PANDORAFMS

Guidelines:Pandora

11-11-2021

Guidelines:Pandora

These are the new rules for development in Pandora FMS. **Any discussion, question or suggestion must be followed in the discussion page**. This development guide also applies to other projects also hosted on Openideas, like Babel Enterprise and Integria IMS.

General approach

Pandora FMS is a project in constant growth from 2003, this means it's codebase has also evolved several times, many times not well planified. Pandora FMS tries to do a mix between cathedral and bazaar FLOSS paradigms. It has a strong direction, but developers has some degree of freedom.

Our main principles are:

1. Gold rule: **Is easier to write code than to read**. Try to code clean and simple **ALWAYS**, if your partner cannot read your code, it's your fault, not him.
2. Silver rule: **Don't feed the monster**: if you find yourself adding the 20th parameter to that terrible function nobody wants to split or “fully understand”, please, raise the red flag and join the team together to rebuild the monster and split into pieces.
3. Broze rule: To put functionality over later code beauty. Always.

Other rules.

1. Keep your code as basic and simple to be easy to maintain. Avoid “magic tricks”.
2. Keep the visual aspect homogeneous, try to use the standard way to render controls, forms, buttons and elements in tables and forms. There are some different ways to do it, even you can create your own (but you shouldn't), but always, keep the same aspect in your pages.
3. Deadlines are deadlines: are important and need to be agreed with the project management. If this conflicts with your priciples, talk BEFORE reach the deadline, NEVER after.

Files

Proposed file structure and naming.

- functions.php: Generic functions that are absolutely required (check_acl, check_login)
- functions_db.php: Only generic database functions. (process_sql(), get_db_value() and similars)

- functions_html.php: Similar, at least it's mostly clean...
- functions_agents.php: Agents functions
- functions_groups.php: Groups function
- functions_users.php: User functionss
- functions_modules.php: Modules functions
- functions_events.php: Events functions
- functions_graphs.php: Graphing and image handling functions
- functions_files.php: Upload, download and other file/http/stream handling

....

When you require a specific set of functions from a file you should reference them as follows at the top of the file you're working on:

```
require_once ("include/functions_groups.php")
```

This should be done after all functions are cleaned up and in place.

Functions

Function naming

There's very simple pattern to use in the functions name:

```
{operation}_{object}_{attribute} (parameters)
```

In functions returning some value (i.e. from the database) the most extended in all programs and languages is ussing the “get” prefix. If multiple values are returned, they should be returned in an array with the row id as key, if multiple rows are returned, a multidimensional array should be used and the columns should be named (not numbered). Examples:

```
get_event_types ()
get_alert_priority ()
```

Functions that check (compare) something against a specific value should have the “check” prefix and return only true or false.

```
check_acl ();
check_login ();
```

Functions that compose something and output it directly by default should have the “print” prefix:

```
print_table ()
print_error ()
```

Functions that process and create something (insert into the database, create a file) should have the create prefix and return a pointer to what was created (like an insert_id or the filename)

```
create_agent ()
create_alert ()
```

Functions that process something and update (the database or a file) should return false if something went wrong and true in case it went good. The process prefix can also be used if it can handle multiple functions (like aggregate create, process and delete). They should be named with the process prefix:

```
process_sql ()
process_form ()
```

Functions that delete something (database or file) should return false if something went wrong and true in case it went good. They should be named with the delete prefix:

```
delete_agent ()
delete_ip ()
```

Functions that format something in order to be represented should have the format_ prefix. The values returned from such function should not be used to insert into a database.

```
 format_numeric ()
 format_date ()
```

All functions except the print_ functions **should not print ANYTHING** (echo) from within the function. They should either generate an error/warning using the PHP error functions and return true/false or they should return composed HTML/TEXT/XML and the end file should echo the result appropriately formatted or pass it on to one of the print functions for output. All print functions should have an option (true/false) to either return or print.

Functions that format something in order for it to be stored by the database (or a file) should have the prefix safe_

```
safe_string ()
```

Function definitions

Where possible, a function should be able to be called with a minimal of arguments. So functions should have default values as much as possible and fill them in themselves. If you get to have much options (>5), you should consider using a single object (see `print_table` function)

Functions should not use any global variable unless no other option is available (eg. global $config)

Functions *sometimes* might return different values which can all be 'wrong'. You should always follow the principle of least authorization:

BAD:

```
if (get_db_sql ($sql) == 0) {
  echo "Not Authorized";
  exit;
}
```

Why: because get_db_value might return SQL's NULL, it might return false if something went wrong or an empty array (`array ()`) or string (“”) if the result was empty or stored wrong.

