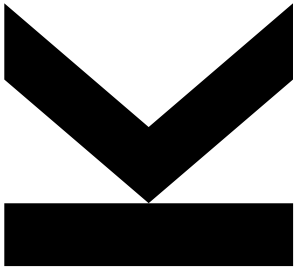
Injection attacks



JOHANNES KEPLER
UNIVERSITÄT LINZ

Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

Injection attacks

- An attacker sends some input to the server, which is incorrectly interpreted there
  - Idea: **Data** is provided, but is then executed as **command(s)**
- Typical examples: SQL/LDAP/XPath queries, OS commands, program arguments…
  - Can be seen as a kind of incorrect/missing input validation
- These are **very** common!
  - Mostly also **very easy to prevent**!
- The impact may be extremely severe: Typically the execution of arbitrary commands – **whatever is possible** where it is injected
- Basic problem:
  - Some data originates from an untrusted source (=client)
  - This data is not clearly and completely separated from data from a trusted source (e.g. source code, server configuration)

SQL injection

- User input is used as part of the input to a database
  - Usually these are SQL databases
    - But problem applies to all kinds of DBs, DB languages & inputs!
  - Typical examples: Login forms, search forms, other forms; URL parameters; other user input
- Concrete example: Search form
  - The following query is used in the software
    - `SELECT * FROM Articles WHERE Text LIKE '%"+searchword+"%';`
  - But what if someone enters the following search term:

    `'; DROP TABLE Articles;--`
    - "--<space>" at the end → Rest of line is comment!
  - Resulting query that will be executed: `SELECT * FROM Articles WHERE Text LIKE '%'; DROP TABLE Articles;-- %';`
    - Selects all articles; deletes the whole table; ignores a comment!
- Also potentially useful: Wildcards: % (multi-), _ (single-char)
- More data can be elicited through illegal SQL

SQL injection

- You can obviously also insert any data - which is interesting for XSS attacks, as input verification is subverted!
  - This doesn't go through any other input validation rules…
- You are typically not limited to the table used in the query
- Any commands are executed with the rights of the webserver
  - This is normally rather much
  - So make sure that your webserver receives as little permissions as possible
    - E.g. cannot read outside its "own" directories
    - "Containment": Separate application → Separate database → Separate user for accessing it through the webserver
    - (Read-only) views, but no table access
- Some special commands/syntax/… work only in some SW
  - Take care your escaping/… applies to **this** product & **this** version!

SQL injection

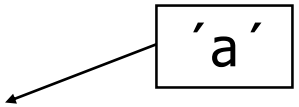
- Blind injection: SQL injection where the result is not immediately apparent to the attacker
  - Time delays: Query will take a long time if assumption is true
  - Conditional error: Error message as a result of the test
    - **`SELECT 1/0 FROM Users WHERE Username='admin';`**
      - Error only if such a user exists!
  - Conditional response: Result page will be somehow different
    - Errors are filtered out: Change SQL until the response is identical
      - Example**: `articleID=1 AND pwd LIKE 'a%'`**
      - Same article shown → Password begins with “a”!
  - Such attacks are difficult and time-consuming, but possible!
    - Note: The attacker can usually try for as long as he wants, with automated software, and usually undetected!

SQL injection

- MS SQL server is particularly dangerous:
  - Stored procedure **`master..xp_cmdshell`** can run any command (permissions: same as SQL Server service account!)
    - **Always** limit access to this procedure (and: **`sp_send_dbmail`**)!
  - Note: Disabled by default for security – must be explicitly enabled
  - Special provisions exist now to run this procedure under a different (→hopefully less permissions) account
- What can we learn from this? Some important questions:
  - Do you really need that functionality at all (sending mail from inside a database transaction via the database)?
    - Send the mail from your application – with data from the DB
  - Should it be enabled by default?
    - Enable only those things you really need and know about
  - Can we run it under different permissions?
    - This is additional work and needs more testing, but is imperative

SQL injection: Examples

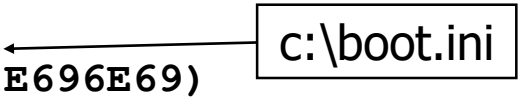
- Escaping from the escape filters:
  - `select * from login where user = char(39,97,39)`



