pLog Security Framework

Name	***Date***	***Status***	***Version***
Oscar Renalias	23/10/03	Draft	01/01/00

1. Motivation

To provide an extendable and flexible security framework for pLog. With this framework it should be possible to provide features such as IP-blocking, content-based blocking and any other feature that might be needed in the future (hence the flexibility requirement)

Users should be able to configure the security subsystem in a per-blog manner, while at the same time the administrator user should be able to provide blog-independent configuration. Global settings will always be used **before** blog-specific settings.

2. Features

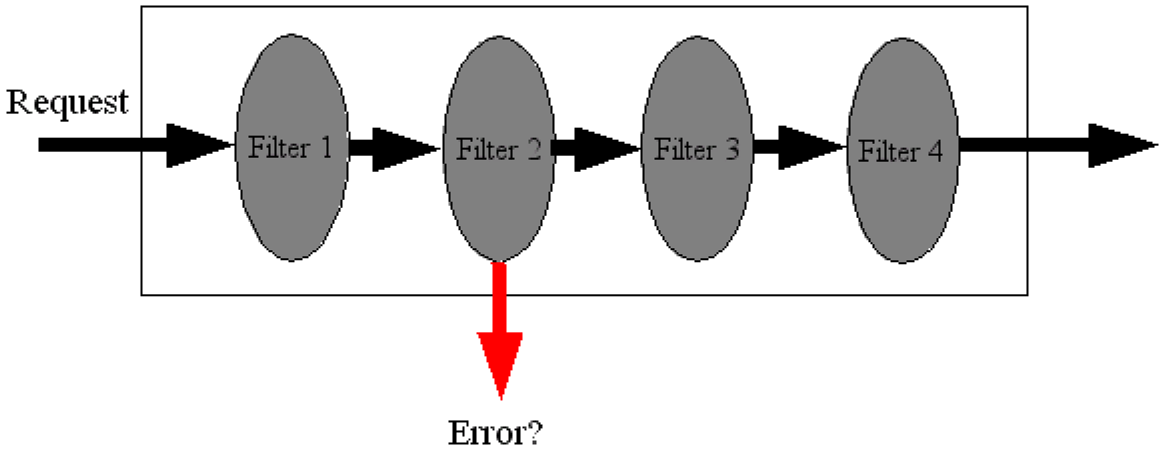
- It should be possible to define a network mask that would block hosts that match the mask from posting comments to our site. The network mask could be as small as a simple host or as big as an A-class network mask.
- It should be possible to define a network mask, in the same way as above, that would block hosts that match the mask from even accessing to the site. That is, they should be forbidden to browse any of our pages.
- It should be possible to invert the aforementioned masks. Instead of blocking all the hosts that match the masks (**blacklist**), **only** the hosts that match the mask should be allowed to access our site and/or post comments (**whitelist**).
- It should be possible to define more than one blocking rule. It should be possible to block as many hosts from posting as needed and block as many hosts from accessing our site as needed. However, the order in which these rules are enforced will matter.
- It should be possible to block certain comments based on the content of the text of the topic or of the text of the comment. This is to protect our blogs against blog-Spam. These rules could be simple words or more complex regular expressions.
- There should be additional 'comment' and 'date' fields for every item that has been blocked, so that we can remember why and when it was blocked.
- The whole security subsystem should be possible to be activated/deactivated easily.
- The security subsystem should add a minimum performance penalty, even though that is somewhat difficult to avoid when using more complex things like regular expressions and network masks.

Not all these features should be present in a first release.

3. How it should work

Every incoming request should be processed by the security subsystem. Therefore, the security subsystem must “live” in a layer within our application where it has knowledge of things like the incoming request (that's an easy one :)) and the current blog we're dealing with. The former is necessary to add things like host blocking features while the latter, is necessary if we wish to provide content-filtering features.

The idea would be to build a pipeline-like structure made up of smaller structures, which we will call "filter". Once a request enters the pipeline it will be, in turns, examined by every filter. If the request happens to match any of the conditions imposed by the filter, then it will be blocked and execution of the pipeline will finish with a negative result.



If the request manages to pass through all the filters, then it will be accepted as a valid request and therefore processed as a such by the appropiate layer.

As we can see from the image, the order in which the filters are organized does matter so we should be careful with that. A suggestion would be to have the filters that work based on IP-addresses/hostnames first and then the content-based filters.

