FCGlob: A New SELinux File Context Syntax

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

Abstract

A new syntax designed to replace the usage of regular expressions in SELinux's file contexts is proposed, named FCGlob. There are several major problems with using regular expressions for file contexts, such as an approximated sorting, ambiguous declarations, common user errors, obfuscation due to cleverness, and the difficulty in finding set relationships. FCGlob is designed to address all of these problems by using a simpler syntax that is more tailored to matching UNIX file paths. The new syntax encourages clear and simple patterns, discourages cleverness and laziness, and makes specifications easy for a computer to analyze. In addition to fixing several problems, FCGlob also opens the door to several enhancements. The basis for most enhancements is the use of a tree data structure as opposed to a linear list that that is used in the current implementation of file contexts. Several benefits result from the usage of a tree structure, such as a faster matchpathcon and the possibility for many new features. Implementing the FCGlob prototype can be done without making any changes to libselinux by converting all the FCGlob specifications into regular expressions and ordering them in a linear list, with an end result that is compatible with the existing toolchain. If the prototype is successful and accepted, the improved syntax can then be integrated into libselinux.

1. 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 specifications, where each specification is composed of three parts. The first part is a regular expression that matches certain paths. The optional second part is an object class, e.g. "-d" for directories. The third part is a security context that will be applied to all files matched by the regular expression in the first part. These specifications are sorted using approximation heuristics 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 follows: if the regular expression A matches a subset of the set of strings that regular expression B matches, then A is more specific than B.

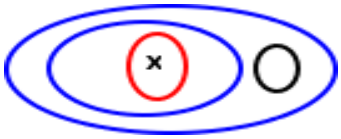

Figure 1: An example demonstrating specificity. The blue circles represent patterns that match file x*, the black circle represents a pattern that does not match* x *and the red circle represents the most specific pattern that matches* x.

To determine what label to apply to a path, e.g., when running *setfiles*, the file contexts file is consulted with the libselinux function *matchpathcon(3)*. 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 matches the given path. 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 are 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. For example, "/dev/.*mouse.*" according to the approximation algorithm is more specific than "/dev/.*mouse1" because "/dev/.*mouse.*" has more characters in it. We believe that the sets formed by regular expressions are extremely hard to analyze and should be replaced by something that can be. Given a new syntax where specificity between two patterns could be easily and quickly determined, the sorting will no longer be an approximation.

Another problem is *ambiguous* file contexts, where two regular expressions match a single path name but neither is more specific than the other. The formal definition of an 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. This definition is illustrated in Figure 2.

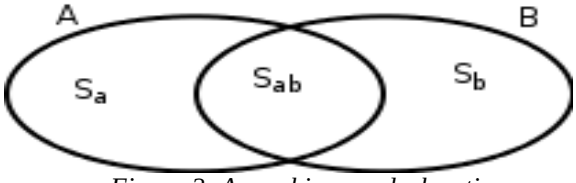

Figure 2: An ambiguous declaration.

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. For example, the two file contexts "/usr/(.*/)?lib(/.*)?" and "/usr/(.*/)?bin(/.*)?" cause an ambiguous declaration: what would be the label of "/usr/bin/lib/foo"? 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. Given 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.

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. For example, there should be an easier way to express "/mnt and all files in the /mnt directory" than "/mnt/(/[^/]*)?". 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. For example, in regular expressions, the "." means "any character," which is entirely counter-intuitive when used to describe filenames, where periods are common. As a result, many policy authors write a context such as "/etc/httpd/httpd.conf" when the correct way would be "/etc/httpd/httpd\.conf". Another major problem is the widespread usage of ".*" because policy writers often intend to use it to match only a single directory level, but it will match any number of them. This introduces lots of ambiguity and oftentimes confuses the current comparison algorithm. A simpler syntax could be less prone to error and tailored to the needs of matching file paths. Finally, regular expressions allow the policy writer to be extremely clever when writing file contexts, and although the result is precise, the intention behind the specification is lost. Since file contexts are shared among everyone, writing file contexts so they are readable and understandable is beneficial to everyone.

Our new syntax, *FCGlob*, addresses these issues because it is designed to express file context specifications in a familiar syntax while also making definitive comparisons between two specifications possible. One of SELinux's major faults is its poor representation of the file system, and FCGlob strives to take SELinux to the next level in functionality. It is 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 files in common. These set comparison operations are extremely difficult for computers to perform on the regular expressions used in file contexts. The syntax for regular expressions is too expressive and too complex because it has been built to match and parse strings quickly and as a trade off it has become hard to analyze. Also, since our 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 does 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 a list of file contexts that satisfies 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 present a graphical view of the file system as described by file_contexts. We will discuss how each of these improvements are implemented later on in this document.

2. 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 since cleverness often leads to obfuscated and erroneous definitions.
- 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 a system in common use.

To satisfy these goals, we devised a syntax that resembles basic shell globbing in many ways. However, standard globbing could not be used due to the several restrictions FCGlob places as well as additional syntax such as the parenthesis for "or", which is borrowed from regular expressions and the double-asterisk which was added.

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. The following characters are reserved as meta-characters:

? Matches any single 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 ".*".
\ The escape character. This is used if a meta-character should be taken literally.

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 an "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. If a policy writer has to use a pattern that breaks this rule, he can split it up into two separate patterns.
- Character strings within an "or" construct cannot contain several directory levels ("/"). The reasons for this are the same as the previous restriction.
- Only one "*" is permitted per directory level. Having multiple wildcards in one directory is a leading cause for ambiguous definitions and we believe it is never actually necessary. 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, and it must stand alone between directory markers, e.g., "/usr/**/lib/*.so".

Some examples of conversions from regular expressions to FCGlob:

```
/bin/.*                  =>   /bin/**
/usr/lib(64)?/amanda =>   /usr/lib(64|)/amanda
/lib64/ld-[^/]*\.so(\.[^/]*)* => /lib64/ld-*.so(.[0-9]|)
```

However, not all of the contexts in the current reference policy are so easy to translate. For example, what does
/usr/(local/)?(.*/)?jre.*/libjvm\.so(\.[^/]*)*
actually specify? To convert such a context, one can check a live filesystem to see what files actually get labeled by the context, or one can consult an expert familiar with the software in question to see what files are actually intended to get this label.

3. Compilation of the FCGlob Contexts and Usage of the Compiled Form

Currently, file contexts are stored in a linear list since this 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.

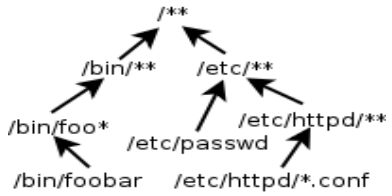

Figure 3: An example FCGlob tree.

This new form of the file contexts makes *matchpathcon* much faster and also makes new features possible. 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. This tree traversal is faster since it is logarithmic in complexity, as opposed 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.

4. 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 *libselinux* or *semodule*. 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 *libselinux*. The general idea of the prototype implementation is we fool *libselinux* so that it thinks it is dealing with an old style file_contexts. 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 creating several programs and methods:

- 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 an 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.

5. Results

Some statistics of speeds and optimizations, how many FCGlobs there are vs. FCs and other general lessons learned during the implementation will be presented once the prototype is implemented. This is expected to be completed by the time of the presentation in March 2007. If possible, this section of this paper will be updated at this time.

6. 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. 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.