- Finding column names:
  - Add the column(s) from the previous error messages
    - `' HAVING 1=1 --`
    - `' GROUP BY table.columnfromerror1 HAVING 1=1 --`
    - `' GROUP BY table.colfromerr1, colfromerr2 HAVING 1=1 --`
- Logging in:
  - `' OR 1=1 --` `admin´ #` `sa´ /*`
  - `' UNION SELECT 1,'user','xyz',1 --`
    - Note: Requires previous knowledge of the query structure!
  - MD5/hash verification (complex; program first retrieves user data by SQL, then hashes the password, and finally compares the result to the database data):
    - Enter username `admin' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055' --`
    - Enter password `1234`

SQL injection: Examples

- MS SQL Server specific
  - Reading files from the file system:
    - `CREATE TABLE aFile (line varchar(5000)); BULK INSERT aFile FROM 'path_to_file'; SELECT * FROM aFile" --`
  - Control Windows services:
    - `EXEC xp_servicecontrol stop, MSFTPSVC` → Stops FTP service
  - Shutdown server:
    - `';shutdown --`
- MySQL specific
  - Checking a table exists:
    - `IF (SELECT * FROM login) BENCHMARK(1000000,MD5(1))`
  - Read a file:
    - `SELECT LOAD_FILE(0x633A5C626F6F742E696E69)`



  - Version detection: `SELECT /*!32302 1/0, */ 1 FROM table`
    - Will cause an error if using MySQL and version > 3.23.02

SQL injection: Examples

- Using subqueries to use complex embeddings
  - `SELECT price FROM Articles WHERE ArtID=(SELECT 100+(SELECT COUNT(user) FROM mysql.user WHERE user=0x726f6f74))`
    - Counting the number of users with name “root”
    - Returns the price of article 100 if none, of 101 if one etc.
    - Can also be used for boolean (TRUE=1, FALSE=0) results
- Extracting data bitwise
  - `SELECT CONVERT(INT,SUBSTRING(password,I,1)) & N FROM master.dbo.sysxlogins WHERE name LIKE 0x73006100`
  - Extracting bit **N** (1, 2, 4 … 128) of character at position **I** (1-?) of the “sa”-user (0x7300=‘s’, 0x6100=‘a’) password as boolean result
    - Would probably have to be executed as a subquery; see above!
  - Requires a loop executed 8 times for each character

SQL injection: Examples

- Combining results, e.g. from subqueries through UNION
  - Add at end " UNION SELECT …"
  - Disadvantage: Must have exactly same number of columns!
    - Needs to be mapped before: Just increase until no error
    - Too few? Add more (take care of column type!)
      - `UNION SELECT a,b,c,1,"",TRUE FROM` …
      - `UNION SELECT a,b,c,a,b,c FROM` …
      - `UNION SELECT a,b,c,NULL,NULL,NULL FROM` …
    - Too many? Use CONCAT to reduce them!
      - `UNION SELECT CONCAT(a,b,c) FROM` …
      - `UNION SELECT CONCAT(a,"-",b,"-",c) FROM` …
    - Wrong column type? The "CAST()" function might help
      - Datatype conversion function (see also "CONVERT")
      - "CAST(" expression "AS" datatype ["(" length ")"] ")"
      - E.g. `UNION SELECT 1,CAST(user AS CHAR(30)),1 FROM` …
        - Expected: Integer – Char(30) – Integer

SQL injection: Examples

- More difficult: Integrated hash function
  - $user = mysql_real_escape_string($_POST['user']);
    $pass = md5($_POST['pass'], true);
    $sql_s = "SELECT * FROM users WHERE username='$user' and password='$pass'";
- Can this be exploited by injection?
  - Theoretically: No. It would involve reversing the hash function

SQL injection: Examples

- Practically: Yes!
  - Definition: string **md5**(string $str **[, bool $raw_output = false]**)
    - Note: the “modern” function hash(string $algo , string $data [, bool $raw_output = FALSE]) has the same problem…
  - Generate random values to hash and check whether the output contains a “suitable” binary string (raw_output=true → binary text!)
    - Suitable: <…>’ OR 1=1; -- → But this would take years!
    - Optimized: ‘||’? or ‘OR’?
      - String starting with any number between 1 to 9 is interpreted as integer when used as boolean; it is not zero → converted to true
      - ||, OR, or, Or, oR → Equivalent (speed up in finding)

