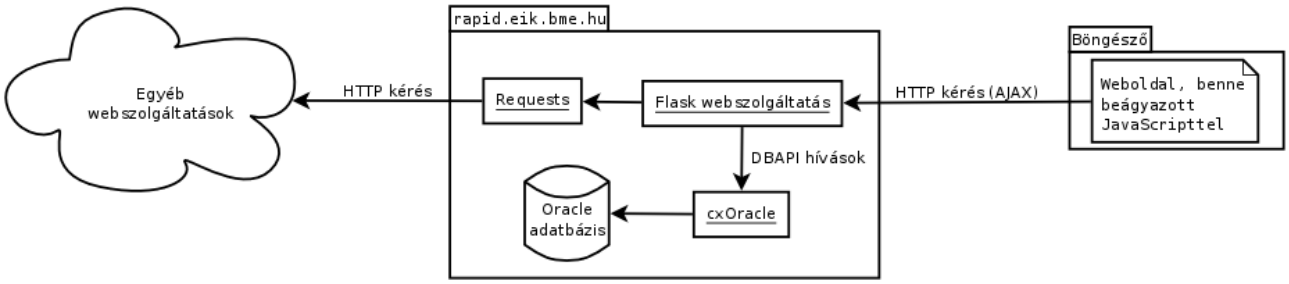
Tartalom

IV. gyakorlat: SOA szemléletű rendszer REST architektúrában

Szerző: Veres-Szentkirályi András

A SOA labor során egy, a SOA elvei mentén működő rendszert fogunk készíteni, szerveroldalon Python, böngészőoldalon JavaScript használatával. A segédletben először a SOA alapjai kerülnek bemutatásra, majd a szerveroldalon futó Python SOA szerver és kliens implementációk, végül a böngészőben futó megoldások. Az alábbi áttekintő ábrán látható az egyes felhasznált komponensek egymáshoz való viszonya, a segédlet olvasása során érdemes időről-időre visszagörgetni ide.



1. SOA alapok

1.1. Bevezetés és történelem: SOA és webszolgáltatások

A SOA (szolgáltatás-orientált architektúra) természetes evolúciós lépés az informatikai rendszerekben. A strukturált, objektum-orientált, komponens-orientált programozásra építve szolgáltatás-orientált rendszerek esetén a kód újrafelhasználása a rendszer szolgáltatásokra bontásán alapul. Ezek jellemzője, hogy elérésük szabványos interfészen keresztül történik, így szemben például az objektum-orientált rendszerekkel, a szolgáltatás és igénybevevője akár eltérő programnyelven is íródhat, eltérő operációs rendszeren vagy számítógépen is futhat. A fent vázolt evolúció vonalát folytatva így csökken a komponensek közti függőség és csatolás, ennek viszont alapvető feltétele a már említett szabványos felületek és formátumok használata. Tipikusan ilyen az XSQL mérésen majd mélyebben bemutatott XML, illetve az ezen a mérésen bemutatásra kerülő, egyre jobban terjedő JSON.

A webszolgáltatások a SOA egyik legnépszerűbb implementációit képviselik, nevüket a HTTP protokoll használatáról kapták. Mint nyílt, egyszerű és elterjedt szabvány, megfelel a SOA rá szabott feltételeinek, és a bináris protokollokkal (OMG/CORBA, Microsoft/DCOM) szemben egyszerűbben feldolgozható és könnyebben továbbítható tűzfallal izolált hálózati szegmensek között is, ezáltal biztonsági szempontból is kedvezőbb. Korábban egyeduralkodó volt és most is elterjedt protokoll az 1998-ban a Microsoft berkeiben megalkotott SOAP, ami XML-t használt a kérések és válaszok leírására, de legtöbbször szintén a HTTP protokoll felett került megvalósításra. Ennek szemantikája leginkább távoli metódushívásnak feleltethető meg, a kérésben a metódus neve és paraméterei kerültek leírásra, majd a válaszban a metódus visszatérési értéke került visszaküldésre.

1.2. A laboron bemutatott megvalósítás: REST

A REST (Representational State Transfer) egy megvalósítása a SOA-nak, inkább megvalósítási stílusként, vezérelvek gyűjteményeként fogható fel, mint protokollként. A SOAP-pal szemben a REST a HTTP eredeti, webes szemantikáját terjeszti ki, az elérhető távoli erőforrások állapotát HTTP kéréssel lehet lekérdezni és/vagy módosítani. Szabvány ugyan nem határozza meg a használt átviteli formátumot, elterjedten használt megoldás az XML és JSON. A szolgáltatás elérésének paraméterei tipikusan a lekért URL-ben kerülnek átadásra, opcionálisan a HTTP kérés fejléce (HTTP fejlécek) is használhatók a célra. Az erőforrások tipikusan az adatbázis tábláival (entitáshalmazokkal) állíthatók párhuzamba, a HTTP verbek (lásd lejjebb) pedig a következő SQL utasításokkal.

Verb	Útvonal	SQL utasítás
GET	/hallgatok.xml	SELECT * FROM hallgatok
GET	/hallgatok/42.xml	SELECT * FROM hallgatok WHERE id = 42
POST	/hallgatok.xml	INSERT INTO hallgatok (...) VALUES (...)
PUT	/hallgatok/42.xml	UPDATE hallgatok SET ... WHERE id = 42
DELETE	/hallgatok/42.xml	DELETE FROM hallgatok WHERE id = 42

Gyakorlatban POST és PUT esetén a kérés törzse tartalmazza a beszúrandó/frissítendő adatokat, GET és DELETE esetén a lekérdezni/törölni kívánt entitás azonosítói az URL-ben kerülnek megadásra.

Az első oszlopban található HTTP verb magyarul leginkább metódusnak nevezhető, a HTTP protokollon közlekedő kérés típusát jelzi. A verb kifejezést használjuk a továbbiakban, hogy megkülönböztessük az objektum-orientált paradigma metódus kifejezésétől. Bár HTTP verb elvileg tetszőleges string lehet, a gyakorlatban a fentiek használata ajánlott, mivel szemantikájuk bejáratott és általánosan elfogadott. Például mivel a GET kérésnek nem szabadna megváltoztatnia az erőforrás állapotát, bármikor újraküldhető, ill. közbenső (pl.

proxy-) szerverek és szolgáltatások gyorsítótárban tárolhatják a választ a szolgáltatás és a hálózat terhelésének mérséklése érdekében.
A weben több lista is található különféle, bárki számára elérhető REST webszolgáltatásokról, ilyen például a ProgrammableWeb, ahol a segédletben bemutatott szolgáltatások is elérhetők.

1.3. HTTP protokoll és a REST kapcsolata

Egy REST architektúrájú webszolgáltatás igyekszik a HTTP protokoll beépített lehetőségeit kihasználni metaadatok átadására – szemben például a SOAP-pal, amely ezt a HTTP felett oldja meg. A labor során használt metaadatok kiterjednek a szolgáltatás-hívás sikerességére, az üzenet (kérés és válasz) tartalmának típusára, és autentikációra is, ebben a pontban az elsővel foglalkozunk részletesen.
Egy tipikus HTTP/1.1 válasz egy ún. státusz sorral kezdődik, amely tartalmazza az alkalmazott HTTP protokoll verziót, és a HTTP kérés feldolgozásának állapotát gépi és emberi feldolgozásra alkalmas változatban is. Előbbi egy háromjegyű szám, tipikusan ez kerül értelmezésre a kliensek által, és mivel hierarchikus felépítésű, mindjárt az első számjegy elárul néhány dolgot a tranzakció állapotáról.

- **1xx** – informális, a kérés feldolgozása még nem történt meg
- **2xx** – sikeres feldolgozás, a kérést elfogadta a kiszolgáló
- **3xx** – átirányítás, a kliens feladata további lépések megtétele a siker érdekében
- **4xx** – kliens hiba, a kérés nem megfelelő valamilyen oknál fogva
- **5xx** – szerver hiba, a kérést képtelen kiszolgálni valamilyen oknál fogva

A kódokat a 2616-os RFC 10. szekciójában szabványosították, a leggyakoribb kód a 200 (OK), melyet a legtöbb szerveroldali megoldás alapértelmezetten küld. Ezen kívül még az alábbi kódok ismerete javasolt a laborfeladatok megoldásához. (A kódot zárójelben követi a szabvány szerinti szöveges, ám kizárólag emberi felhasználásra szánt változat.)

- **201** (Created) – POST kérés esetén a sikeres létrehozási műveletet jelezheti
- **204** (No Content) – PUT és DELETE kérés esetén a sikeres módosítási/törlési műveletet jelezheti, mivel nincs szükség további információ átadására a válasz törzsében (szemben a 201-gyel, ahol például az új elem egyedi kulcsa kerülhet visszaadásra)
- **404** (Not Found) – talán a legismertebb státusz kód, azt jelzi, hogy a kért erőforrás nem létezik, például `GET /hallgatok/42.xml` jellegű kéréseknél mindenképpen javasolt, ha a 42-es egyedi kulccsal nem létezik rekord; ha azonban a kérésre több rekord is visszaadásra kerülhet (pl. kereső vagy listázó végpont), akkor szemantikailag megfelelő egy 200-as válasz is, üres jelentésű törzzsel (pl. JSON formátum esetén `{"results": []}` ilyen lehet).

