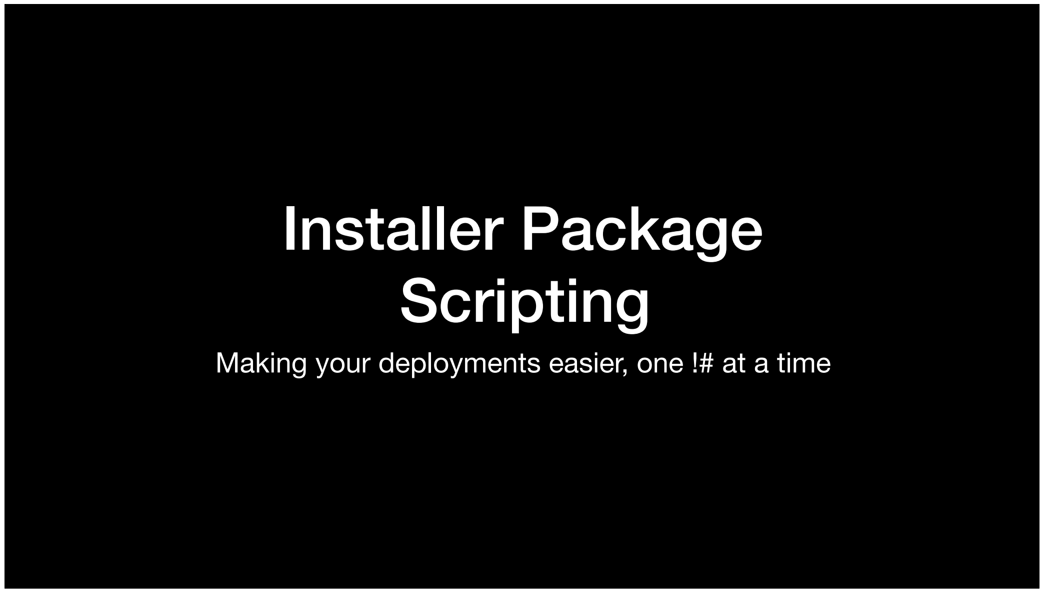
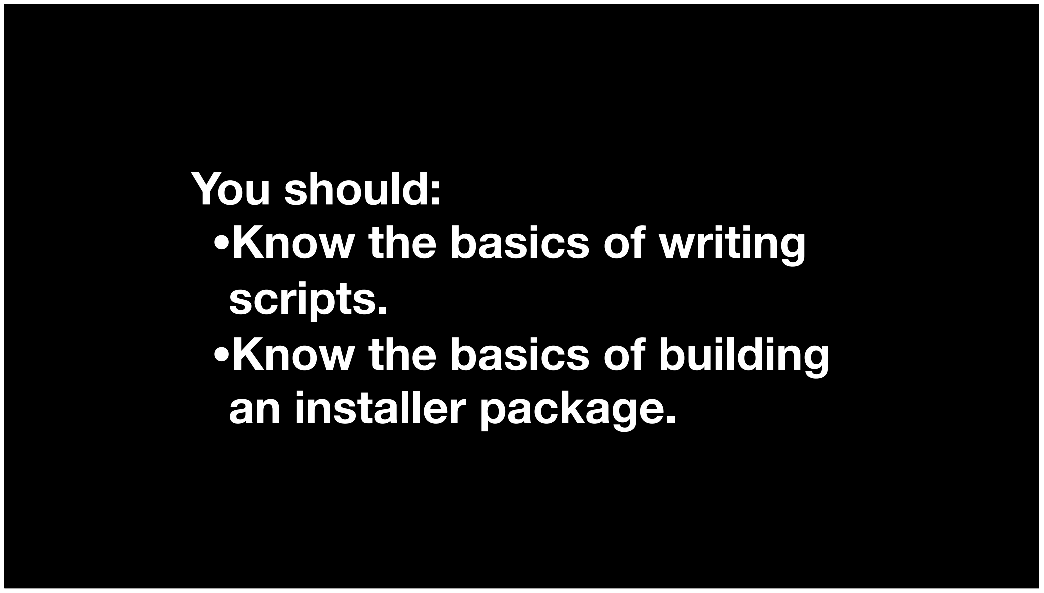


Installer Package Scripting

Making your deployments easier, one !# at a time

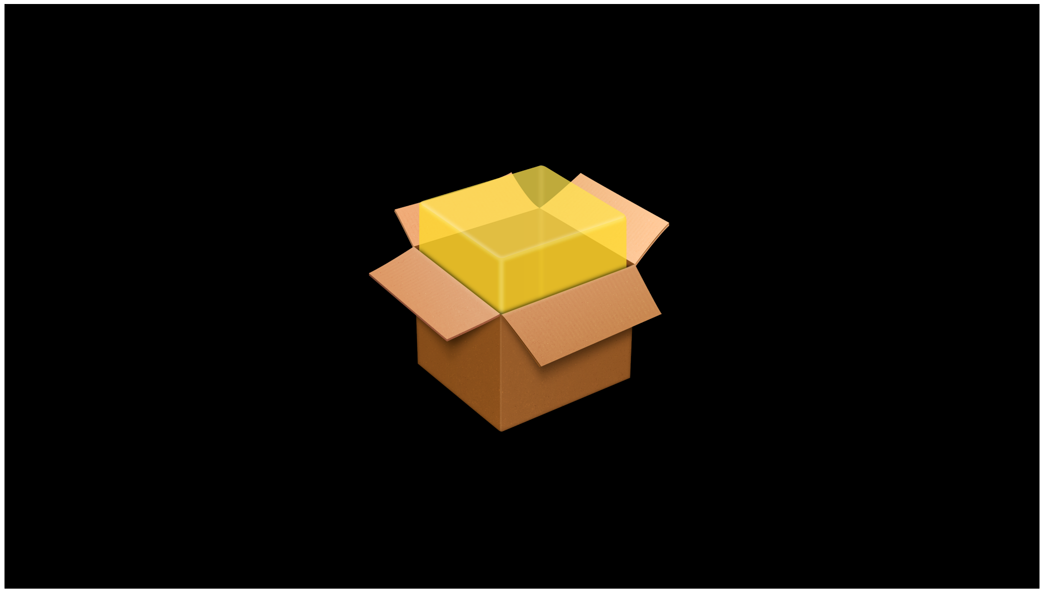
Before we get started, there’s two things I’d like to mention. The first is that, all of the sides, speakers’ notes and the demos are available for download and I’ll be providing a link at the end of the talk. I tend to be one of those folks who can’t keep up with the speaker and take notes at the same time, so for those folks in the same situation, no need to take notes. Everything I’m covering is going to be available for download.



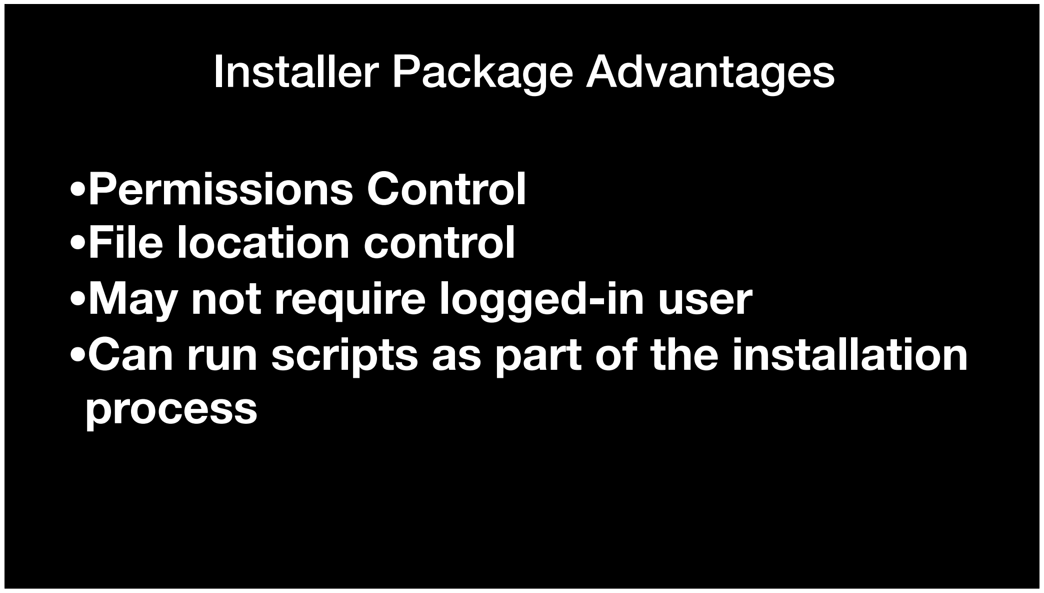
You should:

- Know the basics of writing scripts.
- Know the basics of building an installer package.

To set some expectations management for this session, my assumption is that folks here already know the basics of how to write scripts and build an installer package. We’re going to be discussing how to build on that foundation to solve problems.



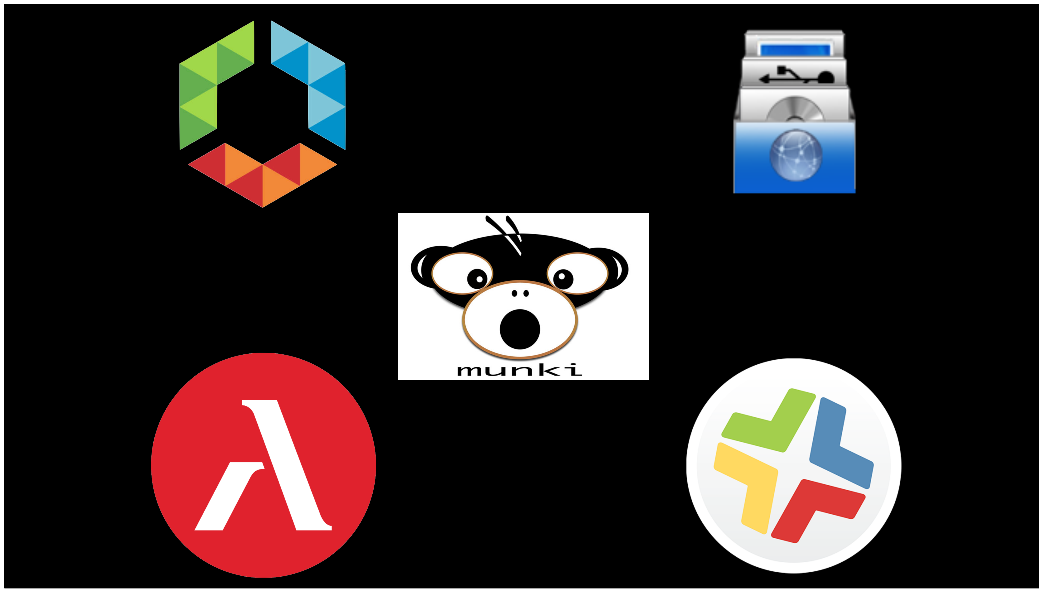
Now, installer packages are pretty great. They’re one of Apple’s two recommended ways to install software, with the other method being dragging and dropping a self-contained application into place.



Installer Package Advantages

- **Permissions Control**
- **File location control**
- **May not require logged-in user**
- **Can run scripts as part of the installation process**

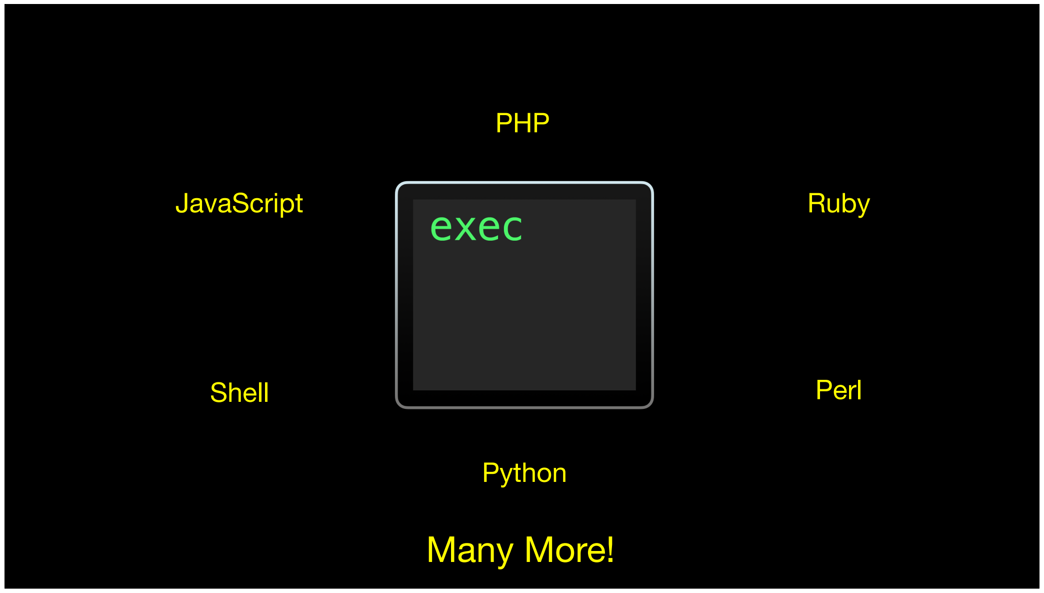
I personally believe installer packages are the superior installation method. You can control the permissions on the installed files. You can control where the installed files are placed. When properly built, a package can be installed with nobody logged in. Last, but not least, you can run scripts as part of the installation process.



Another advantage, outside of the package itself, is that every Mac management system I've run into yet has the ability to install an installer package. For folks who have to support multiple management systems, this common ability means you can build one package and be confident that it will deploy the same regardless of the management system used to install it.

Using Scripts

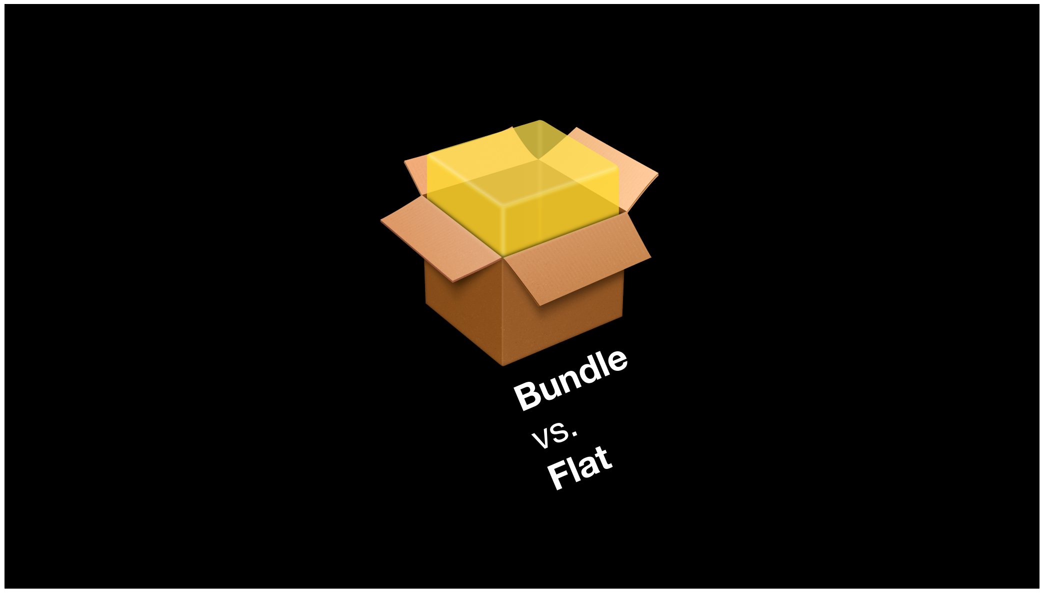
However, the real magic of using installer packages happens when you start using scripts as part of the installation process.



