Script Best Practices

Description

Tips and tricks for running Filesets with scripts

Don't put passwords in scripts

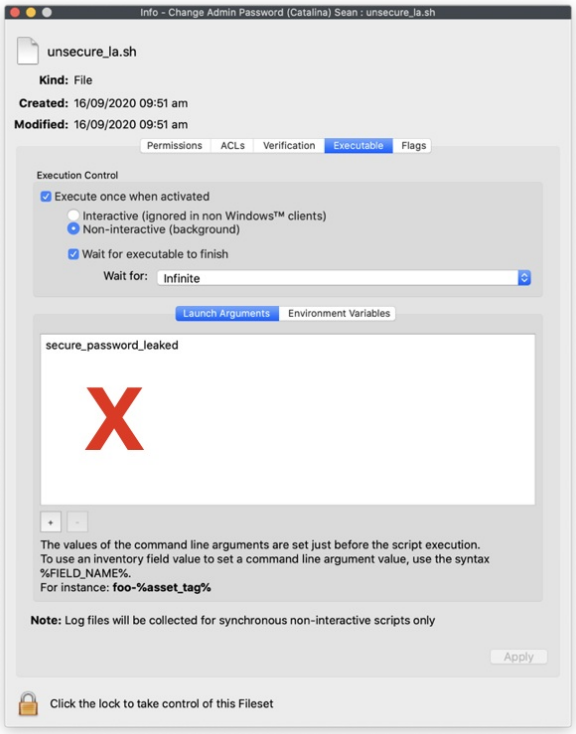
The scripts are stored locally on devices. For security reasons, usernames and passwords should not be included within the body of scripts.

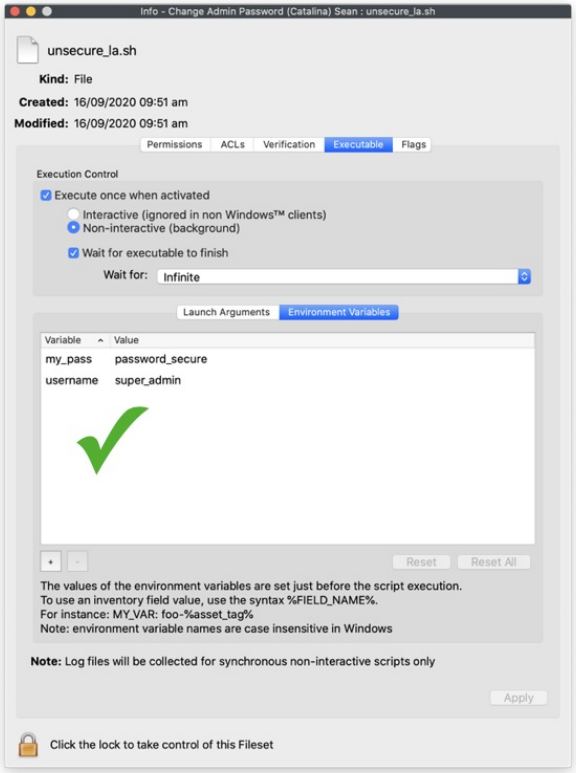
For example:

Example: password in command

```
somecommand -u "USERNAME_HERE" -p "PASSSWORD_HERE"
```

Additionally, DO NOT use Launch Arguments to provide passwords to scripts. Launch Arguments are visible in the process list during script execution. Instead supply the username and password as Environment Variables:





During script execution, the Launch Argument is seen:

Example: Visible Password

```
$ ps -ef | grep secure
    0 73010   155   0  9:51am ??         0:00.01 /bin/zsh /var/scripts/532417/unsecure_la.sh
secure_password_leaked
```

Using the example Environment Variables from the image, they would be addressed as:

OS	Script Type	Command
macOS	shell	`somecommand -u $username -p $my_pass`
Windows	Powershell	`somecommand -u $Env:username -p $Env:my_pass`
	Batch	`somecommand -u %username% -p %my_pass%`
	Batch	In order to not transmit the password to a log file accessible on the device, add @echo off before the line containing %my_pass% and @echo on as the next line. Example: @echo off %SystemRoot%\System32\Reg.exe ADD "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon" /v "DefaultPassword" /d "%my_pass%" /t REG_SZ /f @echo on

macOS & Windows	Python	`import os`
		
		`os.getenv('username')`
		`os.getenv('my_pass')`

Keep Requirements Scripts Small

Requirements scripts are pulled from a fileset and sent before the remainder of the fileset.