1.4. A formátum: JSON

A JSON egy olyan pehelysúlyú adatátviteli formátum, mely a JavaScript nyelv szabvány szerint korlátozott részhalmaza, így minden JSON szabványnak megfelelő karaktersorozat egyben érvényes JavaScript kifejezés is. Hasonló célokra alkalmazható, mint az XML, viszont annál jóval egyszerűbb, így feldolgozása kevesebb kóddal és hibalehetőséggel megoldható. Érvényes JSON kifejezések a következők:

- Unicode stringek idézőjelek közé zárva, különleges karaktereket (ékezetes betűk, sortörés, idézőjel, stb.) escape-elve: `"alma"`, `"M\u0171egyetem"`
- Egész számok: `42`, `-5`
- Lebegőpontos tizedestörtek (tizedes pont használatával): `5.5`, `-273.1`
- Null érték: `null`
- Logikai értékek: `true`, `false`

- Érvényes JSON kifejezésekből alkotott lista: `["alma", 42, null]`, `[]`, `[1.1]`
- Stringek és érvényes JSON kifejezések párjaiból alkotott szótár: `{"telefon": "phone"}`, `{"1": "Jan", "2": "Feb"}`

A *whitespace* karaktereket (újsor, tabulátor, szóköz) a feldolgozók figyelmen kívül hagyják, így ha a maximális tömörség a cél, egy JSON kimenet akár egyetlen hosszú sor is lehet, míg ha az olvashatóság is fontos, a legtöbb könyvtár képes tagolt, behúzott (indentált) kimenet előállítására. A következő két kimenet például ekvivalens:

```
{"targynev": "Szoftver Labor 5", "laborok": ["Oracle", "SQL", "JDBC", "SOA", "XSQL"]}
```

```
{
  "targynev": "Szoftver Labor 5",
  "laborok": [
    "Oracle",
    "SQL",
    "JDBC",
    "SOA",
    "XSQL"
  ]
}
```

Látható, hogy az XML-hez képest nincsenek névterek és a karakterkódolás sincs megadva, mivel a kötelező string kódolás miatt egy szabályos JSON kimenet kizárólag 0-127 közötti bájtokat tartalmazhat, melyeket ASCII szerint kell értelmezni. Ennek megfelelően ASCII-vel kevés átfedést mutató (pl. orosz, kínai, japán) nyelven írt stringek esetében a kimenet mérete többszöröse lehet egy azonos jelentéstartalmú XML-ének – ez az ára az egyszerű szabványnak és feldolgozó szoftvereknek. Megjegyzendő azonban, hogy JSON legtöbbször HTTP felett kerül továbbításra, mely tartalmaz lehetőséget – többek között – *gzip* tömörítésre, így a legtöbb esetben a JSON által okozott méretkülönbség elhanyagolható.

1.5. Teszteléshez svájci bicska: cURL

A *cURL* egy parancssoros kliens többek között HTTP protokollhoz, így webszolgáltatások tesztelésére is használható. A legtöbb UNIX-szerű rendszeren (például a rapidon is) telepítve van, de honlapjáról letölthető szinte minden operációs rendszerre, még kevésbé elterjedtekre is. A programot a `-h` (help) kapcsolóval elindítva részletes súgót kapunk, számtalan paraméterének kimerítő bemutatása nem célja e segédletnek, így alább a REST tesztelés szempontjából releváns use-case-ek kerülnek bemutatásra. GET kérést indítani a legegyszerűbb, ekkor az egyetlen paraméter a lekérni kívánt URL.

```
$ curl http://currencies.apps.grandtrunk.net/getlatest/eur/huf
298.880251501
```

Amennyiben a hibakereséshez szükséges, a `-v` (verbose) kapcsolóval részletesebb kimenet kérhető.

```
$ curl -v http://currencies.apps.grandtrunk.net/getlatest/eur/huf
* About to connect() to currencies.apps.grandtrunk.net port 80
*   Trying 66.33.208.161... connected
* Connected to currencies.apps.grandtrunk.net (66.33.208.161) port 80
> GET /getlatest/eur/huf HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5 OpenSSL/0.9.8b
zlib/1.2.3 libidn/0.6.5
> Host: currencies.apps.grandtrunk.net
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< Date: Sun, 05 Apr 2015 09:11:39 GMT
< Server: Apache
< Cache-Control: public
< Expires: Mon Apr  6 04:11:39 2015
< X-Powered-By: Phusion Passenger 4.0.59
< Content-Length: 13
< Status: 200 OK
< Vary: Accept-Encoding
< Content-Type: text/plain; charset=utf-8
Connection #0 to host currencies.apps.grandtrunk.net left intact
* Closing connection #0
298.880251501
```

Amennyiben adatot kell beküldeni a szolgáltatásnak, a következő kapcsolók lehetnek segítségünkre. Amennyiben egy kapcsoló paramétere szóközt tartalmaz, mindenképp idézőjelek közé kell tenni, egyébként elhagyható. Amennyiben idézőjelek is szerepelnek a tartalomban (ami JSON esetében nem ritka), aposztróf karakter is használható idézőjel helyett a paraméter határolására.

- `--request <verb>` a megadott HTTP verbet használja `GET` helyett
- `--header "Fejlec: ertek"` hozzáfűzi a megadott fejlécet a HTTP kéréshez, leginkább `Content-Type` megadására szokás REST esetén használni, természetesen többször is megadható
- `--data "foo bar qux"` a megadott adatot továbbítja a kérés törzsében, változtatás nélkül

Alább látható egy példa egy POST kérésre, mely a GitHub Gist szolgáltatásában hoz létre egy `SOA labor` tartalmú `szoftlab.txt` nevű fájlt.

```
$ curl --request POST --header "Content-Type: application/json" --data '{"public":
    true,     "files":     {"szoftlab.txt":     {"content":     "SOA     labor"}}}'
    https://api.github.com/gists
{
  "url": "https://api.github.com/gists/5345409",
  "forks_url": "https://api.github.com/gists/5345409/forks",
  "commits_url": "https://api.github.com/gists/5345409/commits",
  "id": "5345409",
  "git_pull_url": "https://gist.github.com/5345409.git",
  "git_push_url": "https://gist.github.com/5345409.git",
  "html_url": "https://gist.github.com/5345409",
  "files": {
    "szoftlab.txt": {
      "filename": "szoftlab.txt",
      "type": "text/plain",
      "language": null,
      "raw_url":
    "https://gist.github.com/raw/5345409/4008674e9bc47bdf54a28260e7a502f30aca834d/
    szoftlab.txt",
      "size": 9,
      "content": "SOA labor"
    }
  },
  "public": true,
  "created_at": "2013-04-09T12:43:11Z",
  "updated_at": "2013-04-09T12:43:11Z",
  "description": null,
  "comments": 0,
  "user": null,
  "comments_url": "https://api.github.com/gists/5345409/comments",
  "forks": [

  ],
  "history": [
    {
      "user": null,
      "version": "05aa55498c23f46abe542c61ec6fd4e4d75910ea",
      "committed_at": "2013-04-09T12:43:11Z",
      "change_status": {
        "total": 1,
        "additions": 1,
        "deletions": 0
      },
      "url":
    "https://api.github.com/gists/5345409/05aa55498c23f46abe542c61ec6fd4e4d75910ea
    "
    }
```

```
  ]
}
```

2. Szerveroldal: Python

2.1. Programnyelv: Python

A Python egy olyan programozási nyelv, mely lehetővé teszi – akár egy programon belül is – több (funkcionális, procedurális, objektum-orientált) paradigma használatát is. E tulajdonsága miatt többek között rövid scriptek nyelveként terjedt el, azonban mára a helyzet megváltozott, és jelenleg egyike azoknak a nyelveknek, melyek jól használhatók webszolgáltatások és -alkalmazások készítésére. Jelenleg a Python 2.x verziói a legelterjedtebbek, így a labor során tett megállapítások erre lesznek igazak.

Kezdők számára legszembetűnőbb tulajdonsága, hogy a blokkok egymáshoz való viszonyát nem kezdő és lezáró karaktersorozatok, hanem a behúzás mértéke definiálja. A behúzás történhet tabulátor vagy szóköz karakterrel és ezek kombinációjával is, a hivatalos Python dokumentáció blokkonként 4 szóközt ajánl. Érdemes Python kód szerkesztéséhez olyan szövegszerkesztőt választani, amelyben beállítható a behúzás megjelenítése vagy automatikus kezelése, különben érdekes hibajelenségekkel találkozhatunk, amelyek forrása nehezen ismerhető fel.

A nyelv erősen típusos, azaz például PHP-tól eltérően sosem történik automatikus típuskonverzió, azonban a változókat és típusukat nem szükséges előre deklarálni, a változók első értékadásukkor jönnek létre. A következő kódrészlet végére például `b` lehet `int` vagy `str` típusú is.

```
if a > 5:
    b = 10
else:
    b = "kisebb"

print b
```

Az alapvető típusok a következők, a példákban látható, hogyan adható meg kódban egy adott típusú konstans érték:

- `int`: egész számok, pl. `42`
- `float`: lebegőpontos számok, pl. `5.5`
- `bool`: logikai érték, pl. `True`, `False`
- `NoneType`: null érték, `None`
- `str`: bájtokból álló string, pl. `"szoftlab5"`
- `unicode`: karakterekből álló Unicode string, pl. `u"M\u0171egyetemi hallgató"`
- `list`: tömb, pl. `[42, 5.5, "szoftlab5"]`
- `tuple`: „ennes”, pl. `(42, 5.5, "szoftlab5")`, `("SOA",)`
- `dict`: szótár, pl. `{"telefon": "phone"}`, `{1: "Jan", 2: "Feb"}`

Látható, hogy az alapvető típusok reprezentációja nagyban hasonlít a JSON-höz, egy különbség a stringek kezelése (ebből a szempontból fejlettebb a Python 3, illetve itt nem kötelező, de megengedett az ASCII-n kívüli karakterek escape-elése). Fontos, hogy a Java és .NET string típusaihoz hasonlóan **a `str` és `unicode` példányok „rögzítettek”**, így az ezeken végzett módosító műveletek (`lower`, `upper`, `replace`, stb.) nem a metódushívás „alanyát” módosítják, hanem **a módosított stringet visszatérési értékben kapjuk meg**.

