i napadač ih može lako otkriti *brute-force* ili *dictionary attack* napadima ili običnim pogađanjem.
- Autentifikacijski token nikad ne ističe.
- Lozinke se pohranjuju u tekstualnom obliku u bazi podataka (nisu *hashirane*).
- Obavijest o pogreški otkriva previše informacija koje napadač može iskoristiti za sastavljanje napada.

I dok su prve tri ranjivosti jasne same po sebi, malo pobliže ćemo se dotaknuti toga što točno znači otkrivanje previše informacija. Otkrivanje previše informacija je nenamjerno otkrivanje podataka koji previše govore o aplikaciji, njenom tijeku izvršavanja ili njenim korisnicima, a napadač ih može iskoristiti kako bi lakše preuzeo tuđi račun.

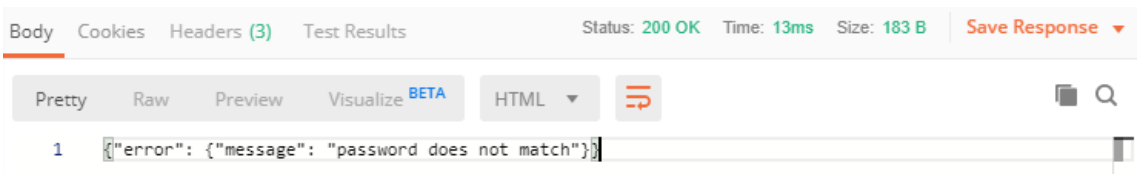
Zamislimo sljedeći scenarij napada na *Vulnerable API*:

1) Napadač pokušava preuzeti tuđi korisnički račun. Pokušava pogoditi kombinaciju korisničkog imena i lozinke kao što je prikazano na slici 10.



Slika 10 ***Napadač pogađa kombinaciju korisničkog imena i lozinke***

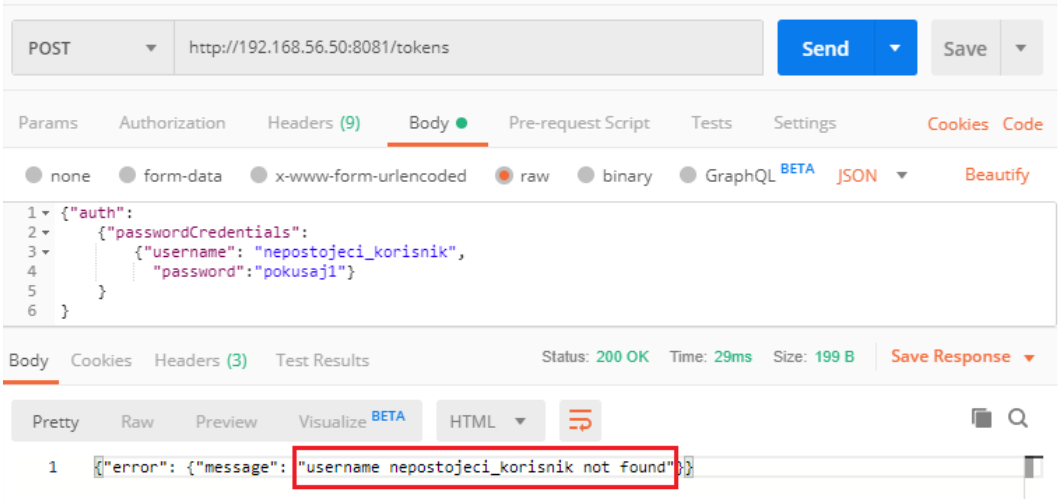
2) Napadač nije uspio pogoditi vjerodajnice i nije se uspio prijaviti, ali dobiva odgovor u kojem mu je izložena informacija da korisničko ime koje je unio postoji, ali je lozinka pogrešna.



Slika 11 ***Pogreška obavještava napadača da korisničko ime postoji, ali lozinka se ne podudara***

Napadač sad zna jedno korisničko ime koje zasigurno postoji u API-ju i na njega može primijeniti npr. *brute-force* napad za pogađanje lozinke. Kad bi napadač

pogriješio i korisničko ime i lozinku, dobio bi sasvim drukčije upozorenje koje mu indicira da takvo korisničko ime ne postoji.



Slika 12 ***Pogreška obavještava napadača da korisničko ne ime postoji***

Takva upozorenja korisna su legitimnim korisnicima koji nisu sigurni pokušavaju li se prijaviti s ispravnom adresom e-pošte (ako ih imaju više), ali nažalost korisna je i napadačima i olakšava im napad na autentifikaciju. Iz tog razloga ključno je ne odavati takve informacije kako ih napadač ne bi zloupotrijebio. Ako korisnik unese ispravno korisničko ime, a pogrešnu lozinku, nikako mu ne treba dati do znanja da korisničko ime postoji, već mu ispisati generičku grešku poput „Korisničko ime ili lozinka nisu ispravni".

Također, ponekad napadač može zaključiti postoji li ili ne postoji korisničko ime koje je pokušao pogoditi na način da prati razliku u vremenu koje je potrebno da dobije odgovor od poslužitelja, pa treba i na to paziti i nastojati ujednačiti vremena odgovora u oba slučaja.

2.3 A3 – Izloženost osjetljivih podataka (engl. *Sensitive Data Exposure*)

Izloženost osjetljivih podataka (npr. OIB, lozinke, brojevi kreditnih kartica, autentifikacijski tokeni…) rezultat je lošeg ili nepostojećeg šifriranja podataka koje aplikacija pohranjuje ili prenosi mrežom. Posljedice izloženosti osjetljivih podataka mogu biti preuzimanje korisničkog računa legitimnog korisnika, krađa novca s bankovnog računa, prikupljanje adresa e-pošte za *spam*, *phishing* i sl.

Kad je riječ o nesigurnoj pohrani podataka, velik rizik predstavlja nešifriranje osjetljivih podataka ili šifriranje (enkripcija) slabim kriptografskim algoritmima. Kad je pak riječ o nesigurnom prijenosu mrežom, velik rizik je nekorištenje TLS protokola (engl. *Transport Layer Security*) ili korištenje TLS protokola sa slabim algoritmima za šifriranje.

O sigurnoj pohrani podataka dotakli smo se u prethodnom poglavlju kad je bila riječ o nesigurnoj pohrani lozinki. Kao i za lozinke, i za ostale osjetljive podatke vrijedi pravilo da se trebaju šifrirati, i to snažnim kriptografskim algoritmom kojeg napadač neće moći dešifrirati u nekom realnom vremenu.

Vezano za siguran prijenos osjetljivih podataka mrežom, REST API-ji trebali bi uvijek i obavezno koristiti TLS protokol za šifriranje podataka koji se šalju mrežom. API-je i *web* aplikacije koje koriste TLS enkripciju moguće je prepoznati prema oznaci HTTPS protokola u URL-u.

