IT biztonság

(VIHIAC07)

Laboratóriumi gyakorlat

Webes software-ek biztonsága labor

Szerző:
Gazdag András

2025. március 6.

Tartalomjegyzék

1. Oktatási célok

A labor gyakorlat során a web fejlesztésben a kliens és szerver oldalon felmerülő problémákat és a különböző beszúrásos támadások veszélyeit fogja megismerni, különös tekintettel a web alkalmazásoknál felmerülő problémákra. A cél, hogy tapasztalatot szerezzen egy való élethez közeli alkalmazáson keresztül egy támadás lehetséges veszélyéről és hatásáról, és megértse a veszélyét az SQL és JavaScript injection (Cross-Site Scripting (XSS)) támadásoknak, mely támadások leggyakrabban az adatbázis kommunikáció során merülnek fel, így a labor alatt is ennek a funkcionalitásnak a megtámadása a cél. A munka során a Juice-Shop alkalmazással szemben kell támadásokat végrehajtani.

2. Háttéranyag

2.1. JavaScript

A JavaScript (JS) nyelv (MDN: What is JavaScript?), egy interpretált, valós időben forduló programozási nyelv. Kliens oldali alkalmazások fejlesztésére használják első sorban, azonban számos más (nem klasszikusan böngészőben futó) területen is előfordul, úgy mint Node.js alkalmazás fejlesztésnél, az Apache CouchDB használatakor, vagy az Adobe Acrobat környezetben. JavaScript egy prototípus alapú dinamikus programozási nyelv, amely támogatja az objektum orientált fejlesztést, az imperatív és a deklaratív programozást is.

A nyelv alap képességeinek segítségével végre lehet vele hajtani a tipikus programozási feladatokat, úgy mint:

- értékek tárolása változókban,
- szövegen végzett műveletek,
- események hatására történő kódfuttatás.

A nyelv által biztosított lehetőségeken túl azonban a technológia igazi erejét az adja, amikor kliens oldali nyelvként, a böngészők által biztosított API-val (Application Programming Interfaces) együtt használjuk. Ezek a programozási interfészek adnak hozzáférést a böngésző belső képességeihez és adataihoz, aminek a segítségével teljes értékű alkalmazássá lehet változtatni egy statikus weboldalt.

Böngésző API: a böngészők által nyújtott kapcsolódási pont, amelyen keresztül lehetségessé válik az adathozzáférés és kommunikáció a számítógép lokális erőforrásaival. Néhány tipikus használati eset:

- A DOM (Document Object Model) API lehetővé teszi, hogy hozzáférjen a HTML és CSS elemekhez. Ezen keresztül lehet új elemeket létrehozni, meglévőket módosítani vagy törölni.
- A Geolocation API a pozíció információkhoz ad hozzáférési lehetőséget. Ez alapján képes például a Google Maps is megjeleníteni a felhasználó helyzetét.
- A Canvas és WebGL API segítségével lehet 2D és 3D grafikákat megjeleníteni.
- Audio és Video APIs úgy mint a HTMLMediaElement vagy WebRTC teszi lehetővé a multimédia tartalmak kezelését és megjelenítését.

Third party API (külső API-k): ezek a programozási felületek nem képezik részét a böngésző által nyújtott szolgáltatásoknak. Tipikusan külső szolgáltatásokhoz adnak hozzáférést, amelyek segítségével új funkciókat tudunk összekapcsolni az alkalmazásunkkal. Néhány példa erre:

- Az X (volt Twitter) API, amely lehetővé teszi például egy X fiók postjainak a megjelenítését egy weboldalon.
- A Google Maps API vagy az OpenStreetMap API segítségével egy külső térkép funkciót ágyazhatunk be egy weboldalba.

A böngészőbe épített biztonsági szabályok garantálják, hogy minden weboldal (böngésző tab) egy önálló külön környezetben működjön. Ezek a környezetek (execution environments) különítik el egymástól az egyes weboldalak végrehajtandó kódját, ami egy kritikus feladat biztonság szempontjából.

A labor foglalkozásra felkészülésképpen, olvassa el a JavaScript nyelv alapjait tartalmazó leírást a Mozilla oldalán.