Szintén különbség a `tuple` típus megléte, mely hasonlít a tömbökre, fontos különbség viszont, hogy stringekhez hasonlóan „rögzített”, azaz létrejötte után sem az elemek száma nem módosítható, sem az elemek nem cserélhetők más értékre. Ennek megfelelően több elem

visszatérési értékként való átadásánál szokás alkalmazni, pl. adatbázis sor lekérdezése. Egyébként minden más szempontból egyezik tömbökkel a használata, például a `[]` operátorral elérhetők az egyes elemek. Fontos, hogy az egyetlen elemet tartalmazó `tuple` nem ekvivalens a benne tárolt egyetlen elemmel (azaz `("SOA",) != "SOA"`), és mivel a zárójel nyelvi elem, ezért egyetlen elem esetén a plusz vessző jelzi, hogy itt nem csak tagolásról van szó, hanem `tuple`-ről.
A Python nyelven írt kód modulokba szerveződik, egy `.py` kiterjesztésű fájl egy modulnak felel meg, melynek neve megegyezik a fájl kiterjesztés nélküli nevével. A modulok egyrészt importálhatók név szerint, másrészt importálhatók belőlük tetszőleges függvények, osztályok, változók, konstansok. A modul karakterkódolása tetszőleges lehet, viszont ASCII-n kívüli (pl. ékezetes) karakterek használata esetén ezt jelezni kell egy kommentben, (BOM nélküli) UTF-8 esetén például az alábbi módon.

```
# -*- coding: utf-8 -*-
```

2.2. Adatbázis-elérés: DBAPI

A Python adatbázis API-ja a JDBC-hez hasonló módon biztosítja, hogy a különféle adatbázis-kezelőket Python nyelven egységes felületen keresztül lehessen elérni. Fontos eltérés azonban, hogy a kapcsolódás és a paraméteres lekérdezések kezelése meghajtónként különbözhet – egyes adatbázis-kezelők ugyanis például csak pozícionális paraméterezést támogatnak (a JDBC például csak ilyet támogat, adatbázis-meghajtótól függetlenül), míg mások nevesített paraméterek használatát is lehetővé teszik. Az adatbázis-kapcsolatot reprezentáló objektum adatbázis-függő módon jön létre, fontosabb elvárt metódusai és tulajdonságai a következők.

- `cursor()`: visszaad egy adatbázis-kurzort, lekérdezések végrehajtására, azok eredményének elérésére
- `begin()`: tranzakciót indít
- `commit()`: commitolja a tranzakciót
- `rollback()`: rollbackeli a tranzakciót
- `close()`: lezárja a kapcsolatot
- `autocommit`: jelzi, hogy aktív-e az automatikus commit

A `cursor` által visszaadott kurzor objektum a következő metódusokat és tulajdonságokat biztosítja.

- `execute(sql, ...)`: végrehajt egy SQL lekérdezést, opcionálisan több paramétert is fogadhat, melyek meghajtó-függő módon kerülnek értelmezésre paraméteres lekérdezések esetén
- `executemany(sql, params)`: végrehajt egy SQL lekérdezést a második paraméter (ami például egy `list` lehet) összes elemére, ezáltal megspórolva az utasítás ismételt feldolgozását
- `fetchone()`: visszaad egy sort az eredményhalmazból `tuple` típusú n-esként, vagy `None`-t, ha már nincs kiolvasásra váró sor
- `fetchall()`: visszaadja az eredményhalmaz összes sorát `tuple` típusú n-esek `list` tömbjeként
- `close()`: lezárja a kurzort
- `rowcount`: kiolvasható belőle az eredményhalmaz számossága

Ezen kívül a kiterjesztett DB API-t támogató kurzor objektumok (ilyen az Oracle is) iterálhatók, így például a következő `for` ciklussal kiíratható egy tábla tartalma:

```
cur.execute("SELECT keresztnev, vezeteknev FROM szemely")
for k, v in cur:
```

```
    print "Keresztnév:", k
    print "Vezetéknév:", v
```

A fenti példa a Python `for` utasításának két jellegzetes tulajdonságára is rámutat.
Egyrészt a `for` mindig valamilyen iterálható (vö. Java `Iterable<E>` ill. .NET `IEnumerable<E>` interfészei) objektumon (`list`, adatbázis eredményhalmaz) halad végig a ciklussal, hasonlóan a Java új `for` és a C# vagy PHP `foreach` utasításához.
Másrészt amennyiben a ciklus aktuális iterációjában lekért érték több elemből áll (mint a példában az eredményhalmaz egy sora), nem szükséges kézzel elemeire szedni. Amennyiben a `for` és `in` kulcsszavak közt felsorolt változónevek száma megegyezik az aktuális érték elemszámával, 1:1 behelyettesítés történik, ellenkező esetben kivételt kapunk.
A fenti példa tehát leírható lett volna `for row in cur:` kezdettel is, ekkor `k` helyett `row[0]`, `v` helyett `row[1]` kifejezést kellett volna írni.

2.3. Kapcsolódás Oracle-höz: cxOracle

A cxOracle egy Pythonban és C-ben írt, a Python adatbázis API-val kompatibilis modul, melynek segítségével a hivatalos Oracle natív könyvtárak használatával lehet Python kódból Oracle adatbázishoz kapcsolódni. Hasonlóan a JDBC-hez, kapcsolódáskor a felhasználónév, jelszó és SID mellett az adatbázis elérhetőségét kell megadni, mint az az alábbi példában látható.

```
import cx_Oracle

conn = cx_Oracle.connect('felhasznalonev', 'jelszo',
        'szerver.hosztnev.hu:1521/SID')
conn.close()
```

A `connect` által visszaadott, kapcsolatot reprezentáló objektum `close` metódussal zárható le, `cursor` metódussal pedig kurzor kérhető tőle, melyen keresztül a Python adatbázis API-nak megfelelő hívásokkal SQL lekérdezések indíthatók az adatbázis felé. JDBC-től eltérően a Python adatbázis API paraméteres lekérdezéseknél engedi az adatbázisonként eltérő hívási konvenciót, ez Oracle esetében mind a lekérdezés szövegében, mind az `execute()` hívásakor nevesített paraméterekkel történik.

```
cur = conn.cursor()
cur.execute('SELECT foo, bar FROM tabla WHERE id = :id', id=42)
results = cur.fetchall()
cur.close()
```

A meghajtó támogatja a Python `with` kontextus-kezelőjét (mely leginkább a C# `using` ill. a Java 7 try-with-resources megoldásához hasonlítható) kurzorok esetében, a `with` blokkon belüli utasítások előtt tranzakciót indít, majd sikeres végrehajtás esetén commitol, kivétel esetén rollbacket hajt végre. Az alábbi két példa tehát ekvivalens.

```
with conn:
    cur.execute('INSERT INTO laborok (labor, sorszam) VALUES ("Szoftlab", 5)')
```

```
try:
    conn.begin()
    cur.execute('INSERT INTO laborok (labor, sorszam) VALUES ("Szoftlab", 5)')
    conn.commit()
except:
    conn.rollback()
    raise
```

Amennyiben a rekord azonosítóját trigger tölti ki (például szekvencia alapján), a következő módon tudjuk megszerezni a beszúrt sor azonosítóját (forrás, természetesen ez a megoldás

nemcsak a rekord azonosítójára, hanem bármely mezőjének a ténylegesen beszúrt értékének a lekérdezésére használható):

```
idVar = cursor.var(cx_Oracle.NUMBER)
cursor.execute("""INSERT INTO tabla (oszlop) VALUES (:ertek)
    RETURNING azon INTO :az""", ertek=42, az=idVar)
azon = idVar.getvalue()
```

2.4. Dátum és idő kezelése

A Python standard könyvtárában szereplő `datetime` modul `datetime`, `date` és `time` osztályai használhatók dátumok, időpontok és ezek kombinációinak reprezentálására. Ennek megfelelően a cx_Oracle is ilyen objektumokkal tér vissza megfelelő típusú oszlop esetén, és paraméterként is elfogad ilyet. A JSON formátum azonban nem tartalmaz ilyet, így itt ad-hoc módszerek terjedtek el, leggyakoribb az ISO 8601 szabványú string használata – ezt ismertetjük alább is – valamint találkozhatunk még Unix időbélyeg (az 1970. január 1. óta eltelt másodpercek számának) számként való reprezentálásával.

Dátum (és dátum-idő) ISO 8601-be történő konvertálására beépített támogatással rendelkezik a modul; mind `datetime`, mind `date` objektumok esetén az `isoformat()` metódus közvetlenül a megfelelő formátumot adja vissza, opcionális paraméterként megadható a dátumot és időt elválasztó paraméter (alapértelmezésben „T” betű, de elterjedt még a szóköz is ilyen célra). Ellenkező irányban (string parse-olása, pl. JSON-ből adatbázisba íráshoz) összetettebb a feladat, a `datetime` objektum statikus `strptime(string, format)` metódusa egy megadott formátumsztring szerint olvassa be a bemenetet és állít elő `datetime` objektumot.

Természetesen `datetime` példányból leválasztható akár a dátum, akár az idő a `date` ill. `time` tulajdonságon keresztül, és mindhárom típusra működnek a standard összehasonlító operátorok (=, <, >, stb.).

2.5. Webszolgáltatás keretrendszer: Flask

A Flask egy Python nyelven írt webes mikro-keretrendszer, mellyel webalkalmazások és webszolgáltatások is készíthetők. Utóbbi szempontjából segítség, hogy készen tartalmaz a HTTP kérések mintaillesztés-alapú kiszolgálására diszpécser logikát, a fejlesztést pedig gyorsítja, hogy külső web- vagy alkalmazásszerver (Apache, Tomcat, IIS) nélkül is működőképes és tesztelhető az elkészült szolgáltatás. Használata az alábbi példa alapján elsőre átlátható:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def szoftlab5():
    return "Szoftlab 5"

if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

Az első két sorban importáljuk a `flask` modulból a `Flask` osztályt, majd példányosítjuk `app` néven. Ezt követően ezen a példányon keresztül építhetjük fel a webalkalmazást, a `route` dekorátort használva. Ennek egyetlen kötelező paramétere az az útvonal, melynek HTTP-n való lekérése esetén a megjelölt függvény fog lefutni. A függvény visszatérési értékéből a Flask előállítja a HTTP választ, string visszaadott típus esetén `text/html` megjelöléssel.