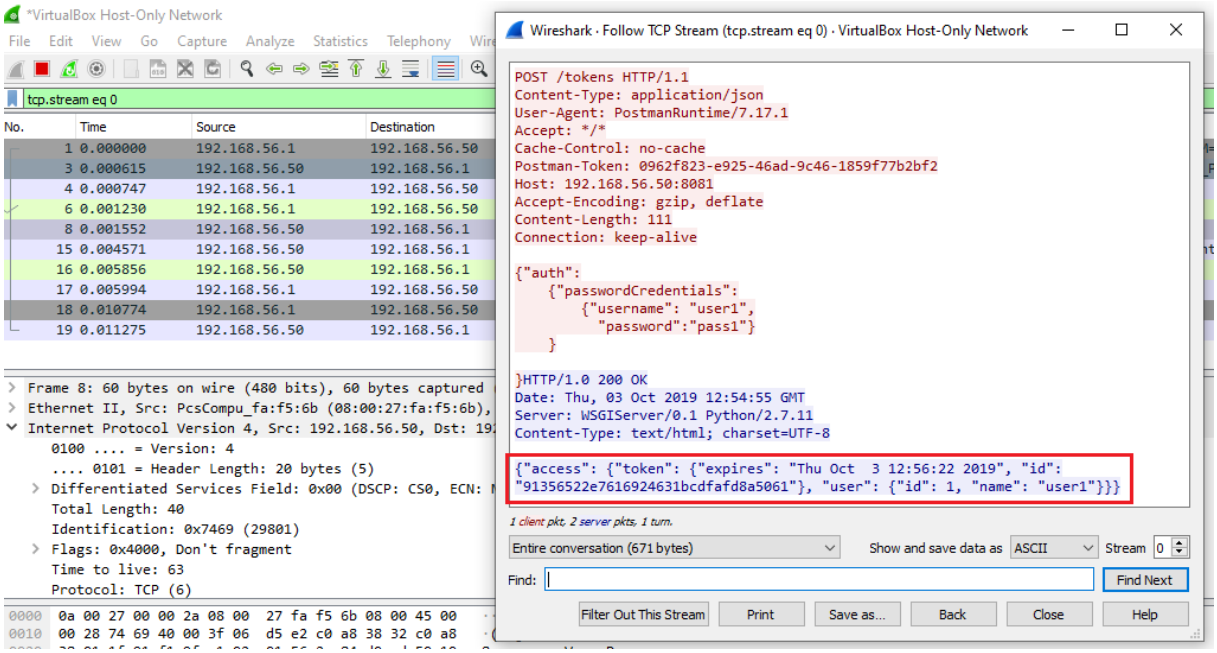
Kakav točno rizik donosi nekorištenje TLS protokola promotrit ćemo na primjeru *Vulnerable API*-ja koji ne koristi TLS protokol, zbog čega napadač prisluškivanjem mrežnog prometa može saznati osjetljive podatke koji se šalju mrežom između klijenta i API-ja. U ovom slučaju je taj osjetljivi podatak autentifikacijski token, korisničko ime i lozinka, ali to može biti bilo koji drugi osjetljivi podatak.

Jedan konkretan scenarij napada u kojem bi napadač mogao zloupotrijebiti nekorištenje TLS-a i preuzeti korisnički račun legitimnog korisnika bio bi:

1) Napadač dolazi u kafić i spaja se na WiFi mrežu na koju su već spojeni ostali gosti.

2) Napadač postavlja svoju mrežnu karticu u način rada za prisluškivanje čime vidi sav promet na istoj WiFi mreži.

3) Napadač pomoću *WireSharka* ili sličnog alata za analizu mrežnog prometa prati promet i pronađe razmjenu paketa između gosta koji komunicira s *Vulnerable API*-jem i *Vulnerable API*-ja. API ne koristi TLS pa napadač vidi prenesene podatke u *plain textu*. Dolazi do osjetljivog podatka o autentifikacijskom tokenu, korisničkom imenu i lozinki.



Slika 13 ***Napadač je prikupio podatke o korisnikovom tokenu***

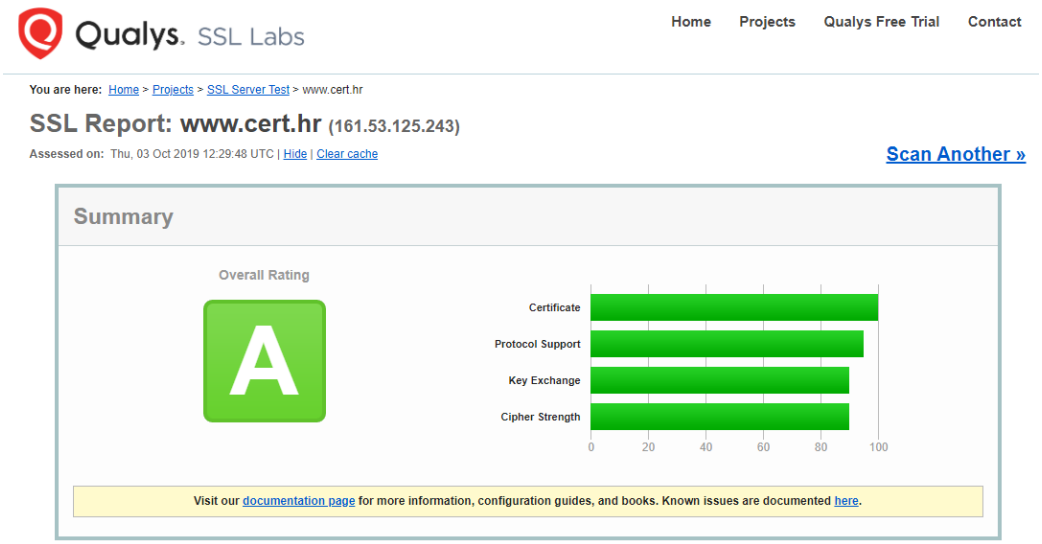
4) Napadač sada može poslati zahtjev API-ju sa žrtvinim podacima i dobiti povjerljive informacije ili provesti neku radnju u ime žrtve.

Kako bi se rizik od izloženosti osjetljivih korisničkih podataka umanjio, potrebno se držati nekoliko osnovnih smjernica. Osjetljivi podaci pohranjuju se samo ako je to uistinu potrebno i treba ih obrisati čim više nisu potrebni. Također, za osjetljive podatke treba

izbjeći (ako je to moguće) implementaciju priručne memorije (engl. *cache*). Kao što je već objašnjeno za lozinke, i osjetljive podatke treba kriptirati snažnim kriptografskim algoritmom, koristiti *salt* i promjenjiv broj iteracija.

Osim sigurne pohrane, potrebno je poduzeti i korake vezane za siguran prijenos osjetljivih podataka mrežom i spriječiti da netko tko prisluškuje promet vidi podatke u *plain textu*. Praktično rješenje je korištenje TLS-a koji osigurava siguran prijenos podataka mrežom i sprječava *Man-In-The-Middle* napad. TLS protokol podržava i snažne i slabije algoritme (npr. API možda komunicira s malim IoT uređajem koji ne može koristiti snažniji kriptografski algoritam jer ima malu bateriju i ograničene resurse). Ako za to ne postoji razlog, TLS bi trebalo konfigurirati na način da ne podržava slabije kriptografske algoritme koji se mogu probiti jer napadač može prisiliti API da komunikaciju šifrira slabim algoritmom i onda je dešifrirati.