2.2. Cross-Site Scripting (XSS)

A Cross-Site Scripting (XSS) egy kódbeszúrásos támadás. Egy ilyen támadás során a támadó kártékony JavaScript kódot juttat egy legitim, azonban sérülékeny alkalmazásba. A sérülékenységek, amelyek kihasználásával ez lehetségessé válik nagyon széleskörűek, a legtöbb változó, amely tartalmát képes a támadó kontrollálni és szerepet játszik a weboldal megjelenésében potenciálisan veszélyes lehet.

Egy támadó felhasználhat egy XSS támadást, hogy kártékony kódot juttasson az áldozat böngészőjébe. A böngészőnek nincs esélye ezt a kódot megkülönböztetni az alkalmazás legitim kódbázisától, így azzal azonos módon a támadó kódot is végrehajtja. A végrehajtás során a legitim kódhoz hasonlóan a támadó is hozzá tud férni a sütikhez, más session információhoz, vagy bármilyen érzékeny adathoz a weboldalon belül. A kód akár a weboldal HTML tartalmát is módosíthatja, ezzel további károkat okozva az áldozatnak. Az XSS támadásoknak 3 fajtáját különböztetjük meg.

2.2.1. Stored XSS (más néven Persistent XSS)

Stored XSS támadásról (eltárolt XSS) akkor beszélünk, ha a kártékony kód a sérülékeny alkalmazásban (tipikusan egy adatbázisban) eltárolásra kerül. Erre tipikus példaként szolgálhat egy fórum alkalmazás, vagy egy kommentelési lehetőség. Fontos azonban szem előtt tartani, hogy maga a támadás ekkor is kliens oldalon történik, azonban a kártékony kód ebben az esetben egy adatbázison keresztül jut el a böngészőbe. A támadást így megelőzi egy másik (időben független) lépés, amely során a támadó a kártékony kódot bejuttatja az adatbázisba. Az előző példákból kiindulva ez úgy valósítható meg, ha közzétesz egy új fórum bejegyzést, vagy ír egy kommentet. A fejlesztő által elkövetett hiba ilyenkor az, hogy egy felhasználótól származó adatot sosem szabad közvetlenül, ellenőrzés nélkül eltárolni (input validáció szükséges), továbbá, adatbázisból kiolvasott adatot, sosem szabad közvetlenül a HTML kódba beleírni (escaping szükséges).

A modern HTML5 alapú technológiák elterjedésével elképzelhető ennek a támadásnak olyan verziója is már, ahol a böngésző nyújtotta adattárolási képességeket használja ki a támadó, így a kártékony kód nem a szerver oldali adatbázisba kerül, hanem a kliens oldalon marad.

2.2.2. Reflected XSS (más néven Non-Persistent XSS)

Reflected XSS esetén a felhasználótól kapott adatot nem tárolja el a szerver oldal, hanem azt közvetlenül felhasználja a válasz elkészítése során. Tipikus példa erre egy keresést megvalósító oldal, ahol az oldal tetején általában megjelenik az a bemenet, amire a felhasználó rákeresett. Egy ilyen funkcionalitás esetén, ha a szerver oldalra érkezett adat ellenőrzés nélkül (input validáció) bekerül a válasz a HTML-be, akkor fenn áll az XSS támadás lehetősége.

Ez a támadás is elképzelhető úgy, hogy a kártékony kód nem jut el a szerver oldalra, ezt azonban egy új kategóriának tekintjük a megvalósítás technikai tulajdonságai miatt: ez a DOM alapú XSS.

2.2.3. DOM Based XSS (más néven DOM alapú XSS)

DOM alapú XSS támadásnak azt tekintjük, amikor a támadás teljes folyamata a kliens oldalon valósul meg, és a sérülékenység a DOM kliens oldali manipulálásában található. Modern web alkalmazások (például az SPA-k: Single-page application vs. multiple-page application) a legtöbb esetben a kliens oldalon renderelik ki a végső weboldalt. A legtöbb keretrendszer rendelkezik alapvető védelemmel, azonban a használatuk során lehetséges hibát elkövetni. Azok az alkalmazások, amelyek pedig keretrendszer nélkül valósítják meg ezt a funkcionalitást, tipikus célpontjai lehetnek egy ilyen támadásnak. Egy gyakran elforduló eset, ha a renderelés során az aktuális URL (vagy egy része) felhasználásra kerül. Ha ilyen esetben, ez az adat közvetlenül megjelenik a HTML kódban, akkor ez kihasználható egy DOM alapú XSS támadás részeként.