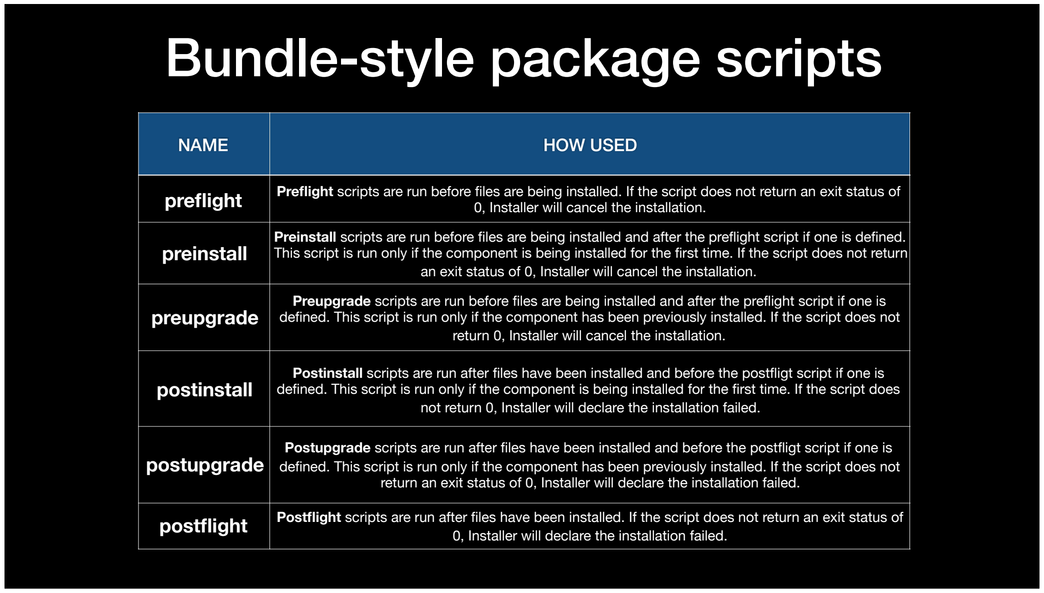
What languages can you use? Well, if a client Mac supports a language, the Installer tool should be able to support running scripts written in that language on that client.

```
#!/bin/bash
```

Probably the most common form is going to be shell scripting using the bash shell, though, so that's what I'm going to focus on.



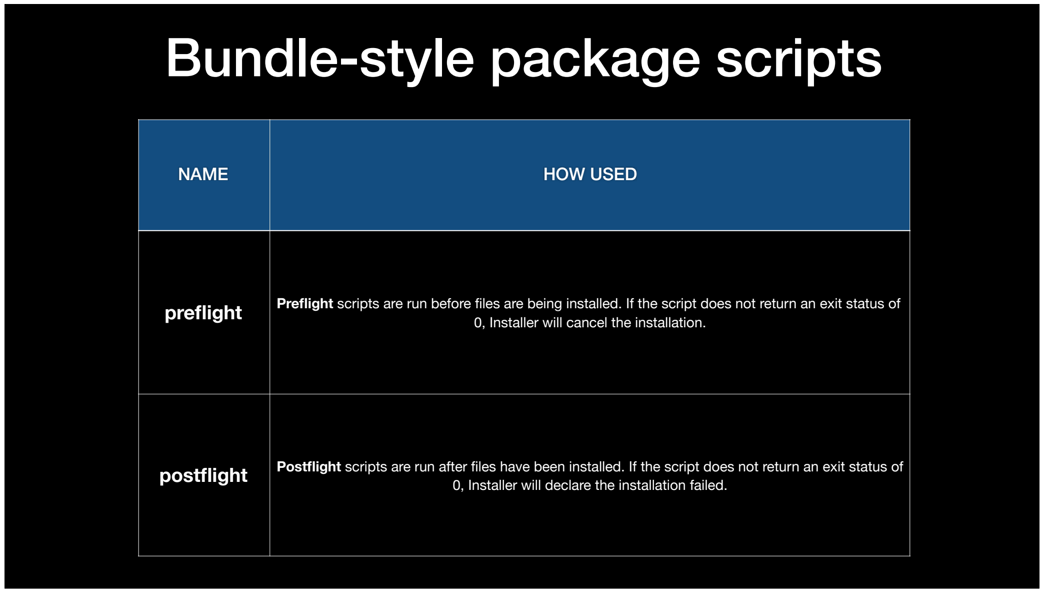
From there, your scripting options are going to depend on whether you're using a bundle-style package or a flat package.



Bundle-style package scripts

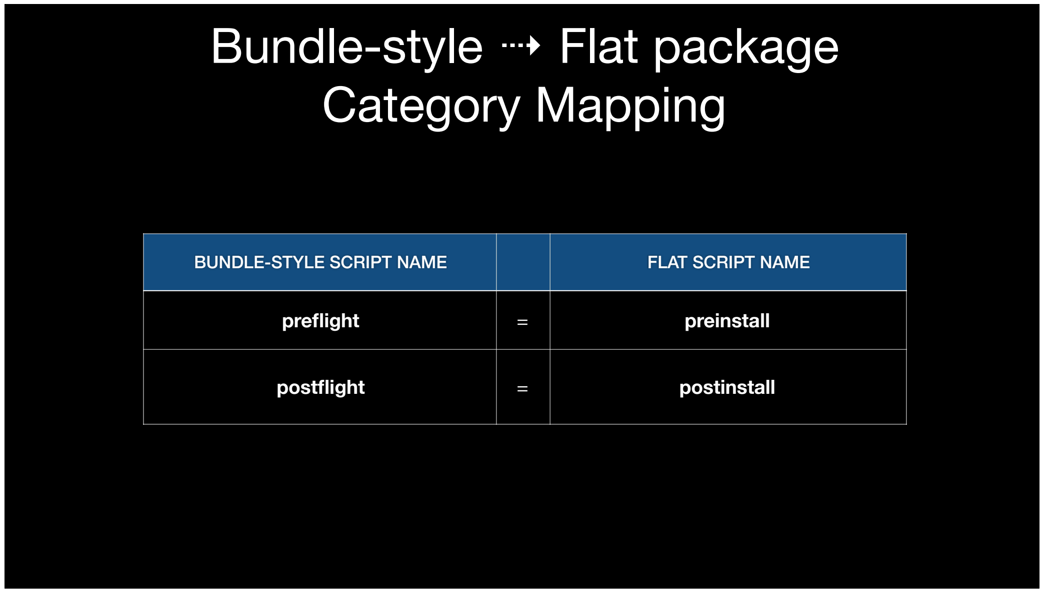
NAME	HOW USED
preflight	**Preflight** scripts are run before files are being installed. If the script does not return an exit status of 0, Installer will cancel the installation.
preinstall	**Preinstall** scripts are run before files are being installed and after the preflight script if one is defined. This script is run only if the component is being installed for the first time. If the script does not return an exit status of 0, Installer will cancel the installation.
preupgrade	**Preupgrade** scripts are run before files are being installed and after the preflight script if one is defined. This script is run only if the component has been previously installed. If the script does not return 0, Installer will cancel the installation.
postinstall	**Postinstall** scripts are run after files have been installed and before the postfligt script if one is defined. This script is run only if the component is being installed for the first time. If the script does not return 0, Installer will declare the installation failed.
postupgrade	**Postupgrade** scripts are run after files have been installed and before the postfligt script if one is defined. This script is run only if the component has been previously installed. If the script does not return an exit status of 0, Installer will declare the installation failed.
postflight	**Postflight** scripts are run after files have been installed. If the script does not return an exit status of 0, Installer will declare the installation failed.

Bundle-style installer packages are created as a Mac OS X bundle that contains scripts and a description of the package requirements and behavior. As part of creating a bundle-style package, you can define scripts that will be executed before or after the installation. Normally, there are six types of scripts can be used in a bundle-style package.


Bundle-style package scripts

NAME	HOW USED
preflight	**Preflight** scripts are run before files are being installed. If the script does not return an exit status of 0, Installer will cancel the installation.
postflight	**Postflight** scripts are run after files have been installed. If the script does not return an exit status of 0, Installer will declare the installation failed.

However, most of the time you'll be using these two types. Preflight scripts get run before files are installed and postflight scripts run after files are installed. The other types of scripts only apply in specific defined circumstances which may not apply to your package.

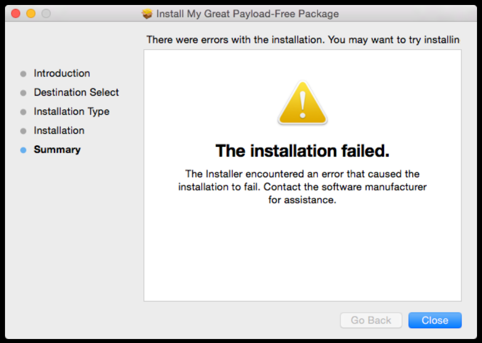


Bundle-style ⇢ Flat package Category Mapping

BUNDLE-STYLE SCRIPT NAME		FLAT SCRIPT NAME
preflight	=	**preinstall**
postflight	=	**postinstall**

When Apple created the flat package format, they simplified the script options and made only two script options available. The bundle-style post/preinstall and pre/postupdate script options were removed. Apple retained the concept of pre- and postflight scripts, but Apple renamed those script categories to preinstall and postinstall to make their functions more clear.

Scripts must return an exit status of zero

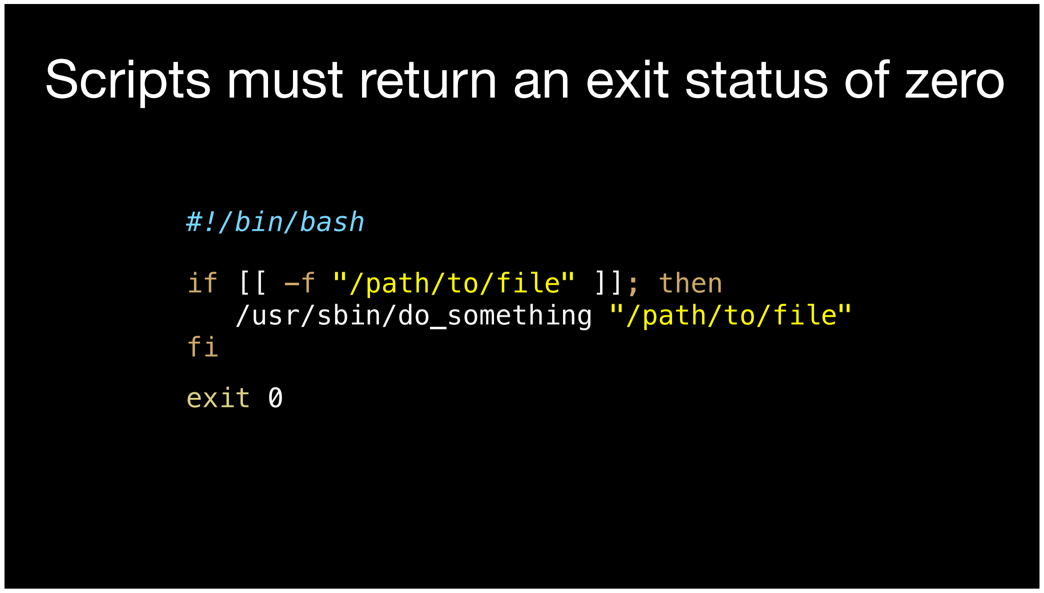


There were errors with the installation. You may want to try installin

The installation failed.

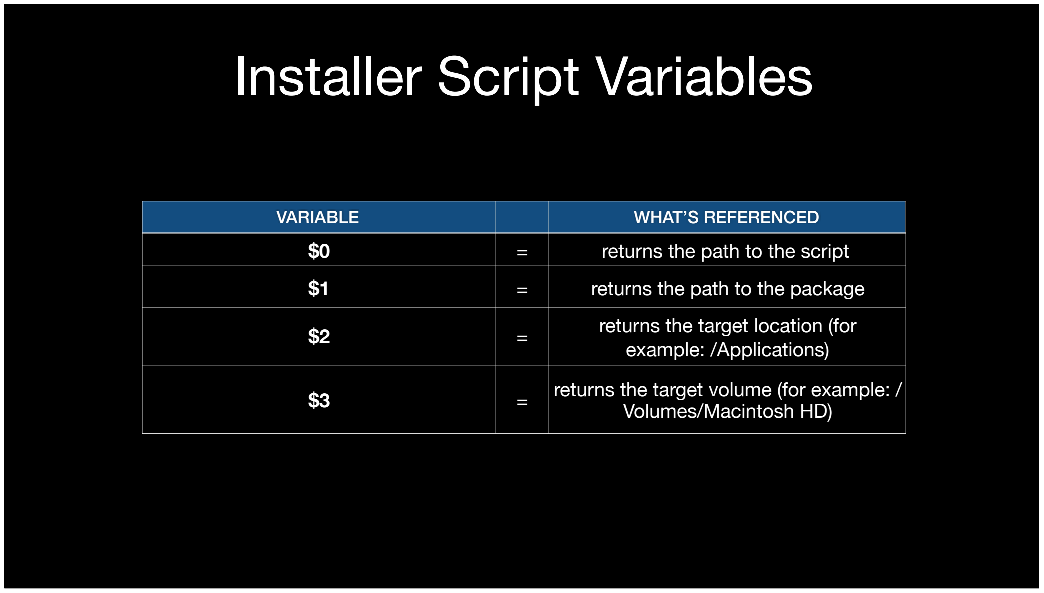
The Installer encountered an error that caused the installation to fail. Contact the software manufacturer for assistance.

A commonality between scripts used in bundle-style and flat packages are that the scripts must return an exit status of zero or else Installer will report failure.


Scripts must return an exit status of zero

```
#!/bin/bash

if [[ -f "/path/to/file" ]]; then
   /usr/sbin/do_something "/path/to/file"
fi

exit 0
```

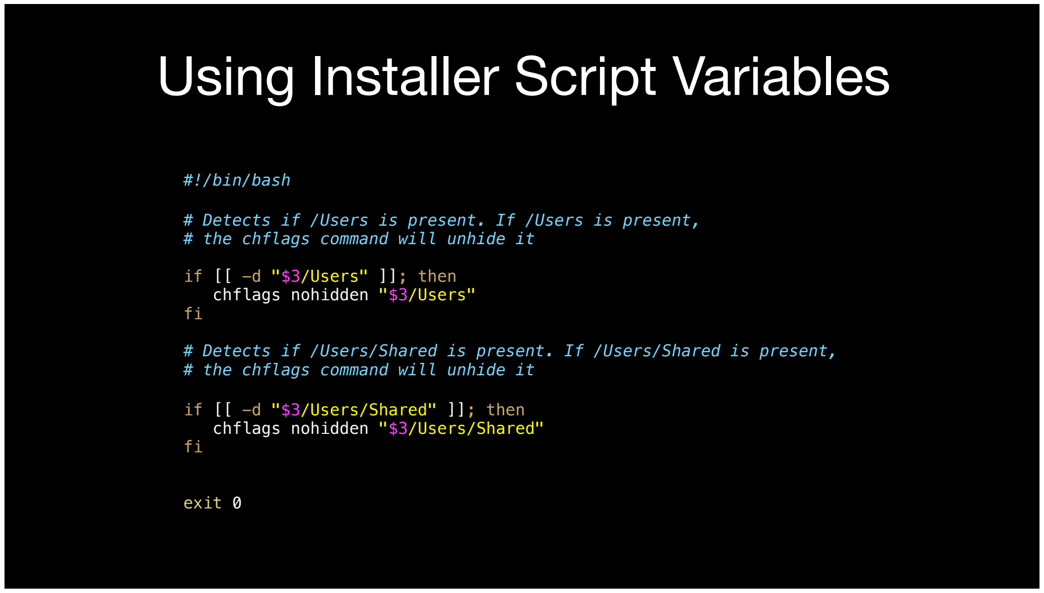
If you're building a shell script for an installer package, one way to ensure that the script returns a status of zero is by adding "exit 0" to the end of your script.



Installer Script Variables

VARIABLE		WHAT'S REFERENCED
$0	=	returns the path to the script
$1	=	returns the path to the package
$2	=	returns the target location (for example: /Applications)
$3	=	returns the target volume (for example: /Volumes/Macintosh HD)

When building a script for use with an installer package, it's helpful to know that the Installer application can automatically pass along information to the script using variables. The ones shown on the screen are for shell scripts.