Jedna korisna stranica na kojoj se može analizirati razina zaštite API-ja ili *web* stranice vezana uz mrežni prijenos podataka je *SSL Labs*. *SSL Labs* će automatski analizirati sigurnost konfiguracije certifikata, protokola, razmjene ključeva i snagu kriptografskog algoritma. Na slici 14 pokazan je rezultat analize sigurnosne konfiguracije stranice Nacionalnog CERT-a.



Slika 14 ***Analiza razine zaštite stranice Nacionalnog CERT-a***

2.4 A4 – XML vanjski entiteti (engl. *XML External Entities*, XXE)

Ova ranjivost javlja se kod API-ja koji koriste XML format za komunikaciju ili dopuštaju učitavanje XML dokumenata koji se zatim automatski obrađuju. Iskorištavanje ove ranjivosti je u stvari napad na modul aplikacije koja čita i obrađuje XML, tj. *XML parser*.

Za vrijeme komunikacije, klijent i poslužitelj se mogu dogovarati o definiciji XML poruke (engl. *External DTD)*. Za takvo dogovaranje podrazumijeva se da su obje strane pouzdane i tad nema rizika od napada.

No, problem nastaje ako i nepouzdan klijent može postaviti svoju definiciju XML poruke jer tad napadač može pripremiti zahtjev kojim će pristupiti datotečnom sustavu poslužitelja, izvršiti neki svoj proizvoljni kôd ili učiniti napadnuti sustav nedostupnim.

Jedan primjer napada na ovu ranjivost bi bio (5):

1) Ranjivi API provjerava koje je stanje na skladištu proizvoda kojeg korisnik traži. Kad bi sve bilo legitimno, XML koji bi došao do poslužitelja kad bi korisnik htio saznati stanje proizvoda s identifikacijskim brojem 381 izgledao bi:

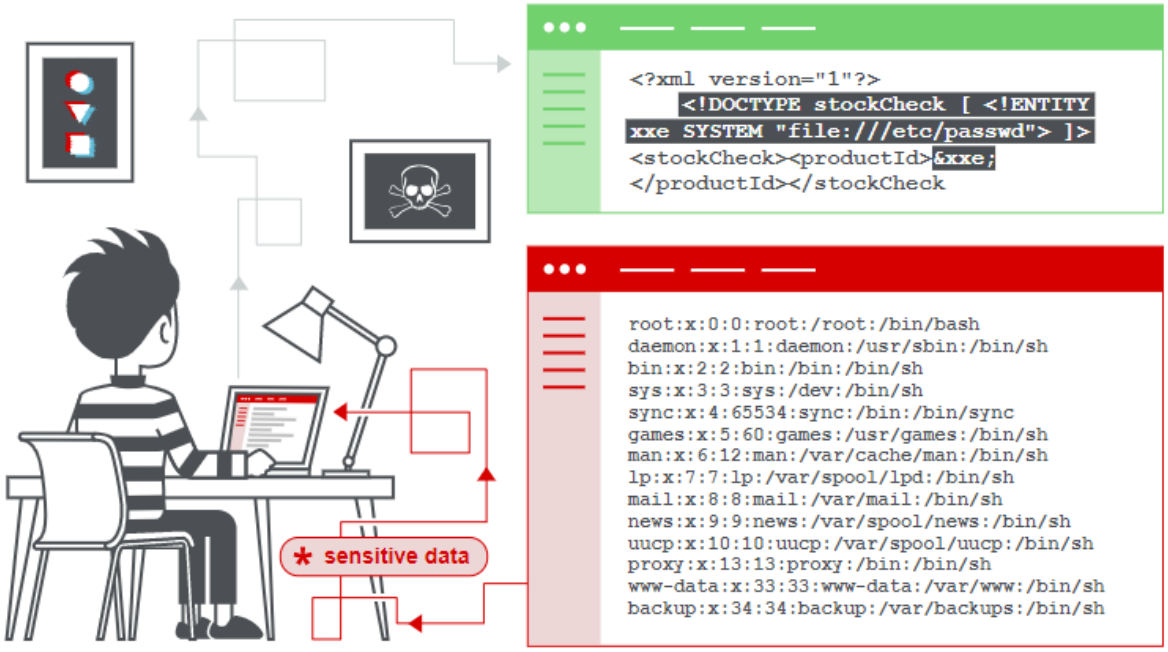
```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

2) Budući da API ne implementira nikakve sigurnosne mehanizme protiv XXE napada, napadač može sastaviti sljedeći XML kako bi došao do datoteke */etc/passwd* s podacima o korisnicima koji se nalaze na poslužitelju:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

3) Ovako sastavljen XML definira vanjski entitet naziva *xxe* čija je vrijednost sadržaj datoteke */etc/passwd*. Taj entitet poziva se umjesto broja artikala na lageru, i umjesto da korisniku vrati odgovor s informacijom o tome koliko je artikala preostalo, poslužitelj će mu vratiti sadržaj datoteke koja pohranjuje informacije o korisnicima sustava na kojem je pohranjen API.

```
Invalid product ID: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```



Slika 15 ***Iskorištavanje XXE ranjivosti*** **(5)**

Za sprječavanje iskorištavanja ovakve ranjivosti preporučljivo je (kad god se to može) koristiti manje složene tipove podataka poput JSON-a te (ako namjena aplikacije to dozvoljava) onemogućiti *XML External Entity* i *DTD Processing* u svim *XML parserima* koji se koriste pri razvoju API-ja ili aplikacije.

Promotrimo kôd *Vulnerable API*-ja koji koristi *XML parser* prikazan na slici 16. *Vulnerable API* napisan je u *Pythonu* i modul API-ja koji se koristi za čitanje i obrađivanje XML-a je *parser etree*. Taj modul ne dopušta vanjske XML entitete i *Vulnerable API* se ne može napasti na prethodno opisan način. Ako bi došlo do pokušaja XXE napada, API ne bi prihvatio obradu takvog XML-a i javio bi pogrešku.

```
if content_type == 'application/xml':
    try:
        # LXML is vulnerable to XXE, etree is vulnerable to Billion Laughs
        # So just have etree try to parse it just to watch it die
        ET.parse(request.body)
    except Exception:
        # But etree will throw an exception for XXE, so ignore that
        pass
    # force unsafe external entity parsing
    parser = etree.XMLParser(load_dtd=True, resolve_entities=True)
    data = etree.parse(request.body, parser)
    username = data.find('passwordCredentials').find('username').text
    password = data.find('passwordCredentials').find('password').text
```

Slika 16 ***Korištenje parsera etree u Vulnerabe API-ju koji nije ranjiv na XXE napad***

Da je programer kojim slučajem koristio *parser* LXML bez da je dodatno isključio opciju za XXE, *Vulnerable API* bi bio ranjiv.

2.5 A5 – Neučinkovita kontrola pristupa (engl. *Broken Access Control*)

Razni korisnici API-ja ili aplikacije imaju različite uloge i razine ovlasti (npr. administrator, privilegirani korisnik, običan korisnik…). Kontrola pristupa ograničava svakog od korisnika na to da smije obavljati samo one aktivnosti unutar API-ja ili aplikacije i pristupiti samo onim resursima za koje ima prava (tj. razina ovlasti mu to dopušta). Pogreške u implementaciji kontrole pristupa mogu dovesti do toga da npr. običan korisnik može obavljati administratorske radnje poput izmjene ili brisanja podataka.