GOOD:

```
if (empty (get_db_sql ($sql))) {
  echo "Not Authorized";
  exit;
}
```

or if an SQL table id or other positive integer should be returned

```
if (get_db_value ($sql) > 0) {
  echo "you're good to go";
}
```

Functions parameters

The name of the parameters in a function must define a clear meaning of what the parameter affects. Sometimes it's easier to see the function declaration than reading all the documentation that explains the meaning of every parameters. Examples:

```
print_table ($table)
```

```
 get_agent_alerts ($id_agent);
```

BAD examples:

```
get_agent_module ($id); // In that case, we are not sure about what
id is it referring
```

Optional parameters

In some cases, a functionality may not need all the parameters, so you may add optional parameters. Because there are many functions which require a lot of these parameters, their prototypes could became extremely large and the use of them very difficult. An example is `print_select`, which at this moment has the following prototype:

BAD:

```
print_select ($fields, $name, $selected = "", $script = "",
$nothing = "", $nothing_value = '0',
              $return = false, $multiple = false, $sort = true,
$class = "", $disabled = false)
```

If you want to use this function but only want to change the `$sort` parameter, you must define all the previous values in the list. This would make the code too hard to write and read. Example:

BAD:

```
print_select ($values, 'select', "", "", "", "", false, false,
false);
```

That's the reason why we suggest a new style for optional parameters, which consists on declaring an optional parameter string indexed array (a.k.a. dictionaries) to set the optional values. Example:

GOOD:

```
print_select ($fields, $name, $return = false, $options = false);
```

In the example, `$options` would be an array allowing the following parameters: selected, script, nothing, nothing_value, multiple, sort, class, disabled. So an example call would be:

GOOD:

```
print_select ($values 'select', false, array ('sort' => true));
```

Functions renaming

At no point should functions be renamed unless approved by all developers. There will be a renaming effort in order to apply the rules on this page to the Pandora console. The renaming effort can be followed here: renaming pandora_functions

Variables

Try to stick (where possible) to these variable definitions:

- Not initialized or not passed: empty string ("") for strings or -1 for unsigned integers (PHP doesn't understand signedness but the database (and we) might), 0 for signed integers and database Ids.
- Something went wrong: false
- Everything was good: true or it's value

All variables that are defined should have (where possible) their standard type defined and initialized especially if we assign the result of a function to a variable. As you can see in the examples below, som functions can return all kinds of things and as developer we don't want to go looking to the table definition to know what type comes down the line especially if we need to reuse it in eg. sprintf:

```
$array = array ();
$integer = (int) get_parameter_post ("integer", -1);
$float = (float) get_db_value ($sql);
$string = (string) get_db_value ($sql);
```

All variables (especially large arrays and objects) should be cleaned up or reused in main code. Especially objects (like $table) should be cleaned up after use (unset ($table);). Result variables can be reused (eg. $result = get_db_row ($sql);). Variables in functions have an automatic garbage cleaner. File and database pointers or open streams within functions should be closed as soon as they are rendered unnecessary (eg. mysql_free_result)

Forms

- Always specify action attribute to avoid weird behavior. Autorefresh add attributes to URL, and if the form is without action attribute, the url sended is the currently one. Then some fields will be sended through POST and GET making several headaches.
- Don't nest forms. The behavior could not be the expected.
- Remember that the unchecked cheboxs and the disabled controls will not be sended

with the form.
- A checkbox unchecked is not sent with the form. It is a problem when you want to set a checkbox checked by default, because the get_parameter function doesn't difference between unchecked one and first execution (when it didn't send). To fix that you must use get_parameter_checkbox to get the value from a checkbox.

Security Issues

Follow all this rules is very important to get code more secure.

SQL Injections

Is very important in applications that uses databases be careful with queries to avoid to return private information.

To get code more secure, be sure you follow these rules to use input values:

- Always use get_parameter function to get parameters from both GET and POST. This function convert blank chars of ASCII table into its HTML entities, so if you need write data from database to a file you must use the function safe_output to revert this conversion and get clear data.
- Convert all values to its type if you know it (do a cast!), for example:

```
$var = (int) get_parameter ('var',0); //We use get_parameter and
also do a cast.
```

- If the value type is string or we unknow the type use function escape_string(), is in functions_db, that returns the input string with special characters escaped. The escape_string() function is not able to insert data in db so, please, use this function only for select queries.

```
$var_string = (string) get_parameter('var_string', "");

$var_string_ok = escape_string($var_string);

$var_notype = get_parameter('var_notype, "");

$var_notype_ok = escape_string($var_notype);
```

- Use sprintf function to create sql queries, for example:

```
$sql_query = sprintf (SELECT field1, field2 FROM table WHERE field1
= %d", $var);
```

Comments

All code should be accompanied by some comment especially when you're nesting or you're doing something clever that not everybody might understand. Also when you're using mathematical formulas to calculate how to spread something on a page, it might be good to have some comment.