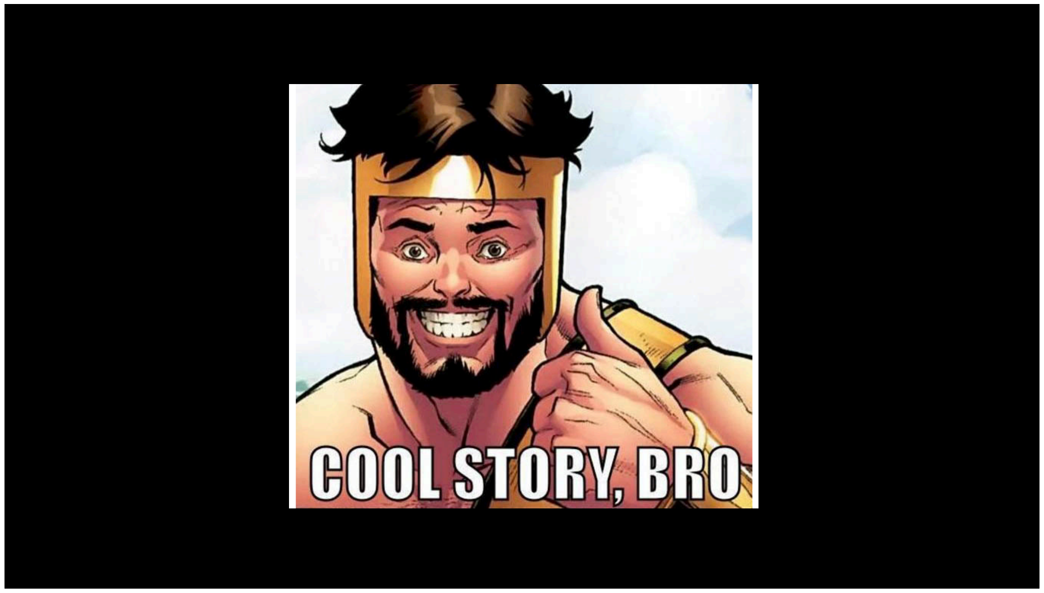


Using Installer Script Variables

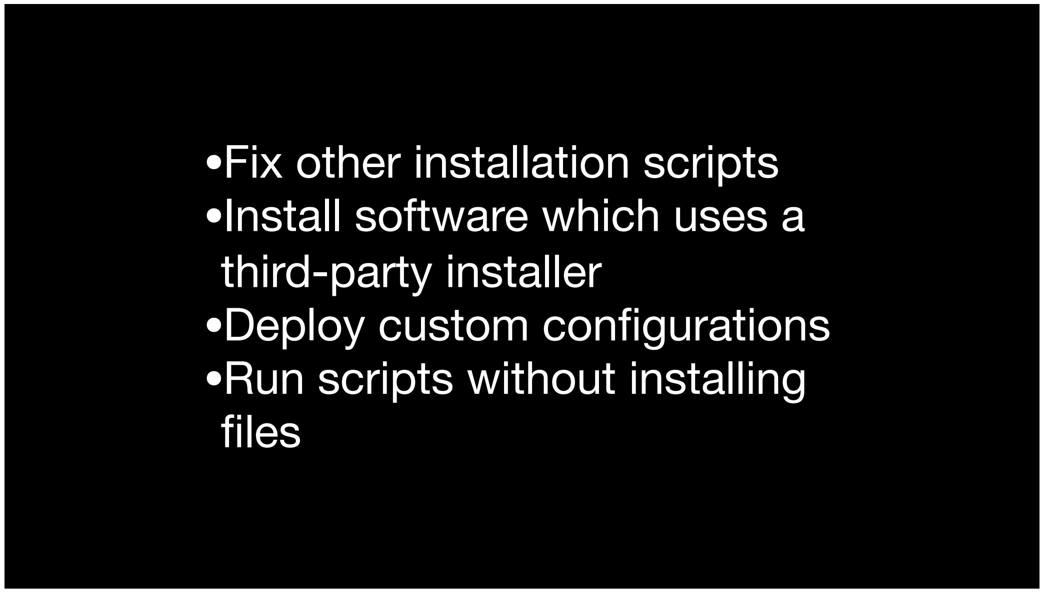
```
#!/bin/bash

# Detects if /Users is present. If /Users is present,
# the chflags command will unhide it

if [[ -d "$3/Users" ]]; then
   chflags nohidden "$3/Users"
fi

# Detects if /Users/Shared is present. If /Users/Shared is present,
# the chflags command will unhide it

if [[ -d "$3/Users/Shared" ]]; then
   chflags nohidden "$3/Users/Shared"
fi


exit 0
```

Here's an example of using the dollar sign three variable in an installer script. In this case, we're able to take advantage of the Installer telling us which drive the package is being installed on to have our script run actions on the targeted drive.

One thing that's important to know is the closest set of variables to the script is going to win. If your system management tool assigns dollar sign 3 to something different than Installer does, the Installer-assigned meaning will be used by the script inside the package.

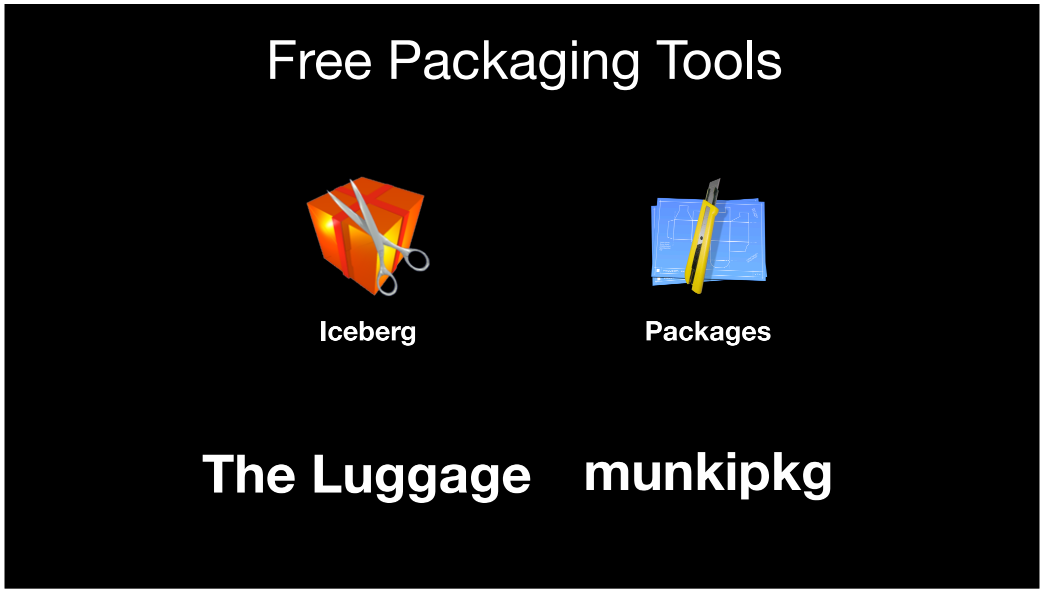


So, big deal right? I can use scripts as part of installer packages. Great. Why have a session about this?



- Fix other installation scripts
- Install software which uses a third-party installer
- Deploy custom configurations
- Run scripts without installing files

There are lots of cool ways to use installer scripts. You can use them to fix problems in other people's installer scripts, install software which doesn't use any of Apple's recommended installation methods, deploy configuration files for other installers, and you can even build an installer package which is only a delivery mechanism for scripts.



Free Packaging Tools

Iceberg

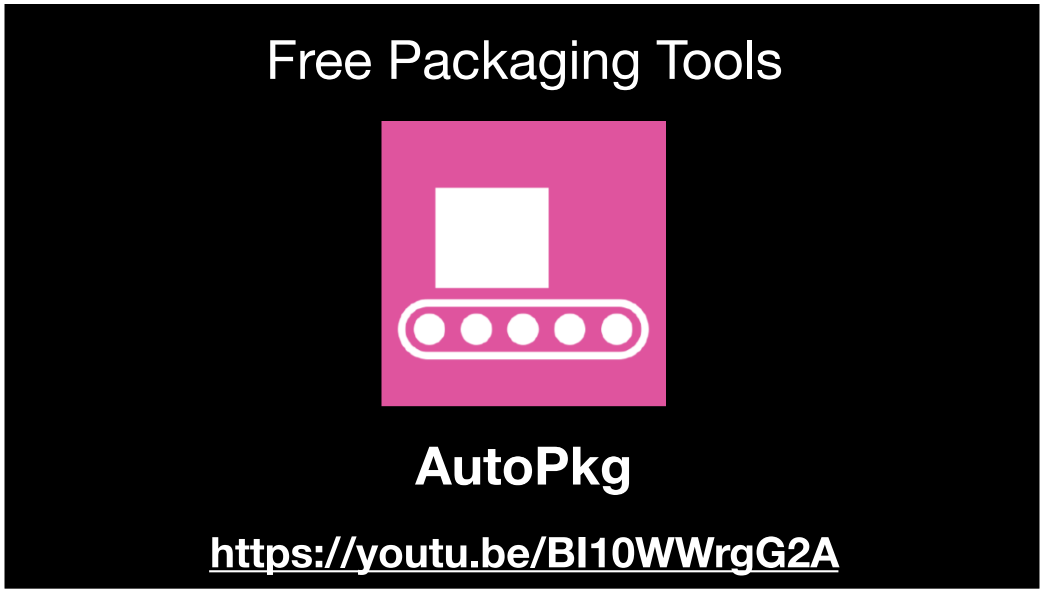
Packages

The Luggage munkipkg

For your packaging needs, there are a number of free tools available. I personally use Stéphane Sudre’s Iceberg and Packages for when I need to manually create a package, but try them all out and use the one that works best for you. Iceberg builds bundle-style installer packages, while Packages builds flat packages.

If you need to use source control for your packaging, I recommend using either Munkipkg or The Luggage. Both are free open source tools which allow packages to be built in a consistent and repeatable way using source files and scripts. In the case of the The Luggage, this was a tool originally created by Joe Block, who had written a tool when he worked at Google which used makefiles to generate installers. This allowed the other members of his group to easily review installer package changes before they were put into production. When Joe left Google, he wanted to have a similar tool available, so he wrote and open-sourced The Luggage.

Jamf’s Composer is also an available tool for creating packages, and it includes a handy feature for taking "before" and "after" snapshots of your system, where you make the initial snapshot, install your software, then take a second snapshot once the installation is finished. Composer will then generate a list of the files and directories that changed and use those changes to generate a package. The main reason it’s not up on this slide is that it isn’t free. It is available for purchase from Jamf and is also included as part of Jamf’s Jamf Pro Suite.



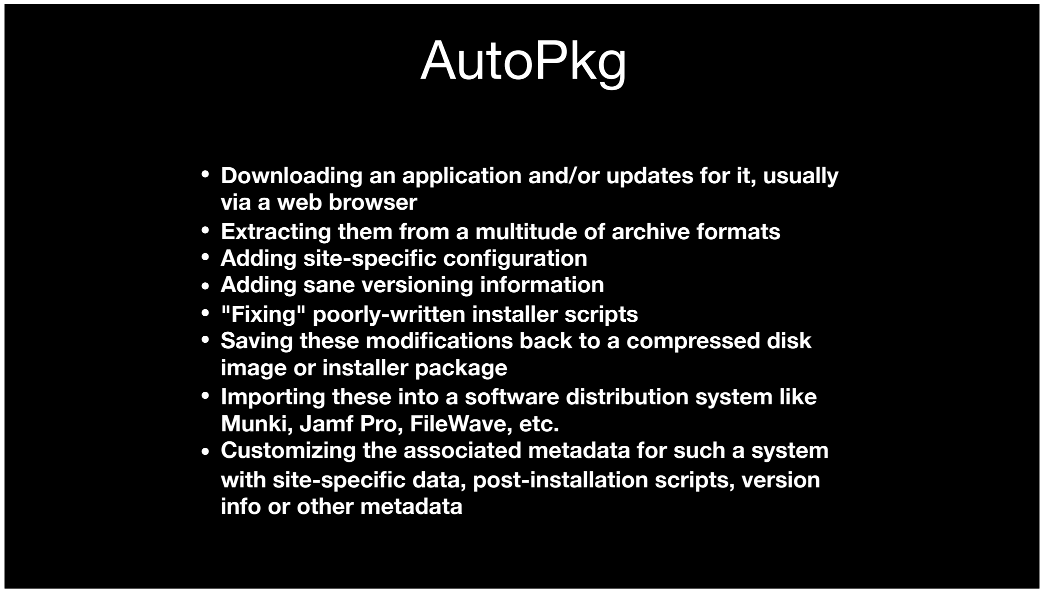
Free Packaging Tools

AutoPkg

https://youtu.be/BI10WWrgG2A

To further automate your packaging, there is another open source tool called AutoPkg. AutoPkg is hugely useful because it’s designed to automate the tasks one would normally perform manually to prepare third-party software for deployment.

For those not familiar with AutoPkg, there was a great introductory talk by Anthony Reimer from the University of Calgary this past summer at the Penn State MacAdmins Conference. The talk was posted to YouTube and the link is available at the bottom of the screen.



AutoPkg

- Downloading an application and/or updates for it, usually via a web browser
- Extracting them from a multitude of archive formats
- Adding site-specific configuration
- Adding sane versioning information
- "Fixing" poorly-written installer scripts
- Saving these modifications back to a compressed disk image or installer package
- Importing these into a software distribution system like Munki, Jamf Pro, FileWave, etc.
- Customizing the associated metadata for such a system with site-specific data, post-installation scripts, version info or other metadata

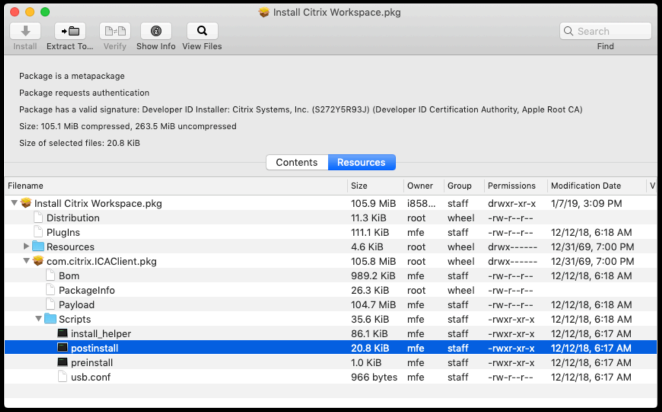
While this talk is going to focus on manually building packages to solve specific problems using installer scripts, I want to mention AutoPkg because it often allows you to solve the same problems in a consistent and repeatable fashion, with the added bonus of also automating the download of the applications you need to package.

In many cases, I will solve a packaging problem first by manually building and testing a package. Then I will write a recipe file for AutoPkg to automate the building of that same package. For those not familiar with AutoPkg and how it works, a recipe is an XML file which describes a sequence of tasks for AutoPkg to run. These tasks can include downloading some piece of software, building an installer package for that software, and then importing the newly-created installer package into your Mac software management tool.

To help illustrate this, I'll be discussing some examples later of how I first solved a problem with manual packaging then later turned it into an AutoPkg recipe.

Fixing Other Installation Scripts

To give some practical examples, let's look at how a vendor's installer package didn't cover all the possible deployment scenarios and how you can use your own script to fix it. For this scenario, we'll be working with the Citrix Workspace installer.