- Where is the problem?
  - We are using/comparing binary data
    - $pass = md5($_POST['pass']**, true**);
  - This is a very bad idea, as in hex encoding, there will never be any “strange” characters – but here anything might appear…
    - But see below – even this is no guarantee if you mess up only slightly

SQL injection: Examples

- Example:
  - password: 129581926211651571912466741651878684928
  - Tries to find: 18.933.549
  - Hex of MD5: 06da5430449f8f6f23dfc1276f722738
  - Text of MD5 (`md5("...",true)`): ??T0D??o#??**'or'8**
    - Question marks are unprintable characters
    - Console shows this as: ▒T0D▒▒o#▒▒'or'8

- Another problematic program example:
  - Text comparison: md5('240610708') == md5('QNKCDZO')
  - This results in TRUE. Why?
    - md5('240610708') = 0e462097431906509019562988736854
    - md5('QNKCDZO') = 0e830400451993494058024219903391
    - Both start with 0e → interpreted as numbers $0^{462\ldots854}$ resp. $0^{830\ldots391}$!
    - 0^x is always the same as 0^y, so both are the same!
  - Solution: Use "===" (strict comparison/identity operator)

Sources: http://cvk.posthaven.com/sql-injection-with-raw-md5-hashes
https://www.php.net/manual/de/function.md5.php

SQL injection: Detection

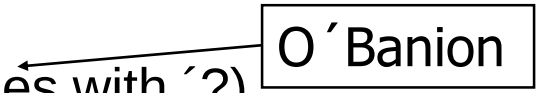
- Code inspection: You need to know what to look for
  - Advantage: Check for using specific “procedures” (like constructing queries as strings), not individual problems (like an incorrect query statement)
- Fuzzing tools:
  - Inspecting forms automatically
  - Submitting form with random modifications/inserted data
  - Verifying output **and DB** (here automation is problematic!)
- Data flow analysis tools
  - Traces data from its source to where it is contained
  - See also “tainting”!
    - Input data is marked as “tainted” with a flag, this is passed on through all uses of a variable and checked in “dangerous” calls
    - Problem: Speed impact, complexity, false positives

SQL injection: Detection

- How to check whether a form is vulnerable:
  1. Find a form in the website with parameters
     - E.g. http://www.site.com/show.php?id=1
     - `´SELECT field FROM table WHERE ID = ´+id+´;´`
  2. (Try to) Inject a query which is certainly empty:
     - http://www.site.com/show.php?id=`1 and 1=2`
       - Note: URL escaping removed here (actually: `id=1%20and%201=2`)!
     - `´SELECT field FROM table WHERE ID = 1 and 1=2;´`
       - Empty result set → Nothing shown (not: error message!)
  3. (Try to) Inject a query which is certainly not empty:
       - This step: Just to make sure!
     - http://www.site.com/show.php?id=`1 and 1=1`
     - `´SELECT field FROM table WHERE ID = 1 and 1=1;´`
       - Result should be the same as in step ❶
       - Shows it was actually executed as SQL and not escape/removed!
  - Result: We know that this form is susceptible to injection
    - We can do whatever we want; no need to search for other forms!

SQL injection: Prevention

- Escaping **'** and **;** are good, but insufficient!
  - Techniques exist to "live without" or use other options
    - Just removing them? → uni'on sel'ect @@version-'-
    - See examples for “char(…”; also: “CONCAT(…, …, …)”
  - You should do it, but never rely on it
- Verify all input data according to a whitelist
  - And strictly enforce length limits → SQL injection is **usually** (but not always!) a long string to be of use
  - Verify which characters **may** occur (e.g. names with ´?)



- Verify data type of everything submitted
- Limit database permissions
  - DB itself should always be separate user with least privileges
  - Each application should have its own DB and user
    - And each application accessing it should also have it’s own user
    - E.g.: Backend (→ write permissions); public frontend (read only on some special views containing only relevant columns)

SQL injection: Prevention

- Parameterized queries
  - Do not construct queries as string by concatenation
  - Prepare all queries & call them with user content as parameter
    - All data is automatically "escaped" → Parameters are **always** and **only** pure data, never commands (or their elements)
    - Note: E.g. XSS is **not** prevented by this, only DB modifications!
  - Trivial and works perfectly (no SQL injection possible at all!)
- Use stored procedures:
  - Like parameterized queries, but “query” is stored in DB
  - Potential danger: You can use other commands in these stored procedures as well
    - E.g. concatenating input to a string to produce a query…
  - If taking care/not doing strange things this is exactly as safe (=perfect) as parameterized queries!

