

FCGlob: A New SELinux File Context Syntax

Authors: Don Miner, University of Maryland: Baltimore County
James Athey, Tresys Technology, LLC

DRAFT: SEPTEMBER 30, 2006

[Italics surrounded by “[]” throughout this paper are comments and hopefully will be resolved eventually.]

Abstract

[Will be completed when the paper's content has been nailed down.]

Introduction

The file context system in place today in *reference policy* is rather simple in the way it works. In a compiled policy, the *file_contexts* file is a list of regular expressions approximately sorted from least specific at the top of the file to most specific at the bottom of the file. Specificity of regular expressions is defined as: if the set of strings regular expression A match is a subset of the set of strings regular expression B match, then A is more specific than B *[This definition seems like it is in an awkward place, where else could I put it, or is this fine?][Have a graphic example of specificity here]*. When the context of a file has to be determined, like during *setfiles*, this file is consulted with *matchpathcon*. What *matchpathcon* does is start at the bottom of the *file_contexts* file and moves towards the beginning and stops at the first regular expression that it is matched by *[Have a graphical diagram showing matchpathcon going up the FC file]*. The purpose of this implementation is the path should be matched by the most specific regular expression in the file. For example, */etc/httpd/httpd.conf* should be matched by *httpd_conf_t (/etc/httpd/httpd.conf)* instead of *etc_t (/etc/*)*. This implementation has some currently unresolvable issues, mainly caused by the use of regular expressions as the syntax to match files.

A point in which this implementation fails is at module link time when the file contexts for each individual modular policy is compiled together. All the file contexts from all the different *.fc files are concatenated together and then sorted with an approximated comparison function, based on heuristics, from least specific to most specific. An incorrect ordering of file contexts may happen because of the usage of an approximation of specificity. We believe that the sets formed by regular expressions are extremely hard to analyze and should be replaced by something that can be. If a new syntax where specificity between two patterns could be easily and quickly determined, no longer will the sorting be an approximation. *[Have a graphical flow chart of the build process][Some examples that show how the approximation can be fooled should go here. They should not be ambiguous.]*

Another problem are *ambiguous* file contexts, where two regular expressions match a single path name but neither is more specific than the other. The formal definition of a ambiguous specificity is: if regular expression A matches strings S_a and S_{ab} and regular expression B matches strings S_b and S_{ab} , then the relationship is ambiguous because the label of S_{ab} can not be determined *[have a graphic that illustrates this example of ambiguity]*. The sorting algorithm orders these based on its approximated comparison function and no warning or error is shown to state that there is ambiguity which in effect causes a silent bug. Since the current sort cannot detect ambiguity, it is assumed ambiguous file contexts do not exist or do not matter and it is up to the policy writer to write contexts so that they conform. If a new syntax where ambiguity could be detected through the definition given earlier, warnings or errors could be raised during the file context file's compile time and alert the policy

writer. *[Some examples that show current file contexts that are ambiguous]*

In our opinion, the regular expression syntax is too complex and expressive for this application and causes several problems outside of the two main ones previously discussed. We believe that regular expressions are very prone to error because many people are not knowledgeable enough in this area to construct them in a perfect manner. One major problem is the widespread usage of “.*” because although policy writers often use it to match only a single directory level, it can match any number of them. This introduces lots of ambiguity and often times confuses the current comparison algorithm. A more simple syntax could be less prone to error, easier to learn and more tailored to the needs of matching file paths. Another problem with using regular expressions is they allow the policy writer to be extremely clever, sometimes obfuscating the meaning. Since file contexts are shared among everyone, writing file contexts so they are readable is beneficial to everyone. *[Some examples that show common errors made]*

Our new syntax, *FCGlob*, addresses these issues because it is designed so that the comparison between two *FCGlobs* always returns a definitive result. It would be able to determine if two patterns are ambiguous, if one pattern is more specific than the other, or if they are disjoint and do not share any strings in common. These set comparison operations are extremely difficult for computers to perform on regular expressions. Also, since the syntax is simpler, we believe it would be less error prone and more readable to humans. The power of regular expressions is not needed in the case of file contexts and we propose that even though *FCGlob* is not as expressive as regular expressions, the trade-off is definitely worth it.