Execution of the pipeline will continue until either a filter matches or the request went through all the filters until the end.

The pipeline should only notify wether the request made it through or not. However, the layers of pLog that use the pipeline should know why the request was blocked, so that some feedback can be provided to the user. Ideally, the pipeline will return some kind of error code offering more information. The error code should be dependant on every filter and great care must be taken so that different filters return different codes, since that could potentially confuse the user. Then, depending on the error code,the user class should be able to decide what to do.

Basically, it is not up to the pipeline to decide wether to stop the execution flow or not. **The pipeline will only provide with a YES or NO answer when given a request**. Such answer will depend of course on the filters, but the filtering pipeline should not take any action but rather provide some information to the calling layer about how to proceed.

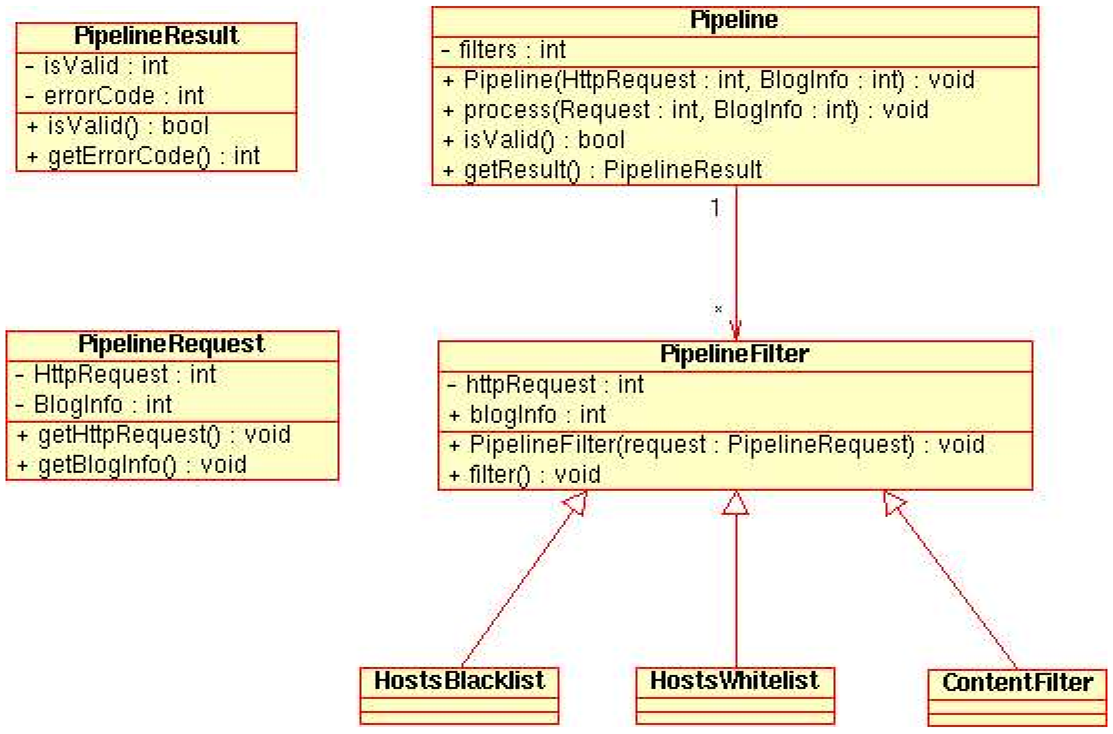
4. Implementation

The implementation will require additional classes as well as a additional tables to the database, in order to store information.

First of all, an additional setting will be added to the global configuration table controlling wether the security framework is enabled or disabled.

The pipeline should be represented by an object of the SecurityPipeline class. This class will support very basic methods such as process() and getErrorCode(). The implementation of the pipeline will be done according to the Pipeline design pattern (http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/AlgorithmStructure/Pipeline.htm) because its features can be divided into smaller units that favour such implementation.

The following is a very simple UML diagram of the implementation of the Pipeline. Every Filter object inherits from the common PipelineFilter class, implementing its methods (if PHP had Java-like “interfaces”, those would be interfaces) The implementation of every class that implements a filter has been left out for readability:



5. References

MT-Blacklist plugin for Movable Type: http://www.jayallen.org/projects/mt-blacklist/

The Pipeline design pattern:
http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/AlgorithmStructure/Pipeline.htm