Végül az utolsó két sor fejlesztés-tesztelés során hasznos, amennyiben a modult közvetlenül indítjuk (pl. parancssorból `python modulneve.py`), elindítja a Flask beépített webszerverét

az 5000-es TCP porton, a hibakeresőt engedélyezve (utóbbit persze éles üzemben nem ajánlott bekapcsolni). Ennek köszönhetően a fenti szkriptet elindítva a böngészőben megnyitható a http://localhost:5000/ URL, melyen a `"Szoftlab 5"` szöveg köszönt minket.
A `__name__` ugyanis az aktuális modul nevét tartalmazza (azaz általában a Python forrásfájl neve kiterjesztés nélkül), kivéve, ha az adott fájl a program belépési pontja, ekkor ugyanis a `"__main__"` értéket kapja. A `Flask` osztály konstruktorának belső működéséhez van szüksége a webalkalmazás/webszolgáltatás fő moduljának nevére, ezért kerül ez átadásra, az utolsó blokkban pedig így oldható meg, hogy a fájl közvetlenül indítva elinduljon a fejlesztői szerver, más modulból (például web- vagy alkalmazásszerverből) importálva viszont ne.
REST webszolgáltatások készítésekor hasznos segítség a `flask` modul `jsonify` függvénye, melynek visszatérési értéke egy olyan HTTP válasz, amely JSON formátumban tartalmazza a függvény paramétereit, és megfelelően be is állítja a `Content-Type` fejlécet. Használata az alábbi példából látható (a beépített webszervert indító részt ezúttal elhagytuk):

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/lab.json")
def lab():
    return jsonify(szoftlab=5, laborok=["Oracle", "SQL", "JDBC", "XSQL", "SOA"])
```

A fenti példát futtatva a következő válasz érkezik a http://localhost:5000/lab.json URL-re irányuló kérésre:

```
HTTP/1.0 200 OK
Content-Length: 98
Content-Type: application/json
Date: Sun, 05 Apr 2015 09:06:03 GMT
Server: Werkzeug/0.10.4 Python/2.7.9

{
    "laborok": [
            "Oracle",
            "SQL",
            "JDBC",
            "XSQL",
            "SOA",
    ],
    "szoftlab": 5
}
```

A diszpécser logika lehetővé teszi paraméterezett útvonalak létrehozását is, ilyenkor az útvonalban kisebb-nagyobb jelek közé írt azonosítóval jelölhetjük meg a változó bemenet helyét és nevét. A `route` dekorátorral megjelölt függvény fejlécében ezután minden azonosítóhoz kell tartozzon egy azonos nevű paraméter is, ebben kerül ugyanis átadásra a mintaillesztés eredménye, ezt a következő példa szemléletesen bemutatja (itt az import részt is elhagytuk).

```
@app.route("/laborok/<n>.json")
def labor(n):
    return "Szoftlab {0}".format(n)
```

A fenti példát futtatva a következő válasz érkezik egy, a http://localhost:5000/laborok/5.json URL-re irányuló kérésre:

```
HTTP/1.0 200 OK
Content-Length: 10
Content-Type: text/html; charset=utf-8
Date: Sun, 05 Apr 2015 09:07:47 GMT
```

```
Server: Werkzeug/0.10.4 Python/2.7.9

Szoftlab 5
```

A beépített diszpécser miatt (pl. PHP, ASP, CGI megoldásokkal szemben) nem fájlrendszer alapján történik a kérések kiszolgálása, így statikus fájlok kiszolgálásához az alapértelmezett Flask konfigurációval az aktuális könyvtár `static` alkönyvtárát lehet használni. Az ezen belül található fájlok a `/static/` útvonalon belül kerülnek kiszolgálásra, tehát például a `static` könyvtáron belül elhelyezett `foo.png` kép a beépített webszervert használva a `http://localhost:5000/static/foo.png` URL-en érhető el. Bár tetszőleges plusz könyvtár felvehető, ezt a mintát követve később, éles környezetben egyszerűen megoldható, hogy a statikus fájlok különálló, erre optimalizált szerverről (lighttpd, nginx) kerüljenek kiszolgálásra.

2.6. Távoli webszolgáltatások elérése: Requests

A Requests egy Python nyelven írt, fejlett HTTP kliens könyvtár, melynek funkcionalitása REST szolgáltatások igénybevételét is megkönnyíti. A modul HTTP verbekkel egyező nevű függvényeivel egyszerűen elérhető bármilyen webes forrás, ezek visszatérési értéke pedig egy olyan `Response` típusú objektum, melynek metódusain és tulajdonságain keresztül magas szinten kezelhető a HTTP válasz tartalma. Az alábbi, első példában egy egyszerű GET kérést küldünk, majd kiíratjuk a választ a standard kimenetre.

```
import requests

response = requests.get("http://currencies.apps.grandtrunk.net/getlatest/eur/huf")
print response.text
```

A szkript kimenete:

```
298.880251501
```

URL paramétereket a `params` nevű paraméterben átadva a könyvtár magától elvégzi a megfelelő kódolási feladatokat, megfelelő `Content-Type` érték esetén pedig a JSON formátumú válaszok feldolgozásra kerülnek, és strukturált módon elérhetők a válasz `json()` metódusán keresztül. A következő példa az *OpenStreetMaps Nominatim* szolgáltatásán keresztül lekérdezi, majd kiíratja Dobogókő koordinátáit. A válasz JSON szótárak listája, melynek minden elemén végigiterálunk, majd kiíratjuk a `lat` és `lon` elemét. Látható, hogy a *Requests* a JSON szótárból *Python* `dict`-et épített, mely a tömbhöz hasonlóan címezhető, hasonlóan, mint a PHP asszociatív tömbjei, a *Java* `Map` vagy a *C#* `IDictionary` interfésze esetében.

```
params = {'format': 'json', 'city': u'Dobogókő', 'country': 'Hungary'}
response = requests.get('http://nominatim.openstreetmap.org/search',
params=params)
results = response.json()
for result in results:
    print result['lat'], result['lon']
```

A szkript kimenete:

```
47.7193734 18.8959654
```

3. Böngészőoldal: JavaScript

3.1. Programnyelv: JavaScript

A Pythonhoz hasonlóan a *JavaScript* is egy olyan programozási nyelv, mely lehetővé teszi – akár egy programon belül is – több (funkcionális, procedurális, objektum-orientált) paradigma

használatát is. Az elterjedt böngészők mindegyike támogatja weboldalakba illesztését, azonban ma már szerveroldalon is használják. A böngészők és azok egyes verziói között kisebb-nagyobb implementációbeli különbségek tapasztalhatók a webes funkcionalitás tekintetében, így gyakran JavaScript keretrendszerek (*jQuery*, *Prototype*, stb.) egészítik ki a beépített könyvtárak lehetőségeit. Szintaktikájában a *Java* nyelvre hasonlít, annak típus-meghatározásai nélkül.
A nyelv dinamikusan típusos, a változók típusát nem szükséges előre deklarálni, az értékadáskor kerül meghatározásra. A következő kódrészlet végére például `b` lehet `Number` vagy `String` típusú is.

```
if (a > 5) {
    b = 10;
} else {
    b = "kisebb";
}

alert(b);
```

A beépített típusok a következők; a példákban látható, hogyan adható meg kódban egy adott típusú konstans érték:

- `Number`: számok, pl. `42`, `5.5`
- `Boolean`: logikai érték, pl. `true`, `false`
- `String`: karakterekből álló Unicode string, pl. `u"M\u0171egyetemi hallgató"`
- `Array`: tömb, pl. `[42, 5.5, "szoftlab5"]`
- `Object`: szótár és objektum, pl. `{"telefon": "phone"}`, `{1: "Jan", 2: "Feb"}`

Látható, hogy a típusok reprezentációja megegyezik a JSON-nel, amely pont ezért lett népszerű olyan környezetekben, ahol JavaScript kód is érintett a feldolgozásban.

3.2. Keretrendszer és lehetőségek: jQuery és AJAX

A weboldalakba ágyazott JavaScript egyik felhasználói élményt javító lehetősége az oldal adatainak, megjelenítésének frissítése a teljes oldal újra betöltése nélkül. Ennek fejlődése során fontos lépés volt, hogy a JavaScript a háttérben HTTP kéréseket indíthasson az oldal újratöltése nélkül, ezt 2005-ben a *Google Suggest* tette igazán népszerűvé. Mivel eleinte XML volt a felhasznált formátum, a megoldás AJAX (*Asynchronous JavaScript and XML*) néven terjedt el, azonban semmi sem korlátozza a felhasználható formátumok körét, így a JSON is felzárkózik az XML mögé e területen.
Bár AJAX megoldás készíthető közvetlenül a böngésző API-jának felhasználásával, egyszerűbb és stabilabb megoldást jelent egy JavaScript könyvtár használata. A laboron a jQuery használatát mutatjuk be. A jQuery használatához a HTML dokumentum fejlécében (a `<head>` részben) a következő sort kell beszúrni – az `integrity` attribútum kriptográfiai hashfüggvény (256-bites SHA-2) segítségével biztosítja, hogy a külső félnél (*jquery.com*) elhelyezett aktív, így weboldalunk működését befolyásoló tartalmat ne cserélhesse le rosszindulatú személy észrevétlenül:

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js" integrity="sha256-
ZosEbRLbNQzLpnKIkEdrPv7lOy9C27hHQ+Xp8a4MxAQ=" crossorigin="anonymous"></script>
```

AJAX megoldás készítéséhez három jQuery funkciót érdemes ismerni. Az első az oldal betöltődésekor lefutó kód beállítását teszi lehetővé, a `$()` nevű függvény segítségével. Az alábbi kód az oldal betöltődését követően feldob egy „Szoftlab5” üzenetablakot.

```
<script>
    $(function() {
```

```
        alert("Szoftlab5");
    });
</script>
```

A megoldás jól szemlélteti a JavaScript funkcionális jellegét, ugyanis egy függvényt (`function() { ... }`) adtunk át paraméterként egy másik függvénynek (`$()`). A kódot természetesen írhattuk volna közvetlenül a `<script>` tagek közé is, ekkor azonban a kód már akkor lefutott volna, amikor a böngésző az adott `<script>` taghez ér, így például nem hivatkozhat a dokumentum további részében definiált objektumokra, valamint az oldal betöltését is lassítja (a példánál maradva, amíg a felhasználó nem zárja be az üzenetablakot, az oldal betöltése szünetel). A `$()` függvény tehát amennyiben egyetlen paramétereként egy másik függvényt kap, annak végrehajtását akkorra időzíti, mikor a teljes weboldal betöltődött a böngészőbe.

