Use of Cfengine for Automated, Multi-Platform Software and Patch Distribution

David Ressman & John Valdés – University of Chicago

ABSTRACT

Good UNIX system administration practice includes among its many tasks the proper configuration of system files, installation and maintenance of third party software, and maintenance of system security, including regular updates of operating system (OS) patches. For a small number of systems running only one or two OSes, keeping up with these tasks isn't too difficult. However, as the number of systems and OSes increase (and the number of staff remains constant), these chores can quickly become overwhelming.

This paper describes our planning, development, and deployment of a system that provides automated software distribution, patch installation, and OS configuration through the integration of GNU cfengine [Bur95], MySQL [MySQL00], and a few custom written Perl scripts. It is meant to be less of a tool description and more of a discussion about the various aspects of designing a multi-platform software and patch distribution system, and the benefits of integrating those systems into a configuration management system such as cfengine. Designing and developing our system has been a time-consuming endeavor, but it has proven to be well worth the effort.

Background

Our network was out of control. In our department, we have over 100 UNIX systems running more than half a dozen UNIX based OSes (more than a dozen when counting different OS versions). The vast majority of the systems share a similar role, but they are all configured slightly differently to suit their particular users' needs. With only two systems administrators, it was extremely difficult to handle the day to day maintenance for each of these systems. We fell so far behind that the majority of our days were spent merely fighting fires. Because of this, we had very little time to respond to requests for software updates or new software installations and even less time to make sure that all of our systems were running at the current OS patch level. This left us with two large problems:

- Most of our systems had very old copies of software. Since there was so little time to spend upgrading software, we would only upgrade or put new software on a machine when a user would specifically ask for it. It's not hard to imagine what this led to; we ended up with a number of different versions of the same software all installed slightly differently across our systems. This made upgrading software a much more difficult task than it had to be, so unless there was a specific need for it, software wouldn't get upgraded at all.
- Most of our systems were unpatched against known security problems. The chore of manually applying patches to over 100 systems one by one every time a new patch report comes out is enough to give most systems administrators nightmares. Because of the great effort involved, most of our computers only had whatever patches were current at the time of OS install and whatever patches were applied to the systems after mass break-in attempts.

It was clear to us that we needed some way to manage the distribution and installation of software and patches for our systems. We felt that eliminating these two problems more than justified however much time would be spent developing a new management system.

Requirements for a New System

Over the period of several days, we brainstormed about what our ideal system should include and came up with the following rough list:

- Software distribution and management
- Patch distribution
- Centrally controlled configuration
- Ease of use
- Low cost
- Security
- Flexible host configuration
- Centrally controlled "pull" of software and patch distribution
- Portability
- Autonomous operation

Software Distribution and Management

Our system needed to be able to handle the distribution, installation, upgrade, and removal of most or all of the software not supplied by the OS vendor that was needed on our computers. Additionally, it would be useful if our system could track information about installed software packages, such as a list of all files included in a software package, the original source of the software, and how it was compiled.

Patch Distribution

Our system needed to be able to handle the distribution and installation of all vendor supplied OS patches. It should also be able to handle any post-installation steps required by the patch, such as restarting a patched daemon. For maximum security, it would need to make sure that all of our computers were as up to date with their specific OS's patch list as possible.

Centrally Controlled Configuration

As much of each machine's configuration as possible (e.g. automounter configuration, printer setup, firewall configuration, syslog configuration, etc.) should be initiated and tracked by a central system. Likewise, the list of all software and patches which should be installed on each machine should be managed by a central system. The exact configuration state of each machine in the system should be reproducible in the event of a hardware failure or any other event that would require a fresh OS install.

Ease of Use

Our system had be easy to use once placed into production. A system that was complex and difficult to use wouldn't be much of an improvement over manually maintaining our systems and would likely go unused. The system should also be easy to maintain and require a minimal amount of work to keep operational. The less work it required, the more time we could devote to other projects. The more time we could devote to other projects, the more productive we could make our users.

Low Cost

Our system must be inexpensive. We have a minimal operating budget and can't afford multi-host licensing fees and yearly software maintenance costs. This unfortunately ruled out most commercial software options.

Security

Security was a primary motivation for our undertaking this project. Our new system must allow us to achieve a higher level of security across our network than we previously had. This necessitated that our system be able to maintain a current OS patch level on all of our users' computers. It should be able to quickly upgrade software across all systems whenever security bugs were found in third-party software. Any central servers and processes used by our system should also be as secure as possible.

Flexible Host Configuration

We can classify our machines as belonging to one or more specific groups (i.e., NFS servers, laboratory workstations, members of specific research groups, etc.). Depending on what group(s) a machine belongs to, our management system should be able to install specific software packages and make specific operating system configuration changes. For example, machines grouped as laboratory workstations may need additional data analysis packages installed on them, while only machines belonging to a specific research group will need a print queue defined for a printer which belongs to that research group.

We also needed to be able to manage software and OS configuration on a host by host basis. Our system should also be able to customize a machine's software and OS configuration beyond the configuration it receives by virtue of its group classification.

Finally, it should also be possible to customize a machine's configuration files based on any particular third-party software packages or vendor supplied patches that are installed. For instance, for machines with the Apache web server installed, we need a way to manage the server daemon's configuration and log files.

Centrally Controlled "Pull" of Software and Patch Distribution

The software and patch distribution, while configured centrally, should work on a system where the patches and software are "pulled" from a central server, rather than "pushed" by the server. By having "smart clients" and a "dumb server," we hoped to minimize the amount of damage that could be done to our system by the temporary loss or compromise of any one host. In our ideal situation, any or all of the functions of the server could be moved to different computers with a minimal impact to the system.

Portability

Our system must be portable. Our needs require it to run across multiple OSes, including SunOS 4.x, SunOS 5.x, IRIX, AIX, Digital/Tru64 UNIX, OpenBSD and Linux (including multiple architectures, such as Intel and Alpha). Fortunately, we do not need to support any non-UNIX OSes at this time.

Autonomous Operation

Most importantly, our system needed to do all of the above with as little intervention from us as possible.

Home-grown or Public Domain Software?

At this point, the only thing we knew for sure about our new system was that it would have to have three major parts to it: software distribution, patch distribution, and configuration management. The question we next asked ourselves was how much existing software could we use, and how much would we have to write ourselves? Our problems certainly weren't unique, and we didn't want to reinvent the wheel. Likewise, we aren't software developers and didn't have the time to write a complete system from scratch, so we wanted to make use of as much pre-existing, maintained software as possible.

Software Distribution

We spent a few days looking around on the web for software distribution systems but were only able to find a handful of references. When we took a more careful look at the few systems we were able to find, we noticed that most of them relied on package directory trees being pushed out from a server (rdist style), copied over from an NFS server, mounted from an NFS server with symlinks set up in the client's local directory structure, or some combination of the three (Xhier [Sel91], Depot [Col92], Depot-Lite [Rou94], GNU Stow [Gli96], opt_depot [Abb97], SEPP [Oet98]).

We had already ruled out a server pushed distribution system, so the only other choices for existing systems were distribution through AFS or NFS from a master server. AFS was not an option we could consider because it was not available for all of our target platforms nor was it freely available for most of our platforms. While NFS may have been an easy choice for the authors of these other distribution systems because of a pre-existing NFS architecture, it was not an easy choice for us. We had made very little use of NFS in our department for software sharing or distribution.

Historically, we had always avoided running software mounted from an NFS server in favor of each machine having local copies of all of its software. In doing so, we minimized the amount of damage the loss of any one computer or the loss of our network could do to any other computer. We're employed to keep our users as productive as possible by allowing them to think about astronomy and astrophysics and not about whether the NFS server that is holding the software they need to use is available. Inevitably, computers and routers will crash or have to be brought down for maintenance; we didn't want to use a distribution system that would cripple all of our users' computers through the unavailability of a single server. Having our software distributed from or mounted on an NFS server would increase our dependence to a specific server more than we felt comfortable with.

Additionally, given our requirement to support multiple operating systems and architectures, using an NFS-based distribution system would have required that we either maintain a separate NFS server for each OS/architecture combination or maintain a complicated, multiple-architecture directory structure on a single NFS server. Lastly, we have software which must be installed locally on each system, such as software containing kernel modules and security related software (e.g. tcp-wrappers, ssh, etc.); if we used an NFS-based system, we would still need something else to manage the locally installed software.

After ruling out server push and AFS/NFS, we were unable to find any free software distribution systems which we could put to use in our department. It was clear that our best option was to write our own software distribution system.

Software Packaging

We needed our system to handle the installation, removal, and upgrade of 100 or more different third-party software packages. Since we had ruled out a network filesystem based distribution method, we immediately realized that we would have to use some sort of packaging system to get our software from our distribution server(s) to our client computers.

We came up with the following three options:

- Use the native package format for each OS.
- Create a home-built package format using shell scripts and tar.
- Use an existing, multi-platform package format.

Of these three, we immediately ruled out using each OS's native package format for software distribution. While it (arguably) might have allowed for the most trouble-free integration of the software distribution system with the client computers and their OSes, it also would have been the most work to set up and maintain. Since each OS has its own unique packaging format, every piece of software we wanted to distribute would have to be packaged up in as many as eight completely different ways. The last thing we needed was to make packaging more complicated than it had to be. Besides, we had had enough experience with some of the package formats to know that a few of them were complicated at best and downright obtuse at worst.

A home-built package format using tar and shell scripts would certainly be the easiest to put together, since every OS we were using had tar and a Bourne-compatible shell. Given that, no additional software would have to be added on the computer in order to install, upgrade, or delete packages. All we would need to do is:

- Write shell scripts to install, remove, and query our available packages,
- Compile each software package once for every OS for which we wished to have the package available,
- Tar up the software into our package format,
- Finally, put the tarball up on our distribution server(s) to be pulled in by our clients.

It did have one drawback; we'd still need to get the package from the server to the client. Since we'd ruled out NFS, the only reasonable ways we could think of to get the package to the client over the network were HTTP or FTP. That alone wouldn't be enough to stop us from using a home-grown package manager. We could install a program like GNU wget or NcFTPget when we loaded the shell scripts on the client. It would be bare-bones and wouldn't have a lot of fancy bells and whistles, but it wouldn't be a bad solution.

When looking around for a Few Good multi-platform package formats, we came across a couple lesser known ones that looked like they might be able to work, but the clear leader in that field is the Red Hat Package Manager (RPM) [RPM00]. Aside from already having been ported to every operating system on which we would need to run it, RPM has a very active development group, a very broad user base, and has been thoroughly tested.

RPM also extends beyond just a packaging format into a development environment that can control the entire packaging process from the compilation of software from source code on up to the installation of the packages. It also has the added benefit (or hindrance, depending on how you look at it) of extensive package dependency awareness, and has a built in FTP client so that it can retrieve packages from a remote FTP server and install them in one step.

