What Windows Admins Need from **Configuration Management**

puppet labs

Table of Contents

Acknowledgements

Authors: Rob Reynolds and Aliza Earnshaw
Editor: Aliza Earnshaw
With thanks to Alanna Brown, Ethan Brown, Daniel Dreier, Molly Niendorf, Richard Raseley and Reid Vandewiele

Windows Admins and Configuration Management

As a Windows systems administrator, you leverage a wide variety of tools to accomplish your day-to-day tasks and long-term goals. These probably include (but are not limited to) Active Directory Domain Services, the System Center family of products, PowerShell, old-school Windows batch scripts, and a myriad of application and function-specific MMC snap-ins.

While the use of these tools and their associated processes has served you well in the past, Microsoft's vision of a "mobile first, cloud first" world suggests we need to work toward a toolset that supports multiple platforms. As businesses continue to transition towards more agile and responsive IT and service delivery methodologies — by adopting DevOps practices, for example — the demands placed upon you, and consequently your tools, will continue to grow. These demands are difficult enough for those managing pure Windows environments, but admins managing multiple platforms — for example, a combination of Windows, Linux and/or OS X, or storage and network devices in addition to servers — will find it even more challenging.

A lot of Windows admins use PowerShell to automate some of this work. But PowerShell scripts aren't portable across other platforms, and they won't help you when it comes to managing non-compute resources.

Most people would prefer to simplify, and manage all their systems with a single configuration management tool, whether Windows servers, systems running on another OS, or other networked devices. It's also great if you can track changes easily for better infrastructure auditing, and store versions of your configurations for easier rollback when something breaks.

If you're looking at tools out there in the market to do all this, you probably already know there are a lot of configuration management choices available. Now you need to figure out what's going to work best with your Windows environment.

What To Look For in a Configuration Management Tool

What to look for

A declarative approach. A tool that's declarative allows you to describe the end state you want systems to be in, rather than describing the process it will take to get there, which is what a procedural tool requires of you.

A graphical tool, such as those commonly used by Windows admins, executes the steps for you, resulting in an end state. But it does not provide a description of the end state, making it difficult to automate a task across more than one machine without manual intervention.

Reading declarative code tells you what you can expect the end state to look like. Contrast that to reading procedural scripts, where you have to keep the starting state in your head, then imagine what will happen as the steps are executed, to figure out what the end state will be, once the script has run. With a graphical tool, you can't read a description ahead of time to know what the end state will actually be; you just execute the task. And it's a one-off; it's not repeatable.

What Puppet offers

Puppet's domain specific language is declarative: You write manifests that describe how your systems should look, not the steps that are needed to get there. So you don't have to know the specifics of configuring every piece of software on servers or other devices: If Puppet supports it, Puppet will do it. Check out all the supported platforms for open source Puppet and for Puppet Enterprise, and supported network and storage devices.

Puppet manifests are concise, and written in language that's easy for people outside of operations to read and understand. That makes it possible for developers and other colleagues outside of ops to see what's being done in operations, and even to write their own manifests as needed. The Puppet language is so easy to understand, some companies provide their Puppet manifests to auditors to demonstrate compliance with regulatory requirements.

Declarative tools have another advantage over graphical-interface tools: They allow you to treat infrastructure as code. That means you are defining the desired state of your infrastructure in a form that can be treated exactly like any other code:
It can be shared with others outside the ops team who may need to manage machines, but don't know the exact steps for doing that; and it can be checked into a version control tool, just like any other artifact of your software development and maintenance pipeline. (See more on infrastructure as code below.)

The declarative approach is especially useful when you're looking to enhance collaboration between teams. Developers, for example, can configure test servers for themselves, without needing to know the details of how it's done. The same thing goes for your quality engineers.

Pre-written modules >>

What to look for

Pre-written modules. Look for a system that is highly extensible, with many widely used and tested Windows-specific modules for common admin tasks, such as installing Windows features, configuring file permissions, managing Registry settings and more. It should also have modules for other operating systems and platforms that you currently manage, and also those you don't. The future is unpredictable: You may not be managing specific platforms now, but as business requirements change, you could end up managing them. It's a lot less work to position yourself for the future now, versus re-architecting later.

What Puppet offers

You can choose from about 3,000 Puppet modules to automate your admin tasks, including a wide variety of modules written specially for Windows. Among these, you'll find modules that are fully supported for use with Puppet Enterprise, including:

- puppet-registry for managing keys and values in the Registry
- puppetlabs-reboot for managing reboots of Windows systems
- puppetlabs-powershell for executing PowerShell commands on Puppet-managed systems
- puppetlabs-sqlserver for installing and managing MS SQL
- puppetlabs-acl for managing ACLs (find detailed examples for using this module in the section below)
- windows_feature for turning features on or off for Windows Server
- download_file for downloading files to use on Windows server

Puppet Enterprise customers get support from Windows administration experts not only for Puppet itself, but also for all Puppet Supported modules. You'll also find Puppet Approved modules designed for Windows — modules created by Puppet community members that have been approved by Puppet Labs as properly designed, regularly maintained and well-documented.

What to look for	What Puppet offers
Community. There should also be a big, active community of users who create, test and discuss modules and use cases. You'll want a community to turn to, and you want a system that's established and still growing, to protect your investment into the future.	The Puppet community has been around for about a decade, and is continually growing. Puppet users are in all industries, and in government, education and nonprofits around the world. Many of the most useful modules on the Puppet Forge were written by community members, and even more people can be found actively helping others learn and solve problems in online groups and blogs and at Puppet Camps and meetups.
Cost efficiency. You shouldn't have to add a lot of expensive hardware plus licenses to use your configuration management tool. It should be easy and lightweight to install, and you should be able to manage potentially hundreds of servers from a small footprint.	Puppet is cost-effective, both in terms of hardware and licensing costs. You can start with just a single server to get Puppet up and running in your environment, and grow only as you choose to expand the number of servers and other resources managed by Puppet. Organizations using Puppet to manage their infrastructure range from fewer than 100 machines to hundreds of thousands of nodes managed by Puppet.

What to look for

Ability to enable other teams. Your configuration management tool should make it easier for you to refine resource access for different users, so you can give other teams (such as software development and quality engineering) access to the resources they need, in accordance with your internal policies — without having to go through ops every time.

What Puppet offers

Puppet Enterprise allows you to determine access by user role, so it's much easier for you to give other teammates the ability to service their own infrastructure needs (in accordance with your organization's policies), without having to go through Ops every time. You can give people what they want while freeing up your own time — a win for everyone.

Ability to manage infrastructure as code >>

What to look for

Ability to manage infrastructure as code. Your configuration management tool should allow you to manage your infrastructure as code. You should be able to check your configuration files into a version control system, the same way developers do to manage software development. There are some big advantages to adopting this approach:

- It's a lot easier for you and your ops colleagues to know quickly what everyone is doing, and to cover for each other when someone's out.
- It increases your confidence in your changes, because once you're using configuration management and version control, you know you can easily audit your configurations; view the diffs between configurations; and quickly and easily roll back to the last known good state when needed.
- It's much easier to collaborate with your colleagues in software development, testing and quality assurance, which in turn makes it easier for the technical teams to align with the organization's business strategy.

What Puppet offers

Puppet manifests are text files, so you can check them into a version control tool, just like any code. That allows you to audit your infrastructure, roll back to the last known good change, and easily share information with colleagues on other teams in your organization.

Support for platforms beyond Windows >>

What to look for

Support for platforms beyond Windows. More and more IT organizations have to manage multi-platform, multi-cloud environments. Even if you aren't there yet, technology moves so fast that it may not be long before the next big initiative drives big change in your organization. You want to be the one leading that change, not one of those struggling to keep up.

What Puppet offers

You can use Puppet to configure all your servers, whether they're running on Windows, Mac OS X, Linux or another variety of Unix. The same manifests will work across all platforms, saving you time and trouble. You can also use Puppet to configure network and storage devices, so you'll be reading and writing manifests in the same language when you work with these resources — less context-switching, so less disruptive to your workflow.

What you can actually do with Puppet >>

What You Can Actually do With Puppet: Manage Permissions With ACLs

While there is a simple interface for setting permissions on Windows, managing and maintaining permissions has never been simple. Puppet has a module that makes working with permissions much easier. You can work directly with ACLs (Access Control Lists) and to a degree, security descriptors, through the puppetlabs-acl module. While the ACL module makes managing permissions easier, it still satisfies admins who need to work with very advanced permissions.

The ACL module became a Puppet Enterprise supported module as soon as it was released in May 2014, and works with Puppet Enterprise 3.2+ (and open source Puppet 3.4.0+). The module adds a type and provider for Windows so you can manage those pesky permissions without a ton of hassle.

Let's talk about a couple of real-world scenarios:

- You need to set permissions appropriately for something like IIS/Apache.
- You have a directory or file that you need to lock down to just admins.

These are very doable with the ACL module and just a few short lines of Puppet code. We'll show you that in the Examples section below, but first let's talk about ACLs and ACEs for those who want a refresher.

