Kompromitace dat pomocí SQL Injection

část I

Jakým způsobem nejčastěji probíhají útoky na webové systémy napojené na databáze? Jak se útočníci dostávají k cenným údajům uloženým v takové databázi? Jaké možnosti mají? Jaká jsou nejčastěji využívaná zranitelná místa?

Zranitelnost SQL Injection (SQL injekce či zkratka SQLi) je známá již od dob prvních webových informačních systémů napojených na databázi. Přesto je i v dnešní době zranitelnost často útočníky zneužívána pro krádeže citlivých dat [1]. Cílem první části článku je popsat nejběžnější způsoby využití této zranitelnosti ke krádeži databázových dat. V dalších pokračováních popíšeme méně známé způsoby útoku a možnosti obrany.

Moderní databázové systémy (Database Management System – DBMS) již neslouží pouze pro manipulaci s daty v relačních (a jiných[1]) databázích prostřednictvím strukturovaného dotazovacího jazyka SQL, ale umožňují např. zápis a čtení souborů z disku či spouštění příkazů na úrovni operačního systému. Tyto funkčnosti přispívají k možnostem databázového systému napojeného na informační systém. Avšak v případě kompromitace aplikace prostřednictvím SQL Injection se tyto možnosti dostávají do rukou útočníka. Výsledkem je, že útok prostřednictvím SQL Injection nemusí nutně spočívat jen v úniku či změně dat, ale může končit až kompletní kompromitací databázového serveru na úrovni operačního systému [3].

Na obr. 1 je zobrazen typický útok SQL Injection pro obejití autentizace do běžné webové aplikace.

Kompromitace dat prostřednictvím SQLi

Zranitelnost SQL Injection spočívá v tom, že je uživatelský vstup bez ošetření předán na zpracování databázovému systému [9]. Pro představu uvažujme jednoduchý informační systém, který na základě hodnoty předaného identifikátoru zobrazí odpovídající článek – jeho nadpis a tělo[2]:

```
http://example.com/?id=2
```

SQL dotaz zpracovávaný databázovým systémem vypadá následovně:

```
SELECT header,body FROM
articles WHERE id=3
```

Červeně je v rámci dotazu znázorněn vstup uživatele prostřednictvím parametru id. Při testování na výskyt SQL Injection v daném parametru[3] aplikace se často používá jako vstup apostrof:

```
http://example.com/?id=3'
```

Výsledný databázový dotaz se tím stává syntakticky nesprávným:

```
SELECT header,body FROM
articles WHERE id=3'
```

Dle typu databázového systému je obvykle vypsáno chybové hlášení informující o nesprávné syntaxi zpracovávaného dotazu. Je-li vypisování chybových hlášení potlačeno, lze pro detekce zranitelnosti SQL Injection využít klauzuli ORDER BY s číselným parametrem, který udává pořadové číslo sloupce, podle nějž bude prováděno řazení [6].

V případě výskytu zranitelnosti SQL Injection v parametru id se rozložení stránky při následujících dvou vstupech bude měnit:

```
http://example.com/?id=3
ORDER BY 1
http://example.com/?id=3
ORDER BY 2
```

V případě řazení podle třetího (neexistujícího) sloupce již ke změně stránky oproti původní nedojde. Tímto způsobem je možné potvrdit výskyt SQL Injection v daném parametru, i pokud má aplikace potlačená chybová hlášení. Metod detekce zranitelnosti je více, lze např. využít logické výrazy, konkatenace řetězců či další konstrukce [7].

[1] Např. objektových či objektově relačních
[2] Tento příklad bude používán i v dalších kapitolách článku
[3] Může se jednat o HTTP GET i POST parametry

Jakmile má útočník potvrzeno, že se v daném parametru opravdu zranitelnost SQL Injection nachází, jsou jeho dalším cílem zpravidla data uložená v databázi. Další postup, jak se k těmto datům dostat, záleží na konkrétním použitém databázovém systému a webové technologii. Nejběžnějším způsobem je využití zřetězených dotazů či konstrukce UNION. Těmto postupům se věnují následující podkapitoly.