Building RPM packages would require more initial work in that we would have to write a package specification (spec) file for each package. The spec file contains all the information that is needed to compile, install, uninstall, and upgrade the package for every OS for which the package would be available. However, we didn't think this was necessarily a bad thing. It would enable us to document from where we got the package's source code (not always an easy thing to remember or find). It would also very clearly show us every step we would need to take in order to compile and install the software (again, not always an easy thing to remember). Additionally, once we had written the spec file, updating packages to newer versions of the software would usually be very easy. Often, all that's required to build a new RPM package when updated versions of software are released is a one-line change to the package's spec file followed by an "rpm -b -a package.spec." The RPM manual [Bai99] contains an excellent description of the entire process of building an RPM.

The only problem we saw with using RPM would be the initial installation of RPM itself. We decided that if we were going to use RPM, we would build an RPM package in each OS's native package format, install that package on each system, and let RPM install all the additional software packages.

While the tar and shell script option would certainly have sufficed, we felt that the added functionality of RPM (especially the FTP client) was worth the effort of learning how to write the RPM spec files and building RPM packages for each OS. The tar and shell script option also would have required that we reinvent much of the functionality already in RPM. Therefore, we felt our best option was to write our software distribution system around the Red Hat Package Manager with packages distributed from an FTP server.

Patch Distribution

However disappointing the lack of information on available software distribution systems was, the lack of information on patch distribution systems was doubly so. We were unable to find any information on multiplatform patch distribution and installation systems. In fact, the only useful information we were able to find about any kind of automatic patch download and installation software was a reference to a program named PatchReport in an article in the October 1997 issue of ;login: [Sin97]. PatchReport was a Solaris-only Perl script that would compare the current patch state of the machine it was being run on against Sun's patch cross-reference file. PatchReport would download any patches that were missing and install them. Since a large fraction of our computers were running Solaris, this was a pretty good start, but since it was a Solaris-only utility, it was clearly not going to be our multi-platform patch distribution and installation utility.

We were unable to find any such utilities for IRIX, Red Hat Linux, AIX, or any other OSes we would have to support. If we wanted a true multi-platform patch distribution and installation system, it appeared we would have to write one ourselves.

Since patches are supplied by the OS vendor or development team, there was no way to use a common patch format. Our only choice would be to write a system that was intelligent enough to recognize what OS it was running on and download and apply the right patches. Since we were already going to use an FTP server to distribute our software, we decided to use an FTP server to distribute the OS patches as well.

Configuration Management

When we started the search for configuration management software, we immediately found hundreds of references to a program named cfengine. After a cursory glance at the online documentation we found on the cfengine web page, we decided that this software package was definitely worth a closer look. We downloaded the cfengine reference manual [Bur99] and went home for the weekend to read it. The more we read, the more we realized that cfengine was a perfect choice for our configuration management system.

Cfengine would allow us to control every aspect of a machine's configuration that we thought would be necessary and then some! A complete description of cfengine is beyond of the scope of this paper, but briefly, cfengine will let you manage:

- Copying of files, both locally and remotely
- Editing of files
- Creation, removal, and maintenance of symbolic links
- Filesystem access control lists (ACL)
- File and directory permissions and deletions
- Filesystem tidying
- External command execution
- System and user processes

among many other things.

All of cfengine's actions can be conditionally applied based on whether or not certain "classes" (cfengine's name for "groups") are defined. Cfengine provides predefined classes based on the OS a given system is running, the hostname of the system, the day of the week, etc., and also allows you to define your own classes. Cfengine can also define classes dynamically at runtime, so that classes can be defined, for example, only if a certain action was carried out. Given cfengine's actions and ability to group actions based on class, it was clear that we could use cfengine to manage system configuration which applies globally to all of our computers, to any individual computer, or to any predefined or dynamically defined group of computers, as we required.

As an added benefit, cfengine has a framework for allowing users to write their own modules in any programming or scripting language and merge them into a stock cfengine run. Through this, you can add nearly any functionality to cfengine without having to actually modify cfengine itself. Modules can use and define classes as well, giving them the same control and flexibility as cfengine's builtin actions. Because of this capability, we decided that we would implement our software and patch distribution system as cfengine modules so that we could use cfengine's class mechanism to manage configuration based on installed software and patches.

Summing It Up

We now knew that our solution would include a home-written software distribution system built upon RPM software packages, a home-written patch distribution system, and cfengine for configuration management.

Drawing Up a Rough Design

Having decided on the features we would want and the software we would use and create, it was time to figure out what we would need to write into our software so that it would meet our requirements.

Class Definitions

As we've mentioned, all of our computers can be classified as being part of one or more larger groups that share a common functionality. There are personal workstations, laboratory workstations, data servers, computational servers, etc. The systems can be further divided into smaller groups; there are groups of machines that are used for a specific research project, share a common user base, need the same software, etc. Depending on which group a machine belongs to, it will have different software installed and its configuration files will be different.

To handle this varied configuration, we would simply make use of cfengine's class mechanism. For this, we needed to define classes which reflected the grouping of our systems, and then create cfengine input files which would associate our systems with our classes and run whatever actions are necessary in order to configure the systems as needed.