Neke od ranjivosti kontrola pristupa su:

- Zaobilaženje kontrole provjere pristupa izmjenjivanjem URL-a, unutarnjeg stanja aplikacije ili HTML kôda stranice ili korištenjem nekog prilagođenog alata za napad na API.

  Npr. korisnik vidi da za pregled stanja svog računa mora poslati zahtjev oblika:

  ```
  GET /showBalance/user123
  ```

  Korisnik zatim pokušava unijeti

  ```
  GET /showBalance/user124
  ```

  i ako dobije odgovor, API ima problem s neučinkovitom kontrolom pristupa jer nije provjerio ima li korisnik ovlasti da pregledava te podatke.

- Manipulacija metapodacima, npr. ponovno slanje ili namještanje JSON *web* tokena (JWT), kolačića ili skrivenih polja

- Pristupanje nekoj nedokumentiranoj stranici namijenjenoj administratoru

- Nepravilno konfigurirane ovlasti za različite korisnike ili grupe

Za kontrolu pristupa se podrazumijeva da se implementira na strani pouzdanog poslužitelja gdje napadač ne može steći pristup i modificirati proces kontrole pristupa ili metapodatke.

Pogledajmo primjer ove ranjivosti na aplikaciji s ranjivim API-jem *Vulnerable API*. Traženjem nedokumentiranih URL-ova (API endpointova) namijenjenih administratoru, napadač može otkriti da će slanjem zahtjeva `GET /tokens` dobiti odgovor od *Vulnerable API*-ja sa svim podacima o svim korisnicima, pri čemu za slanje takvog zahtjeva nije potrebna nikakva autentifikacija ni autorizacija – taj zahtjev može poslati i neregistrirani korisnik bez korisničkog imena i lozinke.

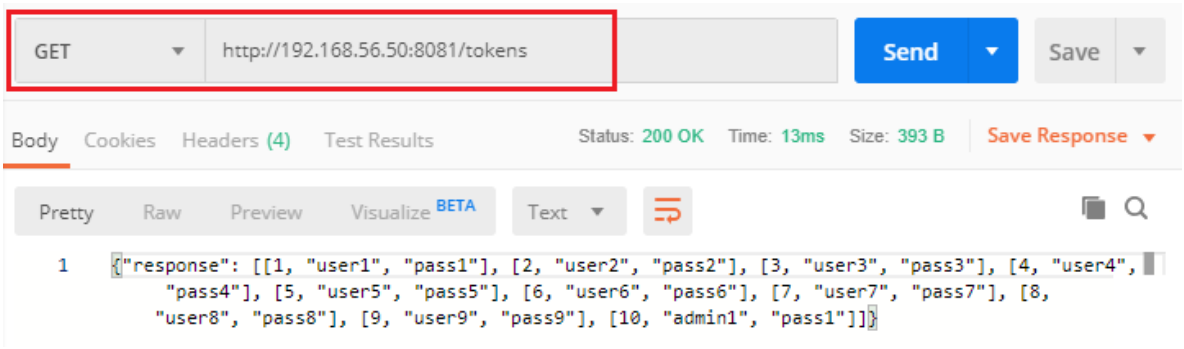
Jedini sigurnosni mehanizam koji sprječava napadača da pristupi tom resursu je tajnost URL-a na koji treba poslati GET zahtjev, tj. zahtjev nije naveden u dokumentaciji pa se pretpostavlja da napadač neće znati kako točno glasi taj URL na koji treba poslati zahtjev. No, napadač može koristiti tehnike pogađanja i može pogoditi takav URL. Oslanjanje isključivo na tajnost URL-a upravo zato je vrlo loša sigurnosna mjera - napadači mogu pretraživati razne kombinacije URL-ova upravo kako bi otkrili one skrivene na kojima mogu doći do informacija bez ikakve autentifikacije. Jedan primjer napada na ovakvu ranjivost na našem *Vulnerable API*-ju bio bi:

1. Programer *Vulnerable API*-ja u uputama za korištenje je dokumentirao samo URL-ove za koje je implementirao autentifikaciju. Osim njih, postoje i neki tajni URL-ovi za koje nije potrebna autentifikacija, ali njih napadač ionako ne zna, tj. ne zna kako mora oblikovati zahtjev da dođe do tih informacija. Kod HTTP REST API-ja je vrlo bitno poslati zahtjev s točno određenom metodom, zaglavljem i tijelom zahtjeva (pratiti upute iz dokumenacije) kako bi se dobio odgovor, i napadaču nije trivijalno otkriti takve tajne URL-ove i kakav zahtjev koji se šalje mora biti da bi dobio željeni odgovor.

2. No, napadač može uzastopno pokušavati razne kombinacije metoda slanja zahtjeva (GET, POST, PUT…) na dokumentirane i nedokumentirane URL-ove (pogađa ih, npr. *fuzzingom* parametara u *Postmanu*).

3. Između ostalih, jedan pokušaj je i GET zahtjev na URL `/tokens`. Legitimni zahtjev `POST /tokens` je dokumentiran i korisnik ga može koristiti, ali `GET /tokens` nije nigdje dokumentiran pa je programer aplikacije *Vulnerable API* pretpostavio da korisnik neće ni pokušavati poslati takav zahtjev.

4. Napadač je bez ikakve autentifikacije došao do osjetljivih podataka (popisa svih korisničkih imena i pripadajućih lozinki) koje bi u stvari smio vidjeti samo administrator.

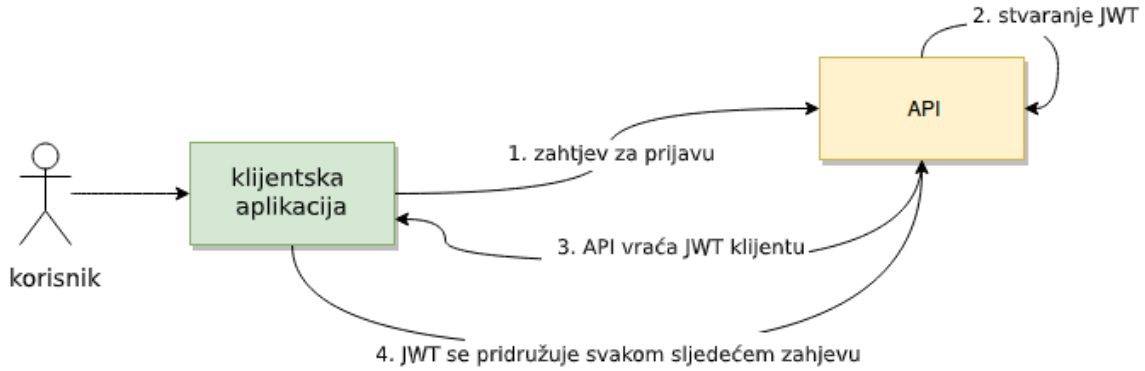


Slika 17 ***Napadač je uspješno došao do popisa svih korisničkih imena i lozinka u Vulnerable API-ju***

2.5.1 *JSON Web Token* (JWT)

Jedan od karakterističnih načina za autentifikaciju i autorizaciju u *web* API-jima je *JSON Web Token* (skraćeno JWT). JWT je standardni (RC 7519) način za sigurno potvrđivanje korisnikovog identiteta u HTTP komunikaciji. Prisjetimo se da je REST *stateless*, što znači da ne postoji mehanizam za pamćenje tko je tko za vrijeme interakcije, već se korisnik mora na neki način identificirati prilikom slanja svakog zahtjeva (kao što na primjeru *Vulnerable API*-ja to radi *X-Auth-Token* zaglavljem). Kad dva sustava razmjenjuju podatke, umjesto njihovih privatnih vjerodajnica sa svakim zahtjevom može se slati JWT.

