

# Advanced PHP Security

By: *Ilia Alshanetsky*

# What is Security?

- Security is a measurement, not a characteristic.
- It's is also an growing problem that requires an continually evolving solution.
  - A good measure of secure application is it's ability to predict and prevent future security problems, before someone devises an exploit.
- As far as application design goes, security must be considered at all times; initial spec, implementation, testing and even maintenance.

# PHP & Security

- PHP keeps on growing as a language, making headway into enterprise and corporate markets.



- Consequently PHP applications often end up working with sensitive data.
  - Unauthorized access to this data is unacceptable.
  - To prevent problems a secure design is needed.

# Input Validation

- One of the key concepts you must accept is that user input is unreliable and not to be trusted.
  - Partially lost in transmission between server & client.
  - Corrupted by some in-between process.
  - Modified by the user in an unexpected manner.
  - Intentional attempt to gain unauthorized access or to crash the application.
- Which is why it is absolutely essential to validate any user input before use.

# Accessing Input Data

- As of PHP 4.1, there are a series of super-globals that offer very simple access to the input data.
  - `$_GET` – data from get requests.
  - `$_POST` – post request data.
  - `$_COOKIE` – cookie information.
  - `$_FILES` – uploaded file data.
  - `$_SERVER` – server data
  - `$_ENV` – environment variables
  - `$_REQUEST` – combination of GET/POST/COOKIE



# Register Globals

- Arguably the most common source of vulnerabilities in PHP applications.
  - Any input parameters are translated to variables.
    - `?foo=bar >> $foo = "bar";`
  - No way to determine the input source.
    - Prioritized sources like cookies can overwrite GET values.
  - Un-initialized variables can be “injected” via user inputs.



# Register Globals

```
if (authenticated_user()) {  
    $authorized = true;  
}  
if ($authorized) {  
    include '/highly/sensitive/data.php';  
}
```

- Because `$authorized` is left un-initialized if user authentication fails, an attacker could access privileged data by simply passing the value via GET.

<http://example.com/script.php?authorized=1>



# Solutions To Register Globals

- Disable `register_globals` in `PHP.ini`.
  - Already done by default as of PHP 4.2.0
- Code with `error_reporting` set to `E_ALL`.
  - Allows you to see warnings about the use of un-initialized variables.
- Type sensitive validation conditions.
  - Because input is always a string, type sensitive compare to a Boolean or an integer will always fail.



```
if ($authorized === TRUE) {
```



# Hidden Register Globals Problems

```
$var[] = "123";  
foreach ($var as $entry) {  
    make_admin($entry);  
}  
  
script.php?var[]=1&var[]=2
```

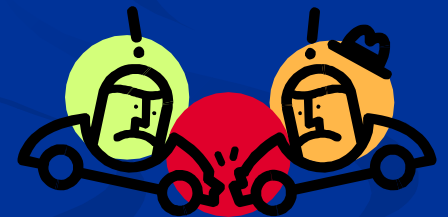
The link above will allow the attacker to inject two values into the `$var` array. Worse yet PHP provides no tools to detect such injections.

# \$\_REQUEST

- The `$_REQUEST` super-global merges data from different input methods, like `register_globals` it is vulnerable to value collisions.

```
PHP.ini: variables_order = GPCS
```

```
echo $_GET['id']; // 1  
echo $_COOKIE['id']; // 2  
echo $_REQUEST['id']; // 2
```



# `$_SERVER`

- Even though the `$_SERVER` super-global is populated based on data supplied by the web-server it should not be trusted.
  - User may inject data via headers  
`Host: <script> ...`
  - Some parameters contain data based on user input  
`REQUEST_URI, PATH_INFO, QUERY_STRING`
  - Can be fakes  
Spoofed IP address via the use of anonymous proxies.

# Numeric Value Validation

- All data passed to PHP (GET/POST/COOKIE) ends up being a string. Using strings where integers are needed is not only inefficient but also dangerous.

```
// integer validation
if (!empty($_GET['id'])) {
    $id = (int) $_GET['id'];
} else
    $id = 0;
// floating point number validation
if (!empty($_GET['price'])) {
    $price = (float) $_GET['price'];
} else
    $price = 0;
```

- Casting is a simple and very efficient way to ensure variables do in fact contain numeric values.