Central Configuration and Control

We needed our system to centrally manage all system configuration. Having all configuration information resident on a central server makes it easy to change configurations for multiple systems with a single edit, minimizing mistakes and eliminating inconsistencies between systems. Cfengine naturally lends itself to central configuration. In the most common method of cfengine use, master copies of cfengine's input files (the files that tell cfengine what commands to perform on the computer) reside on the cfengine server where they are pulled in by the client systems through cfengine's internal network file copy protocol.

We also needed our system to centrally manage all of the available software and patch information for all of the OSes we use with easy expandability for any future OSes we might need to support. Every single system needs to be able to talk to the central system and be told exactly what software packages should be installed and what patches should be applied.

We came up with two different possible ways of distributing this information:

- Have each client pull in a group of text files that describe the available software, each machine's software configuration, and each OS's patch configuration.
- Have each client query a relational database to obtain its software profile and patch information.

A text file option would be the easiest to set up. However, when we did the math to figure out how much data would be passed around in those files, it became clear to us that keeping track of over 100 software programs on over 100 machines would require more than 10,000 different entries in our text file – for the software subscription information alone.

Having two people maintain this text file was a recipe for disaster. No matter how careful we were, we were bound to make syntax mistakes and typographical errors. The more entries we had in this file, the harder it would be to track down any mistakes. While we could probably keep it under control with some difficulty, it was very clear that this option would not scale as the number of hosts and software packages increased. Maintaining a separate file for each computer with its own software information would have been just as bad.

Having this information stored in a database was clearly the better choice. With properly written front-end scripts to maintain the database information, it would be very easy to keep the database relatively error-free, and make the inevitable errors very easy to find and correct. In addition, most databases are designed with scalability in mind and could hold far more detailed information for far more hosts than we would ever need to support.

Hence, we decided to store all the software and patch information in a central database. Of the free database packages available, we chose MySQL because we had had some familiarity with it in the past and because we felt that the development community was a little larger than the other two databases we considered, mSQL and PostgreSQL. Any of the three would have been a suitable choice for our project.

Portability

RPM and cfengine had already been reported to run on every OS on which we would need to run them. To assure portability, all we needed to do was to make sure that our cfengine modules were written in a language that was portable across all of our systems and had the ability to communicate with a MySQL server.

We made a list of all the scripting and programming languages we could think of with MySQL support and came up with the following: Perl, Python, Tcl, C, C++ and Java.

Of those, we immediately ruled out three choices. We decided not to write our modules in Python out of personal preference. We ruled out Tcl because the MySQL support was not as strong as we would have liked. We also ruled out Java because a Java virtual machine was not available for every operating system we would need to run it on.

With Python, Tcl, and Java gone, we were left to chose either Perl, C, or C++ which prompted an interesting question: were there any specific benefits or hindrances that made scripting languages a better choice than compiled languages (or vice versa)?

While personal preference was bound to affect our decision, we tried to be as objective as possible. These modules would not be doing large amounts of heavy computation, so any speed increase given by choosing a compiled language was likely to be negligible. These modules would be distributed from our central server through cfengine, which would mean very little difference between distributing one Perl script to all OSes or distributing a binary for each OS built from the same source code.

If we wrote our modules in Perl, we would need to make sure that every computer had a copy of Perl and a copy of the MySQL module for Perl. Whereas if we used C or C++, no additional software would need to be installed on the client machines; all we would need is a copy of the MySQL libraries and header files on one system running each OS in order to compile the C or C++ modules.

However, if we used Perl, it would be highly unlikely that we would ever use any of its more complicated features that might cause our scripts to react differently depending on what operating system they were being run on. As a result, if our scripts compiled and ran on one OS, they would likely compile and run on all the others. We wouldn't need to recompile our modules on eight different OSes every time we made a small change to either of our modules. Releasing new versions of the modules would also be easier because we wouldn't have to compile and test the program on all of our OSes with every minor change.

We eventually chose Perl because of the ease of development and maintenance it would give us over C or C++. Every computer we would be responsible for maintaining would have a copy of Perl on it anyway, and it wasn't much trouble to add the MySQL module to it. It's not clear that this was the best choice, and in the future, we may rewrite our modules in C or C++ as an experiment.

Autonomous Operation

Building our system through the use of cfengine modules was the perfect solution for us. Because of the way cfengine handles class definitions through its modules, any of cfengine's internal functions could be performed on our systems based on what patches or software packages we installed – all automatically! All that is required is that our modules define classes for the patches and packages that they install; we can then have cfengine run commands based on these class definitions.

The possibilities are almost endless. We can have cfengine restart daemons after installing OS patches that update them. Cfengine can change inetd.conf and send inetd a hangup signal if we install or uninstall a software package that is launched from inetd. Nearly any change that you would want to make to a computer due to the installation or uninstallation of a software package or OS patch can be performed on all of the applicable hosts by merely adding a couple of lines to cfengine's input file.

Below, we show an example of a cfengine input file which illustrates how classes can be used by cfengine to reconfigure inetd based on whether or not the IMAP server package is installed.

```
editfiles:

 imapd::
  { /etc/inet/inetd.conf
  AppendIfNoSuchLine  "imap  stream  tcp
    nowait  root  /usr/sbin/tcp  imapd"
  DefineClasses "inetd"
  }

 !imapd::
  { /etc/inet/inetd.conf
  DeleteLinesStarting  "imap"
  DefineClasses  "inetd"
  }

[later in the input file]

processes:

 inetd::
  "inetd -s"
  action=signal
  signal=hup
```

In this example, we show the use of two builtin cfengine commands, editfiles: and processes:, together with a dynamically defined class called imapd::. The editfiles: command provides a number of actions for manipulating text files, while the processes: command provides actions for manipulating UNIX processes. Through a process detailed in the next section, our software module will dynamically define the imapd::

class based on whether or not the IMAP server package is installed; if the package is installed, the class will be defined, and if the package is not installed, the class will not be defined.