A. Zřetězené dotazy

Některé databázové systémy v kombinaci s webovou technologií umožňují SQL dotazy jednoduše řetězit za sebe oddělené středníkem. Následuje orientační výčet kombinací webových a databázových technologií, které zřetězené dotazy (tzv. stacked queries) podporují [5]:

- MySQL/ASP.NET
- PostreSQL/ASP.NET
- PostreSQL/PHP
- MSSQL/ASP.NET
- MSSQL/PHP

Pokud chce útočník přečíst data z jiné tabulky a databáze, než nad kterou operuje dotaz, jednoduše svůj dotaz připojí za ten, do kterého je prováděna SQL injekce:

```
http://example.com/?i-
d=3;SELECT username,-
password FROM ebank.users
```

Výsledný zřetězený dotaz vykonaný databázovým systémem vypadá následovně:

```
SELECT header,body FROM
articles WHERE id=3; SE-
LECT username,password
FROM ebank.users
```

V případě podpory zřetězených dotazů se však útočník nemusí omezovat pouze na čtení dat. Může např. zahltit informační systém, tedy způsobit jeho nedostupnost pro legitimní uživatele (útok typu Denial of Service). Toho dosáhne smazáním tabulky či celé databáze:

```
http://example.com/?i-
d=3;DROP ebank.users
```

Dalším způsobem, jak nadměrně vytížit databázi a tím nedostupnost informačního systému, je použití vestavěných funkcí náročných na výpočet výkonu. Takovou funkcí je např. funkce benchmark() databázového systému MySQL či Oracle. V případě MySQL první parametr udává počet opakování, kolikrát bude vykonán příkaz uvedený v druhém parametru. Opakovaný následující požadavek tak rychle vytíží dostupné zdroje databázového serveru:

```
http://example.com/?i-
d=3;BENCHMARK(99999999,
MD5(,'Denial of Service'))
```

B. UNION dotazy

Ne všechny webové technologie však podporují zřetězování dotazů, kdy lze jednoduše cílový dotaz zapsat přímo za stávající. Typickým zástupcem této skupiny je poměrně rozšířená kombinace databázového systému MySQL a skriptovacího jazyka PHP. PHP dovoluje svými interními funkcemi spustit vždy pouze jeden SQL dotaz[4]. Přesto existuje řešení, jak i v této situaci pomocí injekce upravit databázový dotaz tak, aby bylo dosaženo požadovaného výsledku. Tímto řešením je syntaktická konstrukce jazyka SQL nazvaná UNION. Lze si ji představit jako sjednocení dvou množin. Nutnou podmínkou pak je, že databázový dotaz přidaný pomocí UNION musí vracet stejný počet sloupců jako dotaz originální.

Jelikož však útočník obvykle nemá k dispozici zdrojové kódy aplikace, na niž útočí, nemá ani informaci o počtu sloupců vracených daným SQL dotazem. Tento počet je možné zjistit prostřednictvím klauzule ORDER BY s číselným parametrem. Číselný parametr uvádí pořadí sloupce, podle kterého budou data řazena. Útočník tak může začít číslem 1 a pokračovat dále, dokud mu chybové hlášení neoznámí, že sloupec, podle kterého chce řadit, neexistuje [4]. Předpokládejme, že následující dotaz se provede korektně:

```
SELECT header,body FROM
articles WHERE id=3 ORDER BY 1
```

Stejně jako tento:

```
SELECT header,body FROM
articles WHERE id=3 ORDER BY 2
```

Avšak následující dotaz, kdy chceme řadit podle třetího sloupce, již skončí chybou:

```
SELECT header,body FROM
articles WHERE id=3 ORDER BY 3
```

Tímto způsobem je možné postupně enumerovat počet sloupců, který databázový dotaz vrací. V tomto případě se jedná o dva sloupce.