Cijeli postupak prikazan je na slici 18:



Slika 18 ***Proces dodjele i korištenja JWT*** **(6)**

1) Korisnik, tj. klijentska aplikacija šalje zahtjev za prijavu (engl. *sign in*) API -ju skupa sa svojim korisničkim imenom i lozinkom (ili nekim drugim podacima kojima se korisnik autentificira).

2) API provjerava vjerodajnice, i ako su ispravne stvara JWT i potpisuje ga koristeći svoj tajni ključ.

3) API vraća JWT klijentskoj aplikaciji.

4) Klijentska aplikacija prima JWT i nastavlja ga koristiti za sljedeće zahtjeve. Klijentska aplikacija više ne mora slati korisnikove vjerodajnice svaki put.

JWT token se sastoji od šifriranog znakovnog niza sastavljenog od različitih dijelova:

1) Zaglavlje (engl. *Header*) – podaci o vrsti tokena i algoritmu koji se koristi za njegovo generiranje. Česti algoritmi koji se ovdje koriste su HS256 (HMAC SHA256) i RS256 (RSA SHA 256).

2) Sadržaj (engl. *Payload*) – sadrži podatke koji se odnose na korisnika koji ga upućuje.

3) Potpis (engl. *Signature*) – šifrirani znakovni niz kojim poslužitelj provjerava autentičnost sadržaja.

Sigurnosni problem vezan uz JWT tokene je u tome što neke inačice biblioteka koje se koriste za generiranje i rukovanje JWT tokenima imaju ranjivost zbog koje napadač može natjerati API da prihvati JWT token potpisan javnim ključem poslužitelja (koji nije tajan) i HS256 algoritmom (simetrično potpisivanje) umjesto RS256 algoritmom (asimetrično potpisivanje).

Vulnerable API ne koristi JWT pa će se za demonstraciju napada koristiti aplikacija koju je razvio Sjoerd Langkemper upravo za demonstraciju ove ranjivosti. Kad je API ranjiv na podmetanje JWT-a, moguć je ovakav primjer scenarija napada (7):

1. API dodjeljuje korisniku JWT token oblika *header.payload.signature* u base64 notaciji:

   ```
   eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1vLnNqb2VyZG
   xhbmdrZW1wZXIubmxcLyIsImlhdCI6MTU0NzcyOTY2MiwiZXhwIjoxNTQ3NzI5NzgyLCJkYXRhI
   jp7ImhlbGxvIjoid29ybGQifX0.gTlIh_sPPTh24OApA_w0ZZaiIrMsnl39-B8iFQ-
   Y9UIxybyFAO3m4rUdR8HUqJayk067SWMrMQ6kOnptcnrJl3w0SmRnQsweeVY4F0kudb_vrGmarA
   XHLrC6jFRfhOUebL0_uK4RUcajdrF9EQv1cc8DV2LplAuLdAkMU-
   TdICgAwi3JSrkafrqpFblWJiCiaacXMaz38npNqnN0l3-
   GqNLqJH4RLfNCWWPAx0w7bMdjv52CbhZUz3yIeUiw9nG2n80nicySLsT1TuA4-B04ngRY0-
   QLorKdu2MJ1qZz_3yV6at2IIbbtXpBmhtbCxUhVZHoJS2K1qkjeWpjT3h-bg
   ```

   Kad se taj JWT dekodira, dolazimo do zaglavlja i sadržaja:

   ```
   {"typ":"JWT","alg":"RS256"}.{"iss":"http:\\/\\/demo.sjoerdlangkemper.nl\\/",
   "iat":1547729662,"exp":1547729782,"data":{"hello":"world"}}
   ```

   U zaglavlju vidimo informaciju da je za šifriranje korišten asimetrični RSA algoritam.

2. Sastavljamo novo zaglavlje i sadržaj za JWT token, s tim da ovaj put navodimo da je za šifriranje korišten simetrični HS256 algoritam:

   ```
   {"typ":"JWT","alg":"HS256"}.{"iss":"http:\\/\\/demo.sjoerdlangkemper.nl\\/"
   ,"iat":1547729662,"exp":1547799999,"data":{"NCC":"test"}}
   ```

   Kodiramo zaglavlje i sadržaj i dobijemo sljedeći base64 zapis:

   ```
   eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1vLnNqb2VyZGx
   hbmdrZW1wZXIubmxcLyIsImlhdCI6MTU0NzcyOTY2MiwiZXhwIjoxNTQ3Nzk5OTk5LCJkYXRhIjp
   7Ik5DQyI6InRlc3QifX0
   ```

3. Još samo nedostaje potpis za koji nam je potrebno saznati javni ključ (engl. *public key*) kojeg API koristi. To se može saznati iz više izvora, od kojih je jedan i TLS certifikat. Naredbom:

   ```
   openssl x509 -in cert.pem -pubkey -noout > key.pem
   ```

   U datoteci *key.pem* pohranit će se javni ključ poslužitelja kojim se potpisuju zaglavlje i sadržaj i generira se JWT token:

   ```
   -----BEGIN PUBLIC KEY-----
   MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqi8TnuQBGXOGx/Lfn4JF
   NYOH2V1qemfs83stWc1ZBQFCQAZmUr/sgbPypYzy229pFl6bGeqpiRHrSufHug7c
   1LCyalyUEP+OzeqbEhSSuUss/XyfzybIusbqIDEQJ+Yex3CdgwC/hAF3xptV/2t+
   H6y0Gdh1weVKRM8+QaeWUxMGOgzJYAlUcRAP5dRkEOUtSKHBFOFhEwNBXrfLd76f
   ZXPNgyN0TzNLQjPQOy/tJ/VFq8CQGE4/K5ElRSDlj4kswxonWXYAUVxnqRN1LGHw
   2G5QRE2D13sKHCC8ZrZXJzj67Hrq5h2SADKzVzhA8AW3WZlPLrlFT3t1+iZ6m+aF
   KwIDAQAB
   -----END PUBLIC KEY----
   ```