Package is a metapackage

Package requests authentication

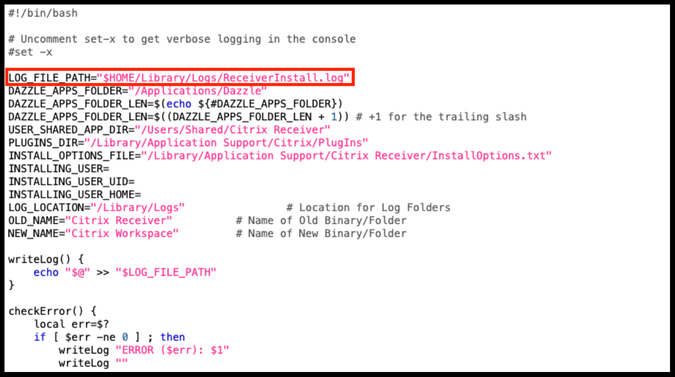
Package has a valid signature: Developer ID Installer: Citrix Systems, Inc. (S272Y5R93J) (Developer ID Certification Authority, Apple Root CA)

Size: 105.1 MiB compressed, 263.5 MiB uncompressed

Size of selected files: 20.8 KiB

Filename	Size	Owner	Group	Permissions	Modification Date	V
Install Citrix Workspace.pkg	105.9 MiB	i858...	staff	drwxr-xr-x	1/7/19, 3:09 PM	
Distribution	11.3 KiB	root	wheel	-rw-r--r--		
PlugIns	111.1 KiB	mfe	staff	-rw-r--r--	12/12/18, 6:18 AM	
Resources	4.6 KiB	root	wheel	drwx------	12/31/69, 7:00 PM	
com.citrix.ICAClient.pkg	105.8 MiB	root	wheel	drwx------	12/31/69, 7:00 PM	
Bom	989.2 KiB	mfe	staff	-rw-r--r--	12/12/18, 6:18 AM	
PackageInfo	26.3 KiB	root	wheel	-rw-r--r--		
Payload	104.7 MiB	mfe	staff	-rw-r--r--	12/12/18, 6:18 AM	
Scripts	35.6 KiB	mfe	staff	-rwxr-xr-x	12/12/18, 6:18 AM	
install_helper	86.1 KiB	mfe	staff	-rwxr-xr-x	12/12/18, 6:17 AM	
postinstall	20.8 KiB	mfe	staff	-rwxr-xr-x	12/12/18, 6:17 AM	
preinstall	1.0 KiB	mfe	staff	-rwxr-xr-x	12/12/18, 6:17 AM	
usb.conf	966 bytes	mfe	staff	-rw-r--r--	12/12/18, 6:17 AM	

The Workspace installer package provided by Citrix runs both a preinstall and postinstall script.

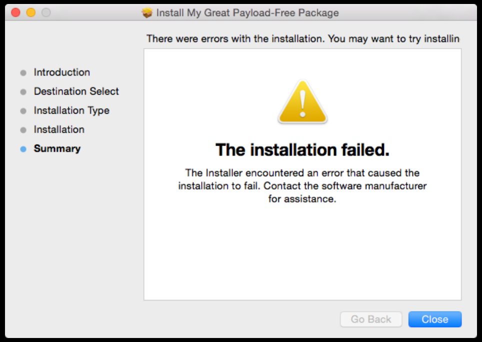


```
#!/bin/bash

# Uncomment set-x to get verbose logging in the console
#set -x

LOG_FILE_PATH="$HOME/Library/Logs/ReceiverInstall.log"
DAZZLE_APPS_FOLDER="/Applications/Dazzle"
DAZZLE_APPS_FOLDER_LEN=$(echo ${#DAZZLE_APPS_FOLDER})
DAZZLE_APPS_FOLDER_LEN=$((DAZZLE_APPS_FOLDER_LEN + 1)) # +1 for the trailing slash
USER_SHARED_APP_DIR="/Users/Shared/Citrix Receiver"
PLUGINS_DIR="/Library/Application Support/Citrix/PlugIns"
INSTALL_OPTIONS_FILE="/Library/Application Support/Citrix Receiver/InstallOptions.txt"
INSTALLING_USER=
INSTALLING_USER_UID=
INSTALLING_USER_HOME=
LOG_LOCATION="/Library/Logs"                # Location for Log Folders
OLD_NAME="Citrix Receiver"          # Name of Old Binary/Folder
NEW_NAME="Citrix Workspace"         # Name of New Binary/Folder

writeLog() {
    echo "$@" >> "$LOG_FILE_PATH"
}

checkError() {
    local err=$?
    if [ $err -ne 0 ] ; then
        writeLog "ERROR ($err): $1"
        writeLog ""
```

LOG_FILE_PATH="$HOME/Library/Logs/ReceiverInstall.log"

As part of the postinstall script, one of the script variables references HOME. HOME is a variable which is set at login to be the path name for the user’s home directory. Citrix’s working assumption is that someone will be logged in when Citrix Workspace is installed.

Scripts must return an exit status of zero

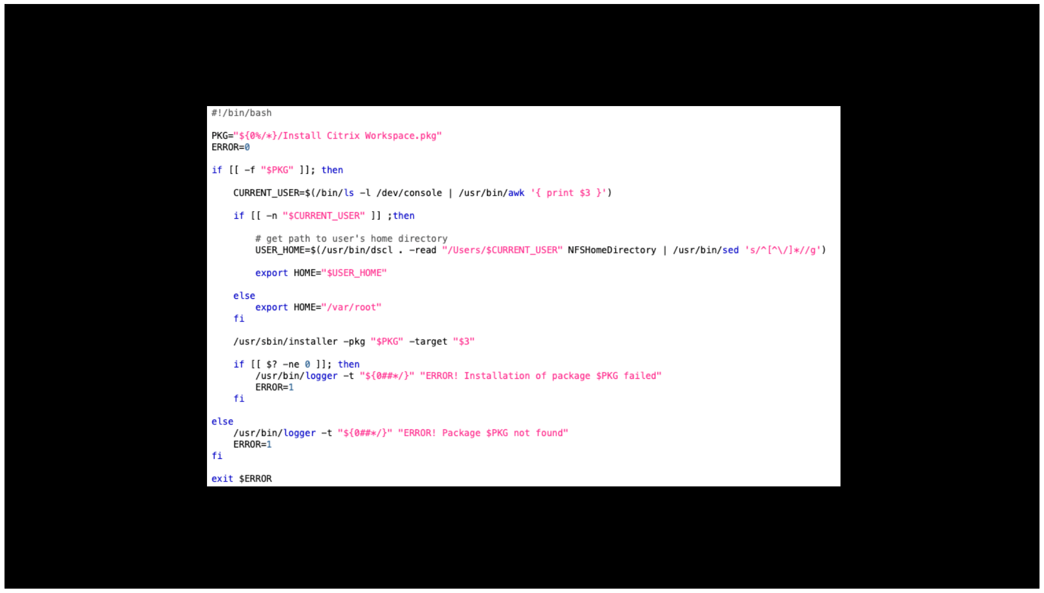


There were errors with the installation. You may want to try installin

The installation failed.

The Installer encountered an error that caused the installation to fail. Contact the software manufacturer for assistance.

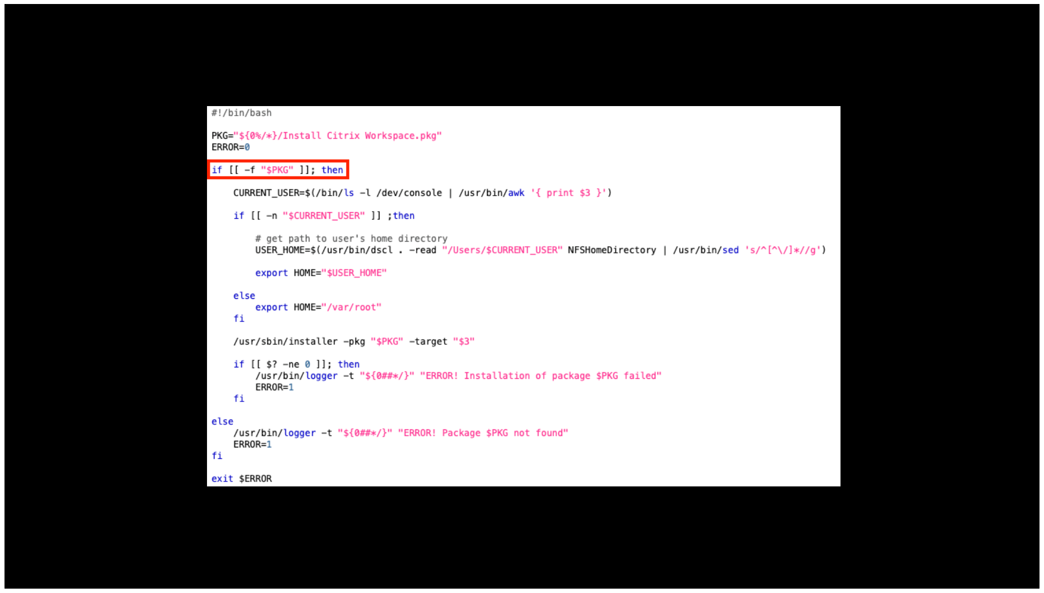
However, there will be times when nobody is logged in. In these cases, an error will be generated and the script will return an exit status other than zero. One thing to remember is that installation scripts must return an exit status of zero or else Installer will report failure.



```bash
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

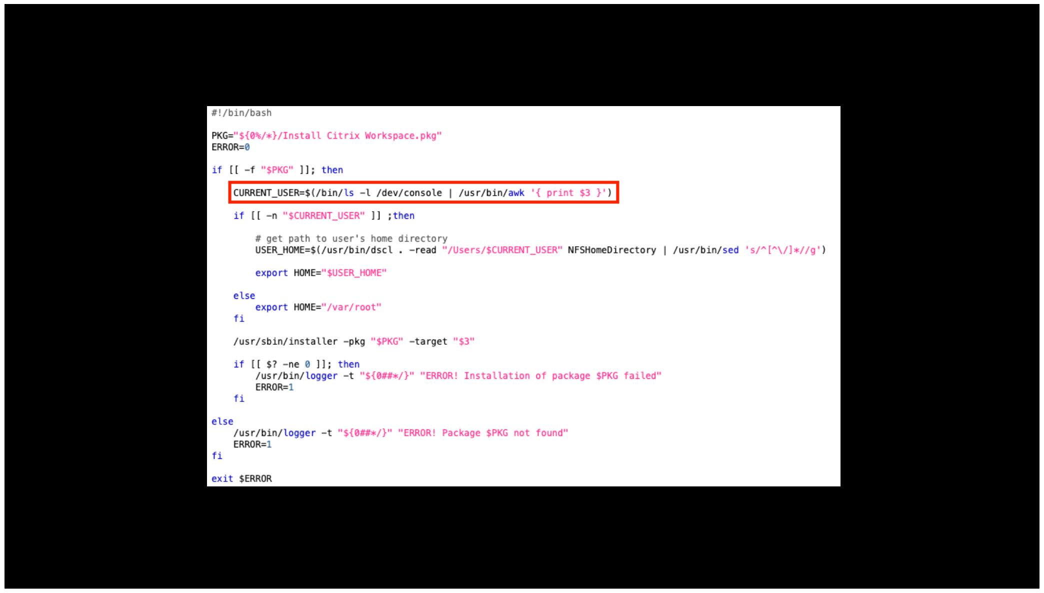
How to fix this? By avoiding the error condition. We can do this by writing a script that does three tasks:

1. Verifying that the Citrix-built installer is available at a location defined by the script.
2. If the installer is present, making sure HOME always returns a valid result by resetting HOME to be a value controlled by the script.
3. Running the Citrix-built installer while using the updated HOME value.



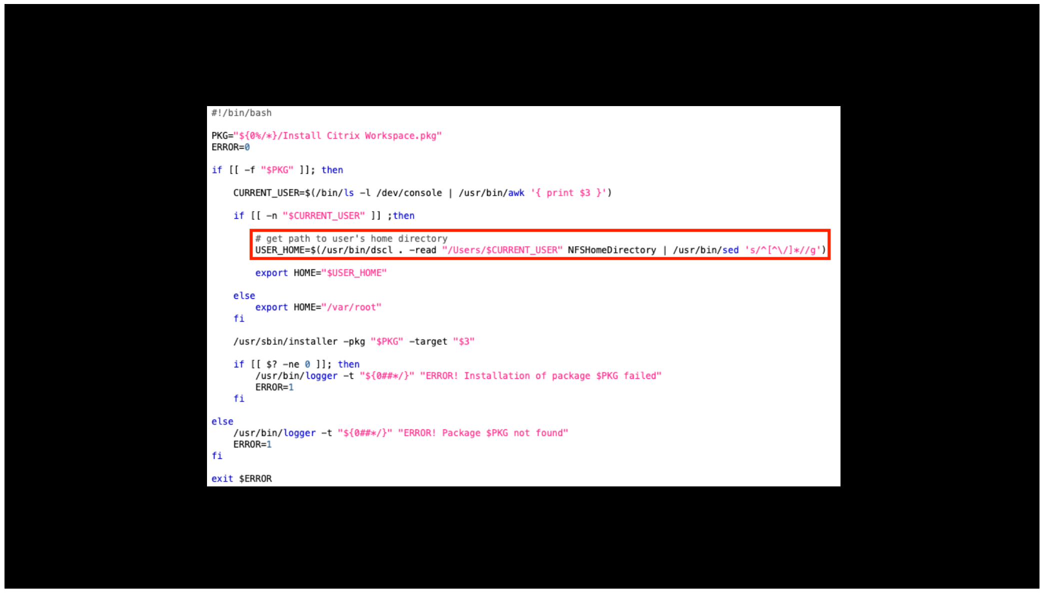
```
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

First step is verifying that the installer is available at the script’s defined location.



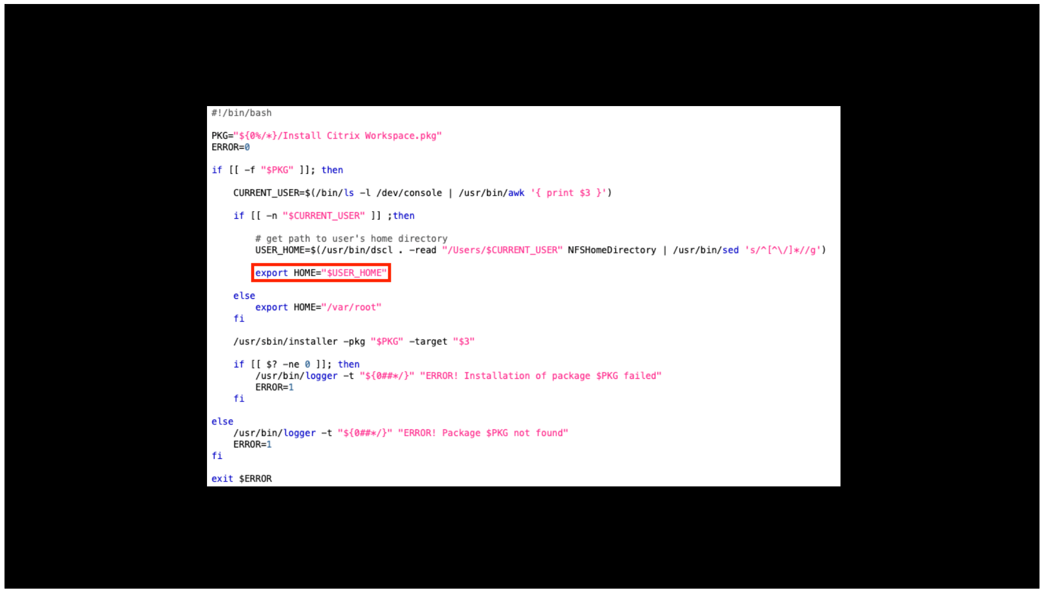
```
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

Next step is trying make sure we match the Citrix installer’s assumption if at all possible. We do that by first figuring out if there’s a logged-in user.



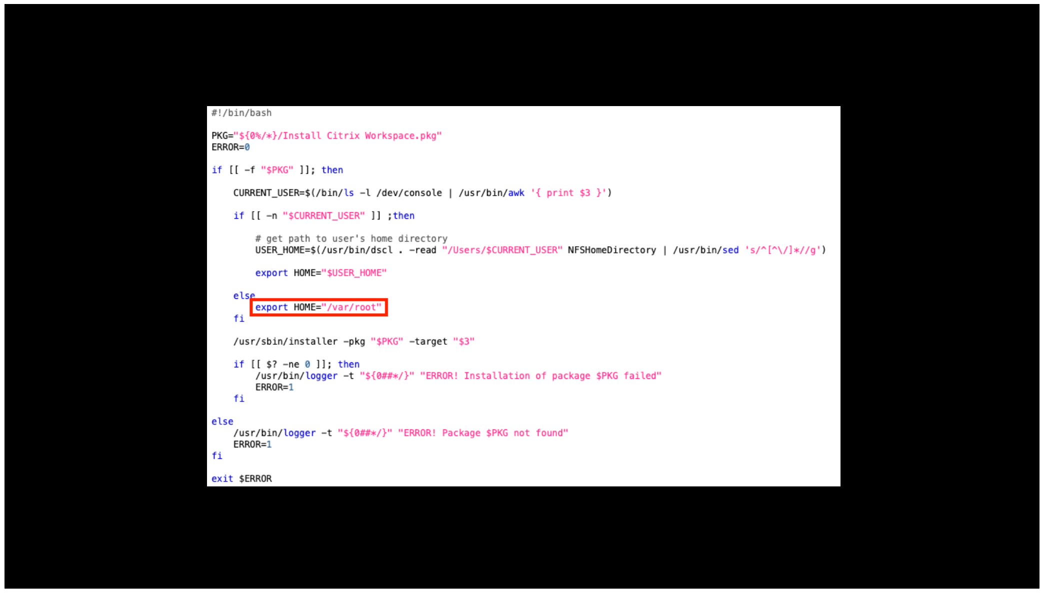
```bash
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

After that, figure out the location of the logged-in user’s home folder.