2.3. XSS támadás megakadályozása

A következő ajánlások célja, hogy az összes korábban említett XSS támadást megállítsák egy alkalmazásban (Cross Site Scripting Prevention Cheat Sheet). Ezek alkalmazása esetén nincs lehetőség külső forrásból származó bármilyen tetszőleges HTML tartalom megjelenítésére, azonban ezen korlátozás mellett elérhető biztonság egy általában elfogadható mértékű kompromisszumnak számít. Egy átlagos szoftver fejlesztése során valószínűleg nincs szüksége az összes itt megemlített szabály betartására. A legtöbb esetben az első két pont elegendő lehet, azonban érdemes minden projekt során megvizsgálni a felmerülő veszélyeket, és az alapján dönteni.

1. HTML escape-elést kell alkalmazni minden nem megbízható tartalom HTML elembe történő beágyazása előtt.
2. Attribútum escape-elést kell alkalmazni minden attribútum érték megadása előtt.
3. Tetszőleges helyre sosem szabad nem megbízható tartalmat beágyazni. (Csak az erre szándékosan meghatározott helyre.)

4. JavaScript escape-elést kell alkalmazni minden nem megbízható adat esetén, amit értékül adunk egy JavaScript változónak.
5. HTML escape-elést kell alkalmazni minden nem megbízható JSON értékre, és adatot csak a JSON.parse metódus segítségével érdemes beolvasni.
6. CSS escape-elést kell alkalmazni, és szigorú érték ellenőrzést végrehajtani, mielőtt stílus leírást illesztünk be egy HTML oldalba.
7. URL escape-elést kell alkalmazni mielőtt egy értéket felhasználunk URL paraméterként.
8. A HTML tartalmat mindig kifejezetten erre a célra kifejlesztett könyvtár segítségével ellenőrizzük a felhasználás előtt.
9. Kerüljük a JavaScript URLek használatát.

2.4. Kódbeszúrás

Egy kódbeszúrásos támadás során a támadó kártékony kódot tud bejuttatni egy rendszerbe sérülékenységen keresztül, amit a rendszer az eredeti alkalmazás kódjával együtt végrehajt. Ilyen támadások közé tartozik, amikor a támadó egy sérülékenységen keresztül megvalósít egy operációs rendszer hívást, egy külső alkalmazás elindítását shell parancs segítségével, vagy egy kommunikációt az adatbázisszerverrel (SQL utasítások felhasználásával). Teljes szkripteket (pl: Python, Perl, stb.) lehet bejuttatni, és végrehajtani egy rosszul megírt alkalmazáson keresztül. Minden helyzetben, ahol interpretált nyelv segítségével készül egy alkalmazás, így a forráskód valós időben kerül értelmezésre és végrehajtásra, egy kódbeszúrásos támadás veszélye fennáll.

2.5. Adatbázismotorok

Adatbázismotornak nevezzük azt a szoftver komponenst, amelyet egy adatbázis-kezelő rendszer használ az adatok beszúrásához, kiolvasásához, módosításához, vagy törléséhez. A legtöbb adatbázis kezelő rendszer definiál valamilyen saját API-t, aminek a segítségével a felhasználó hozzáférhet az adatbázis motor által kezelt adathoz a felhasználói felület manuális használata nélkül.

2.5.1. Relációs adatbázisok

A relációs adatbázis egy olyan adatbázis típus, amely valamilyen kapcsolatban álló adatok tárolására lett kifejlesztve. A felépítés a relációs adatmodellre épít, amely egy intuitív és átlátható formája az adatok táblában tárolásának. Egy táblában minden sor egy rekordnak felel meg, amelyet egy egyedi érték (kulcs) azonosít. Az oszlopok jelentik az adatok attribútumait, amelyek általában a legtöbb rekord esetén mind ki is vannak töltve.

A relációs modell egy standardizált módja az adatok tárolásának és manipulálásának, melynek nagy erőssége, hogy a táblákban tárolt adatleírást könnyű átlátni, és használni. Idővel egy másik erőssége is megjelent a technológiának, ez pedig a strukturális lekérdező nyelv (SQL - Structured Query Language). Hosszú időn át ez a nyelv volt az egyeduralkodó az adathozzáférés terén. Relációs algebra alapon megvalósítva, az SQL nyelv egy konzisztens matematikai leíró nyelv, aminek a segítségével hatékony lekérdezések küldhetőek az

adatbázisnak.