# Validating Strings

- PHP comes with a `ctype`, extension that offers a very quick mechanism for validating string content.

```
if (!ctype_alnum($_GET['login'])) {  
    echo "Only A-Za-z0-9 are allowed."  
}  
  
if (!ctype_alpha($_GET['captcha'])) {  
    echo "Only A-Za-z are allowed."  
}  
  
if (!ctype_xdigit($_GET['color'])) {  
    echo "Only hexadecimal values are allowed";  
}
```

# Path Validation

- Values passed to PHP applications are often used to specify what file to open. This too needs to be validated to prevent arbitrary file access.

```
http://example.com/script.php?path=../../etc/passwd
```

```
<?php
```

```
$fp = fopen("/home/dir/{$_GET['path']}", "r");
```

```
?>
```



# Path Validation

- PHP includes a `basename()` function that will process a path and remove everything other than the last component of the path, usually a file name.

```
<?php
```

```
$_GET['path'] = basename($_GET['path']);
```

```
// only open a file if it exists.
```

```
if (file_exists("/home/dir/{$_GET['path']}")) {  
    $fp = fopen("/home/dir/{$_GET['path']}", "r");  
}
```

```
?>
```



# Better Path Validation

- An even better solution would hide file names from the user all together and work with a white-list of acceptable values.

```
// make white-list of templates
$tmp1 = array();
foreach(glob("templates/*.tmpl") as $v) {
    $tmp1[md5($v)] = $v;
}
if (isset($tmp1[$_GET['path']]))
    $fp = fopen($tmp1[$_GET['path']], "r");
```

<http://example.com/script.php?path=57fb06d7...>





# magic\_quotes\_gpc

- PHP tries to protect you from attacks, by automatically escaping all special characters inside user input. ( `\`, `"`, `\`, `\0` (NULL) )
  - Slows down input processing.
    - We can do better using casting for integers.
    - Requires 2x memory for each input element.
  - May not always be available.
    - Could be disabled in PHP configuration.
  - Generic solution.
    - Other characters may require escaping.



# Magic Quotes Normalization

```
if (get_magic_quotes_gpc()) { // check magic_quotes_gpc state
    function strip_quotes(&$var) {
        if (is_array($var)
            array_walk($var, 'strip_quotes');
        else
            $var = stripslashes($var);
    }

    // Handle GPC
    foreach (array('GET','POST','COOKIE') as $v)
        if (!empty($_{"_".$v}))
            array_walk($_{"_".$v}, 'strip_quotes');

    // Original file names may contain escaped data as well
    if (!empty($_FILES))
        foreach ($_FILES as $k => $v) {
            $_FILES[$k]['name'] = stripslashes($v['name']);
        }
}
```

# Exploiting Code in Previous Slide

- While the code on the previous slide works, it can be trivially exploited, due to its usage of recursive functions!

```
<?php
```

```
$qry = str_repeat("[", 1024);
```

```
$url = "http://site.com/script.php?a{$qry}=1";
```

```
file_get_contents($url);
```

```
// run up in memory usage, followed by a prompt  
crash
```

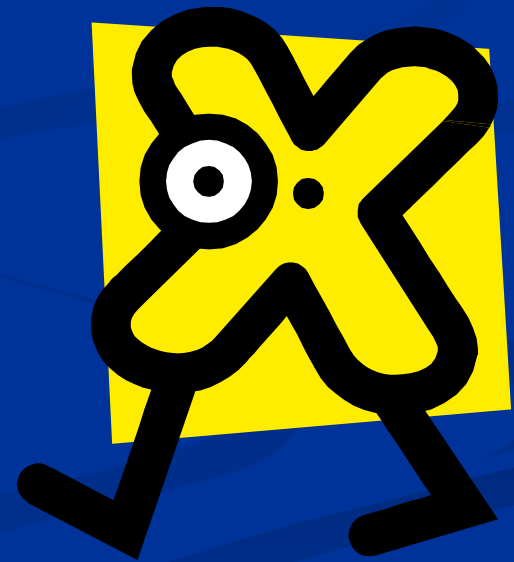
```
?>
```

# More Reliable & Faster Solution

```
if (get_magic_quotes_gpc()) {  
    $in = array(&$_GET, &$_POST, &$_COOKIE);  
    while (list($k,$v) = each($in)) {  
        foreach ($v as $key => $val) {  
            if (!is_array($val)) {  
                $in[$k][$key] = stripslashes($val);  
                continue;  
            }  
            $in[] =& $in[$k][$key];  
        }  
    }  
    unset($in);  
}
```

# XSS

- Cross Site Scripting (XSS) is a situation where by attacker injects HTML code, which is then displayed on the page without further validation.
  - Can lead to embarrassment.
  - Session take-over.
  - Password theft.
  - User tracking by 3<sup>rd</sup> parties.