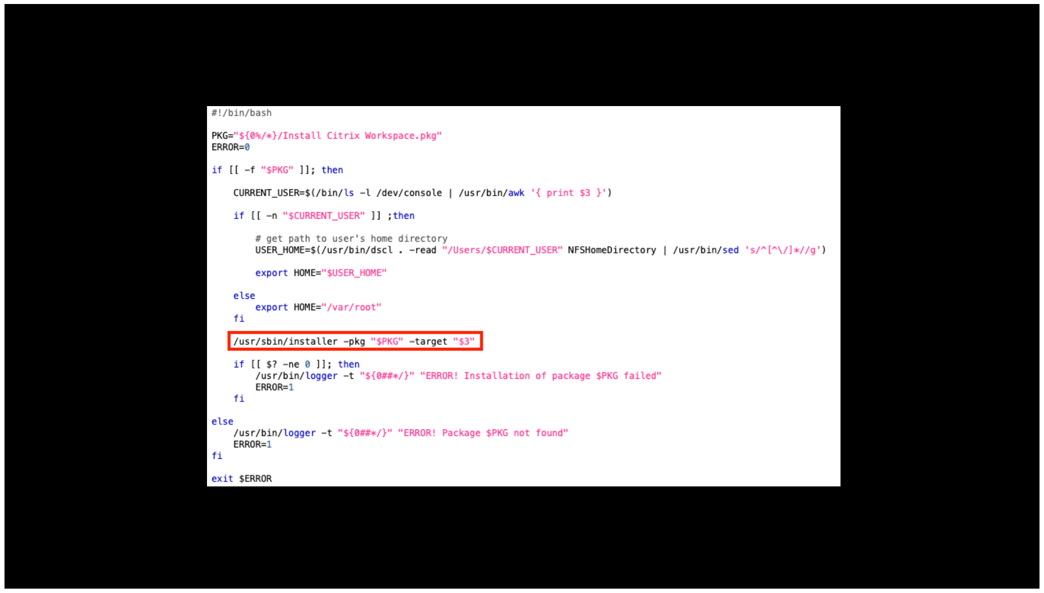
```
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

Next step is setting the HOME value to match the path of the logged-in user's home folder. This matches the default behavior of the HOME variable, so it matches the assumptions Citrix made when building their installer.



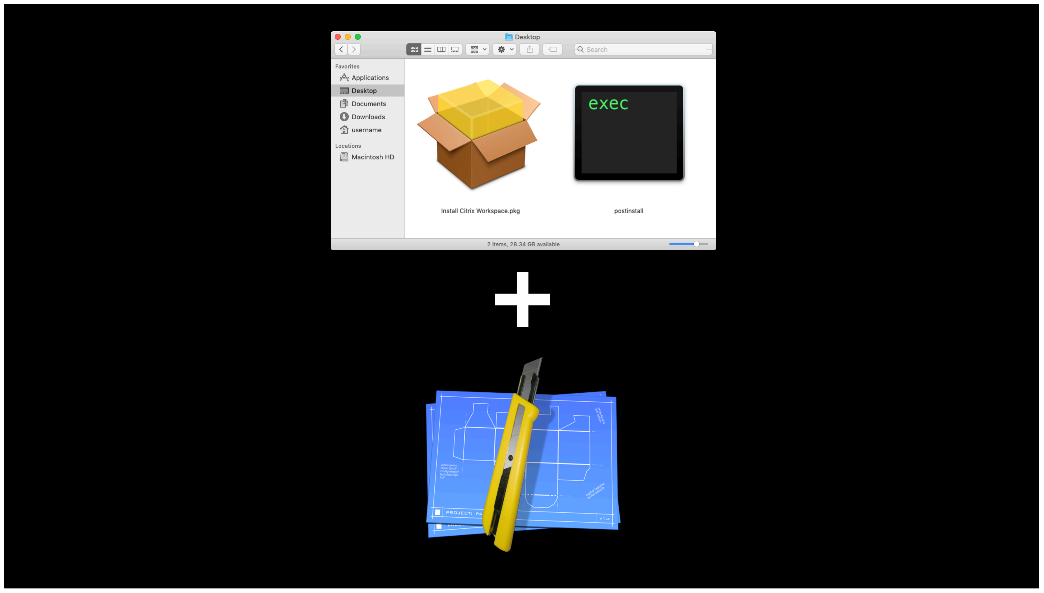
```
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

Here’s where we set HOME to always return a value even in those conditions where the Citrix-built installer would error. If a logged-in user can’t be determined, HOME is set to use the var root directory, which is the home directory of the root user on macOS.

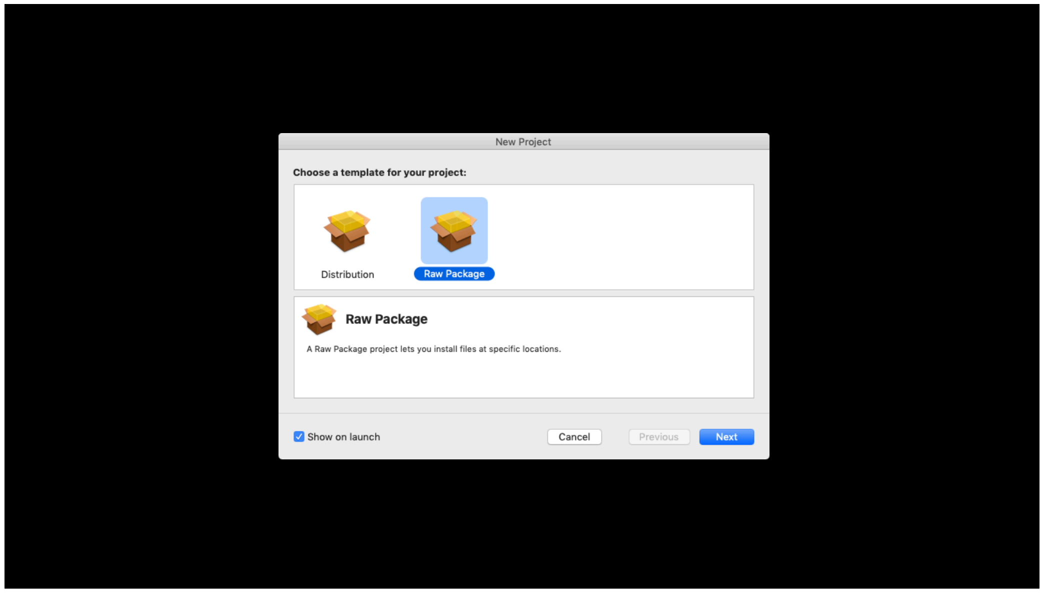


```
#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

    CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

    if [[ -n "$CURRENT_USER" ]] ;then

        # get path to user's home directory
        USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDirectory | /usr/bin/sed 's/^[^\/]*//g')

        export HOME="$USER_HOME"

    else
        export HOME="/var/root"
    fi

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

else
    /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
    ERROR=1
fi

exit $ERROR
```

Last step, the script runs the Citrix-built installer. Now that the script is setting the HOME value to be valid in all conditions, the Citrix-built installer should not error.

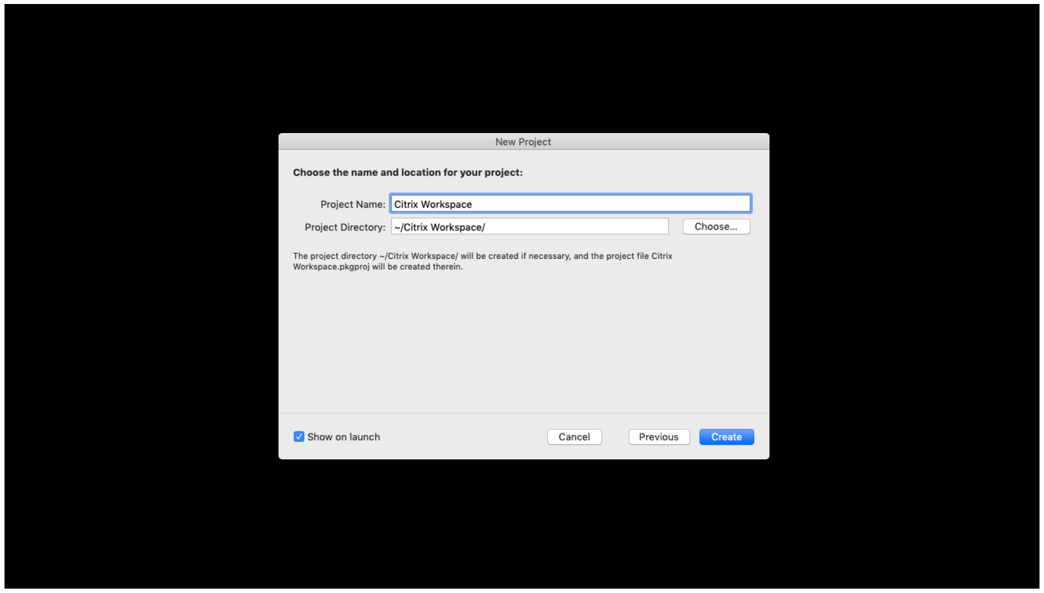


Let’s put all of this together into a package. To do this, I’ll be using the script I just described as a postinstall script, a copy of the the latest Citrix Workspace installer and the Packages app to create the package.



A Raw Package project lets you install files at specific locations.

First step is to open the Packages app and set up a raw package project.



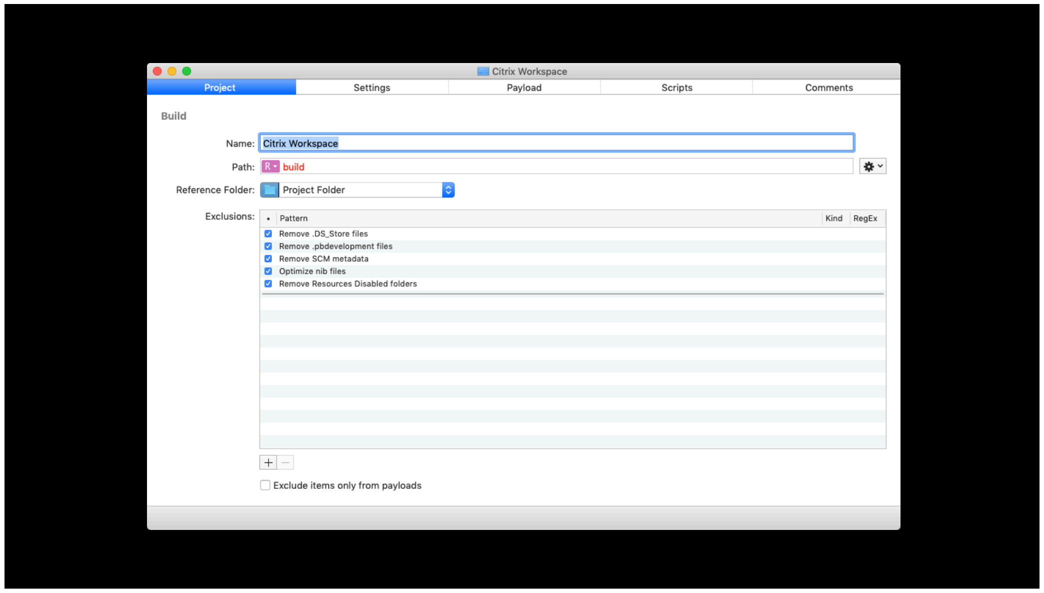
Choose the name and location for your project:

Project Name: Citrix Workspace

Project Directory: ~/Citrix Workspace/

The project directory ~/Citrix Workspace/ will be created if necessary, and the project file Citrix Workspace.pkgproj will be created therein.

Next, we name it and define where the Packages project file will be stored. In this case, I’m naming the package as Citrix Workspace and storing the project files in a Citrix Workspace directory in my home folder.



Citrix Workspace

Project | Settings | Payload | Scripts | Comments

Build

Name: Citrix Workspace

Path: build

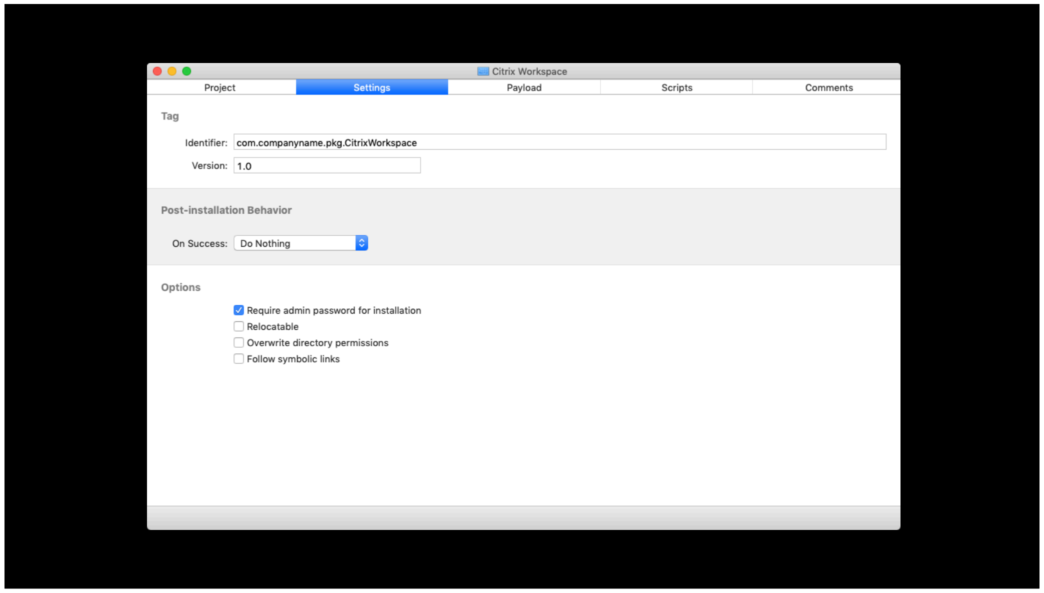
Reference Folder: Project Folder

Exclusions:

•	Pattern	Kind	RegEx
✓	Remove .DS_Store files		
✓	Remove .pbdevelopment files		
✓	Remove SCM metadata		
✓	Optimize nib files		
✓	Remove Resources Disabled folders		

☐ Exclude items only from payloads

In the Project pane, the main thing I'm checking here is the name to make sure it's right. As long as that's correct, there's no need to change anything here from the defaults.



