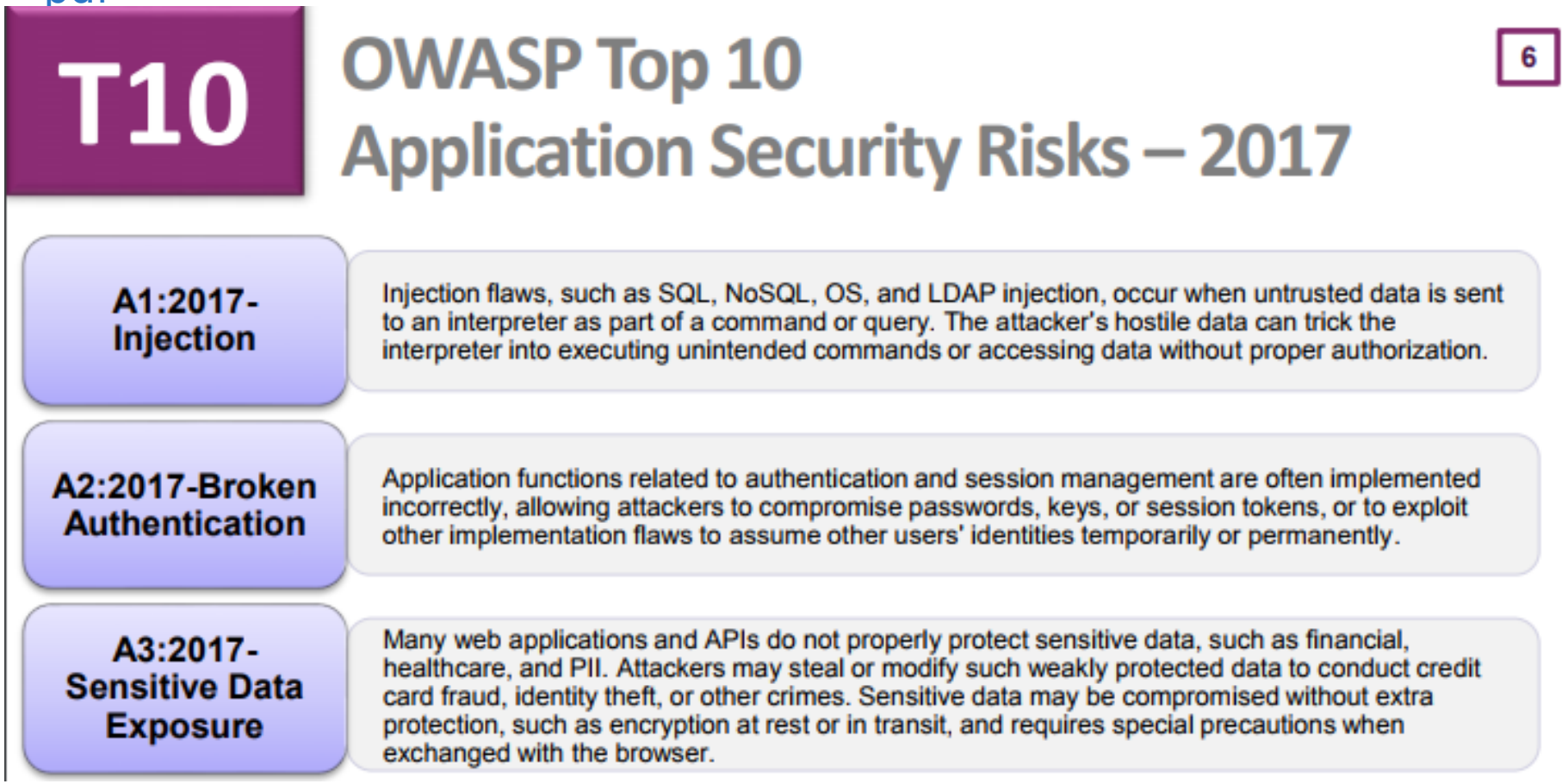
UNIVERSIDAD COMPLUTENSE MADRID

Inyección SQL

Gestión de la Información en la Web
Enrique Martín - emartinm@ucm.es
Grados de la Fac. Informática

¿Qué es?

- El **principal problema de seguridad según** el Top 10 de OWASP 2017: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf



Explicación

- Para acceder a la BD utilizamos **sentencias SQL**, tanto para leer como escribir.
- Casi siempre utilizamos alguna **entrada del usuario** en la sentencia SQL: nueva contraseña, término a buscar, id de producto, etc. → **el usuario participa en la sentencia**.
- Un usuario malicioso puede proporcionar datos formados de tal manera que convierten nuestra **consulta** en una **que hace algo diferente** (y posiblemente no deseado).