4. Svaki daljnji zahtjev API-ju šalje se skupa s tim JWT-om kojeg poslužitelj (koji je ranjiv) prihvaća.

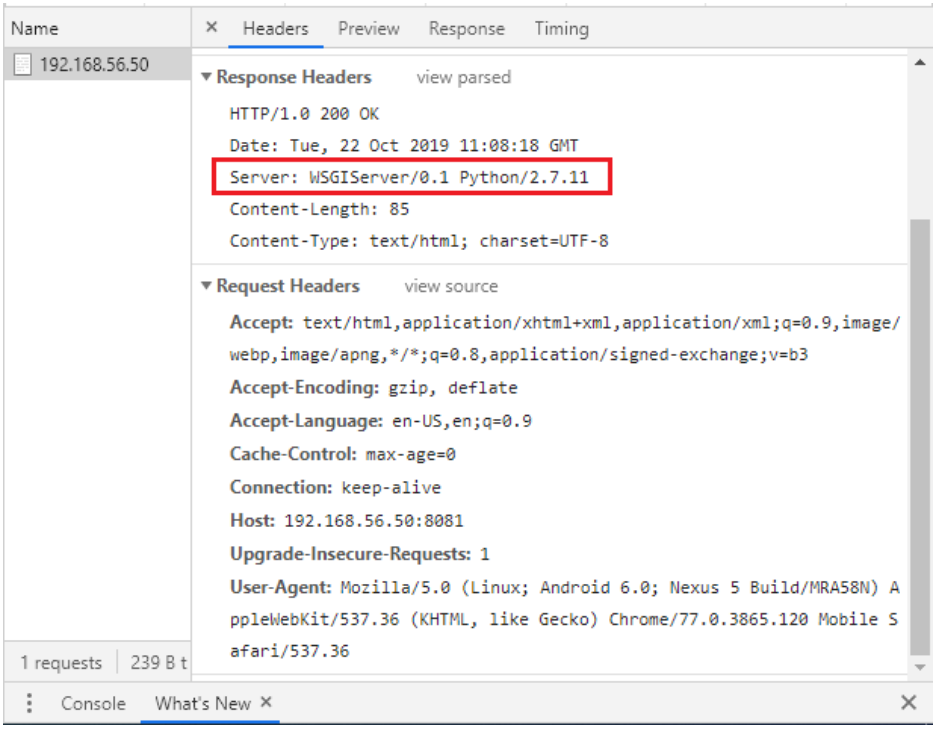
2.6 A6 – Loša sigurnosna konfiguracija (engl. *Security Misconfiguration*)

Loša sigurnosna konfiguracija je, prema OWASP-u, najčešći problem u aplikacijama i API-jima (2). Operacijski sustavi, razvojni okviri, biblioteke i aplikacije moraju biti sigurno konfigurirani i redovito ažurirani kako napadači ne bi uspjeli zloupotrijebiti neku njihovu ranjivost. Pogreške u konfiguraciji mogu se dogoditi u bilo kojem sloju API-ja: na mreži, platformi, *web* poslužitelju, aplikacijskom poslužitelju, bazi podataka, razvojnim okvirima, kôdu aplikacije, virtualnim strojevima, kontejnerima itd. Na većinu ovakvih problema mogu nas uputiti alati za skeniranje ranjivosti, a neki od čestih problema koji se javljaju vezano uz lošu sigurnosnu konfiguraciju su:

- Omogućene su ili instalirane nepotrebne funkcionalnosti (npr. nepotrebni *portovi*, usluge, stranice, korisnički računi ili privilegije).
- Omogućeni su podrazumijevani (engl. *default*) korisnički računi i njihove su lozinke i dalje nepromijenjene (npr. admin/admin).
- Rukovanje pogreškama otkriva previše informacija korisnicima.
- Najnovija sigurnosna ažuriranja softvera su onemogućena ili nisu konfigurirana.
- Poslužitelj ne šalje sigurnosna zaglavlja ili direktive klijentima ili one nisu postavljene na sigurne vrijednosti.

Promotrimo primjer scenarija napada iz ove kategorije ranjivosti na Vulnerable API-ju:

1. Napadač šalje zahtjev poslužitelju i dobije odgovor u kojem je, između ostalog, navedena i inačica poslužiteljskog softvera.



Slika 19 ***U HTTP odgovoru prikazana je i poslužitelj na kojem je pohranjen Vulnerable API***

2. Izložena informacija o inačici poslužitelja pomaže napadaču u napadu jer sad napadač (ili automatizirani alat koji traži ranjive poslužitelje) može zaključiti je li riječ o inačici softvera koja nije ažurirana neko vrijeme (npr. ako je inačica poslužitelja koji poslužuje API npr. 1.0, a jednostavnom pretragom na *webu* napadač sazna da je trenutna verzija 2.5, poslužitelj je definitivno neažuriran dugo vremena i ranjiv). Poslužiteljski softver koji nije ažuriran sigurnosnim zakrpama neko vrijeme vjerojatno je ranjiv. Napadač pretragom na *webu* može pronaći ranjivosti vezane za tu inačicu poslužitelja, kao i možda neki gotov *exploit*.

Još neke pogreške u konfiguraciji koje dovode do izloženosti informacija su npr. stranica 404 *Page Not Found* koja je karakteristična za određen poslužitelj. Također, različiti programski jezici različito odgovaraju na neke karakteristične upite preko kojih (ako ih programer drukčije ne konfigurira) napadač može saznati u kojoj je npr. razvojnoj okolini i programskom jeziku razvijen API. Još jedan sigurnosni problem je i konfiguracija koja dopušta izlistavanje direktorija. Dokument Nacionalnog CERT-a Apache HTTP poslužitelj detaljnije je opisao smjernice za sigurnu konfiguraciju poslužiteljskog softvera na primjeru Apache HTTP poslužitelja, a općenite informacije o sigurnoj konfiguraciji za svaku vrstu poslužitelja mogu se pronaći na *webu*, pri čemu je korisna pretraga ključne riječi „*security hardening*“ koja označava ojačavanje sigurnosti.

Jednom kad se implementiraju sigurnosne mjere, korisno je implementirati i automatizirani proces provjere konfiguracije i postavki i u razvojnoj, i u testnoj, i u produkcijskoj okolini.

2.7 A7 – XSS

Cross-site scripting, tj. XSS napad, odnosi se na umetanje vlastitog zlonamjernog (napadačevog) kôda u tuđu legitimnu (ranjivu) stranicu koji će se kasnije izvršavati u

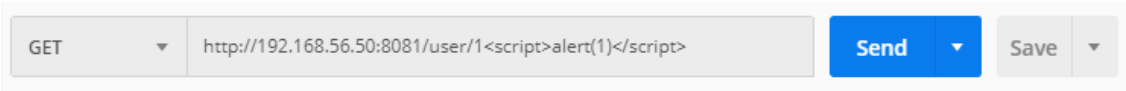
pregledniku posjetitelja napadnute stranice. Taj zlonamjerni kôd ima neograničene mogućnosti unutar žrtvinog preglednika i može npr. pristupiti kolačićima i ostalim osjetljivim korisničkim podacima te će moći provoditi radnje na *web* stranici u ime žrtve (jer *web* preglednik ne može znati da taj kôd i te radnje ne dolaze od legitimnog korisnika).