SQL injection: PHP example

- Login form:

```
<form method="post" action="login.php">
    <input type="text" name="login" id="login" ><br />
    <input type="text" name="password" id="password" ><br />
    <input type="submit" value="Login!" />
</form>
```

- Vulnerable code:

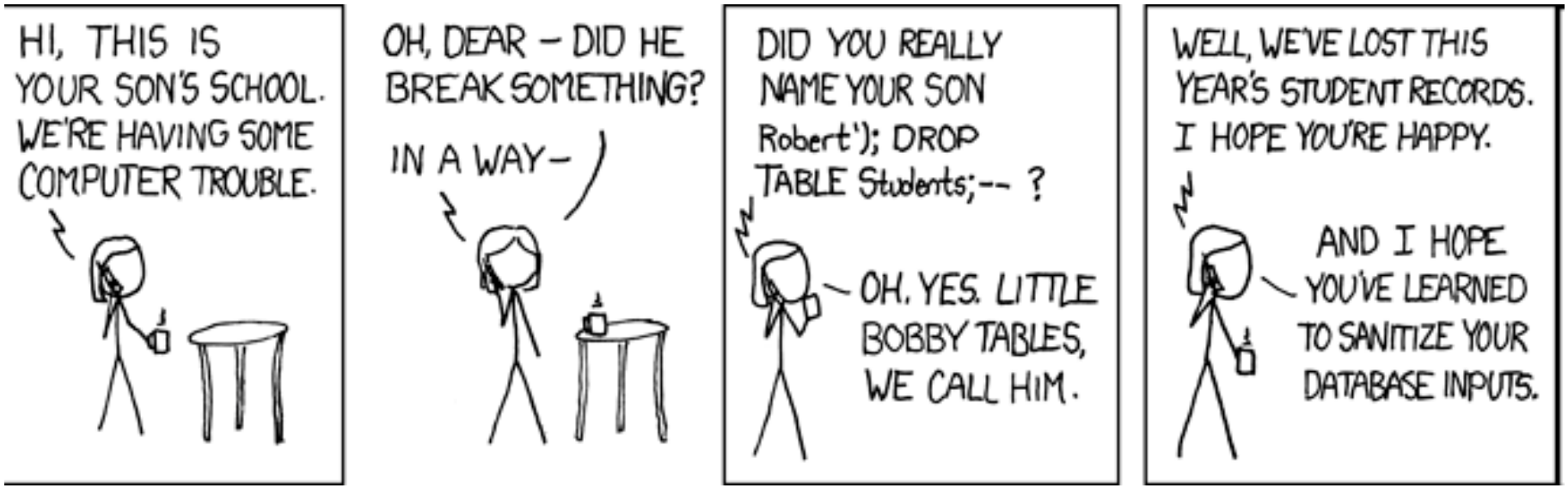
```
$query="SELECT id,login FROM user WHERE login='".$login."' AND
    password='".$password."';";
$result=$conn->query($query);
```

- Corrected/Secure code:

```
$stmt=$conn->prepare("SELECT id,login,password FROM user
    WHERE login=? AND password=?;");
$stmt->bindParam(1,$login,PDO::PARAM_STR);
$stmt->bindParam(2,$password,PDO::PARAM_STR);
$res=$stmt->execute();
```

Or first line and `$res=$stmt->execute(array($login,$password));`

SQL injection: Paper based ☺



Source: xkcd: Exploits of a Mom, http://xkcd.com/327/

SQL injection: Car based ☺

Source: http://cache.gizmodo.de/wp-content/uploads/2010/03/for_traffic_cameras.jpg

XPath injection

- Same as SQL, but different database: XML files/XML-DB
  - Disadvantage: Comments are not possible. The query must always be correct as a whole!
- Example:
  - Original query: xmlobj.selectNodes(“ //users/admins/[user/text()=‘ ”+username+” ‘ and pass/text()=‘ ”+password+” ‘] ”)
  - Injection (empty password): **`admin' or 'a'='b`**
    - Note the non-terminated string at the end!
    - To log in as user “admin” (to log on in anyway insert “ or 1=1 “!)
  - Resulting query: //users/admins/[user/text()=‘admin’ or ‘a’=‘b’ and pass/text()=‘’]
- Other examples: //* (all elements everywhere), */* (current level and all children), @/ (attributes), count(//*) (number of elements)