Not only would *FCGlob* fix problems with the current system, it also introduces the ability to improve on the capabilities of file contexts. The programs *matchpathcon* and *setfiles* could run faster. A policy writer could give *matchpathcon* a *FCGlob* and have it return the list of file contexts that would satisfy the pattern given. New file context definitions could be dynamically inserted into the file contexts file without having to resort the entire file contexts file. The analysis tool *apol* could now analyze the file contexts. We will discuss how each of these improvements are implemented later on in this document.

One of SELinux's major faults is its poor representation of the file system. *FCGlob* strives to take SELinux to the next level in functionality.

Description of the Syntax

FCGlob's syntax was created with several goals in mind in order to make it easy for a human to use and easy for a computer to analyze. We believe that *FCGlob* is a improvement over regular expressions in both these criteria. The goals are as follows:

- The syntax should be expressive enough to provide the features expected in a path matching syntax.
- The syntax should not be expressive enough to allow for cleverness in order to avoid obfuscated definitions and errors.
- The syntax should be designed so that a computer can easily and quickly analyze how the sets of strings two patterns match relate to one another.
- The syntax should be easy to learn and similar to something people already use commonly (more commonly than regular expressions).

To satisfy these goals, we devised a syntax that is similar to basic shell globbing in many ways. A few structures were borrowed from regular expressions and a few restrictions are imposed on the policy writer to avoid ambiguous definitions.

One of the most important differences between *FCGlob* and regular expressions is *FCGlob*

treats the directory dividers, “/”, as special markers, not characters. One of the first steps of parsing is splitting the path into a list of directories, for example “/etc/httpd/httpd.conf” will become the list [“etc”,“httpd”,“httpd.conf”]. This difference is much like the difference between “*” in shell globbing and “.*” in regular expressions where in globbing “*” does not capture more than one directly level and “.*” does. This follows the model of the UNIX directory structure closer and will prevent ambiguous definitions.

A FCGlob can contain meta-characters as well as regular characters, just like in globbing. The following characters are reserved as meta-characters:

- \ The escape character. This is used if a meta-character should be taken literally.
- ? Matches any character. This is different from “.” because “.” can match the “/” dividing a directory level.
- [...] A character class. Matches any one of the characters inside of the []. This is very similar to and is borrowed from regular expressions.
- (...|...) The “or” statement. Match any of the strings separated by “|”. This is very similar to and is borrowed from regular expressions.
- * The star. Matches zero or more of any characters confined within a single directory level.
- ** The double-star. Matches any number of characters over several directory levels. This is much like “.*”.

Several restrictions are imposed on these constructs to satisfy the goals we have set forth. If a policy writer violates these rules, an error will be shown at module compile time. The restrictions are as follows:

- Character strings within a “or” construct must be of fixed length. For example, (z?!z??) is valid, but (z*!x) is not. This makes it much easier for the computer to analyze the pattern and prevents cleverness.
- Only one “*” is permitted per directory level. Having two stars in one directory is a leading cause for ambiguous definitions. We feel that not allowing this is more obvious to the policy writer than figuring out why two patterns are ambiguous.
- Only one “**” is permitted per pattern. The reason for this restriction is the same as the one for “*”.

[A couple examples of actual current file contexts converted into FCGlobals. At least one example of a current FC that would have to be split up into several lines. The examples should demonstrate every point made in this section.]

The FCGlob syntax meets all of our goals. We believe that FCGlob is expressive enough because all of the current file contexts can be expressed in some way with FCGlob. FCGlob discourages cleverness and forces policy writers to write out their definitions on several lines if needed to. We have devised an algorithm that quickly and easily determines the set relationship between the two patterns with absolute certainty. Since FCGlob is very similar to globbing, which Linux users used all the time while using the command line, it should not be had to pick up fast. *[Is just saying the algorithm exist enough for the symposium? Should the algorithm be described in this paper? If so, where? Is a basic overview of the algorithm needed?]*

Compilation of the FCGlob Contexts and Usage of the Compiled Form

Currently, the file contexts are stored in a linear list. This was done this way because it was probably the easiest way to implement the system. However, since we can determine the set relationship

with FCGlob, a tree makes more sense and provides several benefits. Instead of concatenating all the file context definition files for each module and then sorting them, they will be constructed into a tree. This tree's nodes and edges are defined as follows:

- A node is a file context pattern. The node also contains the file label.
- A directed edge exists from a node A to node B if A is a subset of B and no node C exists such that C is a superset of A and C is a subset of B. This defines an ordering of specificity in the tree.