ACLs and ACEs

ACLs (also called Discretionary Access Control Lists) typically contain a list of access control entries (ACEs). An ACE is a defined trustee (identity) with a set of rights, and information about how those rights are passed to (and inherited by) child objects — for example, files and folders. For each ACE, the ACL contains an allowed/denied status, as well as the ACE's propagation strategy. You cannot specify inherited ACEs in a manifest; you can only specify whether to allow upstream inheritance to flow into the managed target location (the location where you are applying the ACL).

ACL module features

Here are some features of Puppet's ACL module you should know:

- Puppet can manage the complete set of ACEs or ensure that some Puppet-specified entries are present, while leaving existing entries alone.
- You can lock down a path to only the specified permissions.
- Identities can be users and/or groups, domain users/groups, and/or SIDs.
- You can ***point multiple ACL resources at the same target path.***
- Propagation and inheritance of ACEs is set to Windows defaults, but can be specified per ACE.

Here's what sets Puppet's ACL module apart from other configuration management tools:

- **ACE order**. The order of your ACEs matters. If they are incorrectly ordered, it can cause issues. Puppet will apply ACEs in the order you've specified in the manifest. And we insert them in the correct location when merging with unmanaged ACEs.
- **SID (Security ID) support**. We support specifying identities as SIDs.
- **Very, very granular permissions**. Do you need to apply read attributes (RA)? Yes, we can do that.

Getting started

Let's take a look at what a typical ACL resource looks like:

```
acl { 'c:/temp':
  permissions => [
    { identity => 'Administrator', rights => ['full'] },
    { identity => 'Users', rights => ['read','execute'] }
  ],
}
```

If you were to run the above on a system that had the module installed, you would be giving the Administrator account full access to the temp folder, and giving the Users group access to `read` and `execute` (and `list` for folders). All the other options are set to Windows defaults, but if we need to get to them, we can. Let's look at that same ACL resource with all the options specified:

```
acl { 'c:/temp':
  target      => 'c:/temp',
  target_type => 'file',
  purge       => 'false',
  permissions => [
   { identity => 'Administrator', rights => ['full'], type=> 'allow',
child_types => 'all', affects => 'all' },
   { identity => 'Users', rights => ['read','execute'], type=> 'allow',
child_types => 'all', affects => 'all' }
  ],
  owner       => 'Administrators',
  group       => 'Users',
  inherit_parent_permissions => 'true',
}
```

We have just defined a resource with all parameters and properties specified; what you are seeing is how the defaults line up. The only exceptions here are `owner` and `group`, which by default are not managed unless specified. The defaults for these depend on the user that created the target (the folder) and could be different based on who the user has as their default group and owner.

- With both of the above examples we have done the following:
- We've given `Administrator` full access to `c:\temp`.
- We've given `Users` read/execute access to `c:\temp`.
- We are by default not removing explicit unspecified permissions (`purge => 'false'`).
- We are by default inheriting the permissions from the parent folder (`inherit_parent_permissions =>'true'`).
- For each ACE, we are allowing the permission by default (`type =>'allow'`).
- For each ACE, we apply the permission to all child types by default, both folders and files (`child_types =>'all'`).
- For each ACE, we propagate the permission to self, direct children and all those further down by default (`affects => 'all'`). Note that all types below child types are called grandchildren, no matter what the level of depth.

By default, if a user is not granted access through an ACE (whether individually or as a member of a group), then Windows will deny access. So if a user is not the `Administrator` account and not a member of the `Users` group, they will not have access to `c:\temp`. With the ACL module, this also means that access could be granted outside of Puppet (when `purge => 'false'`) and/or it is an inherited ACE (when `inherit_parent_permissions => 'true'`).

ACL type structure

Now let's take a look at the ACL type and all it has to offer.

Parameters

The parameters are (**bold** means the parameter is required):

- **name** - The name of the ACL resource, used as target if target is not set explicitly (see the target parameter).
- purge - Determines whether unmanaged explicit access control entries should be removed. Combine `purge => 'true', inherit_parent_permissions =>'false'` to really lock down a folder. Supports `true`, `false` and `listed_permissions`. Defaults to `false`.
- target - The location; defaults to the same value as name.
- target_type - Currently supports only `file`; defaults to `file`.

Properties

The properties are (**bold** means the property is required):

- **permissions** - The list of ACEs as an array. This should be in the order you want them applied. We cover permissions in more detail below.
- owner - User/Group/SID that owns the ACL. If not specified, the provider will not manage this value.
- group - User/Group/SID that has some level of access. If not specified, the provider will not manage this value. Group is not commonly used on Windows.
- inherit_parent_permissions - Whether we inherit permissions from parent ACLs or not. Default is `true`.

