Janet Bass and David Pullman
National Institute of Standards and Technology (NIST)
janet.bass@nist.gov and david.pullman@nist.gov

Configuration Management for Mac OS X, It's just Unix Right?

Tags: security, configuration management, Mac OS X

Introduction

For some time we have worked to automate our host configurations across a number of operating systems. Configuration of Windows clients was fairly straightforward with the information (INF) file provided by NIST through the Federal Desktop Core Configurations (FDCC) program (http://fdcc.nist.gov). Linux and Solaris were also fairly simple because we used Cfengine community version to provide self-healing configurations and to support exceptions for our research community. What about our Mac OS X clients?

Second Class Citizens

Mac OS X was the relative newcomer to our merry band of operating systems. They entered our environment by ones and twos, slowly infiltrating the network. There were a few, then there were ten, then there were 50 and so on. Mac OS X was considered anomalous in our environment. We responded by creating a separate build process for Macs including a manual configuration script. We generally ignored them from an infrastructure perspective. Configurations often did not stick due to updates or end-users making changes to settings. Over time the Macs were out of compliance and our only recourse seemed to be repeated application of our installation script. [1]

The Mac script posed several problems. First it was a manual solution. How could we know when to apply the script other than to run it? Not all Macs were configured the same so we couldn't just run the script the same way on every machine. Someone (us?) would have to document which Macs had slight deviations from the standard and then make sure to run the script so it would not apply those fixes, potentially breaking an experiment or impeding someone's research. This was a headache not only for us, but for the end-users.

The script was also hard to keep up to date every time Apple came out with an OS update. Leopard and Snow Leopard had enough differences that our script silently bombed on a some settings. Also, trusting the end-users to care about IT security compliance when their main goal was to perform science was bad news. We were right, it did not work. The end-users ran the script and reported total compliance just to keep us out of their hair. They had admin rights so configurations changed without our knowledge. We were still treating Macs as a lesser-supported operating system, unlike Linux, Solaris and Windows. We weren't giving them our full attention...yet.

An idea!

It is hard to remember when the light bulb came on and someone suggested we put the Macs under Cfengine control. It seemed so obvious. We were all Unix natives with significant experience maintaining the community version of Cfengine. The first thing we always did under Mac OS X was to open the Terminal application, not reach for the GUI, so we weren't afraid of a challenge. What if we took our knowledge of Cfengine for Linux and Solaris and applied it to the Mac operating system? Mac OS X is purportedly BSD underneath, that's just Unix. This is where we show our naïveté'.

We researched our options for Mac configuration management. We looked at Puppet, as another well-documented open source configuration management program, but we saw no real advantage over what we

1 Albert Einstein once said "The definition of insanity is doing the same thing over and over again and expecting different results".

were already doing with Cfengine and we already knew the Cfengine syntax and had a production server. We also considered the commercial product Casper, which only manages Mac clients. Casper suffered from feature overkill and some of the same issues as the open source programs (someone still has to script the settings). It would also come with a commercial per-client cost on top of the effort to implement it. Using Casper would have meant a separate server for Macs apart from our Linux/Solaris configuration (another thing to learn and maintain). Cfengine already works well for our other Unix and Linux clients, so we decided to stick with what we were already using. If it worked, Cfengine would provide us with self-healing Macs, a way to test compliance, exceptions configured on the right machines and we wouldn't have to depend on the end-user for testing anymore. We would have one server for all of our Unixen.

We started by taking the code from our manual Mac script and configuring the Mac settings on our Unix/Linux Cfengine server. Without extensive testing, we thought we had solved the problem. It appeared that Cfengine had configured our Mac clients. During compliance testing, it became obvious that many of the settings never worked at all. There did not seem to be anything wrong with the configuration, it just did not disable the settings.

Starting From Scratch

We decided to start over with a virtual Cfengine test server to allow us to roll up our sleeves and get under the hood of the Mac operating system. Classing would allow us to deal with special systems that needed specific configuration just as we do with Unix systems. We reviewed the documented secure configuration requirements for NIST to determine which settings could be borrowed from the Unix/Linux Cfengine code. There was not much we could reuse. The Mac operating system did not function as Unix we knew. The Unix and Linux secure guidelines could almost always be accomplished by editing a single file or configuration file. We had assumed it would be similar under Mac OS X. (Cue laugh-track.) We were starting from scratch.

We learned that there are additional layers of abstraction between the Mac OS X environment and the Unix below. We also learned that there are many paths to your final destination when trying to configure a setting and the paths are occasionally ordered so you must “discover” the correct order.

Apple script was our first route; it provides a way to make preference settings in the GUI. A Google search provided links to online documentation explaining how to set some of our required configuration parameters with Apple script. At first we thought we had a fix until we realized that some Apple script settings were **only** changing the GUI and **not** the real OS settings underneath. For example, the GUI would report that Bluetooth was off when in reality it was still enabled. This was disheartening.
Back to the drawing board we tried our Google foo again. We discovered some snippets of what other people had done and searched for documentation on Mac configuration settings.

In the end, the solution was a cmbination of complex rules and multiple ways to implement the settings. Some of the lessons learned during this experience include the following:

- New languages: Property List (.plist), mcx controls, *defaults* and *launchctl*.
- *launchctl* commands require the ”.plist” extension while the *defaults* command rejected a file with that extension.
- Some files needed to be unloaded, written and then loaded to properly apply the controls.
- In Snow Leopard the *disable* flag was moved out of the plist file to a separate file so we could not read a plist file for the disabled flag in the same way we did for Leopard.
- Files were sometimes different in Leopard and Snow Leopard and that we had to look at each setting on each system individually.
- There were [at least] two ways to configure a control, the fully configured method being the preferred one:
    1. Partially configured: Configure it so it was off but could still be enabled through the GUI. This gave the end-users a false sense that they could re-enable the control which just creates frustration when Cfengine overwrites the rule at the next pull.