Looking at the example above, you can see that if the imapd package is installed (i.e., the imapd:: class is defined), cfengine will check to make sure that the imap service is enabled in the inetd.conf file. If a line for imap doesn't appear in inetd.conf , that means that the package has just been installed, at which point cfengine will append the line given in double quotes to inetd.conf and then define the inetd:: class (this class will be defined if and only if cfengine edits the inetd.conf file as a result of this check). Later on in the cfengine run when the processes: command runs, cfengine will see that the inetd:: class is defined and send the inetd daemon a HUP signal as instructed by the input file.

If the imapd package is not installed, then the imapd:: class will not be defined by our module, and so cfengine will run the actions in the !imapd:: stanza. These actions tell cfengine to remove the imap entry from inetd.conf if present and set the inetd:: class if the entry was removed. If the imap line is present, that means that the package has just been uninstalled. In this case, cfengine will remove the imap line from inetd.conf and define the inetd:: class, which will later, when the processes: command is run, cause cfengine to send a HUP signal to inetd.

Similar actions can be taken with the installation or removal of any software package or OS patch.

Summing It Up

We now had a good idea of exactly how we wanted to implement our system. It would be driven by cfengine so that every machine could be configured centrally from the cfengine server. We would have a MySQL database holding up-to-date patch information for each of our OSes and software configuration profiles for each of our hosts. We would need to write two cfengine modules in Perl: one to make sure that the computer was up-to-date on patches, and one to make sure the computer had the proper third party software installed.

Putting It Together (The Hard Part)

Cfengine and RPM

The first step in making this system materialize was to build a cfengine server and familiarize ourselves with cfengine in operation. For our cfengine server, security and economy were our top priorities, so we bought a Pentium-based system and installed OpenBSD on it. We disabled every service except for ssh and ftp, and loaded a highly restrictive ipfilter ruleset. We downloaded and installed cfengine and set up cfengine's daemon to share input files with all of the systems on our subnets. We picked a handful of computers on which to install cfengine. These computers would contact our cfengine server every hour to check for updated input files, download them if necessary, and run the commands in the input files. After two or three weeks of leisurely experimenting, we felt we were familiar enough with cfengine to begin designing a software and patch module system around it.

Since a large fraction of the computers we're responsible for are Sun SPARCs running Solaris, we started developing on our Solaris machines first. After making and installing a Solaris pkg package for RPM, we thought of all the third party software that we would want to install, downloaded the source code, and compiled them into RPM packages. This took about two months and was by far the most time consuming step of the entire process.

Software and Patch Database

Once we made all of the RPMs, the next step was to design and create a database to store all of our software information and patch information. Towards this end, we had to get into the specifics of exactly what kind of information we would want our database to hold. For software, we came up with the following list:

- The name of every package available.
- A listing of what OS each package was available for. Some of our software would only need to be compiled for one or two of our operating systems. For example, our Red Hat Linux systems would already have installed out of the box several packages which we would be building for our other systems. We needed to be able to differentiate, for example, between the "screen" package that shipped with Red Hat and the one that we built from our own spec file for OSes that didn't ship with a copy of screen. It would be very nice for our systems to be aware of which packages are installable and which ones not to bother with.
- The current, default version of each software package, applicable to every operating system.
- The ability to specify different current versions of the software packages for different operating systems. Some versions of a few of the software packages we'd be using simply wouldn't be runnable on some of our operating systems. It'd be great to say, "Everybody run version 1.24 of the 'foo' package, except for you AIX machines; I want you running version 1.22."
- A list of every software package that should be installed on every machine.
- A field in each machine's profile that would allow you to specify a specific version of a software package if you wanted a version other than the current. Some of our users are running software on their computers that require specific versions of packages like Perl or Tcl. It would be necessary to keep these packages from being replaced when we released new versions and marked them as being current.

Package Name	Default Version	Solaris Version	Linux Version	Solaris Build	Linux Build
acroread	4.05-1	NULL	NULL	1	1
tcl	8.1.1-1	NULL	NULL	1	NULL
pam_opie	1.1-1	1.0-2	0.21-3	1	1

Figure 1: Database table holding software information.

After we had come up with that list, we decided to use two database tables to hold this information.

One table holds information for each software package we want to distribute (see Figure 1). Each package has an entry in this table. One column in the table specifies the default version number of each package; this is the version of the package that is applicable to all OSes. Additional columns specify the version number of each package for each OS (Solaris Version, Linux Version, etc.). This version overrides the default version of a package for the given OS. If the value in the OS version column is NULL, then that means that the current version of the package for that OS is given by the value in the default version column. Finally, there are columns which are used to flag the availability of each package for each OS (Solaris Build, Linux Build, etc.). For each package, if the value in these columns is non-NULL for a given OS, then that means that we have built an RPM of that package for that OS. This can be used to distinguish between RPMs which we have built and those that are included with the OS.