Tag

Identifier: com.companyname.pkg.CitrixWorkspace

Version: 1.0

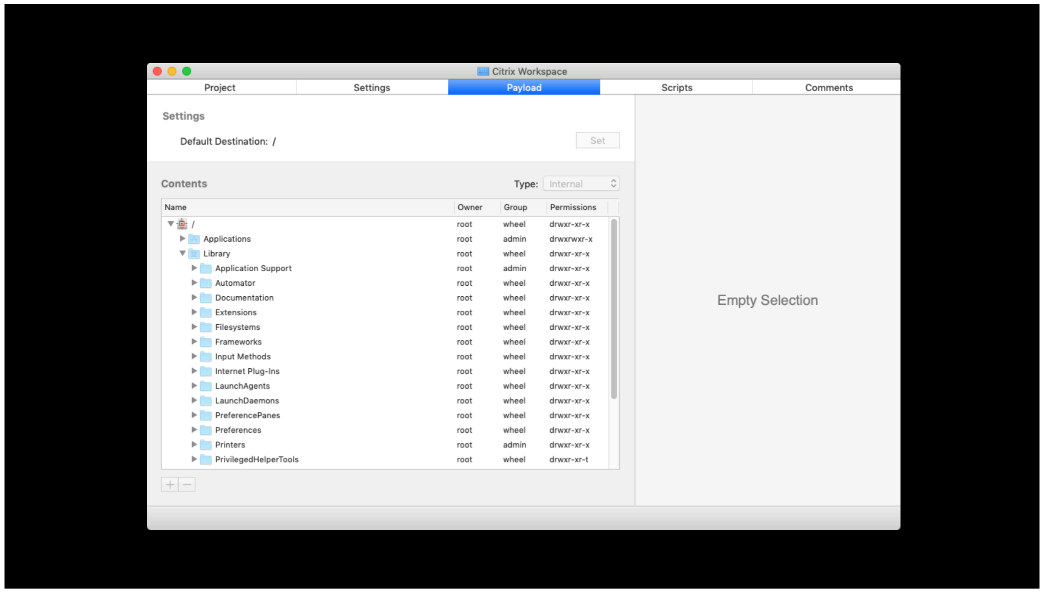
Post-installation Behavior

On Success: Do Nothing

Options

- [x] Require admin password for installation
- [ ] Relocatable
- [ ] Overwrite directory permissions
- [ ] Follow symbolic links

In the Settings pane, we want to require the admin password for installation and on successful installation, we don't want to do anything else.



Settings

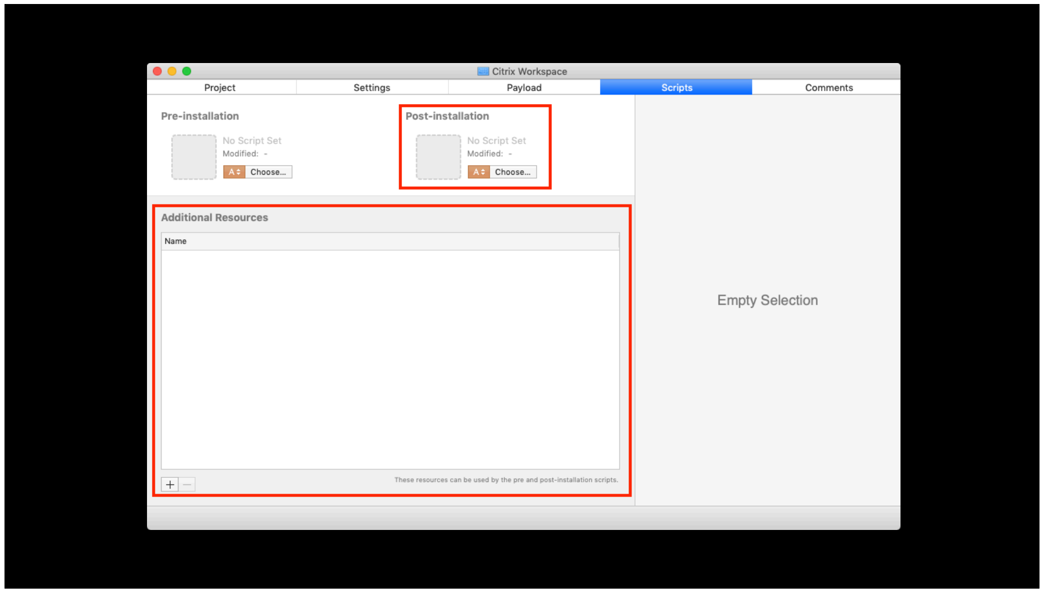
Default Destination: /

Contents

Type: Internal

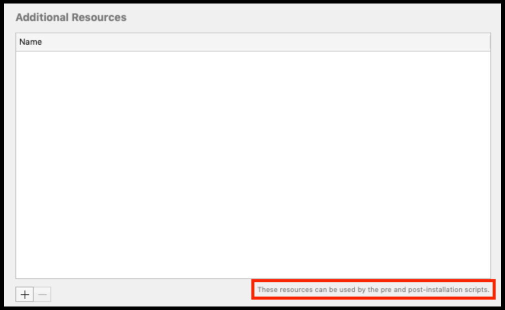
Name	Owner	Group	Permissions
/	root	wheel	drwxr-xr-x
Applications	root	admin	drwxrwxr-x
Library	root	wheel	drwxr-xr-x
Application Support	root	admin	drwxr-xr-x
Automator	root	wheel	drwxr-xr-x
Documentation	root	wheel	drwxr-xr-x
Extensions	root	wheel	drwxr-xr-x
Filesystems	root	wheel	drwxr-xr-x
Frameworks	root	wheel	drwxr-xr-x
Input Methods	root	wheel	drwxr-xr-x
Internet Plug-Ins	root	wheel	drwxr-xr-x
LaunchAgents	root	wheel	drwxr-xr-x
LaunchDaemons	root	wheel	drwxr-xr-x
PreferencePanes	root	wheel	drwxr-xr-x
Preferences	root	wheel	drwxr-xr-x
Printers	root	admin	drwxr-xr-x
PrivilegedHelperTools	root	wheel	drwxr-xr-t

In the Payload pane, we’re not changing anything from the defaults.



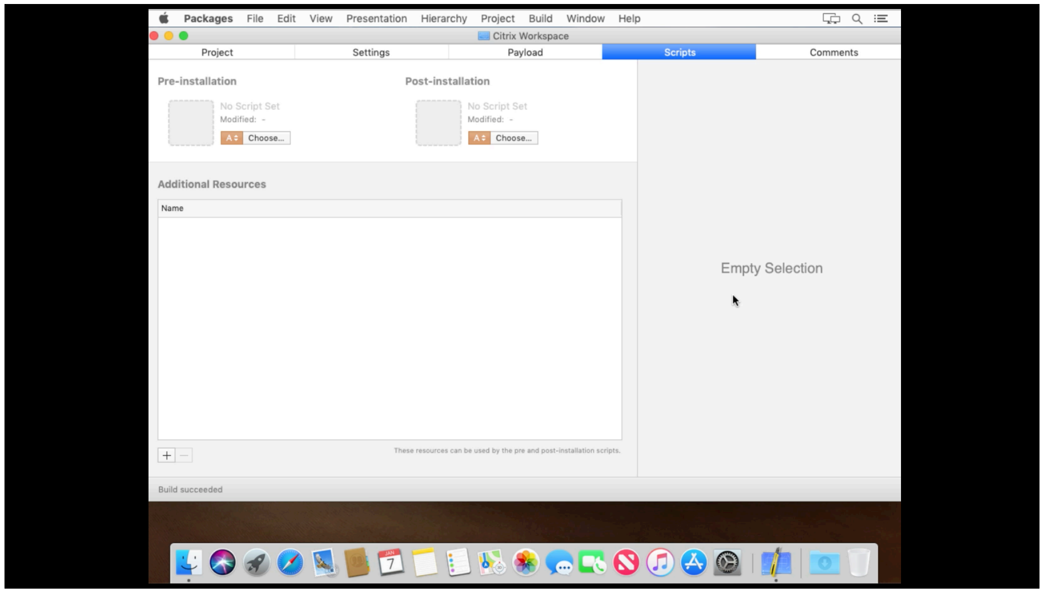
In the scripts pane, we need to add the Citrix-built installer under additional resources and add the postinstall script to the post-installation section.

“These resources can be used by the pre and post-installation scripts”



These resources can be used by the pre and post-installation scripts.

The reason why we add the Citrix installer to the additional resources section is because Packages will store those resources in the same location as it stores the pre and post install scripts. This allows our script to be able to access the Citrix-built Workspaces installer.



Pre-installation

No Script Set
Modified: -

Post-installation

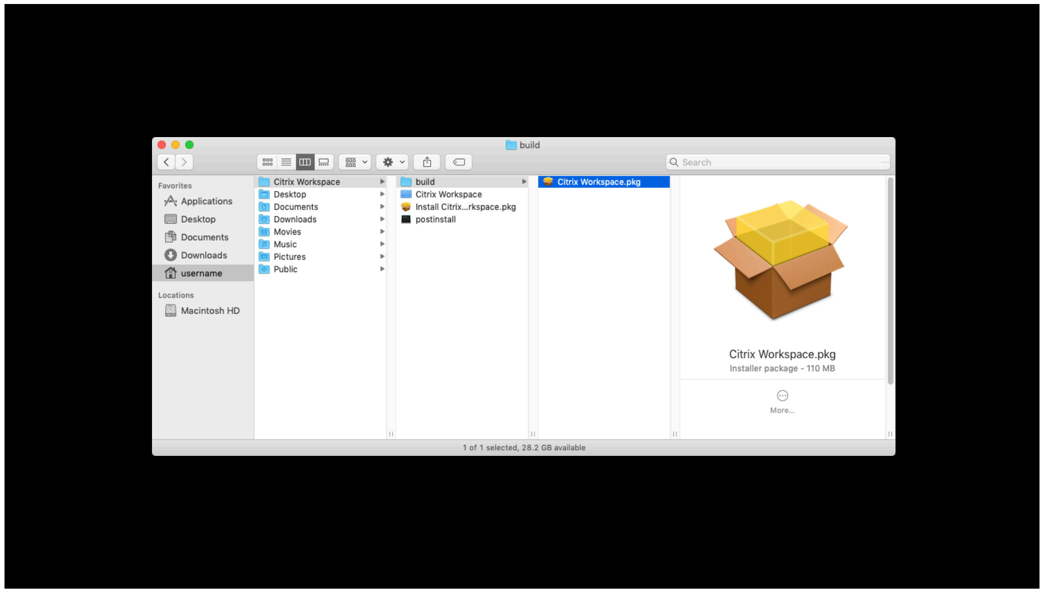
No Script Set
Modified: -

Additional Resources

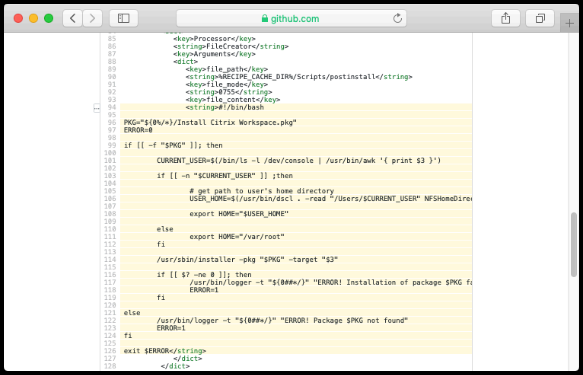
These resources can be used by the pre and post-installation scripts.

Empty Selection

Build succeeded



Once the package is built, you can test it by deploying it onto a Mac while that Mac is at the login screen with nobody logged-in. It should install successfully where previously you would have received an error.



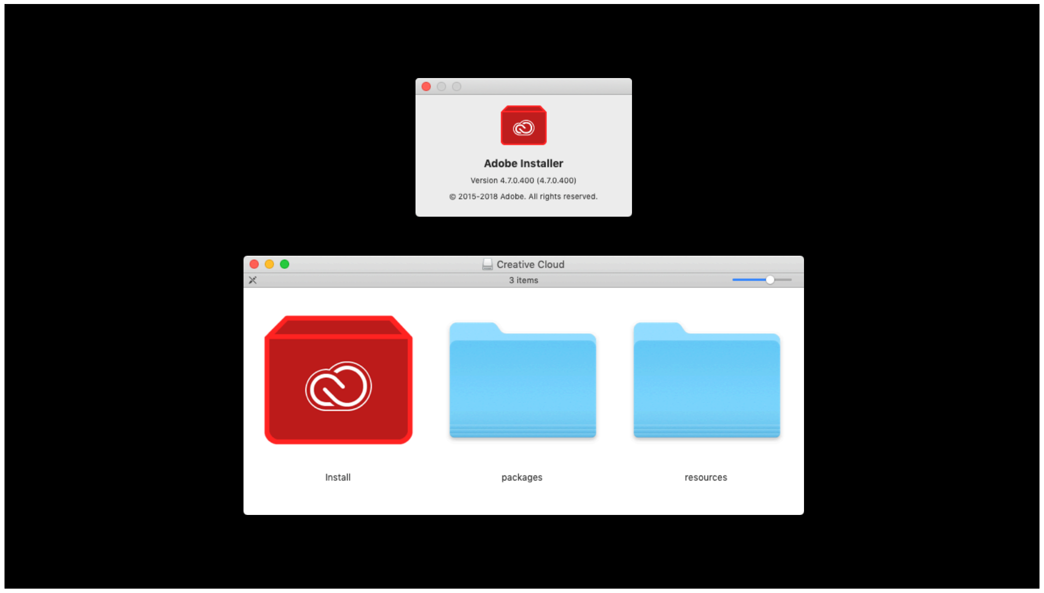
```
            <key>Processor</key>
            <string>FileCreator</string>
            <key>Arguments</key>
            <dict>
               <key>file_path</key>
               <string>%RECIPE_CACHE_DIR%/Scripts/postinstall</string>
               <key>file_mode</key>
               <string>0755</string>
               <key>file_content</key>
               <string>#!/bin/bash

PKG="${0%/*}/Install Citrix Workspace.pkg"
ERROR=0

if [[ -f "$PKG" ]]; then

        CURRENT_USER=$(/bin/ls -l /dev/console | /usr/bin/awk '{ print $3 }')

        if [[ -n "$CURRENT_USER" ]] ;then

                # get path to user's home directory
                USER_HOME=$(/usr/bin/dscl . -read "/Users/$CURRENT_USER" NFSHomeDire

                export HOME="$USER_HOME"

        else
                export HOME="/var/root"
        fi

        /usr/sbin/installer -pkg "$PKG" -target "$3"

        if [[ $? -ne 0 ]]; then
                /usr/bin/logger -t "${0##*/}" "ERROR! Installation of package $PKG fa
                ERROR=1
        fi

else
        /usr/bin/logger -t "${0##*/}" "ERROR! Package $PKG not found"
        ERROR=1
fi

exit $ERROR</string>
            </dict>
         </dict>
```

https://github.com/autopkg/rtrouton-recipes/blob/master/CitrixWorkspace/CitrixWorkspace.pkg.recipe

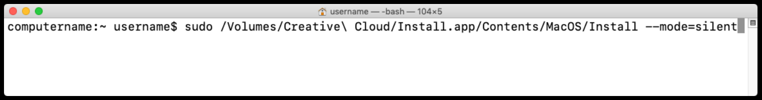
For those who want to automate this process using AutoPkg, I have written a Citrix Workspace package recipe to create a package just like the one we've been talking about. It's available on GitHub via the address shown on the screen and uses the same postinstall script that I've described.

Install software which uses a third-party installer

There are also vendors who prefer to use installers which don’t use any of Apple’s supported methods. These can be challenging, but you can still package these if you can run the provided installer from the command line. For this scenario, we’ll be working with the Adobe Creative Cloud Desktop app installer.


Adobe Installer
Version 4.7.0.400 (4.7.0.400)
© 2015-2018 Adobe. All rights reserved.
Creative Cloud
3 items
Install
packages
resources

Adobe goes their own way most of the time when it comes to installers and the Creative Cloud Desktop app for macOS is a good example. It uses an Adobe-developed installer and in no way leverages either of Apple's supported installation methods.