A második hasznos funkcionalitás az ún. szelektor (amivel CSS stíluslapok írásakor már néhányan találkozhattak), amellyel HTML elemek egyszerűen kiválaszthatók későbbi manipuláció céljára. Erre is a `$()` függvény használható, ám ezúttal `String` paraméter kerül neki átadásra. A visszatérési értéken tetszőleges, az elemet érintő művelet elvégezhető, a `html()` metódus például a paramétereként kapott stringre cseréli az elem tartalmát.

Az alábbi példában felhasználjuk a fent megismert technikát is arra, hogy hivatkozhassunk a `mezo` ID-jű elemre úgy, hogy a `<script>` blokk előbb kerül definiálásra. Amennyiben tehát a `$()` függvény `String` típusú paramétert kap, kiértékeli a benne lévő CSS-szerű kifejezést, majd visszaad egy, a kifejezésre illeszkedő elemeket reprezentáló objektumot. A felesleges ciklusok elkerülésére az ezen végrehajtott metódusok minden illeszkedő elemet érintenek, így például ha a lenti kódban a szelektor több elemre is illeszkedne, mindegyik tartalmát a megadott szövegre módosítaná.

```
<script>
    $(function() {
        $("#mezo").html("jQuery a HSZK-ban");
    });
</script>
<div id="mezo">(egyelőre üres)<div>
```

A harmadik szükséges építőelem maga az AJAX hívás, melyet a `$` nevű objektum `ajax` metódusával érhetünk el. Ennek visszatérési értéke nem a kapott érték, hiszen az aszinkron működés lényege épp az, hogy a hívás azonnal visszatér, viszont megadhatunk eseménykezelőket, amelyek meghívásra kerülnek például sikeres AJAX kérés esetén. Az `ajax` metódus egy objektumot vár, ennek gyakran használt paraméterei az alábbi példában kerülnek bemutatásra. Fontos, hogy biztonsági okokból ezzel a megoldással külön engedély nélkül csak a saját weboldalunkon belülre indíthatunk lekéréseket, ezt az ún. *Same Origin Policy* szabályozza.

Az alábbi példában három függvényhívás történik, ezeket egymásba ágyazva definiáltuk, kihasználva a JavaScript lehetőségeit. A külső `$(...)` hívás az első jQuery példában látott módon beállít egy függvényt, hogy akkor fusson le, amikor az oldal betöltődött. Ennek törzse a `$.ajax(...)` hívás, mely aszinkron kérést indít a `/service.json` címre, a választ pedig `json` típusként kérjük, hogy értelmezze. Mivel a függvény azonnal visszatér, definiálunk egy eseménykezelő függvényt, mely egyetlen paramétert vár, ebbe kerül a válaszként kapott adat. Ennek megfelelően a `data` paraméter változót kizárólag a legbelső függvényben törzsében érhetjük el, ez akkor fog lefutni, amikor az AJAX kérésre válasz érkezett.

```
<script>
    $(function() {
        $.ajax({
            url: "/service.json",
            dataType: "json",
            success: function(data) {
                $("#mezo").html(data.lab);
            }
        });
    });
</script>
<div id="mezo">(egyelőre üres)<div>
```

Ha például a `/service.json` válasza `{"lab": "szoftlab5"}`, a `mezo` ID-jű elembe a *szoftlab5* szöveg kerül. Fontos, hogy amennyiben a JSON szintaktikai hibát tartalmaz, a kérés hibajelzés nélkül meghiúsul, így előbb mindig bizonyosodjunk meg róla, hogy a JSON szintaktikailag helyes-e. Az egyszerűbb érthetőség kedvéért az alábbi példa viselkedésében ekvivalens, viszont a függvényeket kifejtve tartalmazza.

```
<script>
    function success_handler(data) {
        $("#mezo").html(data.lab);
    }

    function page_loaded() {
        $.ajax({
            url: "/service.json",
            dataType: "json",
            success: success_handler
        });
    }

    $(page_loaded);
</script>
<div id="mezo">(egyelőre üres)<div>
```

3.3. JavaScript és AJAX hibakeresés

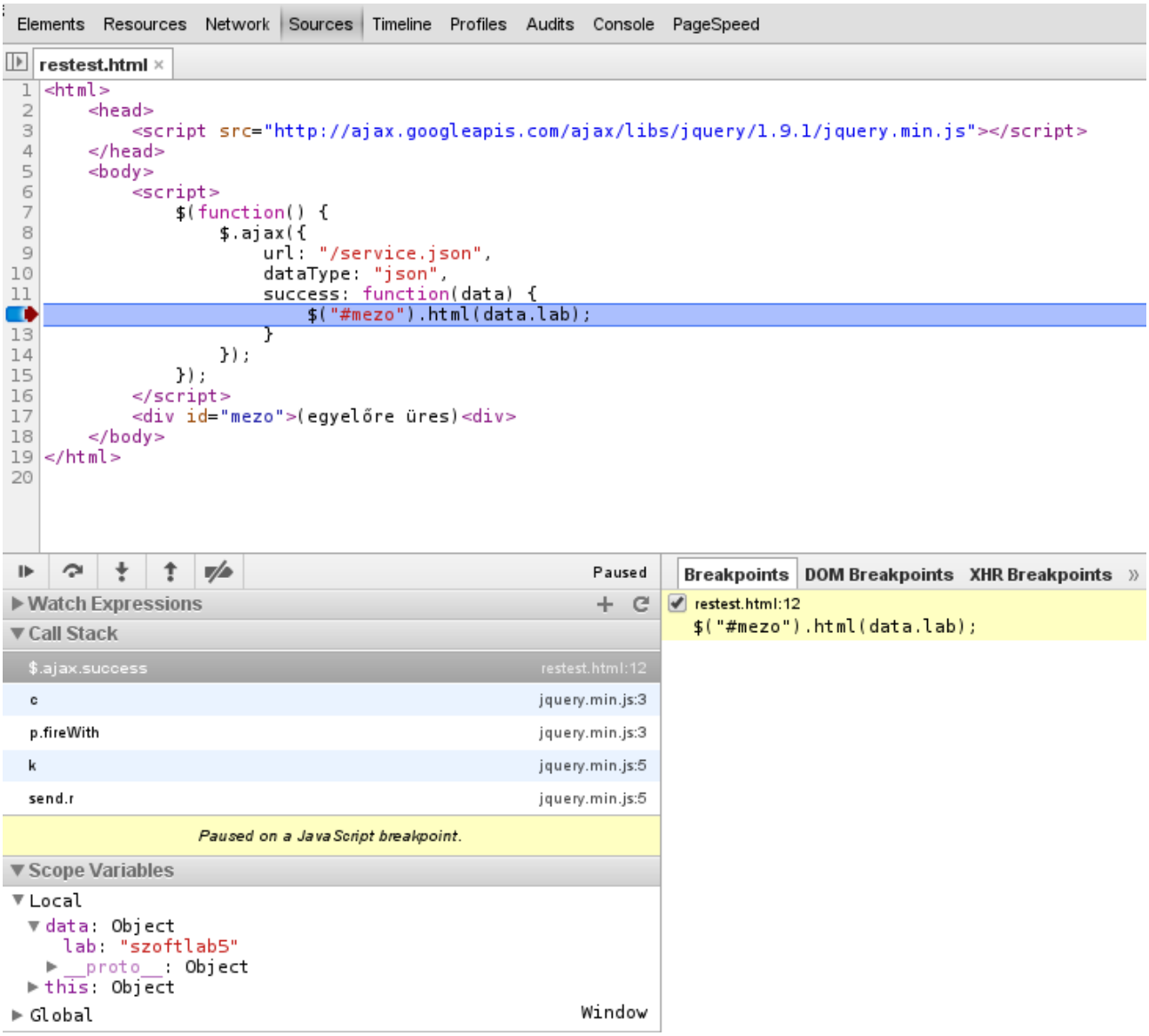
Amennyiben a böngészőben futó szkriptjeink nem a várt eredményt adják, több lehetőség is van a belső működés megismerésére. A legegyszerűbb módszer az `alert()` beépített függvény használata, mely az egyetlen paraméterét felugró ablakban megjeleníti stringként. Ennek megfelelően hasznossága is korlátozott, főleg, ha sok és/vagy strukturált adat megjelenítése szükséges. Jobb módszer a böngészőbe épülő debuggerek használata, ebből a Google Chrome / Chromium debuggerét mutatjuk be, mivel ez régóta a böngészőbe építve érkezik és sok operációs rendszerre elérhető. Hasonló funkcionalitást elérhető természetesen Mozilla Firefox és Microsoft Internet Explorer böngészők újabb verzióiban is és/vagy bővítmények használatával.