For example, looking at the entries given in Figure 1, we can see that an acroread RPM is available for both Solaris and Linux, and that the default version of the acroread package is 4.05-1 for all OSes. On the other hand, a tcl RPM is isn't available for Linux (it wasn't necessary to make one, since Linux already includes tcl) but is for Solaris. Finally, we see that the default version of the pam_opie package is 1.1-1, but that Solaris systems should use version 1.0-2 while Linux systems should use version 0.21-3.

The second table holds information specifying which software packages should be installed on which machines (see Figure 2). For each entry in this table, one column specifies the hostname of the machine for which the entry applies, a second column specifies the name of the package that should be installed on that machine, and a third column specifies which version of the package should be installed on that machine. If the version is listed as "current," then that means that the current version of that package as given in the software table should be used. The collection of all entries for a machine in this table makes up that machine's software profile.

To illustrate, refer to the sample entries in Figure 2. From here we can see that the machine called "mypc" (which is running Linux) should have the current version of acroread installed. As we saw before in Figure 1, the current available version of acroread for Linux is 4.05-1, so that means mypc will have acroread version 4.05-1 installed. Similarly, we can see that this machine will have pam_opie version 0.21-3 installed. Finally, we can see that machine "mysun" (which is running Solaris) will have tcl version 7.6-1 installed, even though the current version of tcl in the software table is 8.1.1-1.

Hostname	Package Name	Package Version
mypc	acroread	current
mypc	pam_opie	current
mysun	tcl	7.6-1

Figure 2: Database table holding host software profiles.

Patch No.	OS	OS Version
107451-04	solaris	5.7
108528-02	solaris	5.8
lpr-0.50-5	linux	6.2

Figure 3: Database table holding patch information.

The list we came up with for the patch database table was much simpler – the only thing we needed was a simple list of every patch for every version of every operating system we'd be maintaining. The table we used for the patch database is illustrated in Figure 3.

Once the database was designed and created, we needed to populated it with our software and patch information. Since we would be the ones making all of the new software packages and deciding what software gets installed on what hosts, it would be easy for us to keep the software database up to date. Every time we released a new package, or wanted to install a piece of software on a host, we would just update the software database tables, either directly through the MySQL command line client, or through a front end (web-based or otherwise).

Keeping the patch database up to date would be a trickier affair. We would have no control over when new OS patches were released, and very few vendors or development teams would send us email whenever new patches were released. We would have to take the initiative of regularly going to each OS's patch information center, downloading any new patches, putting them up on our ftp server, and updating the database so our clients could pull them in.

To be clear, we didn't think going out to every OS's patch information site to check for new patches every morning would be a bad habit to get into, we just thought that we could find better things to do with the time that it would take to do that.

We decided to automate that process and run scripts from our cfengine server to go out and do it for us. Since these scripts would only be run from our central server, we could take existing software (such as PatchReport) and modify it to download new patches to our FTP server and optionally update our database. For OSes that we could not find any such software for, it was easy enough to write quick Perl scripts to check that vendor's WWW or FTP site, download new patches, and update the entries in our database. This way, we could check much more often than we could if we were doing it ourselves. Currently, we check every OS's patch information site every four hours, around the clock. Whenever the script downloads a patch, it emails us a notice containing a one-line description of each downloaded patch.

We should probably mention one precaution, however. In general, it is probably *not* a good idea to install a patch on a system without first examining any documentation that comes with it and trying it out on a test system first. The documentation will mention any special steps which need to be taken before or after the patch is applied, and prudent testing will uncover any potential problems that may be caused by a malformed patch.

In our current system, our patch download script automatically updates our database each time it downloads a new patch from the vendor. This means that our client systems will pull in and install the downloaded patches without our intervention. Given that we are only checking for "recommended" and "security" patches (the two patch types that seem to be well tested by the vendor before release), we aren't overly concerned about any potential damage resulting from automatic updates. We may revisit the wisdom of that decision. If we do, all that will be necessary will be a quick change to disable the portion of the download script which updates the database. We could then just manually update the database after we have inspected and tested a patch.

Software and Patch Modules

Once we had the cfengine server and the software and patch database infrastructure built, all we needed was the client software.

From this point, it was relatively easy to write a cfengine module in Perl to grab a host's software profile, download the RPMs and install them. Since RPM was chosen to be the common package format across all of our OSes, no changes had to be made to this script in order for it to run on the different OSes. When run through cfengine on a system, this module will:

- query the MySQL database to generate a list of all software packages and their versions which should be installed on the system,
- generate a list of all software packages and their versions which are currently installed on the system (by locally running "rpm -q -a"),
- compare the two lists and install, delete and/or upgrade any packages on the system as needed (by locally running rpm),
- define a cfengine class for each software package that remains installed on the system.

After the module completes, cfengine continues its run and can act on any class definitions activated by the module. In this way, cfengine can make any configuration file edits, file copies, process signaling, etc. which are necessary as a result of any software changes, as illustrated with the imapd example in the previous section.

The patch module was more tricky to write. Since all of our OSes handled patches in completely different ways, it was a challenge to make one script that would seamlessly patch as many as eight different OSes. It would be unavoidable to have certain parts of the script dedicated to only one OS, but we wanted to maximize the amount of code that would be common to all the OSes.

As with the software module, the patch module operates by first querying the MySQL database to generate a list of patches that should be applied to the current OS. It then compares this list to a list of patches currently installed on the system. Finally, it downloads and installs or updates patches as needed, using the native patch installation command for the OS. The module defines a class for each patch that it installs for further use during the cfengine run. For example, 107451-xx is a patch for the cron daemon in Solaris 7 for SPARC. Whenever a version of this patch is installed by the patch module, it will define the class 107451 which we can then use in our cfengine input files to tell cfengine to restart the cron daemon.