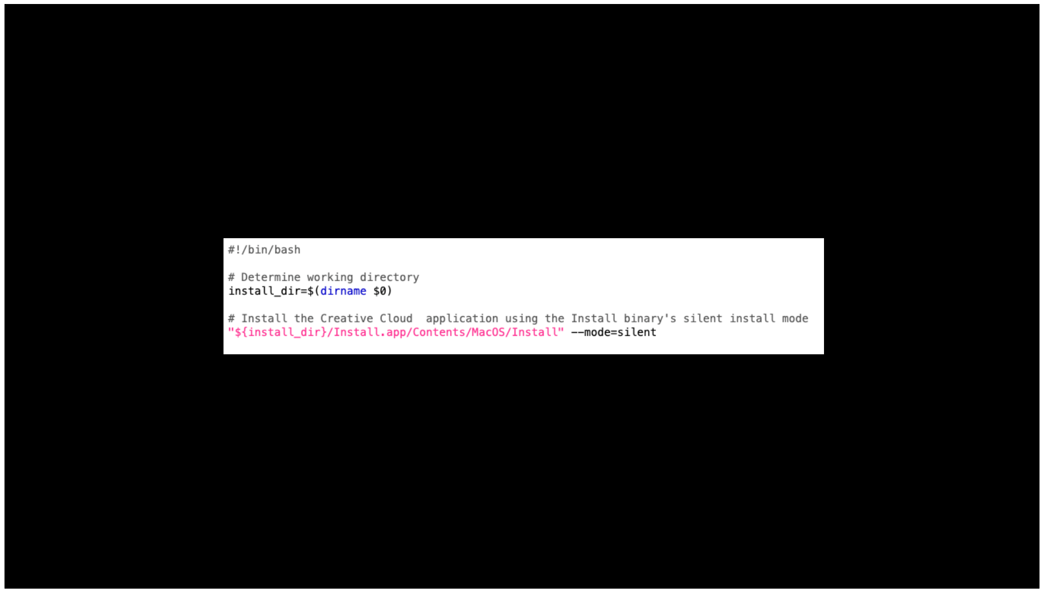


```
computername:~ username$ sudo /Volumes/Creative\ Cloud/Install.app/Contents/MacOS/Install --mode=silent
```

/path/to/Install.app/Contents/MacOS/Install --mode=silent

That said, Adobe did include a way to run a silent install from the command line.

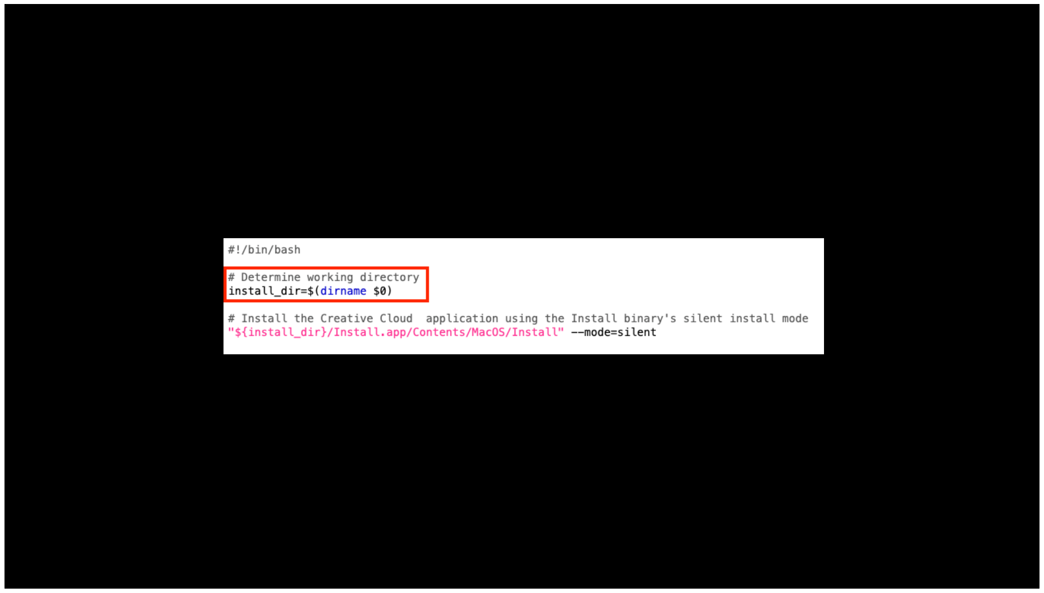
Finder
File
Edit
View
Go
Window
Help
Macintosh HD
Creative Cloud



```bash
#!/bin/bash

# Determine working directory
install_dir=$(dirname $0)

# Install the Creative Cloud  application using the Install binary's silent install mode
"${install_dir}/Install.app/Contents/MacOS/Install" --mode=silent
```

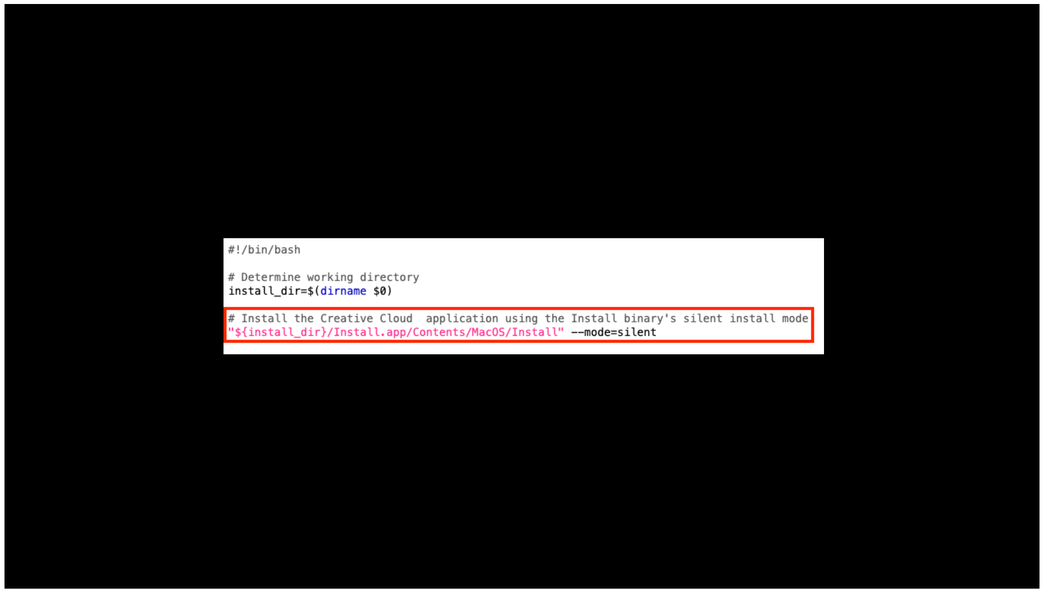
With this information, we can build a script which does two tasks:

1. Identify the directory that the script is running from. The Adobe installer will be placed in the same directory.
2. Running the Adobe installer using the silent install mode.



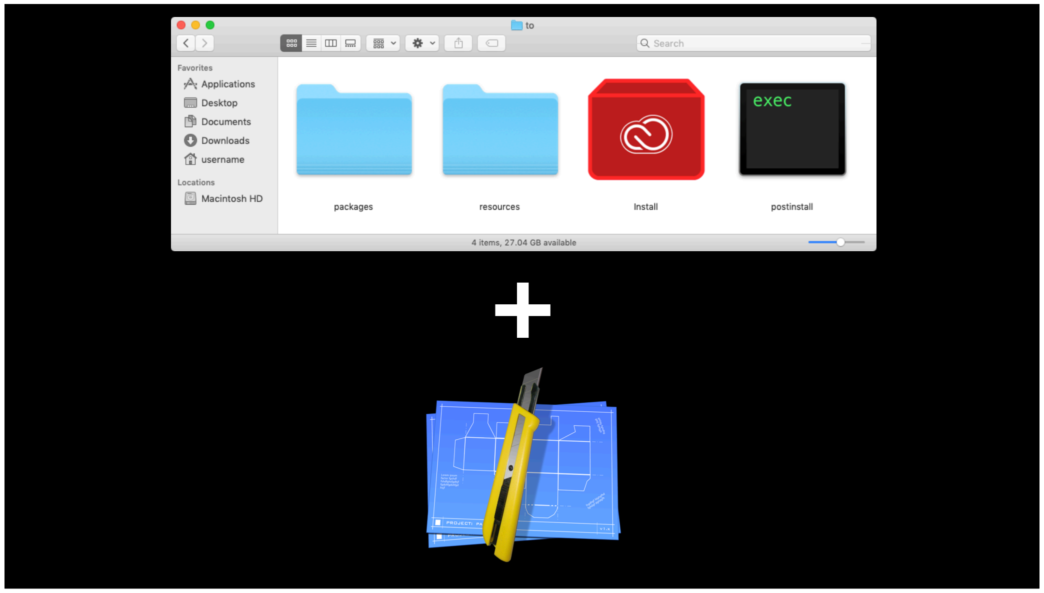
```
#!/bin/bash

# Determine working directory
install_dir=$(dirname $0)

# Install the Creative Cloud  application using the Install binary's silent install mode
"${install_dir}/Install.app/Contents/MacOS/Install" --mode=silent
```

First step is identifying the script’s location.

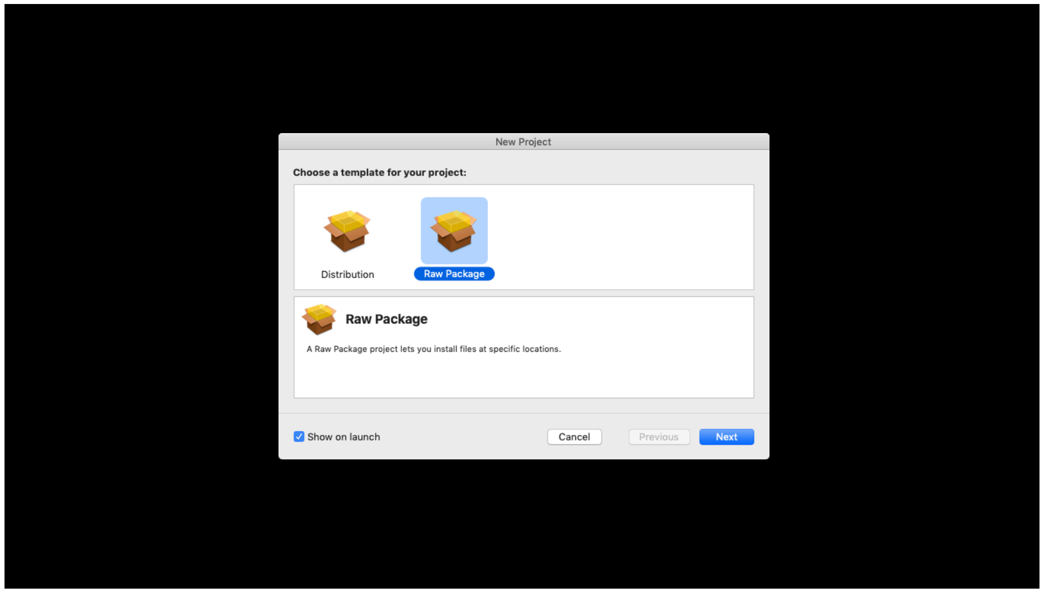


```
#!/bin/bash

# Determine working directory
install_dir=$(dirname $0)

# Install the Creative Cloud  application using the Install binary's silent install mode
"${install_dir}/Install.app/Contents/MacOS/Install" --mode=silent
```

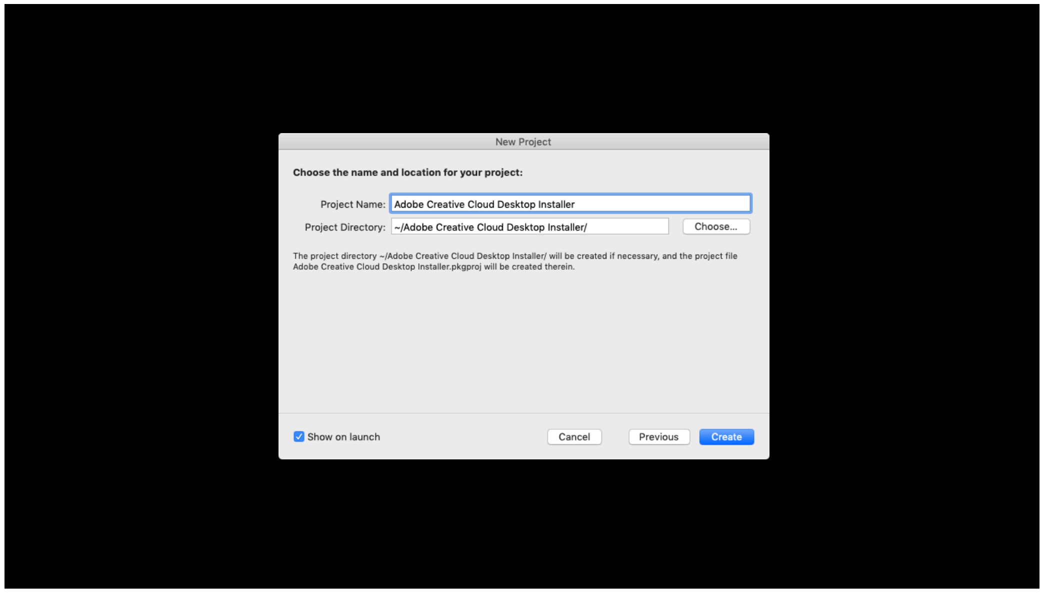
Second is running the installer, using the location information to provide the path to the installer.



Now that we have that, let’s put all of this together into a package. To do this, I’ll be using the script I just described as a postinstall script, a copy of the the latest Adobe Creative Cloud Desktop installer, the installer’s support directories and the Packages app to create the package.


New Project
Choose a template for your project:
Distribution
Raw Package
Raw Package
A Raw Package project lets you install files at specific locations.
Show on launch
Cancel
Previous
Next

First step is to open the Packages app and set up a raw package project.


New Project

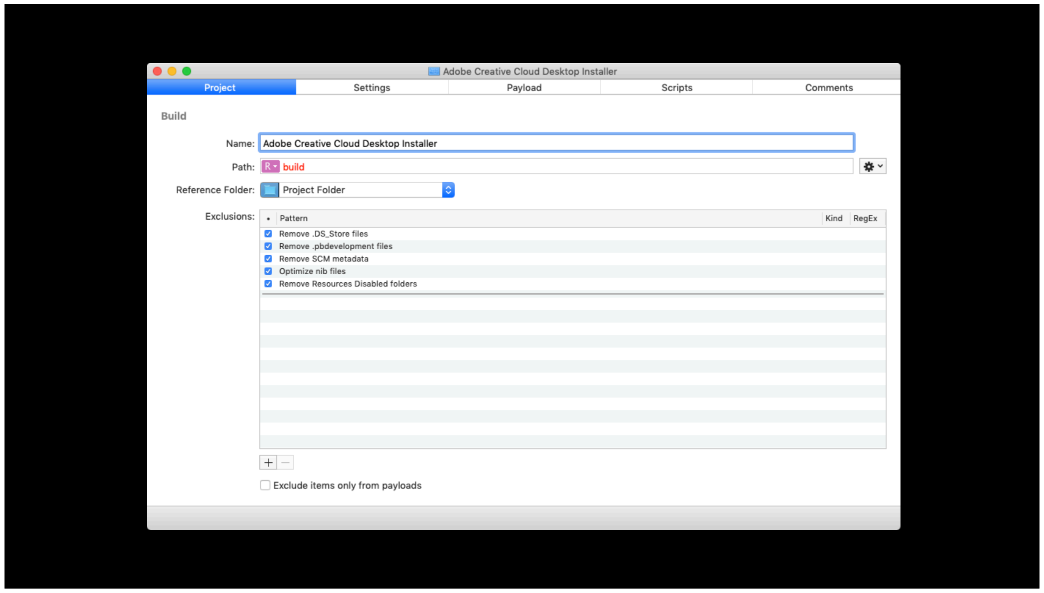
Choose the name and location for your project:

Project Name: Adobe Creative Cloud Desktop Installer

Project Directory: ~/Adobe Creative Cloud Desktop Installer/

The project directory ~/Adobe Creative Cloud Desktop Installer/ will be created if necessary, and the project file Adobe Creative Cloud Desktop Installer.pkgproj will be created therein.