Se znalostí počtu sloupců je pak možné dotaz upravit následujícím způsobem, aby byla vypsána verze MySQL a jméno databázového uživatele:

```
SELECT header,body FROM
articles WHERE id=3 UNION
SELECT version(),user()
```

Pro výpis verze a jména databázového uživatele, pod kterým je dotaz spouštěn, byly použity vestavěné funkce MySQL user() a version().

V praxi útočník databázový dotaz navíc modifikuje způsobem, aby původní dotaz nevracel žádná data, a tudíž aby tak jedinými vypsanými daty bylo pouze to, co útočník požaduje [7]. Toho lze docílit jednoduše nastavením hodnoty id na neexistující (typicky zápornou) hodnotu:

```
SELECT header,body FROM
articles WHERE id=-1
UNION SELECT version(),user()
```

Dosud byly v UNION dotazu využity pouze vestavěné funkce. Vypsání dat

[4] ASP.NET v kombinaci MySQL skládání dotazů podporuje

z databáze však probíhá obdobným způsobem. Následuje příklad, který vypíše uživatelská jména aplikačních uživatelů spolu s jejich hesly:

```
SELECT header,body FROM
articles WHERE id=-1
UNION ALL SELECT username,
password FROM app.users
```

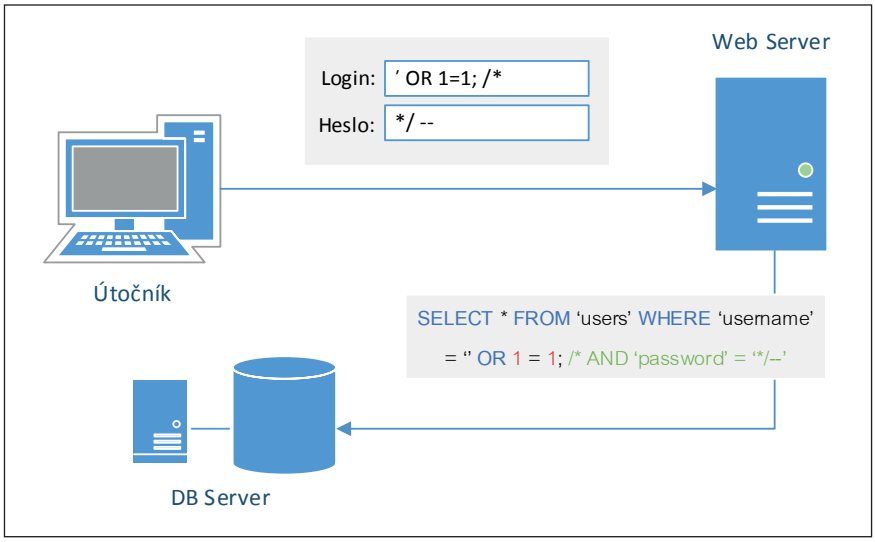
Spuštění výše uvedeného SQL dotazu může být prostřednictvím SQL Injection vykonáno následujícím požadavkem:

```
http://example.com/?id=-1
UNION ALL SELECT username,
password FROM app.users
```

V příkladu byla použita konstrukce UNION ALL, která zajistí vypsání všech záznamů a ne pouze těch jedinečných. Zrovna zde by postačilo taktéž použití standardního UNION, jelikož předpokládáme, že se v aplikaci nenacházejí dva a více uživatelů se stejným jménem a heslem. U složitějších dotazů však má již použití UNION ALL své opodstatnění.

Příkladem může být situace, kdy by útočník chtěl zjistit, kolik odchozích transakcí provedl daný uživatel. Výstupem poddotazu bude počet řádků odpovídající počtu odchozích transakcí uživatele. V každém řádku však bude pouze stále stejné uživatelské jméno. Pokud by v této situaci byla použita konstrukce UNION namísto UNION ALL, byla by vždy vrácena hodnota 0 (uživatel neprovedl žádné odchozí transakce) nebo 1 (uživatel provedl jednu nebo více transakcí). Samotná informace o přesném počtu by byla ztracena.