A Chrome debuggere az F12 gomb megnyomásával indítható, választható, hogy a böngészőablak aljához vagy jobb oldalához legyen dokkolva, utóbbi szélesvásznú képernyőkön lehet kényelmesebb. A debugger felső részén találhatók a funkcionalitást rejtő fülek, ebből számunkra négy lesz fontos:

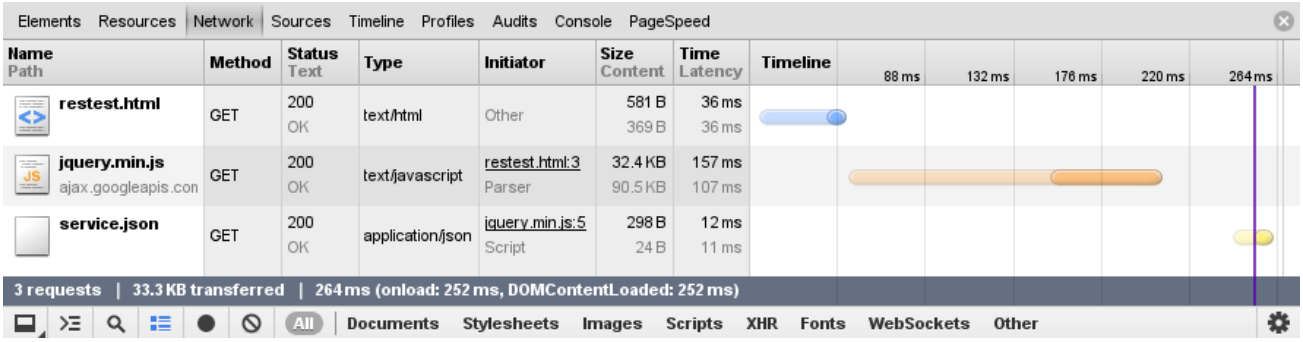
- Elements: az éppen aktuálisan megjelenített HTML oldal szerkezete nézegethető és változtatható benne (elemek törölhetők, módosíthatók, hozzáadhatók) eltérően a forrás megjelenítése ablaktól, mely a betöltéskori állapotot mutatja, amelyet a szervertől kapott

- Network: a hálózatról érkezett válaszokat mutatja, megtekinthető a betöltéshez szükséges idő, illetve a kérések és válaszok tartama és fejlécei is
- Sources: a betöltött szkriptek forrása tekinthető meg itt, töréspontok állíthatók be, szokásos debugger funkcionalitás JavaScripthez
- Console: külön is felnyitható az alsó sávon található *Show console* gombbal, a Python interpreterhez hasonlóan interaktív lehetőséget ad JavaScript kifejezések kiértékelésére

JavaScript hibakeresésnél már a debugger kinyitásakor értesülhetünk arról, ha hibát észlelt a Chrome a futtatáskor, ezt a jobb alsó sarokban egy piros alapon fehér × jelzi, a mellette lévő szám a hibák számát jelenti. Erre kattintva megtekinthető a hibák pontos helye és a hibaüzenet szövege. Amennyiben ebből sem világos a hiba oka, érdemes az összetettebb kifejezéseket akár a Console fülön kipróbálni, akár töréspontot beállítani az adott sorra.
Töréspont (*breakpoint*) a forráskódot megnyitva a bal szélső sorszámra kattintva állítható be és törölhető, ezt követően amint a futás az adott sorra ér, a weboldalon a *Paused in debugger* üzenet látható, az adott sor kiemelésre kerül, és a forrás alatt láthatók a helyi és globális változók értékei és a hívási verem (*call stack*).
A futás ekkor a forráskód alatti gombokkal folytatható (play-szerű gomb), vagy akár lépésenként folytatható, így soronként figyelhető, hogyan viselkedik a kód. Az alábbi ábrán látható például a debugger állapota az AJAX példa `success` eseménykezelőjére állított töréspont esetén. Alul megfigyelhető a `data` paraméter értéke, a JSON-ből dekódolt objektum egyes attribútumai a nyilakra kattintva egyenként kibonthatók.



AJAX hibakeresésnél hasznos még a Network fül, melyben az összes elküldött kérés látható néhány alapadatával együtt. Jobb oldalon megfigyelhető, melyik kérés mennyi időt vett igénybe, így hamar kideríthetők a függőségek (amíg X nem töltődött be, addig Y lekérése el sem indult) és hatékonyabban gyorsítható az oldal betöltődése a lassító elemek felismerésével. Piros és kék függőleges vonal jelzi a betöltött oldal elkészültségét jelző két esemény, alább látható a fenti AJAX példa betöltése után a Network fül állapota. Látszik, hogy a `/service.json` kérés csak az oldal betöltését követően került elküldésre, a 32 kB méretű jQuery pedig összesen 157 ms alatt töltődött be, ebből 107 ms (hálózati) késleltetés.



Az ablak alján lehet szűrni az egyes betöltött elemekre típusuk szerint, alapértelmezés szerint minden megjelenik (*All*), de például XHR-t kiválasztva lehet kizárólag AJAX-szal lekért elemekre szűrni (*XmlHttpRequest*). AJAX hibakeresésnél az ablak több szempontból lehet hatékony segítség: egyrészt megismerhető, hogy egyáltalán lekérésre került-e a kért erőforrás, a lekérés sikeres volt-e, és ha igen, milyen tartalom érkezett meg. Utóbbit az erőforrás nevére (bal oszlop) kattintva lehet megnézni, ekkor a többi oszlop helyén négyfüles ablak jelenik meg.

- Headers: a HTTP kérés és válasz fejlécei tekinthetők meg
- Preview: a válasz feldolgozott formában tekinthető meg, JSON, XML, HTML esetében például faszerkezetben
- Response: a válasz feldolgozatlan, nyers állapotban tekinthető meg
- Timing: az adott erőforrásra vonatkozó várakozási idő

IV. Függelék: Webes és adatbázis biztonsági kérdések

Szerzők: Balázs Zoltán, Paksy Patrik, Sallai Tamás, Veres-Szentkirályi András

1. Bevezetés

A dinamikus weboldalak és a mögöttük lévő adatbázisokban tárolt adatok manapság kiemelt támadási célpontnak számítanak, ezért az adatok védelme kritikus feladat. Sok esetben az adatok bizalmasságának és sértetlenségének biztosítására nem fektetnek nagy hangsúlyt, pedig néhány egyszerű módszerrel már igencsak megnehezíthetjük a támadók munkáját.

A segédlet célja, hogy rámutassunk azokra a főbb problémákra, biztonsági kérdésekre és a hozzájuk tartozó védekezési lehetőségekre, melyekkel dinamikus weboldalak készítése során adatbázis adminisztrátorként vagy fejlesztőként találkozhatunk. Elsőként megemlítünk néhány fontos szempontot, melyeket az adatbázis konfigurálásánál érdemes betartanunk, majd pedig bevezető jelleggel tárgyalunk három gyakori alkalmazás sebezhetőséget, az SQL injection, XSS, és CSRF támadásokat. A támadási felületet elsősorban a bemeneti interfészek biztosítják, így kiemelten foglalkozunk az input ellenőrzésének kérdésével. Fejlesztőként elengedhetetlen, hogy ezekkel tisztában legyünk, és megtegyük a legalapvetőbb lépéseket ahhoz, hogy bizalmas adatok ne kerüljenek illetéktelen kézbe, vagy jogosulatlanul ne lehessen elvégezni bizonyos kéréseket. Minden esetben figyelembe kell vennünk azt a tényt is, hogy a védelmi intézkedéseink mértéke akkor megfelelő, ha arányban van a védendő adatok értékével, de az itt ismertetett módszerek egy jó webes alkalmazásból gyakorlatilag ma már nem hiányozhatnak.

A dokumentum elsősorban áttekintést próbál adni az említett témakörökről. A bővebb információk után érdeklődőknek a segédlet végén található hivatkozások nyújtanak további segítséget.

2. Konfigurációs védelmi szempontok

2.1. Adatbázis jogosultságok

Az egyik legfontosabb biztonsági alapelv, hogy minden program vagy felhasználó a szükséges lehető legkevesebb joggal rendelkezzen. Ez egyrészt igaz az adatbázis-kezelő futtatására (vagyis ne rootként fusson az adatbázis-kezelő), másrészt igaz a webszerver és adatbázis-kezelő kapcsolatára. Bár a fejlesztés során mindig kényelmes, ha nem ütközünk jogosultsági hibába, azonban az éles működés során már tilos a sémafelhasználó vagy adatbázis-adminisztrátori jogokkal csatlakozni a webszerverről az adatbázishoz. Mindig külön felhasználót (ún. proxy felhasználó) hozzunk létre erre a feladatra, amely csak és kizárólag a szükséges funkciókhoz elegendőek. Így például, ha egy támadó a felhasználó nevében

megkerüli a webalkalmazásba épített biztonsági logikát, de ezen felhasználónak csak lekérdezési joga van, akkor a támadó nem lesz képes a különböző adatmódosító utasítások (DML) végrehajtására.

2.2. Puffer túlcsordulás

Mivel az Oracle adatbázis-kezelő core funkcióit C nyelvben írták, ezért szinte természetes, hogy puffer-túlcsordulás alapú hibák is előfordulnak benne. Ezeket a hibákat az Oracle adatbázis biztonsági frissítéseivel lehet javítani.

2.3. Többszintű védekezés – layered security, defense in depth

Gyakori téves feltételezés, hogy mivel az adatbázist tűzfallal szeparálják az internettől, ezért az adatbázis védelmére nem szükséges kellő hangsúlyt helyezni, így például alapértelmezett felhasználói jelszavakat hagynak bent, melyek könnyű hozzáférést adnak a támadók kezébe.
Másik tipikus hiba, hogy az alkalmazások kompatibilitása miatt (vagy egyszerűen lustaságból) nem frissítik biztonsági frissítésekkel az adatbázis-kezelőket vagy akár olyan csomagok is telepítésre kerülnek, amelyek használatát semmi sem indokolja. Ezek olyan sérülékenységet tartalmazhatnak, amelyek kihasználásával jogosulatlan kódvégrehajtás történhet.