Some patches may fail to install on some systems. For example, a patch may be for an operating system component which isn't installed on the system (e.g., UUCP), or it may be intended for a specific type of system and not others (e.g., 64-bit systems and not 32-bit systems). While we could use the MySQL database to specify the applicability of a patch to a given system, we decided to only track the OS and the OS version for which a patch is applicable. As a result, our patch module may try to install inappropriate patches on a given system.

Fortunately, at least on the OSes for which we have so far implemented our patch system, the native patch installation tools which our module uses are intelligent enough to determine on their own that a given patch isn't applicable to the system and fail gracefully without damaging the installed OS. As a result, we are comfortable with our module trying to install inappropriate patches. Should the native patch installation tools not have the requisite intelligence, we will have to expand our patch database structure and module to accommodate. Fortunately, our design is flexible enough that this shouldn't be too difficult.

Given that some patches may legitimately fail to install, the patch module tracks failed patch installations using a text file on each system. When building its list of patches to install during a cfengine run, it will remove from the list any patches listed in this file; that way, it won't needlessly try to reinstall previously failed patches each time cfengine runs.

As a final step, both cfengine modules email us a report of their actions so that we can be kept aware of what software and patches are being installed on our computers. The report details which software packages and patches have been installed and/or removed, and includes any failures, errors or unusual output generated by the modules.

Summing It Up

We had now set up our cfengine server and our MySQL database, made RPMs of all our software, written the software and patch modules, and written the scripts to keep our patch information current. It was at last time to roll it all out!

Making It Work (The Fun Part)

Before we had started any work on this system, we agreed that every computer that was to be integrated into our management system would have to have its OS completely wiped clean and the current version of its OS installed. We wanted to minimize the number of versions of any given OS that were being used in our department, and since security was one of our primary motivations for undertaking this project, we also wanted to be assured that the OSes were clean and untrojaned.

We would start by integrating all of our Solaris systems, then our Red Hat Linux systems, and then our IRIX systems. That would cover about 95% of all of our computers, after which we would declare victory and integrate the other OSes as time permitted.

We picked two Solaris machines with particularly understanding users to use as guinea pigs, completed their software profiles, backed up their home and data directories, and marched off to install Solaris 7.

Once we had a clean installation of Solaris, we loaded on the RPM package (in Solaris pkg format), installed the Perl RPM with MySQL support, and the cfengine RPM. Sun's JumpStart program made this process extremely easy to do, all completely automated. We ran cfengine for the very first time, expecting the worst, and waited for the email report.

Much to our surprise, it worked flawlessly. All of the configuration files were modified just the way we had laid out in cfengine's input files, all of the patches we wanted applied were applied, and all of the software we put in the database configuration profiles was properly installed. After inspecting the systems for anything that was out of place, we declared the debut of our system a smashing success and made plans to upgrade the rest of our Solaris systems. At the time this paper is being written, all of our Solaris systems have been upgraded, and we're about halfway through our Linux systems. Upgrading and integrating the Linux systems has proven to be equally painless.

So far, our system has not caused any serious problems, and we've repelled numerous attacks on vulnerable services because our systems are never more than a few hours behind on vendor-supplied patches.

Successes

In this section, we figured it would be appropriate to include a few examples of where our system has clearly shown itself to be more useful to use than our previous hodge-podge method of systems administration.

Breakin Detection and Termination

Cfengine gives you an excellent interface to interact with your system's processes. You can search for specific processes, start them if they're not running, send them signals if they are, and email you with a report of what cfengine has found or done.

Since the people who break into our systems almost exclusively use the compromised systems to run sniffers, IRC bots, or DoS tools, we decided to make up a list of suspicious process names to have cfengine look for and warn us about every time it ran. Besides the usual suspects (more than one running copy of inetd, anything with ''sniff'', ''r00t'', ''eggdrop'', etc. in the process name, password crackers, etc.), we had cfengine watch for any process with ''./'' in the process name.

One afternoon, we got an email from cfengine on one of our computers that had noticed that the regular user of that machine was running a program as "./irc". It wasn't uncommon to see our users using "./" to run programs, nor do we have objections to our users running IRC, but in this case, it was a bit unusual for this particular user to be running an irc process (good UNIX system administration practice also dictates that you know your users).

Poking around the system, we discovered that the person running this program was not the regular user of the machine, but was someone who had evidently sniffed our user's password from somewhere else and remotely logged into his system just minutes before cfengine had alerted us. This person was in the process of setting up an IRC bot and had not yet tried to get a root shell.

At this point, we had to figure out exactly what to do to minimize the amount of damage that this person could cause with our user's password. We were about 40 minutes from the next cfengine run, so we had 35 minutes to make changes to cfengine's input files so that they would be pulled in by all our clients. The machine that this person had broken into is part of a group of computers in a specific professor's research group. Every user who works with this professor has accounts on every one of his machines, and almost all of the users use the same password on all of the machines. That meant that the person who broke into the one machine we had noticed had a password that would likely let him break into about 10 other machines. It was very important to us that we block them from being able to get into all these other machines. It would only be a matter of time before they figured it out and began installing programs all over this professor's research workstations.