# Preventing XSS

- Prevention of XSS is as simple as filtering input data via one of the following:
  - `htmlspecialchars()`
    - Encodes ‘, “, <, >, &
  - `htmlentities()`
    - Convert anything that there is HTML entity for.
  - `strip_tags()`
    - Strips anything that resembles HTML tag.

# Preventing XSS

```
$str = strip_tags($_POST['message']);  
// encode any foreign & special chars  
$str = htmlentities($str);  
// maintain new lines, by converting them to <br />  
echo nl2br($str);  
  
// strip tags can be told to "keep" certain tags  
$str = strip_tags($_POST['message'], '<b><p><i><u>');  
$str = htmlentities($str);  
echo nl2br($str);
```

- Tag allowances in `strip_tags()` are dangerous, because attributes of those tags are not being validated in any way.

# Tag Allowance Problems

```
<b style="font-size: 500px">
```

```
TAKE UP ENTIRE SCREEN
```

```
</b>
```

```
<u onmouseover="alert('JavaScript is allowed');">
```

```
<b style="font-size: 500px">Lot's of text</b>
```

```
</u>
```

```
<p style="background: url  
  (http://tracker.com/image.gif)">
```

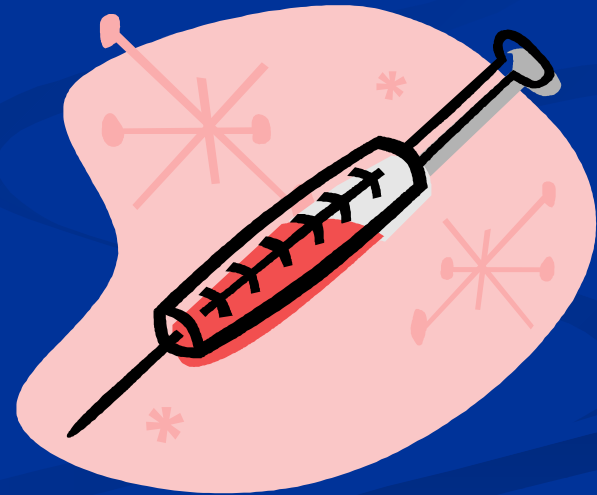
```
Let's track users
```

```
</p>
```



# SQL Injection

- SQL injection is similar to XSS, in the fact that not validated data is being used. But in this case this data is passed to the database.
  - Arbitrary query execution
    - Removal of data.
    - Modification of existing values.
    - Denial of service.
    - Arbitrary data injection.



# SQL Escaping

- If database interface extension offers dedicated escaping functions, **USE THEM!**
  - MySQL
    - `mysql_escape_string()`
    - `mysql_real_escape_string()`
  - PostgreSQL
    - `pg_escape_string()`
    - `pg_escape_bytea()`
  - SQLite
    - `sqlite_escape_string()`



# SQL Escaping in Practice

```
// undo magic_quotes_gpc to avoid double escaping
if (get_magic_quotes_gpc()) {
    $_GET['name'] = stripslashes($_GET['name']);
    $_POST['binary'] = stripslashes($_POST['binary']);
}
```

```
$name = pg_escape_string($_GET['name']);
$binary = pg_escape_bytea($_POST['binary']);
```

```
pg_query($db, "INSERT INTO tbl (name,image)
              VALUES ('{$name}', '{$image}')");
```



# Escaping Shortfall

- When un-quoted integers are passed to SQL queries, escaping functions won't save you, since there are no special chars to escape.

```
http://example.com/db.php?id=0;DELETE%20FROM%20users
```

```
<?php
```

```
$id = sqlite_escape_string($_GET['id']);
```

```
// $id is still 0;DELETE FROM users
```

```
sqlite_query($db,
```

```
    "SELECT * FROM users WHERE id={$id}");
```

```
// Bye Bye user data...
```

```
?>
```



# Prepared Statements

- Prepared statements are a mechanism to secure and optimize execution of repeated queries.
- Works by making SQL “compile” the query and then substitute in the changing values for each execution.
  - Increased performance, 1 compile vs 1 per query.
  - Better security, data is “type set” will never be evaluated as separate query.
  - Supported by most database systems.
  - MySQL users will need to use version 4.1 or higher.
  - SQLite extension does not support this either.