U API-ju, u odnosu na *web* aplikacije, XSS ranjivost ne mora uvijek biti rizična jer se zlonamjerni korisnički unos može samo pohraniti u nekoj bazi podataka i nikad ne izvršiti kao kôd unutar *web* preglednika. Podrazumijevano se XSS kôd umetnut u REST API neće izvršiti u pregledniku jer je zaglavlje *Content-Type* uobičajeno postavljeno na JSON/XML (u tom formatu se i očekuje odgovor od API-ja) i zato preglednik neće pokušati prikazati odgovor kao HTML (kao što to radi u slučaju *web* stranica).

Kako bi se dokazala XSS ranjivost u API-jima, dovoljno je pokazati da API neće provjeriti i sanitizirati korisnikov unos. Iako u slanju zahtjeva *Postmanom* ili nekim sličnim alatom za slanje HTTP zahtjeva neće doći do problema jer je odgovor prikazan u JSON-u, zamislimo da je promatrani API povezan s nekom *web* aplikacijom koja u obliku HTML tablice prikazuje dobivene odgovore iz API-ja.

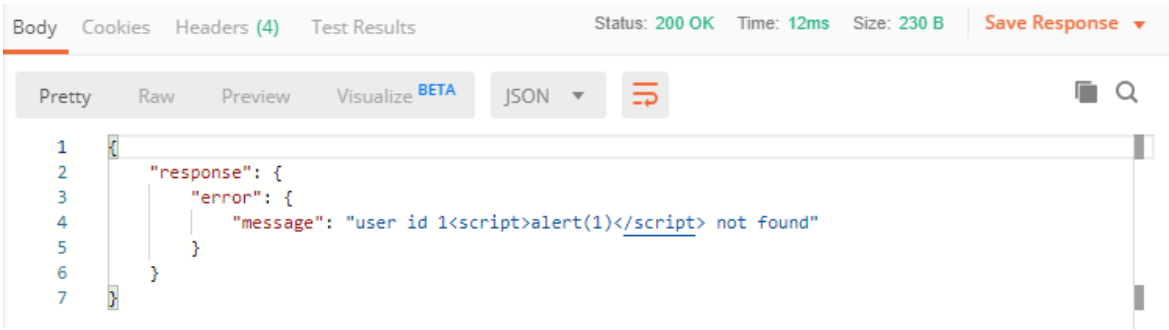
Tad bi, konkretno na primjeru *Vulnerable API*-ja, bio moguć sljedeći scenarij napada:

1) Napadač sastavlja zahtjev s XSS napadom kao što je prikazano na slici 20:



Slika 20 Napadač sastavlja XSS napad

2) Napadač šalje zahtjev i prima odgovor od *Vulnerable API*-ja:



Slika 21 *Napadačev unos nije validiran i filtriran, već pohranjen kakav je*

Vidljivo je da unutar polja „*message*“ nije sanitiziran dio korisnikovog unosa u kojem se nalazi skripta.

3) *Web* stranica bilježi sve pogreške do kojih je došlo tokom pozivanja API-ja i prikazuje ih administratoru. Administrator koji pregledava pogreške postaje žrtva XSS napada jer kad aplikacija bude prikazivala ovaj zapis, izvršit će i JavaScript kôd.

Sprječavanje XSS napada vrlo je složena tema. Postoji više vrsta XSS napada (*stored, reflected, DOM-based*) od kojih svaki ima svoje karakterističnost i sigurnosne mjere zaštite. Ukratko, korisničkom unosu nikad ne treba vjerovati i svugdje gdje se pojavljuje

treba ga ograničiti na dopuštene znakove, validirati, filtrirati i sanitizirati. No, to nije toliko jednostavno kao što se čini, ali organizacija OWASP napisala je dva velika dokumenta na ovu temu koje je svakako korisno pročitati i pridržavati se navedenih smjernica:

- Cross Site Scripting Prevention Cheat Sheet
- DOM based XSS Prevention Cheat Sheet

2.8 A8 – Nesigurna deserijalizacija (engl. *Insecure Deserialization*)

Serijalizacija je pretvaranje objekata u format koji se može pohraniti na disku (npr. datoteka) ili slati putem mreže. Format u koji se objekt može serijalizirati je binarni ili strukturirani tekst pri čemu su najčešći formati koji se koriste u *web* aplikacijama i API-jima JSON i XML.

Deserijalizacija je pretvaranje podataka iz datoteke ili toka podataka u objekte. Većina programskih jezika nudi funkcionalnosti za serijalizaciju i deserijalizaciju podataka, pogotovo u JSON i XML format. Deserijalizacija je uobičajena praksa u razvoju softvera. Nesigurna deserijalizacija je ranjivost koja deserijalizira i nepouzdan korisnički unos koji može dovesti do napada na poslužiteljsko računalo.

I ovdje treba primijeniti iste mjere zaštite/provjere kao i kod bilo kojeg korisničkog unosa.

2.9 A9 – Korištenje komponenti s javno poznatim ranjivostima (engl. *Using Components with Known Vulnerabilities*)

Korištenje komponenata s javno poznatim ranjivostima znači riskirati da je razvijen (ili će se uskoro razviti) *exploit* kojim se ranjivi softver može napasti.

U razvoju modernih aplikacija koriste se raznorazne komponente i biblioteke koje programerima nude gotove funkcionalnosti. Nije rijetkost da se neka komponenta instalira, zatim nikad ne koristi, ali se ne obriše ili ne ažurira. Takve komponente, kao i općenito neažuriran poslužiteljski softver, predstavljaju sigurnosni rizik za aplikaciju jer napadom na njih napadač može steći kontrolu nad aplikacijom, a možda i poslužiteljem na kojem se aplikacija nalazi.

Nažalost, nije dovoljno samo paziti na kôd svojeg API-ja i ažuriranje poslužiteljskog softvera, već programeri moraju voditi računa i o inačicama svih komponenti i njihovih ovisnosti (engl. *dependencies*) koje se koriste (i na klijentskoj i na serverskoj strani). Pri tome mogu pomoći alati poput npr. *versions*, *dependencycheck*, *retire.js* i sl.

Sve nekorištene komponente, njihove ovisnosti, nepotrebne funkcionalnosti, datoteke i dokumentaciju treba obrisati, a softver koji se koristi redovito ažurirati. To se odnosi na sav softver na poslužitelju, dakle i na operacijski sustav, *web*/aplikacijske poslužitelje, sustav za upravljanje bazama podataka, aplikacije, API-je, sve komponente, okoline za izvođenje (engl. *runtime environments*) i biblioteke.

Komponente se moraju preuzimati s izvornih stranica preko sigurne mreže, a za svaki preuzeti softver korisno je provjeriti i digitalni potpis.

Primjer napada na ovu ranjivost bio bi kao i u slučaju napada na lošu sigurnosnu konfiguraciju, uz naglasak na iskorištavanje javno dostupnih exploita za napad na javno dostupnu ranjivost. Drugim riječima, korištenje komponenti s javno dostupnim ranjivostima može rezultirati time da napadač pronađe i javno dostupan *exploit* kojim može napasti API ili da *exploit kitovi* (koji su detaljnije objašnjeni u dokumentu Nacionalnog CERT-a *Exploit kitovi*) automatizirano iskoriste takve ranjivosti i *exploite* za njih.

