Xperteyes - keeping your system under control

Pim Buurman
X|support
Pim.Buurman@summix.nl

Introduction

The increasing complexity of computer systems and infrastructures asks for good solutions. The installation and configuration of all components should be valid (correct syntax), consistent and adhere to the standards of the organization. Fortunately, many configuration settings can be set with a graphical tool, making the task simpler and sometimes more intuitive. Therefore, syntax errors in configuration files hardly occur any more. Installing is made easy with the emerging of install packages, so each file will have the correct owner and group and permission bits. In theory, installing new packages will not interfere with already installed packages, nor will it introduce a security risk.

But configuring a system is still prone to human error, and many inconsistencies may occur. Some errors in configurations are noticed immediately, but many others may slumber for a long time, until an irregularity happens to the system. Then an emergency will occur due to a problem that was introduced three months ago.

Another source of problems are deliberate attempts by people to change the system to their own ideas. These people can be both outsiders (hackers) and regular users. A misguided administrative action may introduce serious security problems, but even a simple typo may lead to a break in. It would be useful for system administrators to detect configuration problems as well as the occurrence of a security breach.

The system administrators are not the only users who want a report on the current configuration. Their manager will be interested to measure the general quality of the computer systems. And the EDP-auditor will have an easier job.

Xperteyes is a solution for these problems. It can check many properties of a system within a few minutes, as well as checking several systems against each other, for instance to check that a fail-over system is configured correctly. Therefore, Xperteyes can be used on a daily basis to find incorrect or insecure configuration settings. Major changes can be tested before the system goes live, important minor changes can be tested immediately. Furthermore, implicit management rules can be made explicit, making it easier to maintain them and to train novice administrators. Experienced system managers will have more time to do complicated jobs, knowing that they can check everything before going on line.

For daring system administrators, Xperteyes offers the possibility to repair the system based on the reported problems. Because this is a dangerous action, repairing needs to be governed by a user.

Each system administrator thinks his computer system has a unique combination of applications and that only he knows to address all of them. This is usually true, so Xperteyes initially handles only the most common configuration properties. But the application is easily extendible. It will take only a few hours to extend Xperteyes to read the contents of a configuration, and check these contents for specific values.

Architecture

Xperteyes is an application containing several programs. The programs can be divided into three classes: collectors, checkers and viewers. The collectors gather all (or at least all interesting) properties from an object of your computer system. The object may be as simple as the password file, or can be as complex as the file system or the domain name configuration. The information is saved in a generic format, which we will call a collection.

The checkers use several collections, together with the criteria the user wants to check, and produce a new collection, based on the original collections and enriched with the failures of the criteria. Because the user can specify both the criteria and the collections, it is possible to check if a system has not changed since yesterday, or that the users on the Mac server are the same as on the Linux clients.

The viewers are used to browse or print the collections. Browsing through the system data of remote machines is possible, but the main purpose is viewing the failures in relation to their environment.

A good performance was one of the design goals. The performance of the collectors is usually limited by the performance of the OS. E.g. the file system collector uses typically only 30% CPU time, because it is disk read bound. The performance of the checkers is CPU bound, checking typically 100,000 properties per minute. The viewers are visually bound; they are only slow when many results need to be shown.

Collectors

The collectors are intended to gather all relevant information about a system. Each collector can handle one object. This object can be a UNIX configuration file (`/etc/passwd`, `/etc/group`, ...) or a more complex object as the file system. The result is either a list of records, or a tree of records. Each record in the result has the same fields, one field per property. It is possible that a special property (e.g. the `rdev` for device files) is only set for a limited number of objects.

The collectors are basically read-only, so the system should be nearly unaffected by it. On purpose, the collectors use the permissions of the owner of the process to probe the system. If the permissions do not allow a full scan of the system, the checkers may give unexpected results. Any error during the collect process is added to the result, including permission errors.

Each collector is principally system specific, so the file system collectors of Mac OS X, generic UNIX and Windows differ. However, they all specify a tree where each node has the name, owner, group and content modification time of a file system object. The Mac file system has amongst others Mac-type, creator and alias as extra fields.

A supercollector is built from a group of collectors. Its result is a tree containing the lists and trees of its subcollectors. In contrast to the simple collectors, this tree has records that depend on the place in the tree.

The currently defined collectors are:

- for generic UNIX:
  - file system - all file object properties (not the data), including optional storage of checksum or content of some files.
    This collector is configurable, e.g. to skip NFS-mounted file systems and to store the contents of `/etc/profile`.
  - many configuration files, mostly in `/etc` (`passwd`, `group`, `fstab`, `named`, ...)
  - configuration API (`getpwent`, `getgrent`, etc.)
- for Mac OS X:
  - extended UNIX file system
  - generic UNIX configuration files and configuration API
  - netinfo, open directory
  - installed packages
- for Linux:
  - generic UNIX
  - rpm information
- for Windows:
  - file system
  - registry
  - user/group/machine database

Checking

The checkers test if the system satisfies some criteria. Unlike many other tools, the checkers in Xperteyes have no internal criteria. The user defines the criteria, and different criteria can be tested on the same collection, either in one run or in several runs.

Currently, three different checkers are available: a general one for checking complex criteria, one for a fast comparison (diff) between two collections, and one for checking the current situation against the installed packages.

The criteria in the general checker consist of a list of requirements. Each requirement states which tests should be applied on which records. Optionally a context and severity may be given, so the failures can be more easily categorized in the viewers. Also the different texts may be adapted to the particular case. In the viewers, the text "The home directory of 'pim' is not owned by him" can be clearer than the default message "uid 0 is not equal to 501".

The general checker uses one or more collections and applies a set of criteria to them. To simplify the criteria, they are most easily expressed as general requirements with exceptions. This means that we write:

```
require('/', perm_le( 'rwxr-xr-x' ), selection=ALL)
require('/tmp', [perm_eq( 'rwxrwxrwt' ), uid_eq(0)])
require('/tmp', perm_le( 'rwxrwxrwx' ), selection=ALL_CHILDREN)
```

The checker applies these requirements so that the more specific tests block the less specific ones, where more specific means that the test is more specific (`perm_eq` is more specific than `perm_le`) or that the object of the requirement is deeper in the tree (`/tmp` is deeper than `/`). Thus, the given requirement are interpreted as: all objects from `/` down (i.e. all objects in the file system) should have permission <= rwxr-xr-x, but `/tmp`

should have permission == rwxrwxrwt and uid == 0 (root!), and all objects from `/tmp` down, but not `/tmp`, should have permission <= rwxwxrwx. Note: permission checks work bitwise.

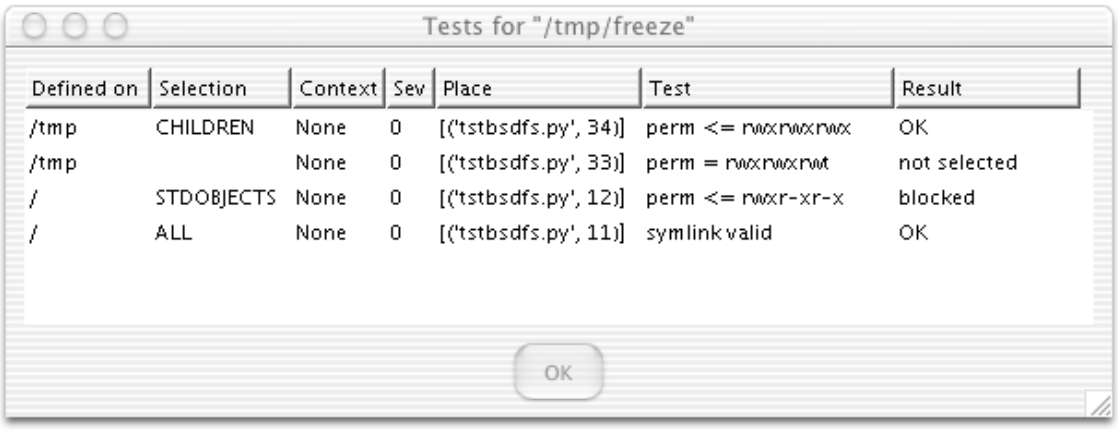

Fig. 1: Test results for `/tmp/freeze`.