XPath injection: Prevention

- Filter out all dangerous input: ( ) < > [ ] / ^ “ * ‘ ; & . @ : = ! $
  - Or use escaping for them, especially for ‘ and “
- Validate type, format, length, and range
- (Potential) problem: No parameterized queries in XPath!
  - But: Use XQuery or other APIs (e.g. XPathCache)
    - They do exactly the same as parameterized queries. However, they are not standardized and depend on the language/platform/…
    - Reduces the portability of the code, but is strongly suggested!
- Take special care if XPath 2.0 is supported: It might access even other documents!

Injection variant: Mail headers

- The user can enter an E-Mail address, to which some data will be sent (recommendation etc)
  - E.g. just printing the user input as the source address
  - Possible input for sender address: "sender@junk.com\nRCPT TO: rec1@org, rec2@org\nDATA\nSpam message\n.\nQUIT\n"
  - This will result in a "strange" SMTP session!
- Whenever the user enters something which ends up in a protocol, a similar attack becomes possible
  - See later: HTTP response splitting (same idea with HTTP!)
- Basic idea: Send data which is then interpreted as part of the protocol to perform (=i.e. as a command)
- How to prevent: Make sure that the data is ONLY data!
  - And doesn't contain line breaks, tabs etc

Mail header injection

- SMTP protocol log (with user supplied data/**attack**):
  - 220 mail.app.com ESMTP Sendmail; Wed, 22 Apr 2015 12:26:03 +0200
  - EHLO localhost
  - 250-mail.app.com Hello [192.168.0.1], pleased to meet you
  - MAIL FROM: **sender@junk.com**
  - **RCPT TO: rec1@org, rec2@org**
  - **DATA**
  - **<Spam message>**
  - **.**
  - **QUIT**
  - RCPT TO: useless@nodomain.com
  - DATA
  - Subject: Recommendation for you!
  - .....

Script injection

- Data can also be sent to the application, which might later be interpreted as server-side code, e.g. JSP/ASP/PHP/… code
  - Note: This is rather rare, as normally the page is compiled/interpreted before user input is involved!
- Examples:
  - Injecting “%> or “?> to close a script tag → Rest of page is not HTML but rather the “source code” of the page
    - Start tag (“’<%” or “”<?”) are even more dangerous as code can be injected directly!
  - Injecting server-side-include commands, like
    - <!--#include file=“/etc/passwd” →
- Causes:
  - Input is echoed as output (but usually must be done indirectly, e.g. writing constructed output to a file, then including this new file)
  - Most likely: When tables or forms are constructed dynamically

Script injection

- Other avenues for injecting commands/scripts:
  - Cookies: If its content value ends up on the page
    - This should never be necessary; use cookies merely for storing an unique random id with the data located on the server!
  - User agent: Many servers produce different output based on the User-Agent string passed in the HTTP header
    - If this UA string is unknown and echoed on the error page, we can inject JavaScript into a page coming from the server!
- Basic idea: **Anything** coming from the client might be used as a vehicle for injecting commands or scripts!
- Countermeasures:
  - Be careful what you accept and take care of escaping
  - Make sure to cover **all** sources of data!
    - Form input, parameters, HTTP headers, cookies, client certificates (e.g. echoing the name in it!)…

OS command injection

- Example code (PHP):
  - $username = $_COOKIE['username'];
  - exec("wto -n \"$username\" -g", $ret);
- Explanation:
  - Retrieve the username from the cookie
    - Note: User can send as cookie whatever he wants!
    - Potential problem: Must get across a login check with a malicious username (but: Need not be actual user name; just display name!)
      - But: Log in normally, then use GUUID from cookie for session and different user name in cookie for attack (if username is in too)
  - Execute check of it on a command line
    - Example: Set username to '$(touch /tmp/file)' (without quotes)
    - Result: Execute **wto -n "$(touch /tmp/file)" –g**
      - Semantics of $(…): Execute this first, then fill in its output and execute rest of command
- Worst: Webserver is running as root…
  - So all commands are executed as exactly this user!

