15

SQLMAP

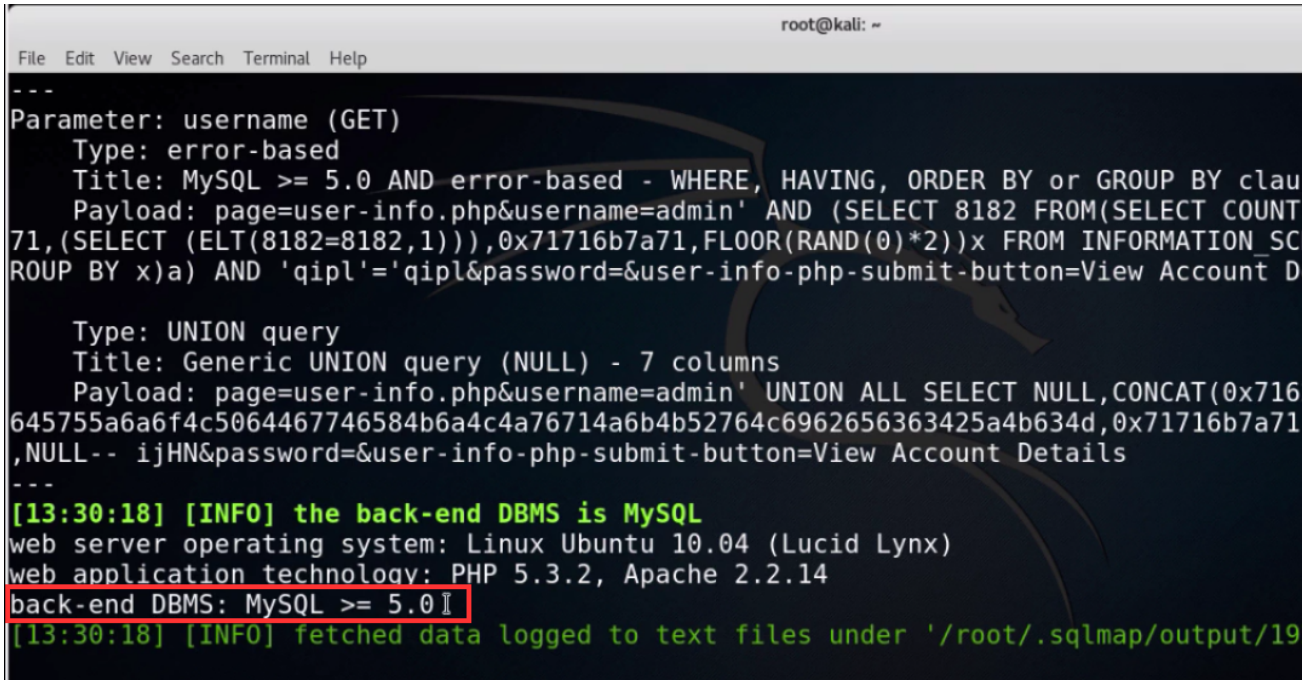
Transcrição

Já descobrimos o nome do banco, a quantidade de colunas e o nome de cada uma delas. Fizemos tudo isso utilizando as *querys* que estão cada vez mais difíceis e complexas. Podemos utilizar uma ferramenta, a SQL Map, para nos auxiliar nessa tarefa. Vamos começar a testá-la! Primeiro, acessamos o site e seguimos pelo caminho "OWASP 2013 > A1 - Injection(SQL) > SQLi - Extract Data > User Info (SQL)". Vamos simular que somos um usuário comum do sistema, portanto, no *Name*, preenchemos `admin` e para a senha não inserimos nada e damos um "Enter". Teremos a seguinte página:

Note que na própria URL podemos visualizar os parâmetros de *name* e senha que acabamos de passar, vamos copiá-la. Pediremos para o SQL Map verificar a vulnerabilidade, assim, abrimos o Terminal do Kali Linux e utilizamos `sqlmap -u` e entre aspas duplas inserimos a URL. O código fica da seguinte maneira:

```
> sqlmap - u "http://192.168.1.37/mutillidae/index.php?page-user-info.php&username=admin&passwo
```

Dando um "Enter" a pesquisa de vulnerabilidades será iniciada e pode demorar um pouco para finalizar. Teremos o seguinte:

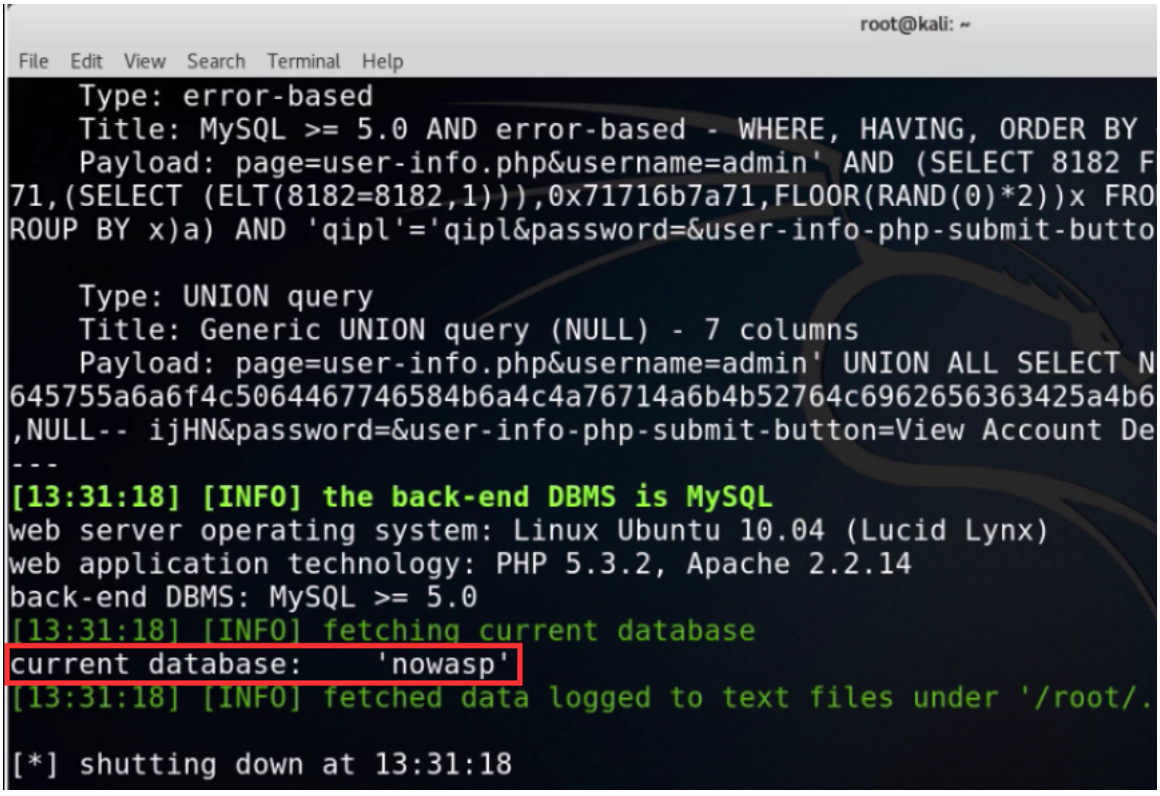


De todas as informações que aparecem uma das mais valiosa é a que afirma: o *back-end* é MYSQL. Assim, temos a confirmação de que estamos lidando com um `MySQL` .

Agora, vamos perguntar ao **SQL Map** qual o banco utilizado, para tanto, damos a seta para cima e aquilo que já havíamos digitado repete-se, assim, acrescentamos `--current-db` :

```
> sqlmap - u "http://192.168.1.37/mutillidae/index.php?page-user-info.php&username=admin&passwo
```

Feito isso temos a seguinte resposta:



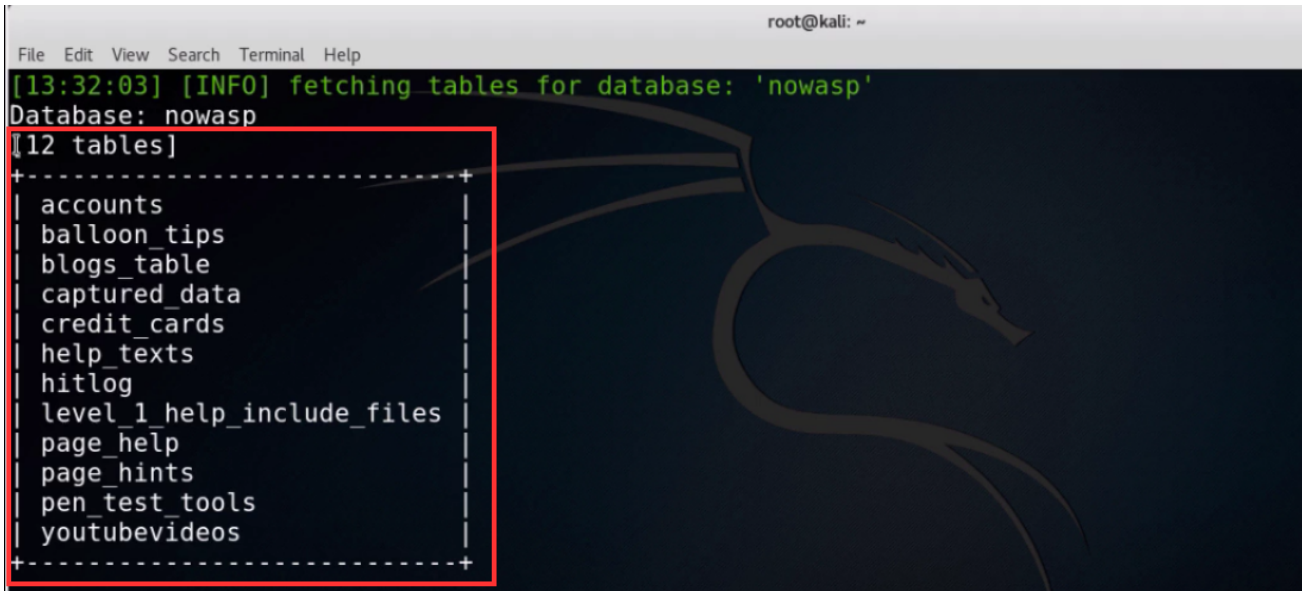
Note que uma das respostas que ele traz é o nome do banco de dados, `nowasp` :

```
current database: 'nowasp'
```