Through cfengine, we had set up syslog on all of our clients to log all messages sent to the AUTH facility to a central log server. Based on the logs on the server, we could say with relative certainty that this person had only managed to break into the one system. We also had the IP address that he had come from. We decided that we would boot the cracker out of the compromised account, lock out our user from all of the machines on which he had accounts, and block all IP access to every machine in this researcher's group from the entire class C from which the attacker had come.

In our Solaris cfengine input file, we added a rule for every computer in that professor's group to look in /etc/shadow for the regular expression "^user:.*:" and replace it with "user:*LK*:" (we don't use NIS or an equivalent, so each system as its own local passwd and shadow file). We also made a change in the global ipfilter ruleset to drop and log every packet coming from the class C from which the breakin had come.

Just before the next cfengine run took place, we killed off the intruder's shell. Two minutes later, the user had been locked out of all of the research group's machines until we could get in touch with him and get a new password. The sniffed password the cracker had obtained was now completely useless, and he had lost all access to our machines from the host that he was using as a breakin staging point. We watched the logs and systems intently for the next few days and saw absolutely no sign that this person was trying to get in again.

All within the time frame of one hour, we had detected the breakin, determined the cause, terminated it, and prevented it from happening again. With the old way of doing things, it's doubtful we would have ever caught this person unless he had launched a DoS and caught the attention of our campus-wide network security team. Score one for the good guys!

Breakins Prevented Through Proper Patching

Far less dramatic, but just as satisfying, was a recent experience with a new Linux bug.

In the old way of doing things, we would be relatively free of breakins (as far as we knew) for periods of time, and then some script kiddie would find an exploit and decide to go to town on the university. We've had days that began with the campus network security team giving us a list of 20 machines on our local subnets that they'd seen compromised and had pulled off the network. We would have to drop everything that we were working on for two days to clean up these systems. These breakins would occasionally be for newly found vulnerabilities, but more often would be for vulnerabilities that had been around long enough to have had patches released. We'd get hit because we hadn't had enough time to patch any of our computers against the vulnerability.

Recently, we've seen several attacks against a bug in rpc.statd on campus. When they first started to show up, we braced ourselves for the worst, but after comparing the CERT advisory that addressed this vulnerability (CERT Advisory CA-2000-17) against Red Hat's errata page and the patches we had installed, we found that our system had already downloaded and installed an updated version of rpc.statd that was invulnerable to this specific attack. We had been safely patched for weeks.

Conclusions

So far, our system has proven to be well worth the effort it took to bring it to fruition. We rarely have to worry about system vulnerabilities because all of our systems automatically patch themselves as soon as new patches are released. We update software regularly now because we can easily compile it once, and install it on dozens of systems simply by editing one entry in one database table. We have regained control of our network.

Availability

All of the scripts that we're using to run our system, including the patch module, software module, and the CGI script we use to configure the software profiles for all of our hosts are freely available at: http://astro.uchicago.edu/~davidr/cfengine-tools/ .

Acknowledgments

Our system has proven itself useful in more ways than we ever could have expected. A great deal of credit goes to the author of cfengine, Mark Burgess. Through the use of his wonderful software, we've been able to bring order to our systems that we would have never had been able to otherwise. We would also like to extend a special thanks to our LISA shepherd, David Blank-Edelman. Without his seemingly endless patience when it came down to the last few days, our paper would not have been nearly as good as we feel it is now.

About the Authors

John Valdés has been playing with UNIX since 1986 and has been working as a System Administrator since 1990. He currently manages systems for the Department of Astronomy and Astrophysics at the University of Chicago. John can be reached via email at valdes@uchicago.edu.

David Ressman has been a UNIX System Administrator since 1996. He currently works for John in the Department of Astronomy and Astrophysics at the University of Chicago. He enjoys referring to himself in the third person, and he hopes to begin college in the Fall of 2001. David can be reached via email at davidr@oddjob.uchicago.edu.

References

[Abb97] Abbey, Jonathan, *The opt_depot Web Site,* http://www.arlut.utexas.edu/csd/opt_depot/opt_depot.html.

[Bai97] Bailey, Ed, *Maximum RPM, Taking the Red Hat Package Manager to the Limit*, August 1997, Macmillan Computer Publishing.

[Bur95] Burgess, Mark, "Cfengine: a site configuration engine", *USENIX Computing Systems*, Vol 8, No. 3 1995.

[Bur99] Burgess, Mark, *Cfengine Reference Manual*, http://www.iu.hioslo.no/cfengine/docs/cfengine-Reference.html .

[Col92] Colyer, Wallace and Walter Wong, "Depot: a Tool for Managing Software Environments", *Proceedings of the 6th Systems Administration Conference (LISA VI),* 1992.

[Gli96] Glickstein, Bob, *The GNU Stow Web Site*, http://www.gnu.org/software/stow/stow.html .

[MySQL] MySQL, http://www.mysql.com/ .

[Oet98] "SEPP – Software Installation and Sharing System", Tobias Oetiker LISA 1998.

[Rou94] Rouillard, John P. and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software", *Proceedings of the 8th Systems Administration Conference (LISA VIII)*, 1994.

[RPM00] *The Red Hat Package Manager Website*, http://www.rpm.org/ .

[Sel91] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach", *Proceedings of the 5th Large Installation Systems Administration Conference (LISA V)*, 1991.

[Sin97] Singer, Daniel E., "ToolMan Meets PatchReport", *;login: The magazine of Usenix and SAGE*, October 1997.

[Yar99] Yarger, Randy Jay, George Reese & Tim King, *MySQL and mSQL*, July 1999, O'Reilly and Associates.