2.5.2. NoSQL adatbázisok

A NoSQL adatbázisok lazább strukturális felépítést engednek meg. Mivel kevesebb a megkötés az adatok felé relációs szempontból, így a NoSQL adatbázisok nagyobb teljesítményt és skálázhatóságot tudnak elérni bizonyos esetekben. Biztonság szempontjából azonban, a relációs adatbázisokhoz hasonlóan, ezek is lehetnek kódbeszúrásos támadás célpontjai, annak ellenére, hogy nem SQL nyelvet használnak az adathozzáféréshez. Sőt, a kommunikáció tipikusan procedurális nyelven történik (pl: JavaScript), így egy támadás hatása még súlyosabb is lehet a deklaratív SQL nyelv elleni támadásokhoz képest.

A NoSQL adatbázis hívások az alkalmazás fejlesztéséhez használt programozási nyelven készülnek. A kommunikáció egyedi API hívásokkal történik, amelyek néha megkövetelnek megfelelően megformázott adatot (pl: XML, JSON, LINQ, stb.). Egy ilyen formátumban elrejtett támadás nem feltétlenül fog fennakadni az elsődleges nyelv által használt biztonsági szabályokon. Például, a HTML vezérlőkarakterek kiszűrése, úgy mint a `<>&;` karakterek, nem fognak megállítani egy JSON API elleni támadást, ahol a speciális karakterek közé tartoznak még a `\{}:` karakterek is.

Manapság már több mint 150 különböző NoSQL adatbázis közül választhatnak a fejlesztők, amelyek a programozási nyelvek széles skáláján nyújtanak API támogatást. Ezek általában mind valamennyire eltérő funkcionalitásokat valósítanak meg, különböző megszorítások mellett. Mivel nincs egy egységes nyelv a NoSQL adatbázisok kezelésére, így az egyes támadások sem lesznek egyformán hatásosak mindegyik ellen. Emiatt, aki NoSQL elleni támadást szeretne megérteni, első lépésként, a háttérben használt konkrét technológia sajátosságait (pl: szintaxis) kell megismernie. Van egy jelentős eltérés a kétféle adatbázis típus elleni támadás között: NoSQL esetben nem magától értetődő, hogy a támadás hol fog végrehajtódni. Egy SQL beszúrásos támadást mindig az adatbázismotor értelmez. Egy NoSQL elleni támadás, a használt API és adatmodell függvényében, vagy az adatbázis szintjén, vagy akár még az alkalmazás szintjén is végrehajtódhat.

Tipikus NoSQL elleni támadás akkor szokott előfordulni, ha a támadó által módosított sztring egy NoSQL API hívásban közvetlenül felhasználásra kerül.

2.6. Az SQL nyelv

Az SQL, ami a Structured Query Language rövidítése, egy strukturális lekérdező nyelv. Ezt a nyelvet használja a legtöbb relációs adatbázis a kommunikációra. SQL utasítások segítségével lehet adatot beszúrni, lekérdezni, módosítani, vagy törölni. Az SQL nyelvet használja, többek között, a népszerű adatbázismotorok közül az Oracle, a Microsoft SQL Server, MySQL, és az SQLite is. Habár mindegyik motor hivatalosan az SQL nyelvet használja, az egyes megvalósítások kis mértékben eltérnek a motor specifikus kiegészítések miatt. Így például egyes esetekben a komment kezdetét jelző karakter eltér. Az alap utasítások, úgy mint `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE` és `DROP` megegyezik mindegyik implementációban, így a legtöbb gyakorlati probléma megoldása során nem érezhető a különbség. A labor teljesítéséhez az SQL nyelv alapvető ismerete szükséges, amely elsajátítható például erről az oldalról: https://www.w3schools.com/sql/default.asp.

2.7. Támadások

A beszúrásos támadások közül az SQL, NoSQL és OS command injection, valamint az ORM vagy LDAP injection szokott a leggyakrabban előfordulni. Ezeknek a támadásoknak az alapötlete elég hasonló. A sérülékenység alapja mindig az, hogy a legitim kód és a külső forrásból származó adat összekeveredik a feldolgozás során.