https://blog.exploitee.rs/2017/hacking_wd_mycloud/

OS command injection

- Solution to a security problem: Use escaping
  - A good and correct idea, but you must do it right
- Checking whether a user is already logged in
  - exec(sprintf("wto -n \"%s\" -i '%s' -c", escapeshellcmd($username), $_SERVER["REMOTE_ADDR"]), $login_status);
- What is the problem? We **did** escape the user-supplied data!
  - Yes, but look at the function used…
- Escaping command lines in PHP
  - escapeshellcmd: Use for complete command line
  - escapeshellarg: Use for argument in command lines
    - See PHP documentation:

> **Warning** **escapeshellcmd()** should be used on the whole command string, and it still allows the attacker to pass arbitrary number of arguments. For escaping a single argument escapeshellarg() should be used instead.

https://blog.exploitee.rs/2017/hacking_wd_mycloud/

OS command injection

- escapeshellcmd/escapeshellarg: Difference
- Example:
  - `$path="/dir/".$input;`
    `$res=shell_exec("ls ".$path);`
- Input = "mySubDir" → "`ls /dir/mySubDir`" → OK
- Input = "mySubDir; rm –Rf /" → "`ls /dir/mySubDir; rm –Rf /`"
  - `$res=shell_exec(escapeshellcmd("ls ".$path));`
    - Result: "`ls /dir/mySubDir\; rm –Rf /`"
    - "rm" and "-Rf" and "/" would all be parameters for "ls"
    - So no deletion anymore, but potentially unwanted results
      - Depends on what the command allows in parameters (e.g. if `ls` had a "and delete those files" parameter)
  - `$res=shell_exec("ls ".escapeshellarg($path));`
    - Result: "`ls '/dir/mySubDir; rm –Rf /'`"
    - Lists a directory with a strange name, which likely doesn't exist

Java Injection

- Java code is injected, which is then interpreted on the server
- Example: OGNL (Object Graph Navigational Language)
  - Expression language, which is a “script” interpreted on server
- Security problems were especially prevalent in Apache Struts (a web framework)
  - Programmer uses a “normal” function
  - But some function parameter passes its text to OGNL interpreter
  - Example:

```
public String getPageTitle() {
    return getText(pageTitle);
}
```

```
public void setPageTitle(String pageTitle) {
    this.pageTitle = pageTitle;
}
```

    - getText(…) is such a dangerous function evaluating OGNL
    - Exploit: Request “!getPageTitle?pageTitle=… some OGNL…”
      - Somehow set page title to specific exploit string (still harmless!)
        - No escaping in “setPageTitle(…)”!
      - Web access → framework retrieves page title → calls “getPageTitle()” → calls “getText(…)” inside → attack is executed!

Example: http://security.coverity.com/advisory/2013/Oct/remote-code-execution-in-apache-roller-via-ognl-injection.html

Escaping

Countermeasures: Escaping

- "Escaping": Changing the meaning of text (in this context!)
  - Characters that have a special meaning (like "SQL command", "HTML tag"…) are "converted" back to normal text
    - Making sure data is only data and not something else (=commands), even if it looks like that
- Typical ways of escaping:
  - Doubling the character: % → %%
    - printf("%d %%d", i) → "5 %d"
  - Prepending with a specific sign: \"
    - System.out.println("Some \""); → Some "
  - Quoting (surrounding with special characters): "…"
    - SELECT * from table WHERE name="TOP OR SELECT™"
- Problem: Where is it going to end up?
  - Is it to inserted into an SQL statement? → Escaping method A
  - Is it to be printed on a webpage? → Escaping method B
  - Is it to be written to a CSV file? → Escaping method C

Countermeasures: Escaping

- What if we don't know yet or there are multiple conflicting ways?
  - Store as original (but perhaps escape for storing ☺) and escape on actual use according to where it is used
    - But be careful not to forget…
    - Useful if we often need it in different escapings
  - Store escaped for the typical use and later un-escape and re-escape for different uses if necessary
    - More secure if we only rarely need it in a different escaping

- Which one is better is a matter of opinion and circumstances

- The important parts are:
  - Do not forget to escape
  - Make sure to use the right form of escaping for the target
  - Do not escape twice ("just to be sure!")
    - Might get malicious things across filters!

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at