## # Personal Picture

Hello. My name is Tony Williams.

Today I'd like to talk to you about a software system I have built to take care of application packaging and patching, I call it “PatchBot”

## # The Problem

So what's the problem I'm trying to address.

If you have a well set up MDM then for me the biggest challenge left was keeping all the packages up to date in Jamf Pro and the fleet properly patched.

And in a high security environment, such as a finance company like Suncorp, it’s not only challenging but vital.

## # The Solution

So what's the solution? Well, it’s a lot of manual work or some automation?

At Suncorp I’ve leveraged AutoPkg, the patch management system in Jamf Pro, and it's API to build an automated solution where most applications are patched on my fleet without me touching a thing.

Patch levels across the fleet have gone from woeful to good in a period of six months.

Today I'll cover how the system works and how it's put together. Along the way I'll cover the advantages of leveraging patch management and how that informed PatchBot's design.

I'll also cover other lessons I've learnt along the way.

## # The End Result

We want to end up with a complete software system that automatically builds application packages when a new version is available, uploads them to our Jamf instance, makes them available for testing and then finally moves them into production without any human intervention, informing us of its progress along the way.

## # How Is It Built

How is it built? What are the key components of the system?

## # AutoPkg

AutoPkg is at the core of it all. While AutoPkg builds packages it can do more.

If we consider that AutoPkg is a process that reads a "recipe" and runs the steps in the recipe with each step performed by what it calls a processor. In fact a semi-independent Python script.

You may already be using jss-importer or munki integration to allow you to do more using custom processors. PatchBot uses three custom processors tightly integrated with Jamf Pro.

## # Jamf Pro API

The Jamf Pro API is the second part of the puzzle. All three custom processors, written, like AutoPkg, in Python, use the API.

I use the classic API throughout as the new Jamf Pro API has nowhere near enough coverage of the Jamf Pro records we require.

## # Patch Definitions

The absolutely vital key to leveraging patch management in Jamf is to have a high quality, reliable patch source feeding you patch definitions.

The patch definitions you get for each application is essentially a version and a release date.

I buy my patch source from Mondada as a Kinobi subscription. Given the quality and convenience the cost is quite small.

They have been swift to add to the application coverage when I start building a new package and to fix the rare problems.

Kinobi have a public domain version of their server software if you want to do it yourself and there is a community that offers the required definitions.

I cannot stress enough how critical a good patch source is.

## # Patch Management

And of course that patch source feeds into the Jamf Pro patch management system.

Crucial to an easy workflow is the patch management system in Jamf Pro.

I've heard other Mac admins mutter about patch management's usability, complaining about it’s so called “lack of coverage” but I think it's great once you have a good patch source.

The one Jamf provides is OK for testing but does indeed lack coverage, in production a better patch source makes patch management a key tool. It works incredibly well.

## # 1200 lines

Then there's the bit I'm proudest of. I have written three custom AutoPkg processors and 4 helper scripts for a little more than 1200 lines of Python.

I'll go over all of these scripts in the next few moments.

## # Quick word about scope

Just a small digression, a quick word about scope.

One of the nice things about using patch management is that patches are automatically scoped.
Patches are only seen by users who have an old version of the application on their Mac.

That means we only need to worry about the test patches and there I only need one computer group for each group of testers.

Since a large part of my fleet are developers I have one group for developer package testing and another for the more general apps.

That's two groups to provide all the scoping I need for over 50 applications.

That makes it considerably less than the number of groups required to do the same job with policies in JSSImporter.

## # Package Lifecycle

What happens in the system? What is the lifecycle of a new package, a new version of an application? It has three parts.

First it's built and uploaded to Jamf Pro. Second, it is put into testing with a test patch policy. Finally, a week later it is put into production and goes out to the whole fleet.

Each of those steps requires a single AutoPkg run and each step uses a custom processor. Each step requires a recipe. Each step uses a Python script to massage the report from AutoPkg and send it as a message or two to Teams.

Let’s have a closer look.

## # Stage One

The first stage is, of course to build the package.

In the same step it is uploaded to Jamf Pro using the first custom processor, `JPCImporter`.

## # Stage One - autopkg command

This is the AutoPkg command we run. It’s the first command in a daily shell script. I wanted to show this to you so I could mention some useful features of AutoPkg and how I use them.

The first thing you’ll notice is that I specify a text file, packages.txt, as a “recipe list”. AutoPkg allows us to do this rather than list all the recipes on the command line.

I do this so I can have this command in the shell script and never change the command, just add and subtract recipes from the file.

Second, you can see I'm running my custom processor, JPCImporter, as a post processor. This means that after AutoPkg has completed the package building recipe for an application it then runs JPCImporter,
without having to change or override the standard package recipe. Notice the "com.honestpuck.Patchbot" on that second line, that tells AutoPkg where to find the processor.