Agora, podemos perguntar quantas tabelas existem no banco do `nowasp`. Vamos dar mais uma seta para cima e completamos com `--tables -D nowasp`:

```
> sqlmap - u "http://192.168.1.37/mutillidae/index.php?page-user-info.php&username=admin&passwo
```

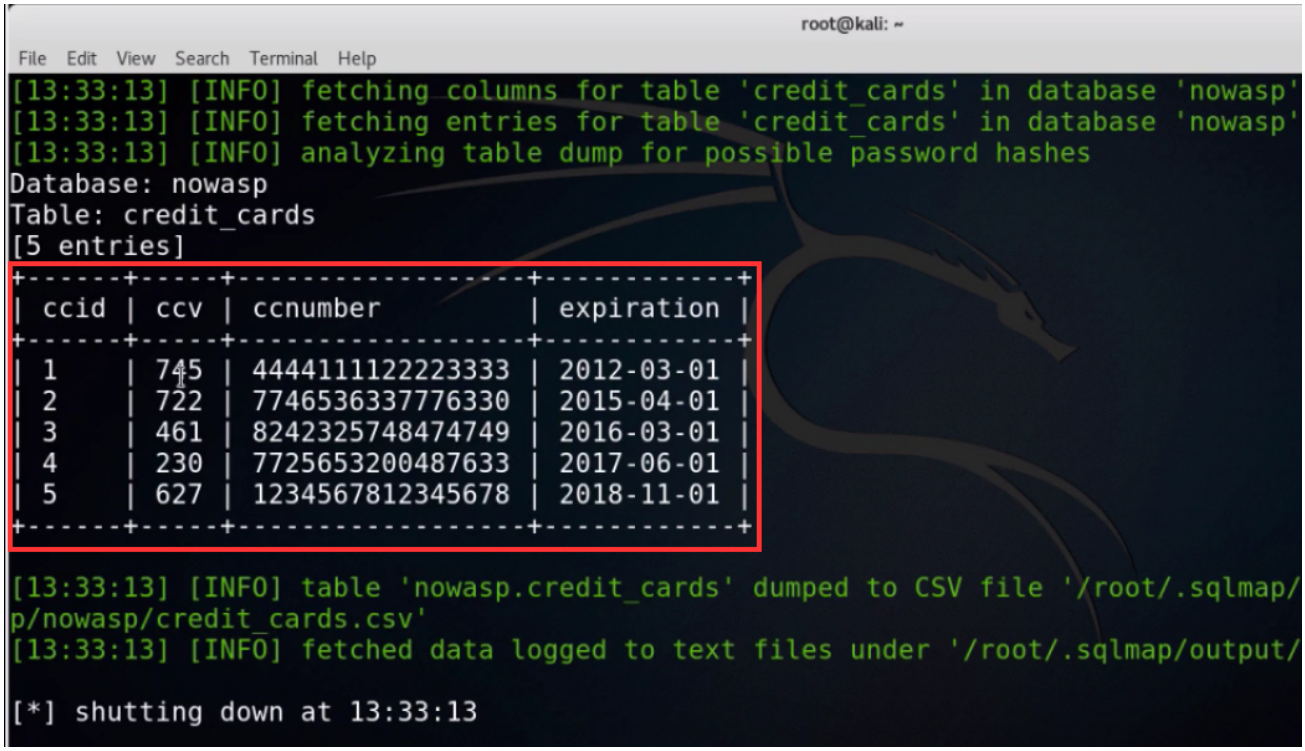
Teremos a seguinte informação:



O `nowasp` possui ao todo 12 tabelas, 11 a mais do que a *Account* descoberta primeiramente! Observe que uma delas tem o nome de `credit_cards`. Imagine uma loja grande ter um furo como esse? Nós chegamos em uma situação na qual através da inserção de códigos, conseguimos descobrir a existência de um banco chamado `credit_cards`. Um hacker desejará ter acesso aos demais dados dos cortoes e obtém isso facilmente utilizando o *dump*. Mais uma seta para cima e teremos o mesmo código, junto disso inserimos o `--dump -T credit_cards -D nowasp`. Assim, estaremos pedindo para que seja feito o `dump` da tabela `credit_cards` onde o banco trabalhado é o `nowasp`:

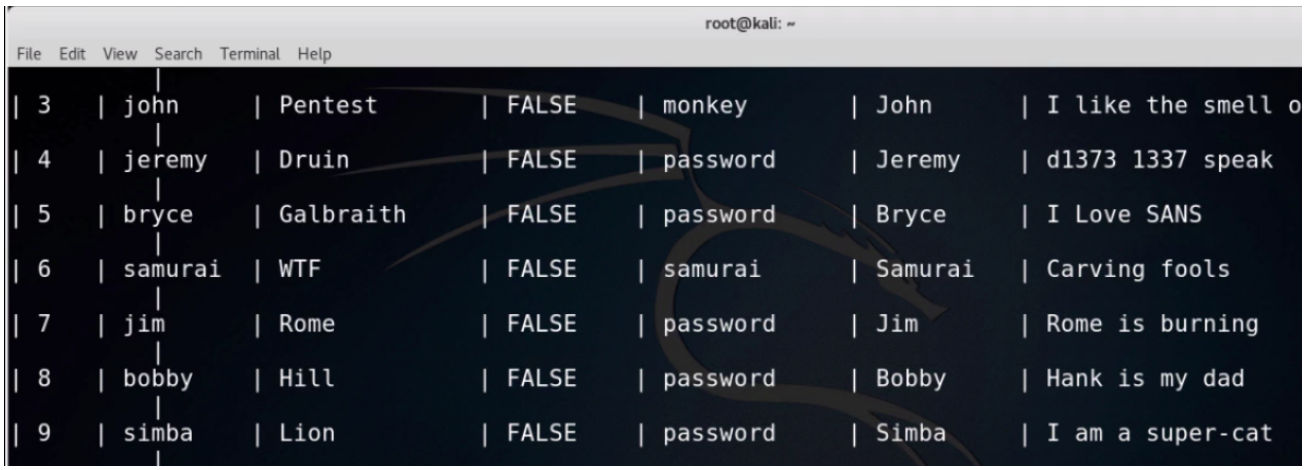
```
> sqlmap - u "http://192.168.1.37/mutillidae/index.php?page-user-info.php&username=admin&passwo
```

Dando um "Enter" nisso teremos:



Na imagem acima é possível observar que conseguimos acesso a uma série de informações de cartões de crédito que a empresa guardou no banco de dados! Normalmente, empresas que trabalham com dados deste tipo, possuem uma política de segurança extremamente rígida, nesse caso, é preciso melhorá-la!

Vamos retornar a tabela *Account* e confirmar se existem as sete colunas que tínhamos descoberto anteriormente. Para isso, damos uma seta para cima e alteramos o nome da tabela pretendida para: `--dump -T accounts -D nowasp` :



Dessa maneira, podemos comprovar que existem sete colunas e, ainda, conseguimos acessar as informações de todos os usuários do site. Ou seja, poderíamos utilizar qualquer usuário e senha para logar no sistema.

Utilizar injeções de SQL e de SQL Map é uma estratégia de verificação permitida apenas na nossa aplicação, fazer uso disso em outros sites da internet é ilegal, é **crime**. Seu uso deve ser limitado somente a aplicação vulnerável que desenvolvemos.

Vamos pensar em maneiras do desenvolvedor prevenir esses ataques! Para explicar utilizaremos exemplos da linguagem *javascript*:

```
String usuario=request.getParameter("usuario");

String senha=request.getParameter("senha");
```

Observe, na primeira `String` nós temos a requisição do usuário e na segunda a da senha. Uma vez com esses valores é preciso separá-los para não estarem vinculados a *query* dirigida ao banco, pois, é justamente esse o ponto que deixou o sistema tão frágil, veja:

```
String sql = "Insert into accounts (username.password) values (?,?)";
```

Dessa maneira, os parâmetros estão diretamente inseridos na *query* enviada ao banco. Assim, o desenvolvedor não verifica o que está sendo enviado e já sabemos que isso pode incorrer em um problema bastante grave! Por exemplo, dados de cartões de crédito facilmente disponíveis a um hacker.

Fazendo uma simples verificação é possível evitar um grande problema. Utilizando o `PreparedStatement` do *java* conseguimos separar os parâmetros que o usuário passa nos campos usuário e senha, conforme mostrado abaixo:

```
PreparedStatement stmt = connection.prepareStatement(sql);

stmt.setString(1,usuario);

stmt.setString(2,senha);

stmt.execute();
```