All functions are **REQUIRED** to have a full comment in this format:

```
/**
 * What the function does
 *
 * If your parameters use abbreviations, they should be defined
here.
 * Also, if the function does something hard or a big operation
 * that you want to describe, do it here.
 *
 * @param int $param1 What it requires
 * @param string $param2 What it doesn't necessarily requires
 * @param mixed $param3 It can be single value or an array.
 *
 * @return bool This function returns false
 */
function this_func ($param1, $param2 = 0, $param3 = false) {
      return false;
}
```

I think all of this is commented in general coding style rules.

DISCUSSION

- We should decide a document generator tool to create a human-readable and indexed function list. I took a look to different tools for PHP, and the best I found is phpDocumentor. You can try to install it using PEAR. Here's a quick steps to generate documentation for include/function.php:

```
~/pandora/ $ pear install PhpDocumentor
~/pandora/ $ mkdir doc
~/pandora/ $ phpdoc -o HTML:frames:earthli -f include/functions.php
-t doc
```

You can see that some changes should be done in the comments style. In the `@param` and `@return` tokens, you need to supply parameter type. I've changed the example to do so. There are other aspects like packages, file aggrupations and similar which I didn't see completely, so maybe we should have more tokens in the future. Esteban 08:04, 10

November 2008 (PST)

Loops

Loops with for should be avoided for looping on arrays. Foreach should be used to loop over an array because sometimes not all keys are defined in an array and a for loop would skip them. It also makes it more readable. For loops should only be used when something purely mathematically repetitious is done. **Always** needs to check if array exists before using it in a foreach loop:

BAD:

```
for (i = 0, i < count ($array), i++) {
   echo $array[$i];
}
```

GOOD:

```
if (isset($array))
    foreach ($array as $value) {
        echo $value;
    }
```

Low-level (basic) PHP functions

The use of the low-level PHP functions for databases (mysql_*) should NOT BE USED at any time. The database functions in functions_db have all functionality that is required and have a standardized return and error handling.

The use of low-level functions for image handling (ImageMagick or GD) should also NOT BE USED in inline code. If you need such functions, we are soon going to be using pGraph or you can create your own wrapper functions to handle certain things. If we switch or upgrade any type of library, we should be able to do so without extensive recoding.

The use of low-level functions that are not implemented in a 'standard' PHP 5 distribution (any recent Debian-based or Red Hat-based distro) or that are server-specific should also NOT BE USED. This includes functions that are only implemented in beta versions, PDF, Apache or IIS-specific functions etc. If you need certain specifics (like virus-scanning or something else) you should also consider implementing either dummy functions or replacement functions/classes using function_exists.

Coding style

Please refer to general coding style rules.

Database tables

Time

Time should be stored and retrieved in Unixtime. This is easier and faster to calculate on and does not leave any doubt as to whether or not there was correction for timezones. This will also allow people in different parts of the world (but with different time zones) to see the time according to their local time. It's unnecessary to store a date/time string as well. Unixtime can be converted in both MySQL or PHP however if there need to be operations performed on it, until it's represented, it should remain in unixtime.

JavaScript & AJAX

If you're making use of JavaScript make sure you do not rely on it for validation. Validation and cleaning should be done at all times server-side. You should try to make use (if possible) of the jQuery framework which is included in Pandora.

General rules about JavaScript:

1. **Dont use Javascript** if you can do things easier with PHP/HTML **even** if this takes more tike or code, take in mind that Javascript code is hard to maintain and make code much difficult to read / modify for average developers.
2. Don't include new frameworks or libraries if you can do something yourself using a current library or even your own function. This is applicable of course, for small things, for big issues a architecture decision has to be made.
3. Again: **don't use javascript** if you can solve the problem using standard PHP/HTML code.

AJAX

Ajax files will be named like the data they are using, so if the ajax call is to get data from an agent, the file name should be agent.ajax.php, in the cases the ajax functionality is very attached to a page, the name of the page with .ajax.php extension should be used.

All the ajax files should be on the directory include/ajax/ this way all the ajax functionality

is on the same place to promote a more general and reusable code.

Today (as 23-III-2010) there are many php files with ajax code inside an *if (is_ajax ())* block that should be moved to separated files to have a cleaner and more maintainable code.

DISCUSSION:

- As addition to the above. We should include the jquery.js (at least the base) within index.php. Right now, we have to remember to include it in our scripts when we use it.
    - I agree, but since it's not widely used in Pandora I didn't include it on index.php yet. It would be nice also to allow pages to add content in the header HTML element, so Javascript can be added there instead of the bottom of each page. That would require some major changes in the index.php file... Esteban 01:50, 29 October 2008 (PDT)
        - A 'simple' solution would be to do this: use the output buffering functions in PHP (ob_start) and use a callback function that evaluates certain global variables and then prepends the existing buffer with a correct header and returns a 'good' header. Exit, die, the end of execution will all automatically flush the buffer so we don't need to mess with ob_end_flush. It will also make it easier to extend beyond the HTML capacity by eg. sending specific XML headers when AJAX or RSS feeds are required.