Third, you can see I'm specifying a plist file as a report. AutoPkg will, when requested, write out the results of a run into a well structured plist file.
It's this file I read to decide what messages to send to Teams.

Finally you can see I'm setting a variable, `FAIL_RECIPES_WITHOUT_TRUST_INFO`. This makes use of a security aspect of AutoPkg.
It allows you to create a recipe override, which is another form of recipe that allows you to add to or modify slightly what the recipe does.
Overrides contain a hash of all it's parent recipes in the override, so called trust information.
So this option tells AutoPkg to not only fail if the hash in the override is wrong, but also fail if the override is missing. This makes sure that nobody is fiddling with our recipes.

## # JPCImporter

So what exactly does JPCImporter do?

Well, as the name implies, the first thing it does is import the package that has just been built into Jamf Pro Cloud.

It does this by uploading the package and then updating the package category and adding a description that says it was built by AutoPkg and the date it was built.

It also points a Test Policy, not a patch policy but a regular policy at the new package. This policy is disabled and scoped to nobody, it's sole purpose is to serve as a database record that holds the package details for the latest version. This information is used in the next stage.

## # <rant>

Before we progress, let me have a short rant. Uploading packages to Jamf Pro Cloud in a script is difficult. Jamf do not provide a supported API method.

There are currently three different Python methods in the wild, the one I use, which uses curl subprocess call and the dbfileupload endpoint.

There is the method used by Graham Pugh in his upload processor that uses the same endpoint but uses the Python requests module and finally there is Mosen's method used in jssimporter that attempts to mimic the web GUI method using requests.

All three methods are a hack. All three have problems, all three lack a method of updating the package details at the same time as the upload,
all three sometimes fail for reasons known only to the gods. We do know that the larger the package the more likely it is to fail.

We also know that even uploading large packages via the web GUI can sometimes fail. For me the tipping point seems to be 2Gb, over that I start having problems.

We also know that once a package fails it may not be possible to upload a package with the same name.

We also know that a good method must be possible because, even though Jamf don't have a reliable method through the GUI, the Jamf Admin application seems not to suffer from the same problems.

The biggest problem facing anyone trying to automate Jamf Pro Cloud workflows is that we lack a reliable, supported API method to upload a package.

It is well past time this was fixed.

## # Stage One - Teams.py

OK, ranting over, back to the process. We haven't quite finished stage one. We still need PatchBot to tell people what it has done.
This is the next line in my shell script.

The Python script, Teams.py, reads the Plist file from the AutoPkg run and sends the messages to Teams.

## # Stage One Messages

Here's an example of what one of those messages looks like. The "Policy" button opens up the policy in Jamf Pro.

I don't have a reason for doing that any more, it's a holdover from an early version and I've left in as a placeholder,  I'm probably going to change it to something else eventually.

I'd really like it to open the patch in self service but Teams doesn't support self service URLs and self service doesn't seem to have URL support for opening patches policies.

## # Stage One Teams errors

This is a Teams message telling me that AutoPkg had errors. The top one tells me that I don't have an override for that recipe.

Remember that FAIL_RECIPES_WITHOUT_TRUST_INFO option?  Having that set made this pop up. I actually added that to my recipe list just to generate the error.

Adobe have such a flawed package and update system that it is impossible to automate.

The next tells me that the DEPNotify recipe is having problems. The error tells me which part of the recipe failed but isn't terribly informative without some digging.

With this one, the author changed how he packages DEPNotify so the recipe breaks. Luckily Ross, the recipe author, keeps it up to date so all I needed to do was get AutoPkg to update the recipe.

If there is an error similar to this that's always a good first step.

## # Stage Two Shell script

The second stage is to update our test patch policy to use the newly uploaded package.

I do this in the second half of the shell script.

You can see that it is almost identical to the first half except for the name of the recipe list is patch.txt instead of packages.txt and it uses a different script to send messages to Teams.

This uses a second custom processor, `PatchManager.py` - it not only gets our test patch policy using the right package and enables it, it also writes a self service description.

This processor doesn't run as a post processor but requires a recipe of it's own.

## # Stage Two - Recipe

This is what a patch manager recipe looks like. It is incredibly simple. It specifies the processor at the bottom and the two arguments. At the moment both arguments are required, even though for quite a lot of packages they are the same. Fixing it so you only need specify one unless the second is needed is an item on my to do list.

If you have a look at the bottom where the processor is defined you once again see “com.honestpuck.PatchBot”, this is because all three custom processors are saved in one particular spot.

## # Stage Two - Package Manager

So what exactly does PackageManager do?

The first thing it does is grab that Test policy we updated in JPCImporter to find out the version number of the latest release.