Konkatenace dotazu pomocí UNION však útočníkovi neposkytuje stejnou sílu jako zřetězené dotazy popisované v předchozí podkapitole. UNION může sice být použit pro čtení dat napříč dostupnými databázemi a tabulkami, avšak již není možné data vkládat (INSERT), upravovat (UPDATE) ani provádět další akce (například DROP).



Obrázek 1: Útok pomocí injekce logické pravdy (1=1) pro obejití autentizace

C. Blind SQL Injection

O Blind SQL Injection nebo také slepé SQL injekci hovoříme v případě, že prostřednictvím zranitelnosti není možné vypsat do stránky žádná data [11]. Prakticky to tedy znamená, že je možné uživatelský vstup injektovat do parametru způsobem vykonání databázového systému, avšak výsledek dotazu není vypsán v rámci webové stránky jako v případě výše uvedených variant zranitelnosti SQL Injection.

I v tomto případě však existují metody, jak informace z databáze přečíst. Útočník se nejprve ujistí, že je v cílovém parametru přítomna zranitelnost SQL Injection, a to např. prostřednictvím logických výrazů. Pokud následující injekce nevrátí žádný výsledek:

```
http://example.com/?id=3
AND 0=1
```

A následující injekce vrátí původní výsledek, pak je pravděpodobné, že je parametr zranitelný vůči SQL Injection:

```
http://example.com/?id=3
AND 1=1
```

Pokud by se útočník pokusil vypsat aktuálního databázového uživatele, nebyl by úspěšný, jelikož výstup dotazu není zobrazen nikde na stránce:

```
http://example.com/?id=3
UNION SELECT NULL,user()
```

Jedná se tedy o klasický scénář Blind SQL Injection. Pro jednoduchost uvažujme, že motivací útočníka je zjistit jméno databázového uživatele, pod kterým jsou spouštěny SQL dotazy. Jde mu tedy o to zjistit výstup funkce user (), který však již nelze dostat tak jednoduše jako v případě klasického SQL Injection.

Řešením je dotaz na jméno zkombinovat s logickým výrazem. Pokud bude výsledek logického výrazu pravda, zobrazí se původní stránka, bude-li výsledek logického výrazu nepravda, nedojde k zobrazení obsahu stránky [11].

Dotazy jsou poté koncipovány níže uvedeným způsobem:

- Je první písmeno jména uživatele ‘a’?
- Je první písmeno jména uživatele ‘b’?
- ...
- Je druhé písmeno jména uživatele ‘a’?
- ...

Injekce představující první z dotazů poté bude vypadat následovně:

```
http://example.com/?id=3
AND 1=(ascii(substring
((SELECT
user()),1,1))=97)
```

V dotazu je použita funkce substring(), která vrátí daný podřetězec (zde konkrétně první písmeno řetězce), a funkce ascii(), která převádí znak na jeho ASCII hodnotu. ASCII hodnota prvního znaku je porovnána s hodnotou 97, což je ASCII hodnota znaku 'a'. Pokud se na tento dotaz vrátí stránka s původním obsahem, ví útočník, že uživatelské jméno začíná na 'a', a pokračuje druhým písmenem. V opačném případě je první znak porovnán s písmenem 'b' (ASCII 98) a tak dále. Stejným způsobem může útočník získat libovolná data z databáze (samozřejmě v rámci hranic daných právy databázového uživatele). Postup lze urychlit využitím metody půlení intervalu.

D. Time Based Blind SQL Injection

Komplikovanější scénář se naplní, pokud daný zranitelný parametr neovlivňuje zobrazení dat na stránce. V takovém případě pak obě níže uvedené injekce (první reprezentuje false – nepravdu, druhá true – pravdu) vrátí stejnou stránku:

```
http://example.com/?id=3
AND 0=1
http://example.com/?id=3
AND 1=1
```

To však ještě nemusí nutně znamenat, že parametr není náchylný na SQL Injection. V takovém případě začne útočník analyzovat průměrné časy odpovědí na uvedené požadavky [8].