It behaves this way because if a requirements script fails, there is no point in downloading and installing the remainder of the fileset.

> ⚠ A fileset whose requirements have failed will not even show up in the kiosk.

Where possible, avoid piping commands. This increases overhead on the scripts. If pipes are required, try to reduce the quantity of pipes. If nothing else, this makes the scripts easier to read.

```
$ time system_profiler SPHardwareDataType | grep "Model Identifier" | awk '{print $NF}'
MacBookPro11,4


$ system_profiler SPHardwareDataType | awk '/Model Identifier/ {print $NF}'
MacBookPro11,4
```

And other commands may achieve the same result more efficiently without the need to pipe.

```
$ time system_profiler SPHardwareDataType | grep "Model Identifier" | awk '{print $NF}'
MacBookPro11,4

real    0m0.192s
user    0m0.071s
sys     0m0.049s


$ time sysctl -n hw.model
MacBookPro11,4

real    0m0.004s
user    0m0.001s
sys     0m0.002s
```

Consider this for all scripts beyond just requirement scripts.

Leverage Dependencies and Scripts

If there is a script that several filesets will need, don't paste the same script into each one. Create an empty fileset with that script, and make the other filesets dependent upon it.

Log Script Output to the Client Log (macOS only)

Have a script that needs to write details steps to a log?

Want a quick status of the script.

Have a script write to the client log.

macOS / Linux

```
#!/bin/bash
exec 1>>/var/log/fwcld.log
exec 2>>/var/log/fwcld.log


... rest of script
```

You can then use Client Monitor and pull and view the log as things are happening.

> ℹ Windows log file is locked such that additional appending may not take place

Testing Scripts

Scripts run by FileWave are run by root or System. As such, scripts should be tested using the same user context to prevent erroneous results. Many commands will yield the same result regardless, but this cannot be relied upon.

Windows

E.g. Run the following on a Windows 10 Professional system locally through Powershell as either user or 'Run As Admin' will see the following result:

```
 (Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion").EditionId
Professional
```

However, as a Custom Field running the same script, the result is surprisingly different:

```
 (Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion").EditionId
Enterprise
```

This is because Windows is providing a different answer based upon either the user running the script or may provide different responses based on 32-bit or 64-bit.

Take a look at Getting a CMD prompt as SYSTEM in Windows Vista and Windows Server 2008 for details about running scripts as System. Note, that by default, this will start an executable as 64-bit, for native 64-bit OS. However, the above example is because the FileWave fwcld process is calling the 32-bit version of PowerShell.

> PsTools: This relies on downloading and installing, onto the test machine, PsTools.

To mimic this experience, consider the guide to starting the CMD. When launching an executable, like PowerShell, the 32-bit version would need to be referenced. For example:

```
PSEXEC  -i -s -d C:\Windows\SysWOW64\windowsPowerShell\v1.0\powershell.exe
```

For CMD, the equivalent would be:

```
PSEXEC -i -s -d %windir%\SysWoW64\cmd.exe
```

Similarly, when attempting to run some commands, it may be necessary to ensure Windows is using the correct version of a binary with the 'sysnative' redirect. An example would be Bitlocker's 'manage-bde.exe'. To use this in a Fileset, try the following:

```
C:\Windows\sysnative\manage-bde.exe -status
```

If you have a requirement to run a particular command through the 64-bit version of Powershell this can be achieved as follows:

```
If ( [IntPtr]::Size * 8 -ne 64 )
{
    C:\Windows\SysNative\WindowsPowerShell\v1.0\PowerShell.exe -File $MyInvocation.MyCommand.Path
}
Else
{
    # Add code here
}
```

Example

Create a new admin account

Two Fileset Environment Variables would be supplied. To add the user 'rstephens' with the password 'filewave'