Then it has a look at the patch software title record in Jamf Pro to find the patch definition for that version and if it doesn’t already, points it at the package. Then update the Test patch policy to the new version, write a self service description, enable the policy and we are done.

## # Stage Two - Self Service description

You might wonder why I wanted to show you a self service description..

It's because it starts the clock for when we move the package into production.

Oh, and I totally sidestep the argument about US date order, or English, they’re both wrong. I use ISO 8601 format dates. That’s obviously the 20th of July, 2020. I like it if for no other reason than it sorts properly.

I also have a self service deadline of three days for the test patch to make sure I get timely screams if the package is broken, but that's set ahead of time when I create the patch policy.

You may notice that unlike 'jss-importer' PatchBot doesn't create groups, doesn't create policies and doesn't worry about scoping. You have to do all that by hand.

There are two reasons for that. One is that it reduces the complexity of the task and the code, reducing the places where things can go wrong.

The second is that PatchBot can run using an account with far fewer privileges for a much more secure system. You only have to do these tasks once when you add a new package, it's not too onerous.

## Stage Two - Patch Teams message

Here's the Teams message telling us packages are now in testing. This can be sent to a more public Teams channel that includes my team and all the testers.

## Stage Two - Teams Patch Errors

Here's one telling me of errors. In this case the "error" is not in the software.

Sometimes you can find that you have downloaded a new package before your patch source has given patch management a definition for it.

Now obviously we can't point a missing patch definition to our package so we have to wait.

This timing problem usually resolves itself by the next morning, but if it doesn't there may be a problem in your patch source. This has happened to me once so far.

Of course these messages are sent to my teams Teams channel, and not to testers.

## # A quick word about names

Another short digression, gee, this is starting to become a habit.

This time I'd like to talk about names. Early on in the design process for PatchBot I realised something about names in Jamf Pro.

The *name* of objects such as packages, policies and patch policies are *never* seen by the end user. The name they see is set in a self service section of the object record.

This means that the names can be entirely for our purpose. In this case to help automate the system.

## # A quick word about names - names list

So I have incredibly tight naming conventions in my Jamf Pro instance and PatchBot is designed to rely on those conventions. These are typical of the rules used.

## # Stage Three

Stage three is moving a package into production after a week.

The key to this is a Python script, Move.py, that decides when we should run the process to move a package into production.

Move.py at it's core is a loop that runs through the patch policy list, if the policy doesn't have "Test" in the name or it isn't enabled it's ignored.

If it is then I grab the patch policy self service description and see if it contains a date more than 6 days ago. If it does then the application title is added to a list.

When it has the list finished it uses it to construct an AutoPkg command command similar to the two previous commands, except it has a third custom processor. `Production`. This custom processor points the production patch policy and the application install package at the tested package and disables the test patch policy so the system knows the process is complete. This custom processor also requires it's own set of recipes.

Then finally it calls another script, ProdTeams.py to send messages to Teams.

We are now at the end of the process.

## # Stage Three  - Why Two Scripts

Before we move on let me explain the design decisions around splitting the logic for when to go in to production out of the custom processor and into a script of it's own.

What we have just explored is the platonic ideal of workflows. Imagine though a couple of different scenarios.

First, imagine if you have moved a package into testing and a tester claims to have discovered a serious bug in the new version. You want to stop it from going in to production until it's checked out.

With the way we have built our workflow, all you need do is go into Jamf Pro and remove the self service description on the test patch policy.

Now the workflow into production will never trigger, while if you discover the tester was wrong you can easily put the package into production. That's one reason for having the logic in Move.py rather than the processor.

The second scenario is even more critical, imagine Google discovers a security flaw in Chrome and releases a patched version.
You want to get that new version tested and into production as fast as possible.

With the current design there is nothing stopping you from running the production recipe manually, well before the seven days are up.
That's the second reason for not putting the decision logic in the processor.

That's the end of our lifecycle. Now to finish the presentation with a look at the final pieces of the system.

## # Plumbing

Before we finish allow me to add a little about the plumbing behind PatchBot.
PatchBot is run by one shell script, and the Python script Move.py but how do we get them to run?

For this I have two LaunchAgents. The only complication is that while packaging and patch management run in the early morning every day I only run Move.py Monday through Thursday to maximise the chance that if a user runs into a problem with a newly installed patch they are likely to find someone who can help.

The next thing I'd like to stress is version control. By the time you finish setting up PatchBot or similar you will have a whole bunch of patch manager recipes, and production recipes before you even start in on your own packaging recipes.

This represents a lot of work so I have separate git repositories for each sort of recipe. If your organisation doesn't have an internal equivalent to GitHub then use private repos there or somewhere equivalent but use it and learn how to use git effectively.

I've learnt a great deal in the last three years about it and it has saved me more than once.