Wrapping superglobals to enforce input sanitizing

One of the cornerstones of secure web applications is input validation and sanitization. Frequently this is an afterthought for developers, because PHP semantics do not actually enforce it.
With the filter extension (since PHP 5.2) this would actually be quite effortless. But not many open source projects have adopted it yet. While it is widely available and technically one of the best solutions, the filter extension has a cumbersome API. And here lies the problem. If developers are to be encouraged to ***always*** validate every piece of input, it must be ultra simple to do so. Everything else leads to laziness-related XSS worries.

This article presents a trick to ***force*** yourself, as developer, to always apply input validation.
We just take the raw input arrays $_GET and $_POST away. - That's it.

Or not quite.
Superglobals can be replaced like every other variable. And this proposal is to replace $_REQUEST and Co. with objects:

```
$_REQUEST = new input($_REQUEST);
 $_GET = new input($_GET);
 $_POST = new input($_POST);
```

So, instead of accessing input variables the old-fashioned and may-or-may-not-forget-to-sanitize-it way, there are now access methods. And these access methods conveniently also provide data validation. Instead of $_GET[„category“] you'd write:

```
 $_GET->int(„category“)
```

Every variable that was an array entry before in $_GET or $_POST, can now be accessed only through one of the sanitization methods. That's it, problem solved.

How so?

With having the old superglobal arrays available, you cannot ensure that every piece of code accesses input variables carefully. You might have legacy libraries in your codebase, or fellow developers who are less stringent about XSS security.
By taking the old arrays away, you can *force a code review*. There is no way around checking every instance and deciding on an eligible sanitizing function.

However, unlike the PHP filter function, the rewrite is not as voluminous. The syntax is rather close to the old. Instead of $_GET[] you have $_GET->int(). That's syntactically a serious difference, but not when it comes to key strokes.

Important for security concepts is not only the technology at hand, but also the effort to implement something in practice. This proposal hopefully strikes the *right balance between security inconvenience and developer laziness*.

Implementation

Internally the superglobals wrapper is pretty easy to set up. As seen in the first example, we just need to feed the old input arrays at instantiation. And we need a couple of sanitization methods:

```
class input {

	function __construct($IN) {
		$this->vars = $IN;
	}

	function int($name) {
		return intval($this->vars[$name]);
	}

	function name($name) {
		return preg_replace("/[^\w_]+/", "", $this->vars[$name]);
	}
```

Of course just the ->int() sanitization ain't gonna cut it for most webapps. So you'll get a couple more basic validation/filtering features.

In the sample implementation attached below, there is a **->name()** method, which for example returns only alphanumeric characters for use as safe IDs or variable names. Then you could have similar methods for **->text()** sans html or even a **->regex()** function for filtering with custom ad-hoc criterias. Remember, that all of your application code can access the input variables only via **$_POST->format(„varname“)** furthermore.

While its use is not recommended, there is also a simple **->sql()** method, which allows escaping in an otherwise totally amateurish way:

```
    mysql_query(„  SELECT * FROM app1 WHERE id='{$_GET->sql(article)}'  „);
```

Note: I do not recommend this, or mysql_query() for that matter. Professional developers should only ever use parameterized SQL. Or else get shot.

Advantages

The power of this method does not lie in the fancyness of variable access, but that security woes for input data get multiplexed at a single point. Gone are the days of possible insecure variable usage/access sprinkled throughout the code.

All the default methods just sanitize variables after basic patters. But since you now can extend this with a simple adaption, reporting and security logging can now be added - without having redundant code at every $_REQUEST access.
It's also helpful that special data formats can now have specific validation methods. As example, following code in our „input“ wrapper class adds an application specific filter for some article id:

```
class input {

	function article($name) {
		if (preg_match(„/^\w{5}\d{2}-\d{4}$/“, $a=$this->vars[$name])) {
			return $a;
		}
		else {	// scare message
			die(„Invalid article id received. This catalogue error has been logged. One of our administrators will look into this incident and contact you.“);
		}
	}
```

And within the e.g. web shop application, a couple of different article ids can be verified like $_POST->article(„order“) && $_POST->article(„stock“). There is no need for duplicating any verification or error handling code. And this can be reused in different code paths or templates. But most importantly, input verification is handled were it structurally belongs and not throughout the application logic.

<aside>Some people are inclined to always provide appropriate error messages for invalid input. But this article is about sanitizing possibly manipulated HTTP content, not user mishaps. Wrong user input in a form field is in fact the domain of the application logic. But the OO input wrappers are strictly about preventing exploits.</aside>

One caveat here. You cannot *always* have a validation function. There are some inputs which cannot be filtered by an regular expression or a few lines of code. Occasionally it's the task of the application logic to bring form to data.
Hence there is a method called **->raw()** which provides unfiltered access to input. Needless to say, this must be used with caution. (But then, you could always reenable the warning message in the ->raw() method for security audits.)

More, More, More

The sample implementation provides further sanitization wrappers. But it does not implement them itself. Instead the magic __get() method delegates some queries to the native PHP filter extension, which -as said earlier- does a good job at filtering.
It adds **->email()** and **->url()** or even **->float()**. Whatever PHPs filter_input() provides, is available through this wrapper class.

Moreover there is a neat little cheat that provides two alternative access syntaxes. As seen before, you have to use $_GET->int(„varname“) for accessing an input variable.

But with the provided wrapper, you can also use **$_GET->int[„varname“]**.

Seen the square brackets?! This makes the transition to the object-wrappers even easier. Since you basically only have to add **„->int“** between the old $_GET and [„varname“] that's currently in your code.

A second additional syntax is **$_POST->int->formfield**. Yes, imagine that. A fully object-style access to filtered input variables. The ->int-> here is the function ->int(). It looks like an object, but it isn't. - Well actually it is, but whatever. It's only important that the ->int() method gets called anyway.
So whatever custom filter method you add, you can access all sanitized input variables through **$_REQUEST->filter->inputvar** or with the array method.

This adds so many levels of coolness to security.

Download

Get it here, before someone else does.

DOWNLOAD LINK

As said, this is just an example implementation. It is imperative that you adapt it for your needs. Add validation methods for your specific application. You might want to change the ->name() or ->text() behaviour even. Or at least remove the warning message from the ->sql() escape filter.
It auto-initializes $_REQUEST, $_SERVER, $_GET, $_POST and $_COOKIE with the new object-wrappers, but spares $_ENV and $_SESSION per default.

Test this approach on a new project with the $_GET->text() methods. For adapting an existing codebase, you can try the $_REQUEST->int[„var“] syntax, just insert ->filter as needed. You'll see that **forcing** a new syntax on an entire webapp takes time, but is a more secure development methodology overall.

Btw, the code implicitely throws E_NOTICE messages. Did not work around this, because I consider E_NOTICE not as proper error messages, but as _DEBUG hints. Loads of isset() checks do not factually contribute to security IMO, but only to very imaginary code cleanliness.
And if, you do want those debug hints whenever a variable is missing or misnamed. The only problem with the input wrappers here is, that the notices don't carry the right line numbers anymore as when you'd access a non-existant $_GET[] variable.