The currently available tests consist mainly of simple checks of one field against one value, or against a range or interval. Also some more elaborate tests are defined, for example whether a file is a hard link to another file. New tests can also be defined as logical combinations of simple tests, e.g. `suid_root_exe()` can be defined as `And(perm_eq('rwsr-xr-x'), uid_eq(0), format="not a setuid root executable")`.

Using these basic tests, one might expect that only a static check can be defined, e.g. principally independent of the current situation on the system. But the general checker is more powerful, because the criteria is a Python script that defines the requirements. It is possible to generate requirements based on the contents of some objects. So it is very easy to define that the home directories of the users are their private area on the system. Based on the password file contents, tests are generated that the home directory trees belong to the correct user, and that its root has permissions equal to rwx------. Note that you want to skip entries for nobody and bin, hence the `REAL_USER_SELECTION`.

```
for el in x.elements('Passwd:File', REAL_USERS_SELECTION):
      x.require('FileSystem:' + el.home, [perm_eq('rwx------')])
      x.require('FileSystem:' + el.home,
                  [uid_eq(el.uid), gid_eq(el.gid)], ALL)
```

The checker can be called with more than one collection. Therefore, it is easy to test several systems against each other. The checker can be used e.g. to find both stale accounts and missing accounts on local systems by comparing it to a server. It is allowed that the local systems are running Windows (in various flavors), while the server is a Mac OS X Server. Two fail-over systems should be compared regularly, because most fail-over software depends (indirectly) on configuration parameters that are not forced to be the same (e.g. passwords). And comparing the collections of one system that are created at two time points shows exactly what is changed. This can be used to test if the installation of a package does not interfere with existing packages, and if a de-installation restores the old situation.

In our view, there is a difference between a test failure because the property has an invalid value, and a test failure because the object is not in the collection. Therefore run

time errors are given if the object of a requirement is not in the collection, or if the test cannot be applied.

The checker adds the errors and failures to the collection, so the viewer can show not only the failures, but also the environment, making it easier to interpret the results.

Viewing

The viewers are used to see the results of the collectors and the checkers. The graphical viewer shows an annotated tree of the results. The criteria failures and the run time errors are shown in two separate windows. Errors and failures are also indicated on the objects self. To guide the user to problem areas, the items in the tree are colored to indicate that in that subtree some problems exist. To understand the requirements on one object, it is possible to view the requirements that are applicable to that object, and the result of those requirements.