# Prepared Statements

```
<?php
$data = "Here is some text to index";

pg_query($db, "PREPARE my_stmt (text) AS
              INSERT INTO search_idx (word) VALUES ($1)");
foreach (explode(" ", $data) as $word) {
    // no is escaping needed
    pg_query($db, "EXECUTE my_stmt({$word})");
}

// de-allocate the prepared statement
pg_query($sb, "DEALLOCATE my_stmt");
```

- ?> Unless explicitly removed, prepared statements “stay alive” between persistent connections.

# Error Reporting

- By default PHP will print all errors to screen, startling your users and in some cases disclosing privileged information.
  - File paths.
  - Un-initialized variables.
  - Sensitive function arguments such as passwords.
- At the same time, disabling error reporting would make bug tracking near impossible.

# Solution?

- This problem can be solved by disabling displaying of error messages to screen

```
ini_set("display_errors", FALSE);
```

- And enabling logging of errors

```
ini_set("log_errors", TRUE);
```

- to a file

```
ini_set("error_log", "/var/log/php.log");
```

- or to system central error tracking facility

```
ini_set("error_log", "syslog");
```



# File Security

- Many PHP applications often require various utility and configuration files to operate.
- Because those files are used within the application, they end up being world-readable.
- This means that if those files are in web directories, users could download & view their contents.

# Securing Your Files

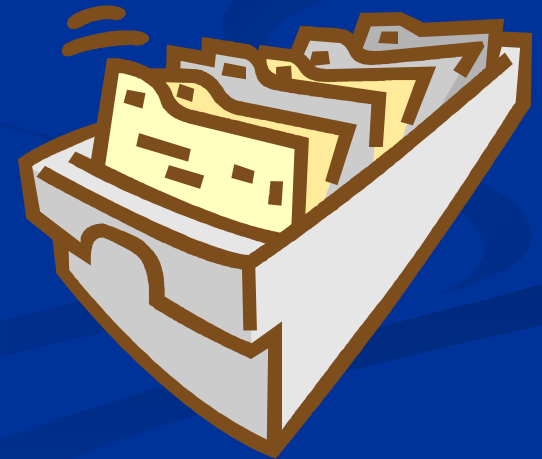
- Do not place files in web root that do not have to be there.
- If nothing is being output by the file, give it a .php extension.
- Use .htaccess to block access to files/directories

```
<Files ~ "\.tpl$" >
```

```
Order allow,deny
```

```
Deny from all
```

```
</Files >
```



# Securing Configuration Files

- Configuration scripts, usually contain sensitive data that should be kept private.
- Just denying web access, still leaves is readable to all users on the system.
  - Ideally configuration files would only be readable by the owner.

# Solution #1

- If the configuration file only stores database connection settings, you can set them via ini directives that will then be loaded by httpd.conf via Include directive.

## mysql.cnf

```
mysql.default_host=localhost  
mysql.default_user=forum  
mysql.default_password=secret
```

## httpd.conf

```
<VirtualHost 1.2.3.4>  
Include "/site_12/mysql.cnf"  
</VirtualHost>
```

- Apache parses configuration files as “root”, so your SQL settings file can have restricted permissions (0600) and still work.

# Solution #2

- For all other settings, Apache environment variables can be used to “hide” data.

## misc\_config.cnf

```
SetEnv NNTP_LOGIN "login"  
SetEnv NNTP_PASS "passwd"  
SetEnv NNTP_SERVER "1.2.3.4"
```

## httpd.conf

```
<VirtualHost 1.2.3.4>  
Include "misc_config.cnf"  
</VirtualHost>
```

```
echo $_SERVER['NNTP_LOGIN']; // login  
echo $_SERVER['NNTP_PASS']; // passwd  
echo $_SERVER['NNTP_SERVER']; // 1.2.3.4
```

# Session Security

- Sessions are a common tool for user tracking across a web site.
- For the duration of a visit, the session is effectively the user's identity.
- If an active session can be obtained by 3<sup>rd</sup> party, it can assume the identity of the user whose session was compromised.

# Securing Session ID

- To prevent session id theft, the id can be altered on every request, invalidating old values.

```
<?php
```

```
session_start();
```

```
if (!empty($_SESSION)) { // not a new session
```

```
    session_regenerate_id(TRUE); // make new session id
```

```
}
```

```
?>
```

- Because the session changes on every request, the “back” button in a browser will no longer work, as it will make a request with the old session id.