Permissions property

The permissions property could be considered the most important part of the ACL resource, because it contains each ACE in the order specified for the ACL. The available elements for each ACE hash are `identity`, `rights`, `type`, `child_types`, `affects`, and `mask`.

The elements are (**bold** means the key is required):

- **identity** - This is the user/group/SID.
- **rights** - An array with the following values: `full`, `modify`, `mask_specific`, `write`, `read`, and `execute`. The `full`, `modify`, and `mask_specific` values are mutually exclusive, and when any of these values are used, they must be the only value specified in rights. The `full` value indicates all rights. The `modify` value is cumulative, implying `write`, `read`, `execute` and `DELETE` all in one. If you specify `mask_specific`, you must also specify the mask element in the permissions hash. The `write`, `read`, and `execute` values can be combined however you want.
- type - Whether to `allow` or `deny` the access. Defaults to `allow`.
- child_types - Which types of children are allowed to inherit this permission, whether `objects`, `containers`, `all` or `none`. Defaults to `all`.
- affects - How inheritance is propagated. Valid values are `all`, `self_only`, `children_only`, `self_and_direct_children_only`, or `direct_children_only`. Defaults to `all`.
- mask - An integer representing access mask, passed as a string. Mask should be used only when paired with `rights => ['mask_specific']`. For more information on mask, see the granular permissions example below.

For more specific details and up-to-date information on the ACL type, see the usage documentation.

Examples

In the following examples we are going to show you how to lock down a folder, set appropriate permissions for a website, and set very granular permissions.

Locking down a folder for sensitive data

Here's what you need to do when you keep sensitive data in a specific folder, and need to limit access to administrators only.

```
acl { 'c:/sensitive_data':
  purge                         => true,
  inherit_parent_permissions    => false,
  permissions => [
    { identity => 'Administrators', rights => ['full'] }
  ],
}
```

We've done the following:

- Since permissions are inherited from parent ACLs by default, we set `inherit_parent_permissions => false` so no permissions are inherited from the parent ACL.
- ACLs will also allow unmanaged ACEs to coexist with Puppet-managed permissions, so we need to specify `purge => true` to ensure that all permissions other than those we have specified are removed.
- We've given the `Administrators` group full permission to the directory.
- If an ACE has not granted permission for a user, Windows will by default `deny access`. So in this case, only users that are part of the Administrators group will be able to access this folder.

That was pretty simple! And very self-documenting as well. It's worth mentioning that one goal of Puppet is for anyone to be able to read and understand the intent of the code, and the ACL module does not disappoint. Now let's try something a little more involved.

Website setup with ACLs

Let's take a look at setting up an IIS site and locking down permissions.

```
$website_location = 'C:\sites\thestuff'
$website_name = 'the.stuff'
$website_port = '80'

# add windows features
windowsfeature { 'Web-WebServer':
  installmanagementtools => true,
} ->
windowsfeature { 'Web-Asp-Net45':
} ->

# remove default web site
iis::manage_site { 'Default Web Site':
  ensure        => absent,
  site_path     => 'any',
  app_pool      => 'DefaultAppPool',
} ->

# application in iis
iis::manage_app_pool { "${website_name}":
  enable_32_bit           => true,
  managed_runtime_version => 'v4.0',
} ->


iis::manage_site { "${website_name}":
  site_path     => $website_location,
  port          => "${website_port}",
  ip_address    => '*',
  app_pool      => "${website_name}",
} ->

# lock down web directory
acl { "${website_location}":
  purge                         => true,
  inherit_parent_permissions    => false,
  permissions => [
    { identity => 'Administrators', rights => ['full'] },
    { identity => 'IIS_IUSRS', rights => ['read'] },
    { identity => 'IUSR', rights => ['read'] },
    { identity => "IIS APPPOOL\\${website_name}", rights => ['read'] }
  ],
} ->


acl { "${website_location}/App_Data":
  permissions => [
    { identity => "IIS APPPOOL\\${website_name}", rights => ['modify'] },
    { identity => 'IIS_IUSRS', rights => ['modify'] }
  ],
}
```

The script above does the following:

1. Ensures IIS is installed and ASP.NET is set up.
2. Ensures the default web site is removed.
3. Ensures our site is set up correctly.
4. Uses ACL to ensure IIS users have read access.
5. Uses ACL to ensure the processes that run the website can modify the `App_Data` directory.

All of that, done in just a few lines of code! You can add more for getting the files there in the first place, using Puppet, but we have left that as an exercise for the reader.