Variable	Value
username	rstephens
password	filewave

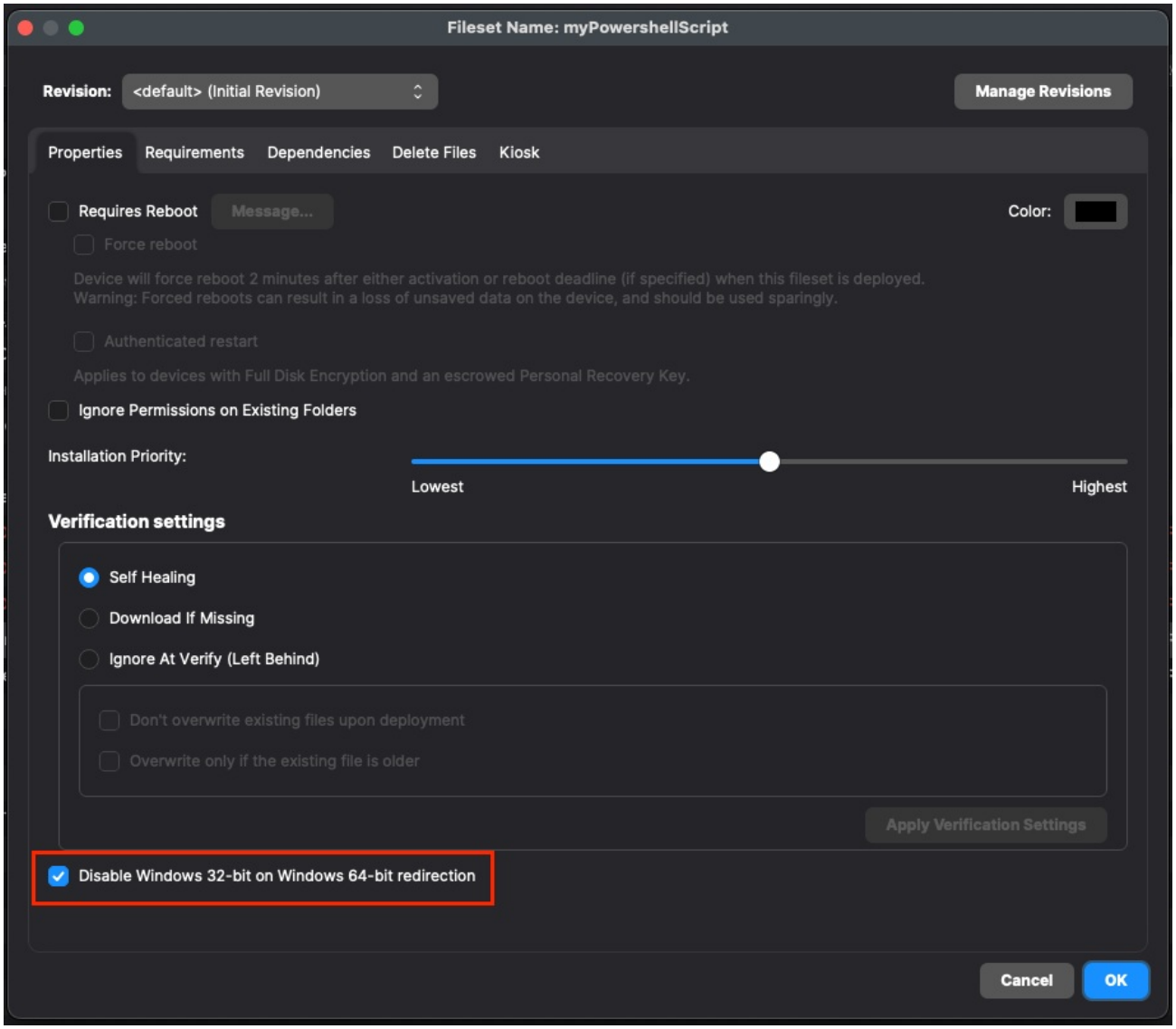
Parameters may be supplied, that can then be added to the execution of Powershell from within the script:

```
Param (
    [string]$MyUsername = $Env:username,
    [string]$MyPassword = $Env:filewave
)
```

```
If ( [IntPtr]::Size * 8 -ne 64 )
{
    C:\Windows\SysNative\WindowsPowerShell\v1.0\PowerShell.exe -File $MyInvocation.MyCommand.Path -MyUsername
$MyUsername -MyPassword $MyPassword
}
Else
{
    (New-LocalUser -AccountNeverExpires:$true -Password ( ConvertTo-SecureString -AsPlainText -Force $MyPassword)
-Name $MyUsername | Add-LocalGroupMember -Group administrators)
}
```