2.7.1. SQL Injection

Az SQL injection vagyis az SQL beszúrásos támadás a kategóriának a legismertebb és legjobban elterjedt példája. Egy ilyen sérülékenység kihasználásához a támadónak először találnia kell egy paramétert, amit a webalkalmazás továbbít az adatbázis felé valamelyik kérés során. Ezután, ha a támadó képes egy speciálisan megválasztott értéket megadni a paraméter értékéül, akkor ráveheti az alkalmazást, hogy egy módosított kérést küldjön el az adatbázis felé. Egy ilyen támadás végrehajtása nem kíván meg speciális szakértelmet, és a felmerülő nehézségek megoldására egyre több automatizált eszköz áll rendelkezésre.

Az ilyen sérülékenységek az alkalmazások minden részében elfordulhatnak, így megtalálásuk a triviálistól a rendkívül nehézig változhat. Egy sikeres támadás hatása szintén nagyon széles lehet, egyszerűbb adatszivárgástól az adatbázis teljes elvesztéséig minden. Általánosságban, egy bonyolultabb alkalmazás esetén érdemes azt feltételezni, hogy a forráskód tartalmaz ilyen sérülékenységet, mivel erre van nagyobb valószínűség.

Egy alkalmazás sérülékeny, ha a következő feltételek közül valamelyik teljesül:

- A felhasználó által megadott adatot nem ellenőrzi az alkalmazás mielőtt azt használja (input validation és filtering hiánya).
- Dinamikus lekérdezések vagy nem-parametrikus lekérdezések vannak használatban a környezetnek megfelelő escape-elés nélkül.
- Felhasználótól származó adat kerül felhasználásra egy ORM rendszer használata során, további érzékeny adatok lekérdezéséhez.
- Felhasználótól származó adat van ellenőrzés nélkül kézzel hozzáfűzve egy SQL utasításhoz (dinamikus lekérdezés vagy tárolt eljárás is lehet sérülékeny).

Egy tipikus **sérülékeny** és emiatt **hibás** megvalósítása egy SQL lekérdezésnek:

```
1 String query = "SELECT * FROM users WHERE name = '" + userName + "' AND
    password = '" + password + "'";
```

Ez az implementáció funkcionálisan elfogadhatóan működik, amíg a felhasználó ténylegesen csak felhasználónevet és jelszót ad meg bemenetként. Egy támadó azonban felhasználhatja bármelyik mezőt arra, hogy egy SQL injection támadást valósítson meg. Egy ilyen esetben a támadó megadhatja bemenetként például a következő értékeket:

- username: `whoever`
- password: `asd' OR 'a'='a`

Ezzel módosul az alkalmazás viselkedése. Az így létrejött lekérdezés már eltér az eredetitől:

```
SELECT * FROM users WHERE name = 'whoever' AND password = 'asd' OR 'a'='a'
```

A pluszban hozzáfűzött `OR 'a'='a'` rész alapjaiban módosítja az eredeti lekérdezés funkcióját. Ahelyett, hogy csak egy rekordot adna vissza az adatbázis, az összes rekordot visszaadja a táblából. A támadó így meg tudta kerülni a jelszó ellenőrzést az alkalmazásban. Amennyiben az alkalmazás kódja a válasz első sorát használja fel automatikusan (mivel arra számít, hogy csak 1 rekordot fog visszakapni), akkor a támadó több célt is elérhet egyből: egy másik felhasználó nevében jelentkezik be. A felhasználók közül az első tipikusan az adminisztrátor szokott lenni, így ez még további különösen nagy problémát is okozhat.

Egy másik tipikus trükk, amit a támadók fel szoktak használni, az az SQL komment karakter használata. Ez a karakter, ahogy korábban is volt már róla szó, eltér az egyes adatbázismotorok között. A komment karakter használata, egy támadás során, képes "megállítani" a lekérdezés végrehajtását, mivel a komment karakter hatására megáll a lekérdezés még hátra lévő részének a feldolgozása. A következő példa szemlélteti ezt az esetet:

- username: `name' OR 'a'='a' #`
- password: `whatever`

Ezen bemenetek hatására az alábbi SQL lekérdezés fog előállni:

```
SELECT * FROM users WHERE name = 'name' OR 'a'='a' #' AND ...
```

A feldolgozás során az adatbázismotor nem fogja már kiértékelni a jelszóra vonatkozó feltételt.