Granular permissions

In our last example, let's get into very granular permissions.

```
acl { 'c:/granular_permissions':
  permissions => [
    { identity => 'Administrators', rights => ['full'] },
    { identity => 'Bob', rights => ['mask_specific'], mask => '1507839' }
  ],
}
```

We've done the following:

- We've given the `Administrators` group full permission to the directory.
- We've given the user `Bob` permission to modify, plus `WRITE_DAC` (the ability to modify the discretionary access control list, or make changes to the ACL) and `FILE_DELETE_CHILD` (the ability to delete all children, including read-only files). Bob now has almost full permissions, except we have not given this user the ability to modify the owner of the ACL.

So you see `1507839` and it looks like a magic number. How does one arrive at this number? See this blog post that shows you how to add up the numbers. Now we have a much simpler way to add those permissions: We have created and made available a worksheet to add up the ACL rights mask! This will allow you to add up the rights and will give you a heads up if you should use named rights from the module (like `read` or `execute` or a combination of `read, execute`).

Tips for using ACLs

Here are some tips to consider when using ACL:

- If you are specifying an ACL, please ensure you don't set `mode` on file resources.
- Don't use fully qualified names except when using domain accounts. When identities (users) are local-machine specific, don't fully qualify the name (e.g., `Machine-Name\Administrator`), as that doesn't translate well to multiple machines with different names.
- Avoid SIDs unless you need them. When ACLs attempt to set dependencies on user resources, the ACL module will go with a fully qualified user name, which you won't usually want to put into manifests (e.g., `Machine-Name\Administrators`), unless you are specifying domain accounts.
- Don't use Windows 8.3 short file names. (1980 just called.)
- Unicode identities don't work yet. This is due to a bug in Puppet, and will be fixed in future versions of Puppet.
- ACEs are checked for uniqueness based on `identity`, `type`, `child_types`, and `affects`. Don't differ based on rights alone. It will confuse the provider.

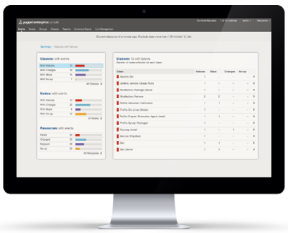
Puppet Makes Windows Administration Easier

Windows admins in all industries, in organizations ranging from small startups to Fortune 100 companies with huge web operations, are using Puppet to do their work more efficiently and collaborate better with their colleagues. For Windows admins, Puppet offers some compelling benefits over other configuration management tools:

- It's easy to use, once you learn the Puppet DSL (and we find most Windows admins learn it quickly).
- Puppet has been used by system administrators for almost a decade, and is used to manage infrastructure in more than 20,000 companies worldwide, from tiny startups to large companies like Google, ADP, PayPal, Sony and many more. More Windows admins adopt Puppet every year, and Windows modules are among the most-downloaded modules on the Puppet Forge.
- Patch Tuesday doesn't have to be painful; Puppet helps you build test environments that are exactly like production, then roll out updates in a controlled and auditable way.
- You'll find many pre-built, extensively tested modules for managing your admin tasks on the Puppet Forge. We've even made it easy for you to search for Windows-compatible modules, for Puppet Enterprise supported modules, and for modules that have met the Puppet Labs standards for Approved status. All these lists are continually growing.
- The Puppet community is large, growing and includes many Windows admins who enjoy discussing (and helping others with) the challenges of automating Windows and mixed environments.
- Puppet Labs offers specialized support for Windows admins.
- On the Puppet Labs website, you'll find a huge trove of learning resources, including special training and introductory courses for Windows admins who have not yet used Unix, the vim text editor or Git and GitHub (tools that are commonly used by Unix sysadmins).
- You'll have the ability to support DevOps and continuous delivery initiatives in your company.

Resources

- Learn how .NET shop IP Commerce uses Puppet Enterprise with GitLab, Jira, Bamboo and MySQL for its continuous delivery system. If you want even more details about the company's continuous delivery workflow, you'll probably enjoy application architect Jason Moorehead's blog post.
- Here's an excellent discussion of the Desired State Configuration (DSC) tool in Powershell. Look for the interesting comment by Rich Siegel — it was the first, so is at the bottom of the post page.
- In case we haven't linked it enough already, here's the Puppet ACL module.
- Good documentation on how permissions work in Windows Server.
- Access rights and access masks
- File generic access rights
- Access mask format
- ACL access mask rights addition worksheet



If you haven't tried out Puppet Enterprise yet, you can download and try it out for free. Do it now! And let us know how we can help you — there are many ways you can get answers to your questions: at ask.puppetlabs.com, in the Puppet Users group, plus more at the Puppet Labs page on GitHub. And of course, there's always our extensive documentation to consult.