Troubleshooting PowerShell scripts

As a best practice, always check the "Disable Windows 32-bit on Windows 64-bit redirection" checkbox in the Properties tab for your fileset. This ensures that any scripts in the fileset will be run in a 64-bit session and built-in Windows executables triggered by those scripts will call the 64-bit versions. A common reason for why your script might not be performing the expected results could be due to 64-bit Only Modules or Cmdlets: Some PowerShell modules or cmdlets are available only for the 64-bit version of PowerShell. If a script relies on these 64-bit modules, it must run in a 64-bit shell.



macOS

On macOS, running commands as sudo is not necessarily the same as actually becoming root.

Root vs As Root

E.g. Run the following commands to evaluate the local variable $HOME, once using sudo and once as root.

```
$ whoami
auser
$ sudo echo $HOME
```

```
/Users/auser
$ sudo su -
$ whoami
root
$ echo $HOME
/var/root
```

Paths

Similarly, the paths used to locate executable files will differ, since FileWave is a service ran as root and is not the root account. On an example device:

User Account	Root Account	FileWave Client

`% echo $PATH | tr ":" "\n"`
`/opt/homebrew/bin`
`/opt/homebrew/sbin`
`/var/root/.cask/bin`
`/usr/local/sbin`
`/usr/bin`
`/bin`
`/usr/sbin`
`/sbin`
	`% echo $PATH | tr ":" "\n"`
	`/usr/local/bin`
	`/System/Cryptexes/App/usr/bin`
	`/usr/bin`
	`/bin`
	`/usr/sbin`
	`/sbin`
	`/Applications/VMware Fusion.app/Contents/Public`
	`/Library/Apple/usr/bin`
	`/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/local/bin`
	`/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/bin`
	`/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/appleinternal/bin`
		`/usr/bin`
		`/bin`
		`/usr/sbin`
		`/sbin`

As such, consider always using the full path within a script to an executable, to be explicit, and ensure the executable is found.

For example, it can be seen from the above that homebrew is installed.

```
 % ls -al /usr/local/bin/brew
lrwxrwxrwx  1 root  _developer  28 Mar 23  2023 /usr/local/bin/brew -> /usr/local/homebrew/bin/brew
```

Running the following command would work as the user or root account, but would fail through FileWave, since the FileWave Client does not search /usr/local at all for executables:

`brew -v`

To ensure the script works and targets the correct brew, the full path should be entered:

`/usr/local/bin/brew -v`

Plist

It is common to see plist files edited with the 'defaults' command. However, this command is unique when it comes to ownership and permissions of files. The 'defaults' command will both take ownership and change permissions of files when used to write to plist files:

```
$ whoami
root
$ ls -al /tmp/example_plist.plist
-rw-r--r--  1 rstephens  staff  66 Feb 28 10:03 /tmp/example_plist.plist
$ defaults write /tmp/example_plist Label example_plist
$ ls -al /tmp/example_plist.plist
-rw-------  1 root  wheel  66 Feb 28 10:05 /tmp/example_plist.plist
```

As such, ensure to add a repair to scripts to reset permissions and ownership after the command has been used or consider using the following command instead (Note the full path is required if /usr/libexec is not in the paths list:

```
/usr/libexec/PlistBuddy
```

Related Content

- Custom Fields
- Filesets / Payloads
- Fileset / Payload Script Exit Code Status

Revision #9
★Created 2 July 2023 14:43:06 by Josh Levitsky
Updated 19 July 2024 15:27:19 by Jared Jones