Megjegyzés: A korábban bemutatott példakódok nem teljesen helyesek szintaktikailag. A céljuk a koncepciók bemutatása, nem pedig konkrét támadások terjesztése.

2.7.2. A támadás megakadályozása

Kódbeszúrásos támadást a leghatékonyabban forráskód ellenőrzéssel, és alapos teszteléssel lehet elkerülni. A tesztelés során érdemes minden paramétert, fejléc mezőt, URL részt, sütit, JSON dokumentumot, SOAP vagy XML bemenetet tesztelni, mert ezek játszanak tipikusan szerepet egy ilyen támadásban. Ilyen teszteléseket érdemes egy a CI/CD folyamat részeként megvalósítani, hogy minden forráskód módosításkor ezek automatikusan lefussanak.

Egy kódbeszúrásos támadás megállításának a kulcskérdése, hogy sikerüljön az eredeti kódbázist, és a feldolgozandó adatot elkülönítve kezelni.

- Ideális esetben a fejlesztés során csak megbízható API-t használjunk, amely nem interpretálja a bemenetet. Erre a legelterjedtebb opció egy parametrizált interfészt (parameterized interface) használata. SQL injection esetén, ezt a legtöbb keretrendszerben vagy nyelven prepared statementnek nevezik. Emellett, szintén segíthet egy ORM (Object Relational Mapping) keretrendszer használata, azonban fontos figyelni rá, hogy ez nem jelent automatikus védelmet, így érdemes ellenőrizni, hogy a keretrendszer fejlesztői hogyan kezelik a problémát.
- Egy engedélyező lista (allow list) használata is javít a helyzeten, az input validáció részeként. Teljes védelmet ez a megoldás általában nem ad, mivel szükséges lehet néhány speciális karakter engedélyezése az alkalmazás funkcionalitásához.

- Ha az előző két módszer nem alkalmazható valamilyen okból kifolyólag, akkor a dinamikus lekérdezések összeállítása során escape-eljük a bemenetet, az adott interpreter által használt szabályoknak megfelelően. *Megjegyzés:* Az SQL adatstruktúra neveit (pl: tábla-, vagy oszlopneveket, stb.) nem lehet escape-elni, emiatt, ha egy struktúra neve külső adatként érkezik az alkalmazásba, az komoly biztonsági kockázatot jelent. Ez a probléma gyakran szokott előfordulni jelentést vagy kimutatást készítő alkalmazások fejlesztése során.
- A `LIMIT` és ahhoz hasonló kulcsszavak használata szintén javasolt, mivel ezzel korlátozni lehet egy támadás során, az egy lépésben kiszivárgó adatok mennyiségét.

2.7.3. Denial-of-service (más néven szolgáltatás megtagadás)

Egy denial-of-service (DoS) támadás során, a támadó célja, hogy leterhelje az elérhető erőforrásokat annyira, hogy a legitim felhasználók számára már ne maradjon szabad kapacitás. Például elképzelhető, hogy egy támadó nagy terhelést generál egy átlag számítógép processzora és hálózati kártyája ellen, amivel rövidebb-hosszabb kiesést érhet el a gépet használni próbáló felhasználó munkájában: akadozhat az internet elérés, email letöltés, vagy bármelyik közepes, vagy nagyobb erőforrást igénylő feladat végrehajtása.

3. Környezet felállítása

3.1. VM használata

A VirtualBox alkalmazást használva importáljuk a kiadott OVA fájlt, majd indítsuk el a VM-et. A kiadott VM-et használva semmilyen további előkészület nem szükséges.

A Juice Shop elindításához szükséges lépések a következőek:

1. Indítsa el a Terminál alkalmazást.
2. Lépjen be a Juice Shop könyvtárába: `cd ~/juice-shop/`
3. Indítsa el a Juice Shop alkalmazást: `npm start`
4. Az indulás a gépek terheltségétől függően pár percig is eltarhat.

A Juice Shop interaktív módban fut, így az eddig használt terminálban nem fog tudni további parancsokat kiadni. Ha bezárja a terminál ablakot, akkor leáll az alkalmazás is. A mérés végén a `CTRL + C` billentyűkombináció segítségével lehet majd leállítani az alkalmazást szabályosan.

Az indulás után az alkalmazás a `http://localhost:3000`-es címen érhető el.