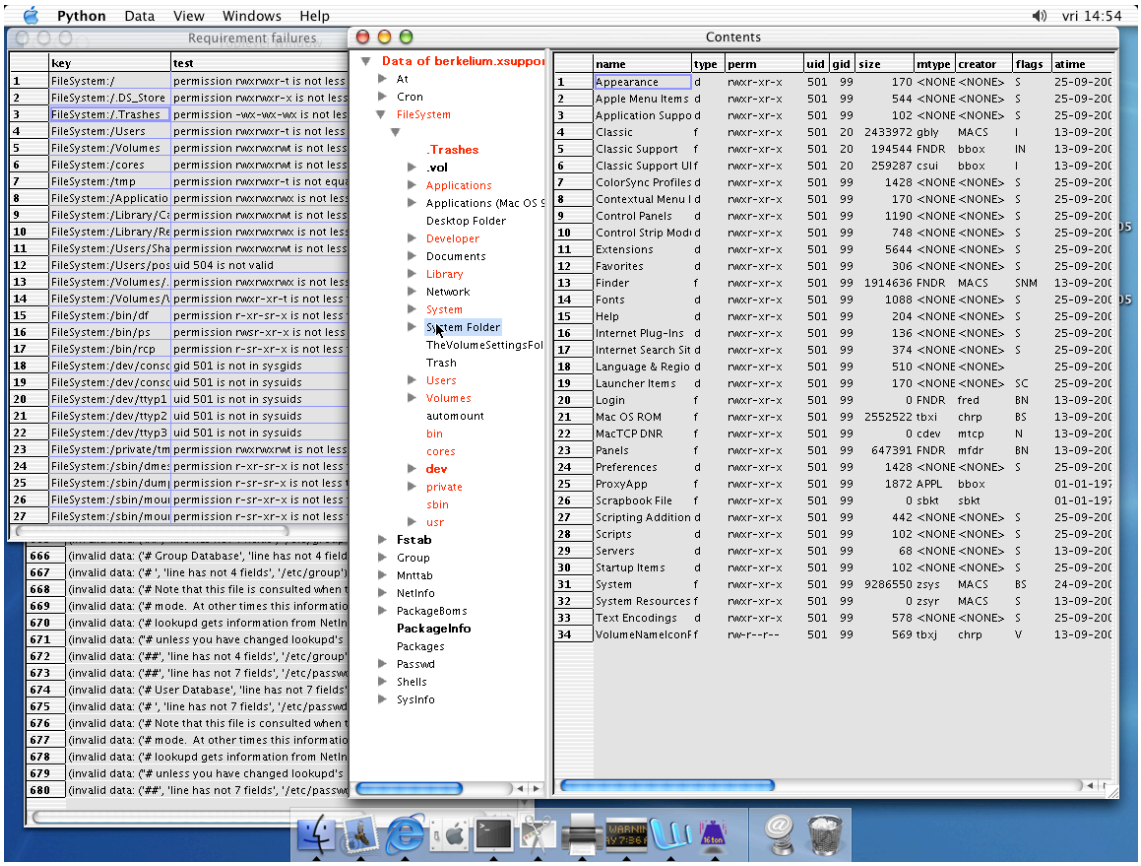

Fig. 2: Annotated tree window and errors window for a Mac OS X collection

By selecting an error or failure, the object is also selected in the annotated tree.

Repairing

For simple failures it can be convenient to repair the failure from within the viewer. Instead of writing a small shell script which may repair too little or too much, or which may even corrupt your system, Xperteyes has the possibility to use the viewer to repair the system.

A repair action can be executed on the objects that are selected in the criteria failures. This repair action should be defined previously in the criteria. The repair action will

only change the system if certain conditions are met: usually the object should still have the same properties as in the collection, but a less restrictive condition may be used, e.g. the contents are not changed.

The person defining the criteria with repair actions should be aware of the dangers of repairing. The person actually applying the repair actions should check that the failures really should be repaired. An example of the dangers is easy. Suppose that a malicious user has hard linked `/etc/passwd` in his home directory as `'.mailrc '`. One of the rules states that all objects in a home directory belong to the user. A repair action may be: set the uid to this user. It is clear that in this case the object should not be repaired, but removed. Very accurate definition of the repair action is necessary, and correct checking is also necessary.

Extending

One of the major design goals was to create an easily extendible tool. This is based on the idea that it should be easy to include some information on the system that is specific for the organization using the system. Suppose that our own company has the policy that all objects in `/xsupport/bin` should be statically linked executables. This information is not collected by default, so we need to get this additional information, and define and apply an additional test.

The additional information is most easily added to the file system collector. An extra field, called 'magic', is defined. A function is defined, which will call the external program `/usr/bin/file` and store the result in the 'magic' field. This function is added to the file system collector in the same way as the checksum and content functions.

In the file system collector configuration the magic is required for all files in `/xsupport/bin`. Now we can run the collector and the collection will contain the file magic of the files in `/xsupport/bin`, as the viewer will show.

A new test needs to be defined, which will test the magic field. This is a straightforward test, having as argument the content of the magic field. In the criteria this test is applied to the children of `/xsupport/bin`. The general checker can now generate failures for this test, and with the viewer we will see if the system satisfies our criteria.

The time estimate for implementing and testing this new feature will be about one day for someone not familiar with the code.

Portability

Xperteyes is currently running on Mac OS X, Linux and Windows. It is implemented in Python with one extension module in C, for compact storage and fast retrieval of large data sets. The graphics are based on the wxPython extension module, which is a portable (Windows, Mac and X11) graphical toolkit.

For the collectors, a few wrappers are written in C to read the system data and convert it to Python structures. This code is the only OS dependent part of the application; all other code is Python and will run on any platform with Python. Also, the data is portable, making it possible to check on a Mac OS server the Windows clients against a Linux server. The data is self-contained, so it is not necessary to use the collection on the same kind of platform.