2. Fully configured: Configure it so the rule was greyed-out in the GUI. The end-user could request an exception (e.g. need bluetooth for a keyboard) but it avoided a tug-of-war between the end-user and Cfengine.

We discovered a wealth of experience and practical advice on a number of sites. We also met some great people at the MacWorld Conference and Expo and learned that much of the Mac community is struggling with configuration management solutions for their Macs too. Some of our sources included the following:

- AFP548,
- MacEnterprise.org,
- support.apple.com (you have to dig to get past the GUI instructions),
- Apple Mac OS X Security Configuration for Version 10.5 Leopard Second Edition,
- Apple Mac OS X Security Configuration for Version 10.5 Snow Leopard Second Edition, and
- SAN's Auditing Mac OS X Compliance with the Center for Internet Security Benchmark Using Nessus.

It was disappointing to us that the Snow Leopard guide came out months after the operating system. New Macs were shipping with SnowLeopard with no downgrade path (official answer from Apple).

Example Settings and Headaches

Mac OS X uses the UUID appended to a file name for *some* settings. You need to retrieve the UUID and then know which file names include the UUID and append it to send a read, write or delete request.

```
/Users/username/Library/Preferences/ByHost   | ls -la
com.apple.Bluetooth.D20410BC-F453-533D-B34F-EXXXXXXXXC.plist
com.apple.CrashReporter.D20410BC-F453-533D-B34F-EXXXXXXXXC.plist
com.apple.DirectoryUtility.D20410BC-F453-533D-B34F-EXXXXXXXXC.plist
com.apple.ImageCaptureExtension2.D20410BC-F453-533D-B34F-
EXXXXXXXXC.plist
com.apple.SoftwareUpdate.D20410BC-F453-533D-B34F-EXXXXXXXXC.plist
com.apple.SubmitDiagInfo.D20410BC-F453-533D-B34F-EXXXXXXXXC.plist
```

Because Mac OS X is so different from Linux, we had to learn Mac design principles. For example Mac OS X likes to empower the User to have the ability to control settings on an individual basis. This removes the system-wide preferences that could be so easily set for all users. System administrators must loop through each user's settings to properly configure the setting for each user.

```
opendir(USERDIR, "/Users") or die"Can't open /Users: $!\n";
@userdirs = grep { $_ ne '.' and $_ ne '..' and $_ ne 'Shared' and $_
ne '.localized' } readdir USERDIR;
my $uuid = &getuuid;
foreach my $user (@userdirs)
{
next if (! -f
"/User$user/Library/Preferences/ByHost/com.apple.ImageCaptureExtension
2.$uuid.plist”);
my $status = `defaults read /Users/
$user/Library/Preferences/ByHost/com.apple.ImageCaptureExtension2.$uui
d shared`;
chomp $status;
if ($status ne "0")
{
 `defaults write /Users/
$user/Library/Preferences/ByHost/com.apple.ImageCaptureExtension2.$uui
d shared -bool FALSE`;
system ( "/usr/bin/logger -p daemon.warning -t cfengine Disabling
Scanner Sharing by editing /Users/
$user/Library/Preferences/ByHost/com.apple.ImageCaptureExtension2 for
```

```
each user");
 }
 }
```

More Complexity

One might think that the extensive use of property lists for configuration control would introduce some consistency, but this can be a trap. Apple has used a number of different constructs in plists for what are very similar controls. Why does one plist use a dictionary construct for values while another uses an array, and another will simply use a number of unstructured, yet related, values? Even value types vary for things that are simply on/off controls, sometimes using type boolean and other times type integer.

Set it, test it, undo it redo it and test again became our mantra. Several times we would get something to work, we would remove it, reset it and then it would fail. Each control needed to be researched, set, and tested for whether or not the configuration would stick.

After testing the various methods to configure Mac OS X, we settled on Local MCX controls where possible because they gave us fully configured settings: grayed out configuration options in the GUI which prevented users from attempting to re-enabling a control. Even with the control grayed out in the GUI, we still had to deal with the MCX concept of persistence. Persistence uses the labels “once”, “often”, and “always” to define how long a setting will remain configured. Some controls like Bluetooth can be set “always” disabled, but others like the idle time for the screen saver are “often” set, as in each time the system reboots. Yet the setting for requiring a password to unlock the screen saver is an “always”! So, for each setting, we had to determine how to set it and how long it would stick. The idle time setting is one that is set in MCX and needs to be checked each time Cfengine runs to make sure it stays configured. For those settings still configured through plist, the GUI allows the end-user to potentially re-enable the setting, albeit temporarily. At least we have Cfengine to re-heal the configuration at a predetermined time every hour.

We also did not benefit from using the Cfengine metalanguage. The Cfengine metalanguage methods lend themselves well to text files and to simpler shell commands, but OS X does not lend itself to these methods. We used this for Linux and Solaris, but it turned out that we wrote a lot of scripts to deal with the Mac issues. It is an open issue whether we can use the Cfengine3 community metalanguage to configure our Mac settings.

In the end we have been able to manage Mac OS X systems with Cfengine. There was a definite learning curve that we initially underestimated. We are prepared for more complication with future releases of the operating system and the mysteries they will bring.

Future

We intend to share our scripts and settings with the Cfengine community. We figured that maybe we can save someone else from the re-inventing the wheel. We do plan to migrate to Cfengine3 community at some point, at which time we'll also post our configurations using the new constructs.