[A sample graph]

This new form of the file contexts makes *matchpathcon* much faster and also opens for the possibility of new features. When *matchpathcon* is asked to find the label of a file, the most specific node is found by traversing the tree. For each level of the tree, the path should be matched by only one of the patterns on that level since the definitions are guaranteed to be unambiguous. To traverse the tree, starting with an iterator on the root node, simply continue level by level, following which nodes match the path. Once a point is reached where the path does not match any of the patterns on the next level, the iterator is pointing to the most specific match [A graphical diagram showing the iterator move through the tree, very similar to the diagram used to show how *matchpathcon* works so the contrast can be made]. This tree traversal is faster since it is logarithmic in complexity, apposed to linear in the current implementation. When adding a new file context definition, perhaps through *semanage*, a similar process is used to figure out where the node should fit into the tree. For example, in order to find a subset of file contexts from a FCGlob, a policy writer wanting to allow access to all files in */etc/* could query *matchpathcon* with *"/etc/**"* and find all subsets of this FCGlob. The possibility for a faster and more flexible *matchpathcon* is a major advantage FCGlob has over using regular expressions.

One major difference and downside to this implementation is the tree would be a binary file and not human readable anymore. However, new tools could replace all reasons why a human would want to look at the file contexts file and this will not be a problem.

Actual Implementation

We propose that at the prototype of FCGlob should be implemented so that only changes are made to *reference policy* so that nothing has to be changed in *libslinux*. The implementation of the prototype with this ideal is described below and then later we state what would have to be done to completely integrate FCGlob in *libslinux*. The general idea of the prototype implementation is we fool *libslinux* so that it thinks it is dealing with an old style file contexts file. This is done by taking the tree and converting it into a linear list of regular expressions sorted from most specific to least specific, like it is now.

Implementing a FCGlob prototype would require several programs or methods would have to be created:

- A syntax parser that would compile a pattern into a parsed form. This way comparisons can be done without having to re-parse patterns every time.
- A comparison function that receives two patterns as parameters and returns the set relationship. Possible set relationships between the set of paths pattern A matches and the set of paths pattern B matches are: subset, superset, disjoint and ambiguous
- A tree compiler that generates the tree based on the comparison function.
- A new *matchpathcon* would have to be created that is able to read the compiled tree and has the new features.
- A tree-to-file_contexts converter that takes the tree and flattens the tree into a linear list. The FCGlob syntax is a subset of regular expressions, so a simple conversion is possible. Since

there are no ambiguous definitions stored in the tree, a list where more specific patterns are lower in the file than less specific ones is possible.

In addition, conversion of all the old file contexts is needed. Some of these can be done automatically but many will have to be done manually. A major difficulty in this will be deciding between “*” and “**” based on whether a author of a definition meant “.*” to cross directories or not. Also, the *reference policy* makefile will have to be changed so that it compiles the tree as well as the translated old-style file contexts file.

In the future if FCGlob were to be integrated into *libselinux*, the entire prototype would have to be coded directly into the library so the tree could be queried directly. This would be considerably faster than the prototype implementation. However, we are confident that the prototype implementation should show direct results to using FCGlob.

Results

Some statistics of speeds and optimizations, how many FC globs there are vs. FCs and other general lessons learned during the implementation will be presented once the prototype is implemented.

Conclusion

FCGlob resolves the issues currently present in the way SELinux looks at the system's directory structure. It no longer uses heuristics and does not give wrong orderings of contexts. The syntax is simpler, easier to use and easier to learn. Searching for a file's desired context is now logarithmic instead of linear, making *matchpathcon* and *setfiles* much faster. The implementation of the prototype does not require *libselinux* to change, making it relatively easy to code to demonstrate its usefulness. Applications that leverage SELinux could use the tools that query the tree immediately instead of using *libselinux* to reap the benefits right now. A SELinux using the FCGlob syntax is a correct and faster implementation than one that uses regular expressions and has very few, if any, downsides.