3.2. Otthoni környezet használata

A Juice Shop elindításához szükséges lépések otthoni környezetben a következőek:

1. Telepítsd az operációs rendszerednek megelelő módon a Node.js-t az alábbi link segítségével. A támogatott Node.js verziókért nézd meg az alábbi oldalt
2. Töltsd le a Juice Shop GitHub projektet. Például klónozd le terminálból az alábbi módon: `git clone https://github.com/juice-shop/juice-shop.git`

1. ábra. Várt eredmény

3. Telepítsd az alábbi linken megtalálható leírásnak megfelelően.
4. Kövesd a lépéseket a **VM használata** szekcióban leírtak szerint.

Megjegyzés: amennyiben más módon szeretnéd használni az alkalmazást, például Docker segítségével a 3. lépésben megosztott leírásban különféle telepítési útmutatok találhatóak, melyek segítségével különböző módon telepíthető az alkalmazás.

4. Feladatok

Ez a labor gyakorlat 6 feladatból áll: 3 XSS - 3 SQL injection-nel kapcsolatos feladatból. Az egyes feladatok egy "blokkon" belül nehézség szerint növekvő sorrendben vannak, így a megoldásuk a leírt sorrendben javasolt. A labor során a Juice Shop alkalmazást kell használni.

A gyakran használt XSS támadásokat több helyen is összegyűjtötték már. Egy ilyen lista a feladatok megoldásában is segítség lehet, a próbálkozásokhoz innen javasolt ötleteket gyűjteni: XSS Filter Evasion Cheat Sheet.

Amennyiben valamelyik feladat megoldásához szükséges egy bejelentkezett felhasználó, akkor az adminisztrátor felhasználó használata javasolt. Ehhez a **felhasználónév:** `admin@juice-sh.op`, valamint a **jelszó:** `admin123`

A frontenddel kapcsolatos feladatok megoldása során a cél mindig a Javascript kód futtatás demonstrálása, amit a legegyszerűbben a JavaScriptben elérhető `alert()` függvény meghívásával lehet megtenni. A cél az, hogy feldobjunk egy ablakot (`<iframe>` tag), amiben az `xss` szöveg szerepel.

A támadásokat gyakran megnehezíti, hogy string beszúrása során figyelni kell, hogy az idézőjelek megfelelően legyenek használva. Az eredeti forráskód is rend szerint használja a `'` és a `"` karaktereket. Ilyen esetekben célszerű lehet még a ‘(backtick) karaktert is kipróbálni, amely szintén használható string eleje és vége jelként.

Az SQL injection-nel kapcsolatos feladatok megoldása során a sérülékenységek elleni támadásokat először érdemes egy egyszerűbb verzióban kipróbálni, hogy kiderüljön, hogy valóban a megfelelő helyen próbálja megoldani a feladatokat. Erre egy jó példa, hogy egy SQL injection támadást először csak egy olyan minimális bemenettel érdemes tesztelni mint például a `'` vagy a `';` karakterek. Az ilyen bemenetekre adott válaszok elemzésével megállapíthatja, hogy a sérülékenység valóban kihasználható, és így ezután érdemes a feladat megoldását célzó összetettebb bemenetet összeállítani.

4.1. XSS támadások

4.1.1. Hajtson végre egy reflected XSS támadást!

Keressen az alkalmazásban olyan bemenetet, amely tartalma a szerverrel megvalósított kommunikáció során megjelenik a válaszban is.

Segítség: A sérülékenység a megrendelések nyomon követése szolgáltatásban van. (Ezt a korábbi rendelések [Order History] oldalon a kis teherautó ikonra kattintva lehet elérni.)

Hajtson végre XSS támadást, amely során egy HTML elemet szúr be a weboldalba (lásd korábban részletezett várt eredmény)!

4.1.2. Hajtson végre egy DOM alapú XSS támadást!

Ez a feladat nagyon hasonló az előző feladatban végrehajtotthoz.

Ismételje meg a támadást a weboldal keresés funkciója ellen, majd keresse meg a forrás kódban, hogy hol található a sérülékeny kódrészlet. A keresés feldolgozását megvalósító kódot ezen a linken keresztül éri el: https://github.com/juice-shop/juice-shop/blob/master/frontend/src/app/search-result/search-result.component.html.

4.1.3. Hajtson végre egy persistent XSS támadást a kliens oldali védelem megkerülés után!

Minden új felhasználó adata eltárolásra kerül az adatbázisban. A regisztráció folyamata során minden adat megfelelő ellenőrzése egy kritikus feladat a biztonság szempontjából.