2.10 A10 – Nedovoljno zapisivanje i nadzor (engl. *Insufficient Logging & Monitoring*)

Nedovoljno bilježenje događaja unutar sustava (engl. *logging*) i nadzor (engl. *monitoring*) omogućit će napadaču da napravi više štete prije no što je detektiran, spriječiti pravovremenu reakciju na napad/incident i otežati oporavak.

Nadziranje sustava uspješno je ako se zapisuju u dnevnik sve bitne aktivnosti kojima je moguće detektirati i pratiti sve sumnjive aktivnosti i rekonstruirati tijek napada ako do njega dođe. Obavezno se moraju bilježiti, tj. zapisivati u dnevnike, bitni događaji poput prijava, neuspjelih prijava i velikih transakcija (npr. brisanje većeg dijela tablice ili cijele tablice iz baze podataka). Svaka prijava, pogreška u kontroli pristupa i neuspjela validacija korisničkog unosa treba biti zabilježena s dovoljno konteksta da bi se mogli identificirati sumnjivi ili zlonamjerni korisnički računi i držati ih zabilježenima dovoljno dugo da se omogući odgođena forenzička analiza. Velike transakcije trebaju imati reviziju s provjerama integriteta kako bi se spriječilo neovlašteno mijenjanje, brisanje ili dodavanje tablica u bazu.

Uzmimo za primjer pokušaj Brute-force napada kako bi se preuzeo neki korisnički račun na *Vulnerable API*-ju:

1) Napadač se neuspješno uzastopno pokušava prijaviti s istim korisničkim imenom, ali različitim lozinkama. Napadač se već preko npr. 20 puta pokušao prijaviti, a prijave su u otprilike pravilnim vremenskim razmacima, što daje za naslutiti da je riječ o automatiziranom online pogađanju npr. alatom *THC Hydra*.

2) Alati koji nadziru dnevnike (npr. Fail2Ban o kojem smo više pisali u dokumentu Fail2Ban) primjećuju da se događa sumnjiva aktivnost (uzastopne neuspješne prijave ili trošenje puno resursa na API *endpointu* za prijavu) i automatski blokiraju IP adresu s koje dolaze neuspješni zahtjevi za prijavu i upozoravaju sistemskog administratora.

3) *Brute-force* napad uspješno je zaustavljen.

Upozorenja i pogreške zapisane u dnevnik moraju biti jasne i mora biti jasno kakva je točno aktivnost izazvala pojedini zapis. Pragovi za eskalaciju ne smiju biti previsoki ili preniski, već što je više moguće realno postavljeni tako da ne eskaliraju za bilo kakvu sumnjivu aktivnost, a opet da se može u stvarnom (ili barem skoro stvarnom) vremenu otkriti, eskalirati ili upozoriti na aktivan napad kako bi ga se moglo zaustaviti.

Također, dnevnike treba pohranjivati na više lokacija, a ne samo lokalno. Ako je dnevnik pohranjen samo lokalno, napadač koji uspješno napadne računalo i stekne kontrolu nad njim može i izmijeniti ili obrisati taj dnevnik.

Naravno, ne može se pretpostaviti da će svaki napad biti zaustavljen ili spriječen i zato treba biti spreman i za reakciju na incident (engl. *Incident Response*) i oporavak od incidenta (npr. NIST 800-61 rev 2).

Jedan praktičan način na koji se može provjeriti provodi li se dovoljno bilježenja i nadzora je pratiti dnevnike zapisa (engl. *logove*) za vrijeme penetracijskog testiranja ili skeniranja alatima poput ZAP-a ili *Burp Studija*. Takve akcije trebale bi uzrokovati upozorenja i sve akcije koje poduzimaju „napadači“ tad bi trebale biti zapisane u logovima.

3 Zaključak

Digitalno doba dovelo je do sve veće potrebe za *web* aplikacijama – korisnici su navikli svakodnevne obaveze obavljati korištenjem interneta. Kako bi se uspio pratiti tempo brze izrade aplikacija, moderne aplikacije više se ne izrađuju u jednoj monolitnoj cjelini, već se nastoje modulirati na manje komponente koje surađuju u obavljaju nekog posla.

Pritom se za komunikaciju tih komponenti u internetskim i mobilnim aplikacijama koriste HTTP REST API-ji. Takvi API-ji jednostavni su za razumijevanje i korištenje te su široko prihvaćeni među programerima, ali zbog modularne arhitekture i prijenosa mrežom sa sobom donose i određene sigurnosne rizike.

Većina ranjivosti koja se javlja u API-jima podudara se s ranjivostima uobičajenih *web* stranica uz manje iznimke koje su prokomentirane kroz dokument.

Kroz svako pojedino poglavlje navedene su mjere zaštite od pojedinih ranjivosti i napada, ali prvi korak osiguravanja API-ja/aplikacija i pisanja sigurnih API-ja/aplikacija svakako je razumijevanje koje su to česte ranjivosti, zašto su tako česte i kako se iskorištavaju u napadima. Tek tad moguće je na ispravan način izbjeći ranjivosti i uspostaviti učinkovite mjere zaštite.

I uz najveći mogući oprez, nije se uvijek moguće zaštititi od napada – napadač ima mnogo vremena i vektora napada, dok programeri imaju mnogo drugog posla i moraju štititi sustav na svim frontama. Zbog toga je bitno voditi računa o redovnoj izradi pričuvnih kopija te bilježenju događaja unutar sustava u dnevnike pomoću kojih se može shvatiti što se dogodilo i popraviti pogrešku koja je dovela do napada, kao i imati plan reakcije na incident.

4 Literatura

1. **Atlassian.** JSON requests and responses. *Atlassian.* [Mrežno] [Citirano: 24. listopada 2019.] https://developer.atlassian.com/server/crowd/json-requests-and-responses/.
2. **OWASP.** OWASP Top 10 - 2017. *OWASP.* [Mrežno] 2017. [Citirano: 24. listopada 2019.] https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.
3. **Tech Differences.** Difference Between Authentication and Authorization. *Tech Differences.* [Mrežno] 30. siječnja 2018. [Citirano: 28. listopada 2019.] https://techdifferences.com/difference-between-authentication-and-authorization.html.
4. **Brian, Matt.** Bad day for LinkedIn: 6.5 million hashed passwords reportedly leaked – change yours now. *The Next Web.* [Mrežno] 6. lipnja 2012. [Citirano: 24. listopada 2019.] https://thenextweb.com/socialmedia/2012/06/06/bad-day-for-linkedin-6-5-million-hashed-passwords-reportedly-leaked-change-yours-now/.
5. **PortsWigger.** XML external entity (XXE) injection. *PortsWigger.* [Mrežno] [Citirano: 25. listopada 2019.] https://portswigger.net/web-security/xxe.
6. **Doglio, Fernando.** How to secure a REST API using JWT. *LogRocket.* [Mrežno] 11. ožujka 2019. [Citirano: 24. listopada 2019.] https://blog.logrocket.com/how-to-secure-a-rest-api-using-jwt-7efd83e71432/.
7. **Smith, Jerome.** JWT Attack Walk-Through. *NCC Group.* [Mrežno] 24. siječnja 2019. [Citirano: 28. listopada 2019.] https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/january/jwt-attack-walk-through/.