Ejemplo

- Tenemos una página que acepta un argumento *user* y muestra los pedidos de ese usuario.
  http://localhost:8080/orders?user=pepe
- La sentencia SQL construida sería:

```
user  = request.query['user'];
qbody = "SELECT * FROM orders WHERE user='{0}'"
query = qbody.format(user)
```

  - Para la petición anterior tendríamos:
    SELECT * FROM orders WHERE user='**pepe**'
- ¿Qué ocurriría con una petición como...?
  *http://localhost:8080/orders?user=**pepe' or 'a'='a***

Ejemplo

- Al recibir la petición *http://localhost:8080/orders?user=**pepe' or 'a'='a***
- La sentencia SQL generada sería:

```
user → “pepe' or 'a'='a”
query → “SELECT * FROM orders
         WHERE user='pepe' or 'a'='a'”
```

- Devolvería los datos de **todos los pedidos**, independientemente del usuario → *gran fuga de información*.

Ejemplo

- El problema radica en que no hemos escapado/evitado la comilla simple del argumento **user**.

```
"SELECT * FROM orders WHERE user='{0}'".format(user)
```

- Debemos comprobar que **user no contiene comillas simples.** De ser así hay que *escaparlas* o evitar la consulta.
  - Si hubiésemos escapado, la consulta sería:

```
query → "SELECT * FROM orders
          WHERE user='pepe\' or \'a\'=\'a'";
```

    es sintácticamente correcta → ***pero no devuelve ninguna información***.

Ejemplo

- O mejor aún, debemos utilizar los parámetros del método **execute()** ya que que este se encargará de escapar los caracteres peligrosos:

```
query = “SELECT * FROM orders WHERE user=':who'”
user = request.query['user']
conn = sqlite3('fichero.db')
cur = conn.cursor()
cur.execute(query, {'who': user})
```

Y puede ser peor...

- Hemos visto la inyección de código en consultas, pero también afecta a **modificaciones**:

```
age → 38
name → pepe' or '1'='1
query = “UPDATE usuarios SET edad={0}
          WHERE name='{1}'”.format(age, name)
```

puede producir sentencias SQL como

```
UPDATE usuarios SET edad=38
WHERE name='pepe' or '1'='1'
```

- También afecta a **inserciones** y **eliminaciones**.

E incluso peor...

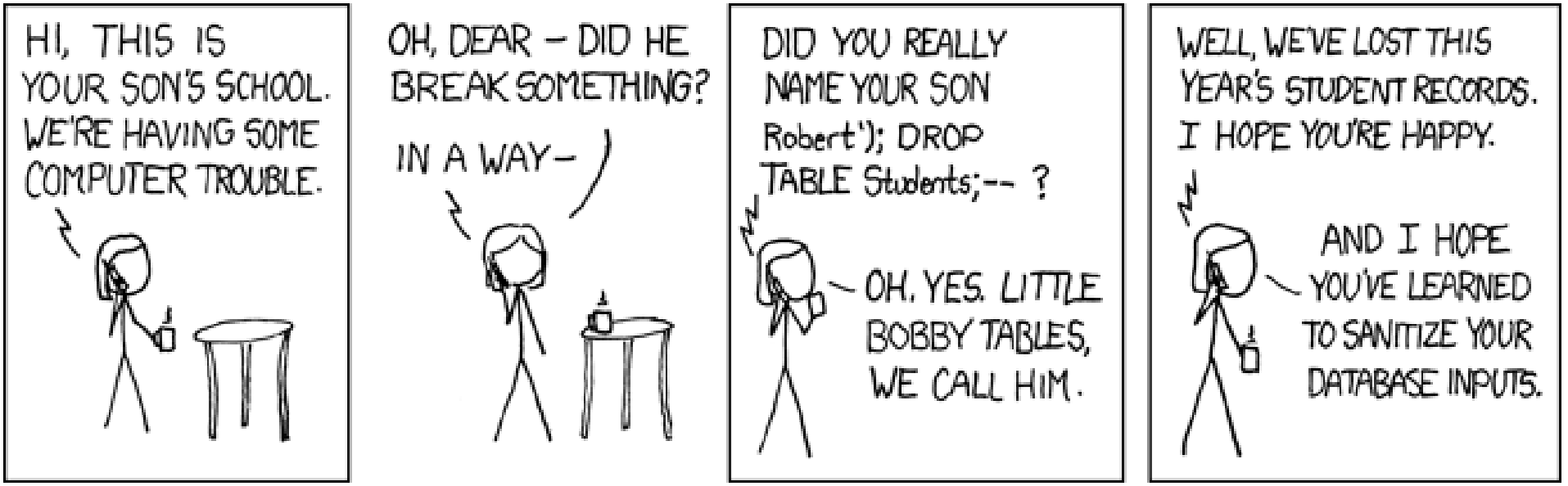
```
user  = request.query['user']
qbody = “SELECT * FROM orders WHERE user='{0}'”
query = qbody.format(user)
```

- Tomando `user → '; DROP TABLE orders; --'` generamos la consulta :

```
SELECT * FROM orders WHERE user='';
DROP TABLE orders; --''
```

- Afortunadamente la función **execute** (cuidado con **executescript**) de *sqlite3* solo permite ejecutar una sentencia SQL, pero versiones antiguas u otras BBDD podrían actuar de manera menos segura.