A Juice Shop alkalmazásban ezt a védelmet a kliens oldalra implementálta a fejlesztő, ami így a támadó által megkerülhető. A kliens oldalra írt védelmek a weboldal HTML tartalmának a kézzel végrehajtott módosításaival könnyen megkerülhető.

Hajtson végre persistent XSS támadást, hogy demonstrálja ezt a sérülékenységet! A sikeres támadás eredményét a `.../#/administration` URL megnyitása után lehet látni!

Ez a kihívás a webes alkalmazások egy nagyon gyakori biztonsági hibáján alapul, ahol a fejlesztők figyelmen kívül hagyták a következő aranyszabályt a bemeneti érvényesítésről:

Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

Segítség: a sebezhetőség a regisztrációs űrlapon található. Az űrlap DOM-fájának manipulálásával megkerülhető a regisztráció "kliens oldali biztonsága".

4.2. SQL injection

4.2.1. Szerezzen adminisztrátori hozzáférést a rendszerhez!

Találjon egy sérülékeny bemenetet, ahol a hibát kihasználva képes az adminisztrátor nevében bejelentkezni. Bár a feladat megoldható a jelszó kitalálásával is, a cél egy SQL injection végrehajtása.

Segítség: A bejelentkezést végző szerver oldali kód az alábbi linken elérhető: https://github.com/juice-shop/juice-shop/blob/master/routes/login.ts.

4.2.2. Hagyja a szervert végre pihenni egy kicsit!

A megoldandó feladat arról szól, hogy a folyamatos terhelés után végre pihenhessen a szerver egy kicsit, ami a felhasználók szempontjából egy leegyszerűsített denial-of-service támadásként érzékelhető.

A Juice Shop web alkalmazás egy MongoDB nevű NoSQL adatbázist használ információk tárolására. Ebben található többek között az egyes termékek értékelése. A feladat során először az értékelések betöltéséhez használt hálózati kommunikáció megkeresése szükséges, majd a kérésben található sérülékenység megtalálása után, a támadás végrehajtása. A támadáshoz a MongoDB által használt sleep függvény használata javasolt, melynek dokumentációja megtalálható itt: https://docs.mongodb.com/manual/reference/method/sleep/.

Segítség: A szerver oldali kód, amely kiszolgálja a kéréseket megtalálható ezen a linken: https://github.com/juice-shop/juice-shop/blob/master/routes/showProductReviews.ts.

Megjegyzés: a szerver elárasztása nagy számú kérésekkel nem fogja megoldani ezt a feladatot. Ezzel valószínűleg csak a teljes környezet túlterhelését lehet elérni, ami nem számít helyes megoldásnak.

4.2.3. Jim jelszavának a megváltoztatása az 'Elfelejtett jelszó' funkció segítségével.

Használja az alkalmazásba épített elfelejtett jelszó funkciót, hogy módosítsa Jim (jim@juice-sh.op) jelszavát. A jelszó helyreállítás rendelkezik egy beépített védelemmel, amely az engedély nélküli módosításokat hivatott megállítani: egy biztonsági kérdésre kell válaszolni. Ezt a védelmet azonban social engineering támadás segítségével ki lehet játszani.

Jim az egyik lehető legrosszabb biztonsági kérdést választotta a regisztrációja során. Ez, azzal együtt, hogy Jim egy híresség semmilyen védelmet nem nyújt a felhasználói fiókja számára, mivel a kérdésre a választ ki lehet találni. A feladat megoldásához a következő lépések javasoltak:

Segítség:

- Keresse meg az 'Elfelejtett jelszó' funkciót a web alkalmazásban!
- Jim e-mail címének a megadása után jegyezze meg Jim biztonsági kérdését!
- Keressen olyan Juice Shop termékeket amelyekhez írt Jim értékelést! A második oldalon érdemes keresgélni. Az így talált információ árulkodhat Jim személyéről.
- Az első feladatból ismert SQL injection módszerhez hasonlóval jelentkezzen be Jim nevében, majd keresse meg az általa megadott szállítási címeket!
- Az értékelésekből és a címekből megszerzett információk alapján derítse ki az interneten, hogy ki Jim valójában.
- A Jimről található elérhető információkból válaszolja meg a korábban megtalált biztonsági kérdést!