3. Alkalmazások biztonsági kérdései

3.1. SQL kódbeillesztés (SQL injection)

Áttekintés

A legfontosabb biztonságot növelő módszer a már szóba került input validáció. Bármely interfészen érkező adatot megbízhatatlanként kell kezelni mindaddig, amíg bizonyos ellenőrzéseken (pl. tartomány illetve értékkészlet vizsgálat, reguláris kifejezések) sikeresen át nem esett az adat. Biztonsági és teljesítmény szempontból is rossz megközelítés, amikor megpróbáljuk felsorolni a rossz, hibás bemeneteket, és ezeket blokkoljuk (black list alapú megközelítés), hiszen az ilyen listák sosem teljesek, és folyamatosan frissíteni kell.
Az ellenőrzés elmulasztása akár egy bemeneti paraméteren is, komoly biztonsági következményekkel járhat. Egy támadónak ilyenkor lehetősége nyílik arra, hogy a fejlesztő által megírt SQL kódot tetszőleges kóddal kiegészítse, és azt a felhasználó jogosultságaival futtassa. Így a támadónak lehetősége nyílhat a felhasználói bejelentkezés megkerülésére, a teljes adatbázis lemásolására, új adatbázis-adminisztrátor létrehozására, operációs rendszer parancsok végrehajtására vagy tetszőleges kód feltöltésére – egyetlen nem ellenőrzött paraméter miatt. Adatbázis-kezelő, illetve szerver-oldali programozási nyelvfüggő, hogy az SQL kódbeillesztéses támadás során csak a már elkezdett utasítást tudjuk folytatni, vagy lehetőségünk van több parancs egymás utáni végrehajtására is – egyetlen webszerver kéréssel. Az utóbbit kód stackingnek nevezzük, melyet az Oracle nem támogat, így csak azt az utasítást tudjuk kiegészíteni, amelyiket a program elkezdte.
Tipikus tévhit, hogy ha egy paraméter legördülő menü, rádiógomb formájában szerepel a weboldalon, vagy ha kliensoldali JavaScript ellenőrzést végeztünk a beviteli mezőn, akkor azon értékek már megbízhatónak tekinthetőek. A szerveroldali input validáció kötelező eleme minden webes alkalmazásnak, míg a kliensoldali JavaScript ellenőrzés elsősorban nem biztonsági szerepet játszik, hanem a felesleges űrlapküldések elkerülésére, ezzel együtt a szerver terhelésének csökkentésére valamint a felhasználók kényelmére szolgál. Az input

validáción kívül kiemelten fontos az output validáció[1] is, az ehhez kapcsolódó támadásokról a későbbi fejezetekben is olvashatunk.

Hibakezelés

Szintén alapelv, hogy éles alkalmazás esetén ne adjunk vissza olyan hibaüzenetet a felhasználónak, amiből következhet az adatbázis vagy az alkalmazás felépítésére. Például egy éles alkalmazásban egy hibás SQL lekérdezés esetén a helyes megjelenítendő hibaüzenet: „Belső hiba történt", míg a kerülendő hibaüzenet típusok: „ORA-01790: expression must have same datatype as corresponding expression". Az utóbbi típusú üzenetek sokat segítenek a támadónak abban, hogy az SQL kódbeillesztéshez érvényes SQL kódot készítsen.
PHP-ban például az error_reporting(0); paranccsal tudjuk forráskódból kikapcsolni a hibaüzeneteket, majd ha a lekérdezés visszatérési értéke üres, akkor lehet egy általános hibát megjeleníteni. Ha nem kapcsolnánk ki minden hibaüzenetet, PHP-ban van lehetőség egy adott parancs futtatásakor keletkező hibának elnyomására is, ehhez a parancs elé az „@" jelet kell beírni.

SQL kódbeillesztés példák

A következő két példa szemléltetésként szolgál az SQL kódbeillesztés alapú támadásokhoz. Természetesen rengeteg más típus is létezik, most két egyszerű módszert mutatunk be.
Példa I. (mindig igaz WHERE feltétel konstruálása)
Tekintsünk egy olyan weboldalt, ahol nem kezelték megfelelően a kódbeillesztési támadásokat. Az oldalon egy bejelentkezési űrlap található, melynek elküldésekor az alábbi SQL kód fut le. Ha találunk az adatbázisban a megadott paraméterekkel felhasználót, tehát a lekérdezés nem nulla sorral tér vissza, akkor sikeres volt a bejelentkezés.

```
SELECT * FROM felhasznalok
WHERE nev = '$nameField' AND jelszo = '$pwdField';
```

A kódbeillesztés fontos lépése, hogy a megfelelő szintaxis megtartásával biztosítsuk a kód lefutását. Ezért figyelnünk kell arra, hogy a megkezdett aposztrófokat megfelelően zárjuk le, a felesleges SQL kódrészeket pedig kommentezzük ki. Az utasítás hátralévő részének kikommentezésére a „--" jel való.

Beillesztett SQL kódrészlet: Teszt' OR 1 = 1; --

```
SELECT * FROM felhasznalok
WHERE nev = 'Teszt' OR 1 = 1; --' AND jelszo = 'TesztJelszo';
```

A beillesztett kóddal a következő lekérdezés fog lefutni, így a bejelentkezés sikeres lesz:

```
SELECT * FROM felhasznalok
WHERE nev = 'Teszt' OR 1 = 1;
```

Példa II. (UNION)
Kódbeillesztésnél gyakran használt módszer, amikor a lekérdezéshez az UNION művelet segítségével kapcsolunk hozzá egy másik lekérdezést, ezáltal kinyerhetjük egy tetszőleges tábla tartalmát. A sikeres támadáshoz az UNION két tulajdonságát kell felhasználni: ugyanannyi mezőt kell megadnunk a SELECT záradékban mindkét lekérdezésnél, és a mezőknek páronként azonos típusúnak kell lenniük (a *null* érték minden típussal azonos).
Tegyük fel, hogy egy oldalon az URL-ben megadott ID-val rendelkező hírt jelenítjük meg az alábbi lekérdezéssel:

[1] Output validáció: https://www.owasp.org/index.php/Output_Validation

```
SELECT cim, szoveg FROM hirek
WHERE id = $urlParam;
```

Célunk pedig a jelszavak listájának megszerzése a *felhasznalok* táblából:

```
SELECT cim, szoveg FROM hirek
WHERE id = 1 UNION SELECT null, jelszo FROM felhasznalok;
```

Beillesztett kódrészlet: 1 UNION SELECT null,jelszo FROM felhasznalok

Védekezés

A legegyszerűbb védekezési módszer, ha escape-eljük a speciális karaktereket, így az SQL injection megvalósításához szükséges aposztróf vagy kötőjel nem speciális jelentéssel bíró szimbólumként értelmeződik. Az escape-elés során minden speciális karakter elé egy „\” jel kerül. Mivel ezek a karakterek többféle formában megadhatók, így önmagában nem mindig nyújt megfelelő védelmet.
Adatkötés használata esetén a lekérdezést paraméteresen, helyőrzőkkel adjuk meg, az adatbázis-kezelő pedig a paraméterek helyét kihagyva készíti el a végrehajtási tervet. Így a behelyettesítés során már nem változhat meg a lekérdezés szerkezete, az esetleges rosszindulatú kód paraméterként fog szerepelni a lekérdezésben, nem pedig SQL kódként. A PHP változókat az oci_bind_by_name() metódus segítségével köthetjük a helyőrzőkhöz.
Példa (adatkötés):

```
$sqlQuery = "SELECT * FROM table WHERE id = :id";
$statement = oci_parse($sqlQuery);

oci_bind_by_name($statement, ":id", 12345);
oci_execute($statement);
```

Ha nincs lehetőség paraméterezett SQL kódra, akkor használjunk tárolt eljárásokat. Tárolt eljárásnál kerüljük a dinamikus SQL használatát, vagy használjunk adatbázisoldali input-validációt (pl. DBMS_ASSERT Oracle esetében).

3.2. XSS

Áttekintés

A Cross-Site Scripting (XSS) támadás során a támadó JavaScript kódot injektál az áldozat gépén a megtámadott weboldalba. Ez azért lesz veszélyes, mert ez a script az oldalon majdnem mindenhez hozzáfér, tudja manipulálni, hogy mi jelenjen meg és kifele kapcsolatokat is nyithat. Általában a weblapok a saját domainjükből jövő tartalmakban megbíznak, más domainből jövőekben pedig nem (ez a Same Origin Policy: ami onnan jött ahonnan az oldal, az szabad kezet kap mindenhez, ami máshonnan, az csak nagyon kevés dolgot tehet meg).
Általában feltételezzük, hogy minden támadáshoz az áldozat rákattint egy linkre, ami a támadótól származik.

A támadás megvalósítása

A támadás alapja, hogy a támadó által megírt JavaScript kód valamilyen formában az áldozathoz kerüljön a weblap által. Ehhez képzeljünk el egy esetet, amikor a weblapon lehet keresni, és a kereső (mint általában szokás) a keresési eredmények fölé kiírja a keresett kifejezést. Ezt szerveroldalon egy egyszerű String összefűzéssel éri el, tehát ami a klienstől jött, az egy az egyben kikerül a kimenetre is.
Például:
Eset I., a megfelelő működés:

- URL: localhost/?search=kif
- HTML forrás: Keresés:kif
- Eredmény: **Keresés: kif**

Eset II., a támadás:

- URL: localhost/?search=kif<script>alert('xss')</script>
- HTML forrás: Keresés:kif<script>alert('xss')</script>
- Eredmény: **Keresés: kif**, valamint felugrik egy popup az xss szöveggel, tehát sikeresen kódot injektáltunk a weboldalba.

Tehát ha az áldozat rákattint a támadó által küldött linkre, akkor a támadó által megírt JavaScript lefut.

Perzisztens és reflektív támadás

Az előző pontban bemutatott megoldás az ún. reflektív támadás, mivel kizárólag a kérésben szerepel a támadó kód. A másik változat a perzisztens XSS, amely úgy futtat kódot az áldozatnál, hogy azt a webszerver tárolja. Ez általában azért lehet veszélyesebb, mert a megtámadott weblap összes látogatóját érintheti.
A perzisztens eset is arra épül, hogy a felhasználótól jövő bemenet módosítás nélkül kikerül az oldalra, csak ebben az esetben valamilyen eltárolt adat által. Ez lehet pl. egy komment egy blogbejegyzéshez, ahova regisztrált vagy anonim felhasználó szabad szöveget írhat.
Például: A weboldalon van egy kommentelési lehetőség, és ezek a bejegyzés alatt megjelennek. Ezután a támadó véleménye ez a cikkről:

```
Nagyon jó, csak így tovább!<script>alert('xss')</script>
```

Ezután bárki megnézi a bejegyzést, a Javascript kód le fog futni nála.

Veszélyesség

Persze egy alert feldobása miatt még nem lenne olyan veszélyes, azonban a Javascript nagyfokú szabadsága miatt változatos (és nem feltűnő!) módon lehet kihasználni az XSS sebezhetőségeket.
A támadó legfontosabb célja a SessionID megszerzése lehet, mivel ha azt sikerül megtudnia, akkor egyszerűen meg tudja személyesíteni az áldozatot. A támadás vázlata az alábbi lehet:

- Az áldozat rákattint a linkre
- Javascript kiolvassa a SessionID-t a sütik közül
- Nyit egy kérést egy, a támadó által irányított gépre, amiben elküldi a SessionID-t
- A támadó a saját SessionID-jét átírja az imént megszerzettre
- Megnyitja a támadott weboldalt, és az áldozat nevében van bejelentkezve

Mivel a DOM-ot és a futtatott Javascripteket is tetszés szerint tudja módosítani, ezért nem nehéz a bejelentkező formot átírnia olyanra, hogy mellékesen a beírt adatokat még a támadó által irányított gépre is elküldje.
Mivel a megjelenítést is tudja módosítani, ezért megváltoztathatja a weboldalon közölt tartalmakat, pl. egy tőzsdeoldalon nem valós árfolyamadatokat mutat, ezzel befolyásolhatja a piaci folyamatokat.
Különösen veszélyes, hogy egy rejtett (1×1-es méretű, keret nélküli) IFrame-be (Inline Frame) be tudja tölteni a weboldalt, és ahhoz szabadon hozzá tud férni. Ez azért lehetséges, mert ugyanarról a domainről lett betöltve, ezért a böngésző megbízhatónak tekinti a kódot.

Védekezés

A legfontosabb védekezés, hogy minden felhasználói bevitt adatot gonosznak tekintünk és validálunk, valamint minden dinamikus adatot amit megjelenítünk escape-lünk. Ez utóbbi a speciális karakterek lecserélését jelenti, így a böngésző biztosan nem fogja azokat végrehajtani.
Az előbbi pedig az esetek többségében whitelistinget jelent, tehát csak azokat a bemeneteket fogadjuk el, ami illeszkedik egy bizonyos mintára. Pl. egy keresőmezőbe csak alfanumerikus karaktereket fogadunk el.
A SessionID ellopása ellen lehetőség van sütiket HTTPOnly-nak megjelölni, így a JavaScript nem fér hozzá, mivel erre általában nincs is szükség. 14-es verziótól felfele a Chrome figyeli, hogy ha futtatható kód van a kérésben, akkor letiltja a JavaScript futtatást, ez megnehezíti a reflektív támadást.

3.3. CSRF

Áttekintés

A Cross-Site Request Forgery (CSRF) során a támadó az áldozat nevében küld kéréseket a megtámadott oldalra. Ennek során a támadó paraméterezi fel a kérést, az áldozat küldi el, az oldal pedig végrehajtja. Ez a web állapotmentessége miatt tud működni, ugyanis nincs olyan információ, hogy a böngészőn belül melyik tabról lett megnyitva a weboldal, valamint a tab becsukásakor a felhasználó általában nem lesz kijelentkeztetve.

A támadás megvalósítása

Képzeljünk el egy üzenetküldő oldalt, ahol a bejelentkezett felhasználó egyetlen GET kéréssel tud üzenetet küldeni egy ismerősének. Amennyiben a támadó ismeri a pontos URL-t és a paramétereket (ezt általában ismertnek tekintjük), akkor össze tud állítani egy olyan oldalt, ahonnan a látogató böngészője elküld egy üzenetet a megtámadott oldalon.
Ehhez a támadó egy olyan oldalt állít össze, amelyben van egy ilyen tag:

```
<img src="http://localhost/sendmessage?to=bela&amp;msg=szia"/>
```

Amikor az áldozat megnyitja ezt az oldalt (figyeljük meg, hogy nem kell azonos domain alatt lennie, tehát a támadó ezt az oldalt minden esetben össze tudja állítani), akkor a böngésző megpróbálja letölteni a képet, ami egy GET kérés lesz. Ezzel együtt elküldi a SessionID-jét is, és ha történetesen be is van jelentkezve a támadott oldalra, akkor az üzenete el lesz küldve.

Védekezési lehetőségek

Ha GET helyett csak POST-tal lehet műveletet végezni, az nem teszi biztonságosabbá az alkalmazást, ugyanis POST kérést egyszerűen össze lehet rakni JavaScripttel. HTTP Referrer ellenőrzése ugyancsak nem jó, mert egyes böngészők ezt nem küldik el.
Általában a védekezés arra épül, hogy a form-ba kiküldünk a kliensnek egy véletlen számot, és figyeljük, hogy az megfelelően visszajön a form visszaérkezésekor. Mivel a támadónak alapvetően nincs lehetősége a form tartalmát megismerni (csak más támadással együtt, pl. az XSS használható erre), akkor a visszajövő véletlen nem fog egyezni, ezért elutasítjuk a kérést.
A védekezés az alábbi lépésekből áll:

- A form generálásakor készítünk egy véletlen számot, egy hidden fieldben megjelenítjük (`<input type="hidden" name="csrf" value="véletlenszám">`), valamint a session-ben eltároljuk

- A form feldolgozásakor kiolvassuk a session-ből az eltárolt értéket, és ha nem egyezik meg a visszakapottal, akkor hibával megszakítjuk a feldolgozást

Ennek megvan az a hátránya, hogy minden form generálása felülírja a sessionben elmentett értéket, emiatt a formokat ki kell töltenie a felhasználónak mielőtt másikat nyit meg. Ez több tabos böngészésnél kényelmetlen lehet. Erre félmegoldás, ha minden formnak van egyedi azonosítója, így csak ugyanazon űrlap megnyitásakor íródik felül a véletlen.

4. További megfontolások

Jelszavak tárolása adatbázisokban

Amennyiben a webes alkalmazásunk saját felhasználó-jelszó adatbázist használ, a jelszavakat tilos közvetlenül az adatbázisba illeszteni. Mindig használjunk egy egyirányú hash függvényt (pl. SHA-1 vagy SHA-256) a jelszón, illetve a hash-elés előtt a jelszó elé vagy után illesszünk egy minimum 10 karakter hosszúságú, felhasználónként különböző álvéletlen értéket, ún. saltot. Pszeudo-kóddal kifejezve:

```
stored_password := SHA-1(clear_test_password ||
  random_salt_different_for_every_user)
for (i=0; i<10000; i++) {
  stored_password := SHA-1(stored_password);
}
```

Ez a salt érték megakadályozza, hogy a hash értékekre ún. előszámított szivárvány-táblák segítségével percek alatt legyen feltörhető a jelszó. Az Oracle 11g-ben az adatbázis felhasználók jelszava három módon lehet letárolva: Vagy az új, 11g kompatibilis, SHA-1 alapú, a felhasználói nevet saltként használó jelszó hashként van letárolva, vagy a régi, saját hash algoritmust használó, nem case sensitive jelszó hash – szintén a felhasználói nevet saltként használva, vagy mindkettő hash. Érdemes megfigyelni, mivel nem véletlen a jelszóhoz tartozó salt, ezért a felhasználókra külön szivárvány-táblát lehet építeni, pl. a SYS v. SYSTEM adminisztátor felhasználó jelszó hash-eire az internetről letölthetőek ilyen szivárvány táblák.

5. Védekezési módszerek összefoglalása

A fentebb ismertetett támadások ellen tehát tartsuk szem előtt a következőket:

- az adatbázis-kezelőt futtassuk csökkentett jogosultságú felhasználó nevében
- az adatbázis-kapcsolathoz használt felhasználó csökkentett jogokkal rendelkezzen az adatbázis sémákban, ne legyen a séma tulajdonosa
- ne használjunk alapértelmezett, vagy egyszerű jelszavakat az adatbázisban, a nem szükséges felhasználókat tiltsuk le
- végezzünk szigorú input és output validációt
- kezeljük a speciális karaktereket escape-eléssel
- használjunk adatkötést avagy ún. paraméterezett SQL kódot (pl. oci_bind_by_name)
- ha nincs lehetőség paraméterezett SQL kódra, akkor használjunk tárolt eljárásokat
- a jelszavakat salttal egészítsük ki és hash-elve tároljuk el

6. Referencia, hasznos linkek, érdekességek

A webes támadások és az ismert védekezések legteljesebb gyűjtőhelye az OWASP (Open Web Application Security Project) www.owasp.org-on elérhető weblapja.

- Cross-Site Scripting
  https://www.owasp.org/index.php/XSS
- Cross-Site Request Forgery
  https://www.owasp.org/index.php/CSRF

- SQL injection cheat sheet
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Useful Oracle security tools
  http://www.petefinnigan.com/tools.htm
- Decrypt any password from password hash and successful network login packet capture (Oracle11g)
  http://www.soonerorlater.hu/index.khtml?article_id=512
- Red database security
  http://www.red-database-security.com/
- David Litchfield
  http://www.davidlitchfield.com/security.htm
- OWASP SQL Injection
  http://www.owasp.org/index.php/SQL_Injection
- NGS Software
  http://www.ngssoftware.com/media-room/WhitePapers.aspx
- XKCD
  http://xkcd.com/327/
- SQLMAP
  http://sqlmap.sourceforge.net/
- Szivárvány táblák működése
  http://en.wikipedia.org/wiki/Rainbow_table
- Advanced out-of-band SQL injection
  http://www.red-database-security.com/wp/confidence2009.pdf