SQL injection en una viñeta



http://xkcd.com/327/

Prevenir inyecciones SQL

- **Ser realmente consciente**. Cualquier entrada puede acabar en una consulta SQL.
  - Hay que **revisar todas las entradas**, incluso:
    - Campos ocultos (*hidden*) de formularios.
    - Campos cuyo valor no puede estar mal formado porque se elige a través de elementos del formulario como: combobox, campos numéricos, etc.
  - GET: puedo generar una URL con cualquier valor para cualquier parámetro.
  - POST: utilizar **POST no protege nada**, cualquiera puede realizar una petición POST maliciosa.

Prevenir inyecciones SQL

- **Comprobar** los **tipos** de las entradas de usuario.
- **Escapar** todos los caracteres problemáticos.
- Desarrollar una **capa abstracta de seguridad** reutilizable entre proyectos. Se encargará de todas las entradas de los usuarios, e incluso de llamadas a la BD.

Prevenir inyecciones SQL

- Aplicar el principio de **mínimos privilegios** en la BD y **segregar usuarios**.
  - Utilizar diferentes usuarios para los distintos accesos a la base de datos, no un solo usuario omnipotente → **segregación**.
  - Los usuarios deben tener únicamente los privilegios necesarios para su tarea y **ninguno más**. ¿Por qué tener permiso de escritura en *usuarios* si solo voy a consultar la tabla *pedidos*?

Prevenir inyecciones SQL

- Utilizar **sentencias SQL preparadas**:

```
query = "SELECT * FROM orders WHERE
         user=':who' AND age=:age"
params = {'who': 'pepe', 'age': 39}

conn = sqlite.connect('file.db')
cur = conn.cursor()
cur.execute(query, params)
```

Inyección en otras BBDD

- Hasta ahora nos hemos centrado en la **inyección SQL**, es decir, en BBDD **relacionales**.
- La causa principal es que la **consulta** que se ejecuta en la BD es **texto plano** que se construye **concatenando** cadenas predefinidas y otras proporcionadas por el usuario.
- ¿Este problema de seguridad se puede presentar en otras BBDD? ¿Qué pasaría en MongoDB?

Inyección en MongoDB

- En principio parece más complicado porque las consultas MongoDB aceptan ***diccionarios*** como argumentos en lugar de ser **texto plano**:

```
query = { 'name': request.query['name'],
          'pass': request.query['password'] }

res = collection.find_one(query);
```

Inyección en MongoDB

- Sin embargo hay que tener mucho cuidado con las consultas que aceptan texto como **$regex**.
- Usuarios con nombre que termina con `sufijo` dado:

```
db.users.find({'_id':{'$regex': sufijo+'$' }})
```

- Correcto: sufijo = "pe" → "pe$"
- Peligroso: sufijo = "(.*)|e" → *"(.*)|e$" (cualquier texto o algo terminado en 'e')*

Inyección en MongoDB

- También hay que tener cuidado con **$where**.
- En este tipo de consultas se construye directamente código **JavaScript**.
- Argumentos cuidadosamente formados podrían generar código JS con comportamiento totalmente diferente al deseado.

Inyección en MongoDB

http://localhost:8080/getUsers?user=1

```
…
ident = request.query["user"]
jsbody = "this.a == this.b && this._id == {0}"
js = jsbody.format(ident)

c = MongoClient()
db = c.test
col = db.test
res = col.find({"$where": js})
…
```

- Consulta final:

```
{“$where”: "this.a == this.b && this._id == 1”}
```

Inyección en MongoDB

http://localhost:8080/getUsers?user=1||true

```
…
ident = request.query["user"]
jsbody = "this.a == this.b && this._id == {0}"
js = jsbody.format(ident)

c = MongoClient()
db = c.test
col = db.test
res = col.find({"$where": js})
…
```

- Consulta final:

```
{“$where”: "this.a==this.b && this._id == 1||true”}
```

Recordatorio: a && b || c → (a && b) || c

Referencias

Referencias

- SQL Injection Attacks by Example:
  - http://www.unixwiz.net/techtips/sql-injection.html
- SQL Injection – w3schools:
  - http://www.w3schools.com/sql/sql_injection.asp
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command:
  - http://cwe.mitre.org/data/definitions/89.html
- OWASP SQL Injection Prevention Cheat Sheet:
  - https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet