Filtering Best Practices

Overview

To prevent itself from XSS vulnerabilities, Tiki applies a very broad sanitization filter on all input. While the technique is mostly safe, it is well known to cause undesired side effects on the end users. This form of sanitization has been present in Tiki for a very long time without problems. However, more aggressive filters have been in place since 2.0 and those have raised a large number of bug reports. Broad sanitization is also very expensive in processing time.

Beginning with 3.0, more control over the filtering is possible. The very broad filter is now used as a last resort in cases where no specific filters are specified. Specific filters are even more secure and can significantly improve performance.

Two types of filters are available:

- Declarative filters, which declare what are the expected data types before sanitization is performed.
- Just in time filters, which defer filtering until the variable is used.

Declarative Filtering

Declarative filtering works by attempting to apply a declared set of rules on the input. Once a rule applies, the remaining part of the sequence is dropped. Matching rules can be of three different types:

- **Static key** lookup applies the rule if there is an exact match between the key name and the provided one.
- **Key pattern** lookup applies if the key name matches a provided regular expression.
- **Catch all** applies to everything reaching it.

The rules can have two different effects:

- They can **filter** the value associated with the matching key or
- they can **unset** the key altogether and destroy the value.

Each script file is expected to define the static key and key pattern rules for their own input. Optionally, they could add a **unset** catch-all clause to the set of rules if every possible input is declared. Doing so would provide the highest possible level of security. Before sanitization occurs, a catch-all clause to the broad XSS prevention filter is added automatically. Scripts are now allowed to set a default catch-all filter clause.

Declarative filters are defined by setting a global variable before inclusion of *tiki-setup.php* or *tiki-setup_base.php*.

Sample declaration

```
<?php $inputConfiguration = array( array( 'staticKeyFilters' => array( 'page' => 'pagename', 'content' => 'wikicontent', ) ), array( 'staticKeyFiltersForArrays' => array( 'categoryId' => 'digits', ) ), array( 'staticKeyUnset' => array( 'page_id' ) ), ); require_once 'tiki-setup.php'; // ...
```

For each row, the key is the rule type and the value is the creation argument. The creation argument varies. Here are the available rule types:

- **staticKeyFilters** applies the specified filter for each key. The input is a map in which the key is the

input key and the value is the filter. Initializing an array here will cause an error.
- **staticKeyFiltersForArrays** is a variant of the previous one. Rather than applying the filter directly on the value, it traverses arrays recursively and applies the filter on end-elements.
- **staticKeyUnset** removes the keys listed in the argument. The argument is a simple array of keys.
- **keyPatternFilters** applies a filter on each key matching a PCRE. The creation argument is the pattern as the key and the filter as the value.
- **keyPatternFiltersForArrays** is a variant on the previous one, analogous to **staticKeyFiltersForArrays**.
- **keyPatternUnset** unsets all keys matching a PCRE. The creation argument is a simple array of patterns.
- **catchAllUnset** removes all keys not declared before. Use *null* as the argument. Declaring this is advisable whenever technically possible.

If more complex filters are required, *tiki-filter-base.php* can be included prior to the declaration. This inclusion will set the Tiki include paths. More files can then be included. More complex filters could require instanciation of filter objects. A filter could be another **DeclFilter** instance to handle structured input.

Declarative filters in plugins

As an addition, the plugin description function must now provide the filters to use on each value passed to the plugin. Wiki syntax can be considered safe, but the plugins may cause security threats if the input is not handled correctly. Unspecified filters will apply the XSS filter.

Sample plugin declaration

<?php function wikiplugin_example_info() { return array('name' => tra('Example Plugin'), 'description' => tra('Does nothing'), 'body' => tra('Some short text string'), 'filter' => 'striptags', 'params' => array('height' => array('required' => false, 'name' => tra('Height'), 'description' => tra('Box height'), 'filter' => 'digits',),),); } /* Using the above definition, this call: {EXAMPLE(height=123abc width=123abc)}Hello World{EXAMPLE} Would produce the following plugin call: wikiplugin_example('Hello World', array('height' => '123')); Notice that width is gone because undeclared. HTML tags were removed by striptags and height only contains '123'. */ ?>

Use a separator when there are more than one possible values

If you can have many possible values, use a **separator**:

'filter_category' => array('name' => tra('Filter category'), 'description' => tra('Limit search results to a specific category. Enter the comma separated list of category IDs to include in the selector. Single category will display no controls.'), 'filter' => 'digits', 'separator' => ',',),

Just In Time Filtering

In many cases in Tiki, it's impossible to define the data type of the input variable before runtime. The applicable filter may depend on other input or various settings. Examples of this are the wiki page content, which can either be HTML or wiki syntax, or tracker field values which depend on the tracker field type. For all these cases, JIT filters can be used. The default filtering, in this case, is also the broad XSS prevention filter.

Before performing sanitization, a pristine copy of the original input is preserved in JitFilter objects. While the data is not accessible directly, no filtering is made on it before it's accessed. The filters can then be defined at runtime when the data type is known. When JitFilter is used to access input, the input keys should be **unset** by the declarative filters for increased security and performance.

The JitFilter objects are accessible through **$jitGet**, **$jitPost**, **$jitCookie** and **$jitRequest** in the global scope.

JitFilter can be used in multiple ways.One interface is meant to act as closely as possible to an array. It was built during the first effort.

Sample usage in array mode

```
<?php $jitRequest->replaceFilters( array( 'content' => 'wikicontent', 'page' => 'pagename', 'categories' => 'digits', ) ); $page = $jitFilter['page']; // Filter applied $content = $jitFilter['content']; // Filter applied // Array access A $categs = array(); foreach( $jitFilter['categories'] as $categ ) // Filter applied on each element $categs[] = $categ; // Array access B (recommended) $categs = $jitFilter->asArray('categories'); // Array access C $categs = $jitFilter['categories']->asArray(); ?>
```

When used in array mode, the JitFilter object applied the filters when an end-node in the array is reached. Otherwise, another instance of JitFilter is returned. The filters must be specified before the property is accessed (it's safe to change the filter afterward and re-request the data). The asArray() method on the filter can be used to return a completely filtered array instead of an object.

Because declarative filters cover the predefinition of parameters in most cases, a different way of accessing filtered values can be used.

Sample usage in object mode

```
<?php // Page name A (recommended) $page = $jitFilter->page->pagename(); // Filter 'pagename' applied // Page name B $page = $jitFilter->page->filter( 'pagename' ); // Page name C $page = $jitFilter->page->filter( new Zend_Filter_StripTags ); $content = $jitFilter->content->wikicontent(); // Filter 'wikicontent' applied // Array access A (recommended) $categs = $jitFilter->categories->digits(); // Applied on values, proper array returned // Array access B $categs = $jitFilter['categories']->digits(); ?>
```

This second technique provides the same results but skips the definition statements.

Available Filters

The two techniques above use a common set of filters. Filters can be provided as a filter object instance (implementing Zend_Filter_Interface) or a filter name, which will be resolved based on rules defined by Tiki. The mapping between names and filters can be found in lib/core/TikiFilter.php. These are very likely to evolve.

Using names allow replacing the filter's implementation. Some shorthand generic filters are available, like 'alnum', 'alpha', 'digits'. However, higher level concepts like 'groupname', 'username', 'pagename', 'wikicontent' are desired, even if they are only alias' of other filters at the moment of definition.

Roadmap

Long term goals

- Use declarative filters on all input
- Unset all non-declared input
- Use JIT for input types that cannot be determined at sanitization time

Want to help?

Specifying filters in Tiki is a huge task. You can help by:

- Specifying filters on plugin body and arguments
- Converting the script pages to use the new filtering techniques, which includes:
  - Specifying the declarative filters
  - Modifying the code to use jit filters when needed
- Testing, testing, testing

Alias

- FilteringBestPractices