# Session Validation

- Another session security technique is to compare the browser signature headers.

```
session_start();  
$chk = @md5(  
    $_SERVER['HTTP_ACCEPT_CHARSET'] .  
    $_SERVER['HTTP_ACCEPT_ENCODING'] .  
    $_SERVER['HTTP_ACCEPT_LANGUAGE'] .  
    $_SERVER['HTTP_USER_AGENT']);
```

```
if (empty($_SESSION))  
    $_SESSION['key'] = $chk;  
else if ($_SESSION['key'] != $chk)  
    session_destroy();
```





# Safer Session Storage

- By default PHP sessions are stored as files inside the common /tmp directory.
- This often means any user on the system could see active sessions and “acquire” them or even modify their content.
- Solutions?
  - Separate session storage directory via `session.save_path`
  - Database storage mechanism, mysql, pgsql, oci, sqlite.
  - Shared memory “mm” session storage.
  - Custom session handler allowing data storage anywhere.

# Shared Hosting

- Most PHP applications run in shared environments where all users “share” the same web server instances.
- This means that all files that are involved in serving content must be accessible to the web server (world readable).
- Consequently it means that any user could read the content of files of all other users.

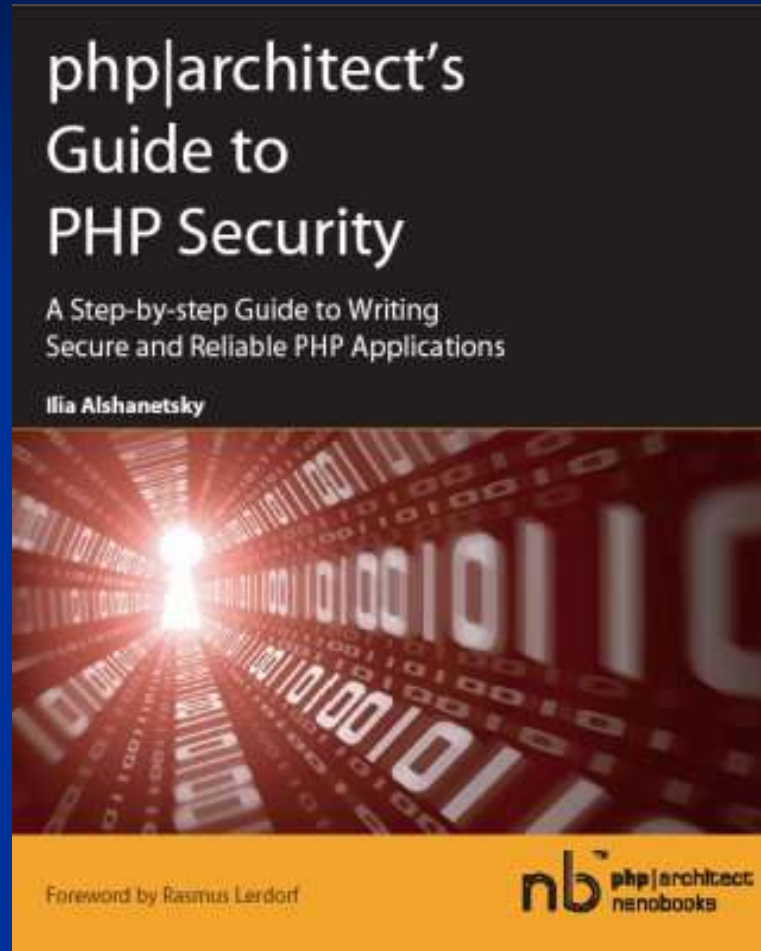
# The PHP Solution

- PHP's solution to this problem are 2 INI directives.
  - `open_basedir` – limits file access to one or more specified directories.
    - Relatively Efficient.
    - Uncomplicated.
  - `safe_mode` – limits file access based on uid/gid of running script and file to be accessed.
    - Slow and complex approach.
    - Can be bypassed with little effort.

# Security Through Obscurity

- While by itself is not a good approach to security, as an addition to existing measures, obscurity can be a powerful tool.
  - Disable PHP identification header  
`expose_php=off`
  - Disable Apache identification header  
`ServerSignature=off`
  - Avoid obvious names for restricted control panels.

`<?php include "/book/plugin.inc"; ?>`



# Questions