Uvažujme, že z deseti pokusů u každého z požadavků zjistí, že odpověď na první požadavek se vrací průměrně za 500 ms, zatímco odpověď na druhý za 1 500 ms. V tomto bodě je zásadní, aby měření doby přijetí odpovědi probíhalo pokud možno na co nejstabilnější lince, jinak mohou být výsledky značně zkresleny. Pokud je mezi odpověďmi true a false znatelná časová prodleva, je možné, že je parametr náchylný na SQL Injection.

Čtení dat z databáze by pak probíhalo stejným způsobem jako v případě klasického Blind SQL Injection, jen s tím rozdílem, že jako pravda by byly brány požadavky s vyšším zpožděním odpovědi nebo naopak.

Postup zneužití ať už základního Blind SQL Injection nebo Time-Based Blind SQL Injection je velice zdlouhavý a vyžaduje velké množství SQL dotazů. Útočník však nemusí dotazy psát ručně, ale může využít nástroje, který tuto činnost automatizuje [7]. Jedním z nejznámějších auditních nástrojů pro testování zranitelnosti SQL Injection je aplikace sqlmap[5].

Závěr

V této první části článku byly uvedeny nejběžnější způsoby zneužití zranitelnosti SQL Injection. Příště bude popsáno, jakým způsobem může útočník SQL Injection zneužít pro přístup k souborovému systému, skenování portů či pro vyvolání XSS.

Lukáš Antal
iantal@fit.vutbr.cz
Maroš Barabas
ibarabas@fit.vutbr.cz
Petr Hamáček
hanacek@fit.vutbr.cz

Ing. Lukáš Antal

Studuje Fakultu informačních technologií v Brně, kde působí jako student Ph.D. se zaměřením na bezpečnost bezdrátových sítí.

Ing. Maroš Barabas

Vystudoval Fakultu informačních technologií v Brně a v současné době působí na této fakultě jako výzkumný pracovník a student Ph.D. studia.

doc. Dr. Ing. Petr Hanáček

Absolvent VUT v Brně, v současné době působí jako docent na FIT VUT v Brně. Zabývá se bezpečností informačních systémů, aplikovanou kryptografií a bezdrátovými systémy.

[5] http://sqlmap.org/

POUŽITÉ ZDROJE

[1] Clark, J.: SQL Injection Attacks and Defense, Second Edition, Syngress, 2012, ISBN 978-1597499637
[2] Mistry, R.: Microsoft SQL Server 2008 Management and Administration, Sams Publishing, 2009, ISBN 978-0672330445
[3] Anley, C: Advanced SQL Injection In SQL Server Applications, NGSSoftware Insight Security Research, 2002, [cit. 7.5.2013], URL https://sparrow.ece.cmu.edu/group/731-s11/readings/anley-sql-inj.pdf
[4] Litchfield, D.: The database hacker's handbook: defending database servers, Wiley, 2005, ISBN 978-0764578014
[5] Stuttard, D., Pinto, M.: The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws: Discovering and Exploiting Security Flaws 2nd Edition, John Wiley & Sons, 2011, ISBN 978-1118026472
[6] Khan, S., Mahapatra, P.: SQL Injection Attack and Countermeasures: Ways to secure query processing, LAP LAMBERT Academic Publishing, 2012, ISBN 978-3659211836
[7] Engebretson, P.: The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy, Syngress, 2011, 978-1597496551, ISBN 978-1597496551
[8] O'Connor, T: Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers, Syngress, 2012, ISBN 978-1597499576
[9] Halfond, W. G., Viegas, J., Orso, A.: A classification of SQL-injection attacks and countermeasures, Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA. 2006, [cit. 9.5.2013], URL http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf
[10] Taylor, A.: SQL All-in-One For Dummies: 2nd Edition, For Dummies, 2011, ISBN 978-0470929964
[11] Litchfield, D.: Data-mining with SQL Injection and Inference, NGSSoftware Insight Security Research, 2005, [cit. 9.5.2013], URL http://www.databasesecurity.com/webapps/sqlinference.pdf