Next, we name it and define where the Packages project file will be stored. In this case, I’m naming the package as Adobe Creative Cloud Desktop Installer and storing the project files in a Adobe Creative Cloud Desktop Installer directory in my home folder.



Build

Name: Adobe Creative Cloud Desktop Installer

Path: build

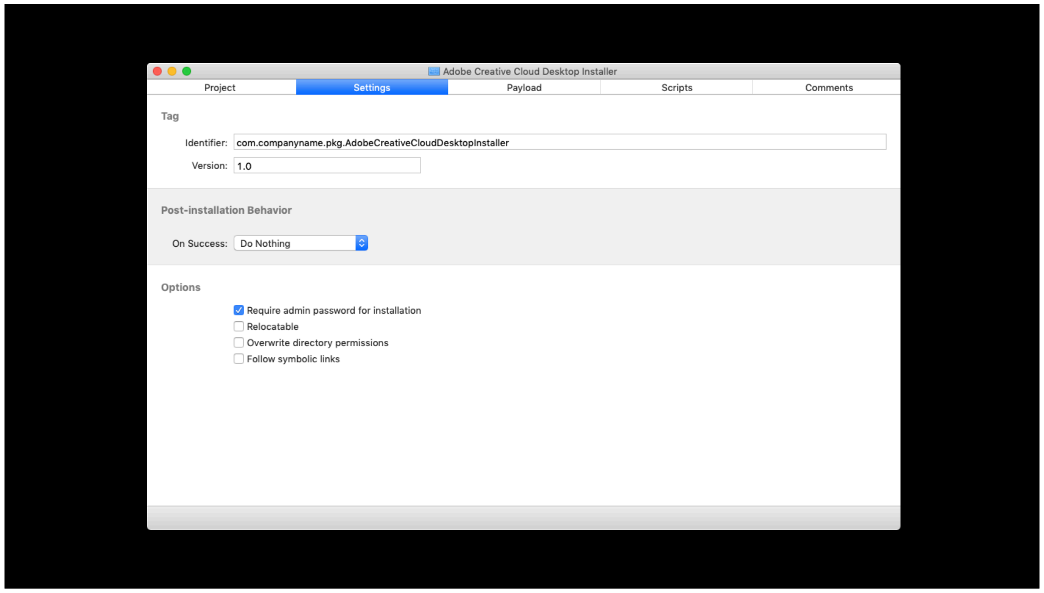
Reference Folder: Project Folder

Exclusions:

•	Pattern	Kind	RegEx
✓	Remove .DS_Store files		
✓	Remove .pbdevelopment files		
✓	Remove SCM metadata		
✓	Optimize nib files		
✓	Remove Resources Disabled folders		

☐ Exclude items only from payloads

In the Project pane, the main thing I'm checking here is the name to make sure it's right. As long as that's correct, there's no need to change anything here from the defaults.



Tag

Identifier: com.companyname.pkg.AdobeCreativeCloudDesktopInstaller

Version: 1.0

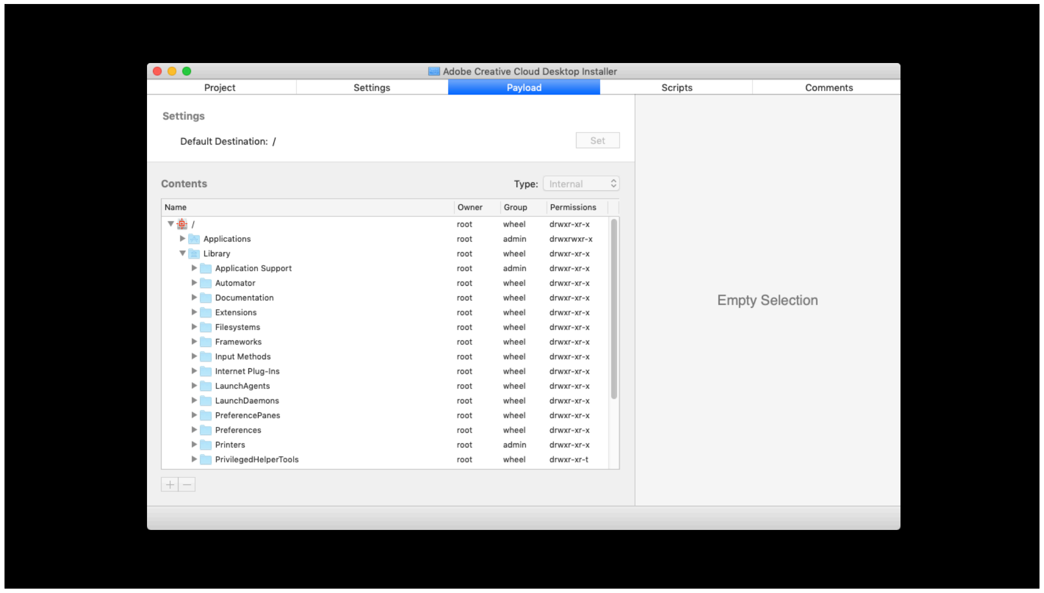
Post-installation Behavior

On Success: Do Nothing

Options

- [x] Require admin password for installation
- [ ] Relocatable
- [ ] Overwrite directory permissions
- [ ] Follow symbolic links

In the Settings pane, we want to require the admin password for installation and on successful installation, we don't want to do anything else.



Settings

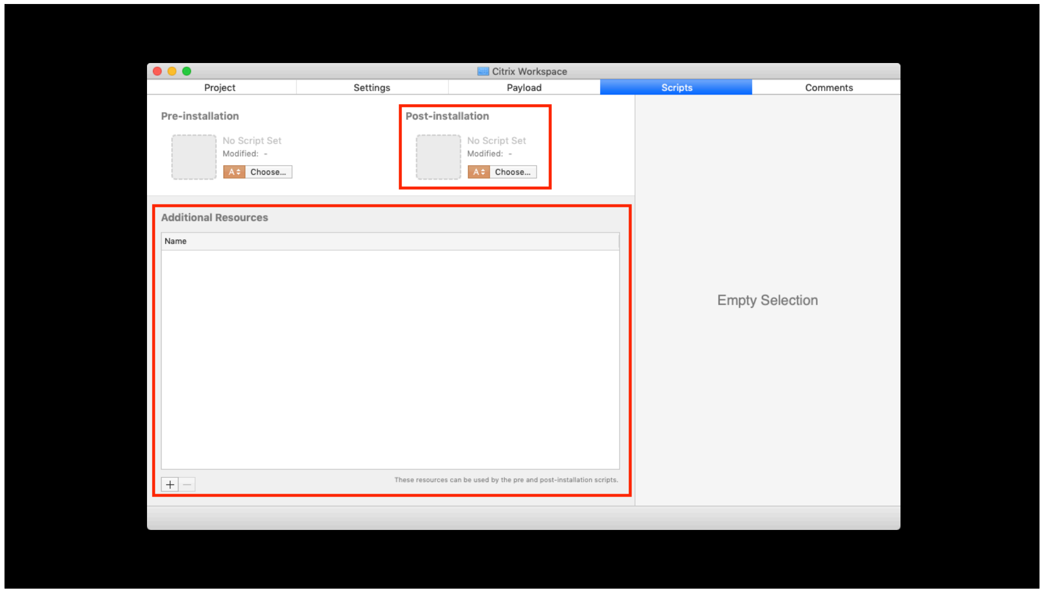
Default Destination: /

Contents

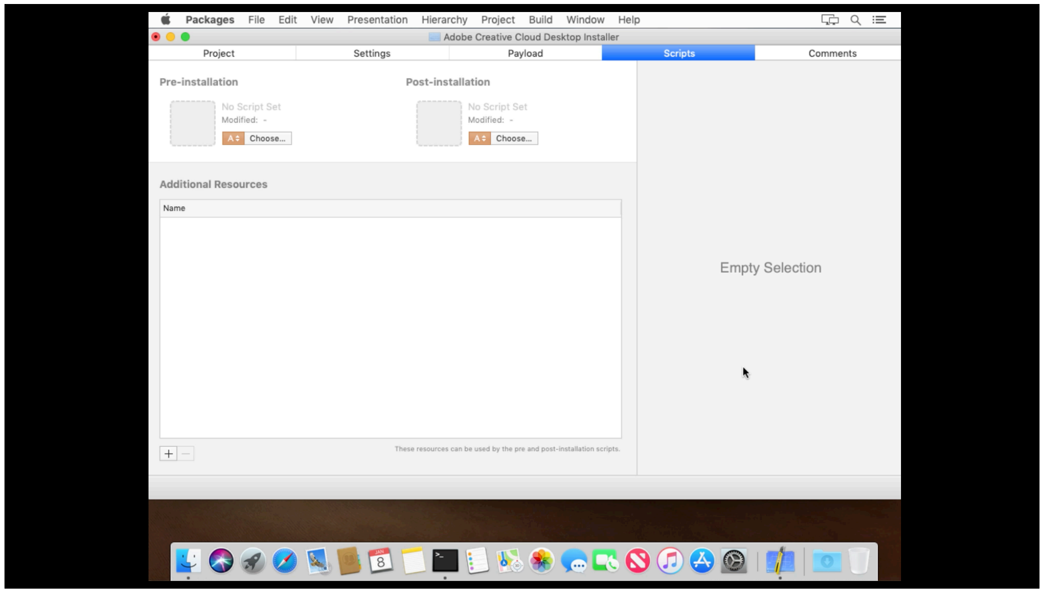
Type: Internal

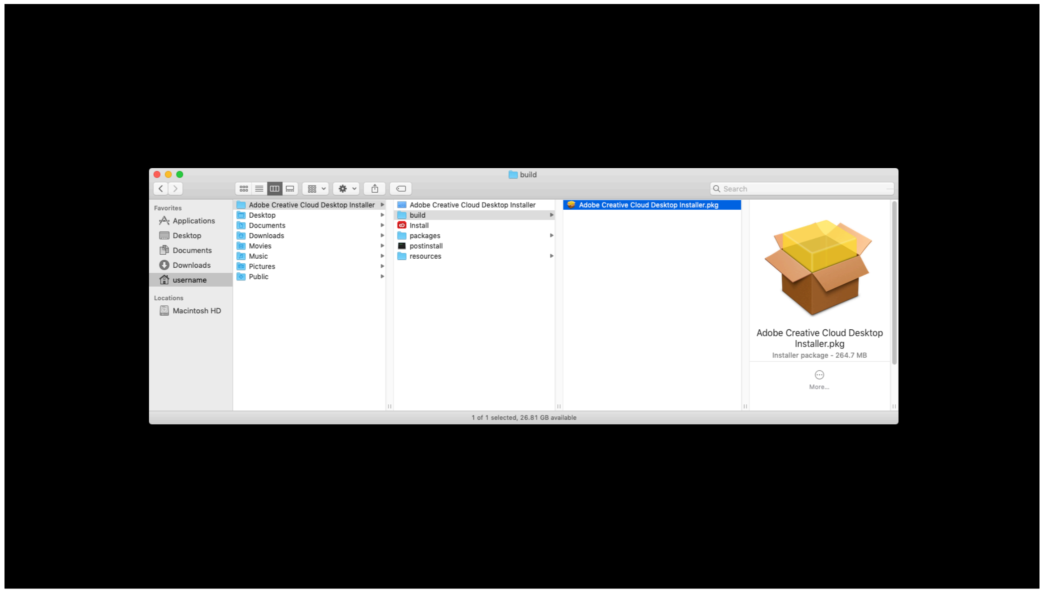
Name	Owner	Group	Permissions
/	root	wheel	drwxr-xr-x
Applications	root	admin	drwxrwxr-x
Library	root	wheel	drwxr-xr-x
Application Support	root	admin	drwxr-xr-x
Automator	root	wheel	drwxr-xr-x
Documentation	root	wheel	drwxr-xr-x
Extensions	root	wheel	drwxr-xr-x
Filesystems	root	wheel	drwxr-xr-x
Frameworks	root	wheel	drwxr-xr-x
Input Methods	root	wheel	drwxr-xr-x
Internet Plug-Ins	root	wheel	drwxr-xr-x
LaunchAgents	root	wheel	drwxr-xr-x
LaunchDaemons	root	wheel	drwxr-xr-x
PreferencePanes	root	wheel	drwxr-xr-x
Preferences	root	wheel	drwxr-xr-x
Printers	root	admin	drwxr-xr-x
PrivilegedHelperTools	root	wheel	drwxr-xr-t

In the Payload pane, we’re not changing anything from the defaults.



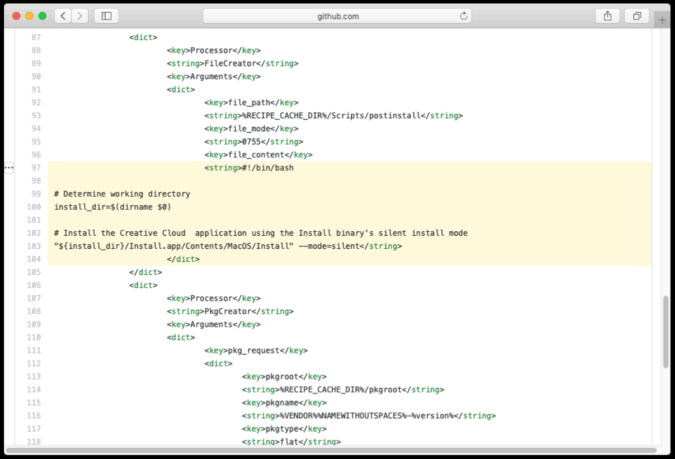
In the scripts pane, we need to add the Adobe installer and its support directories under additional resources and add the postinstall script to the post-installation section. As mentioned during the Citrix Workspace installer, placing things in additional resources means they can be referenced by pre and post installation scripts.


Packages File Edit View Presentation Hierarchy Project Build Window Help
Adobe Creative Cloud Desktop Installer
Project
Settings
Payload
Scripts
Comments
Pre-installation
No Script Set
Modified: -
Choose...
Post-installation
No Script Set
Modified: -
Choose...
Additional Resources
Name
These resources can be used by the pre and post-installation scripts.
Empty Selection


build
Search
Favorites
Applications
Desktop
Documents
Downloads
username
Locations
Macintosh HD
Adobe Creative Cloud Desktop Installer
Desktop
Documents
Downloads
Movies
Music
Pictures
Public
Adobe Creative Cloud Desktop Installer
build
Install
packages
postinstall
resources
Adobe Creative Cloud Desktop Installer.pkg
Adobe Creative Cloud Desktop Installer.pkg
Installer package - 264.7 MB
More...
1 of 1 selected, 26.81 GB available

Once the package is built, you can test it by deploying it onto a Mac that doesn't have the Adobe Creative Cloud Desktop app installed and verify that the desktop app installs correctly.

Finder
File
Edit
View
Go
Window
Help
Macintosh HD
Adobe Creative Cloud D...taller.pkg



```
                <dict>
                        <key>Processor</key>
                        <string>FileCreator</string>
                        <key>Arguments</key>
                        <dict>
                                <key>file_path</key>
                                <string>%RECIPE_CACHE_DIR%/Scripts/postinstall</string>
                                <key>file_mode</key>
                                <string>0755</string>
                                <key>file_content</key>
                                <string>#!/bin/bash

# Determine working directory
install_dir=$(dirname $0)

# Install the Creative Cloud  application using the Install binary's silent install mode
"${install_dir}/Install.app/Contents/MacOS/Install" --mode=silent</string>
                        </dict>
                </dict>
                <dict>
                        <key>Processor</key>
                        <string>PkgCreator</string>
                        <key>Arguments</key>
                        <dict>
                                <key>pkg_request</key>
                                <dict>
                                        <key>pkgroot</key>
                                        <string>%RECIPE_CACHE_DIR%/pkgroot</string>
                                        <key>pkgname</key>
                                        <string>%VENDOR%%NAMEWITHOUTSPACES%-%version%</string>
                                        <key>pkgtype</key>
                                        <string>flat</string>
```

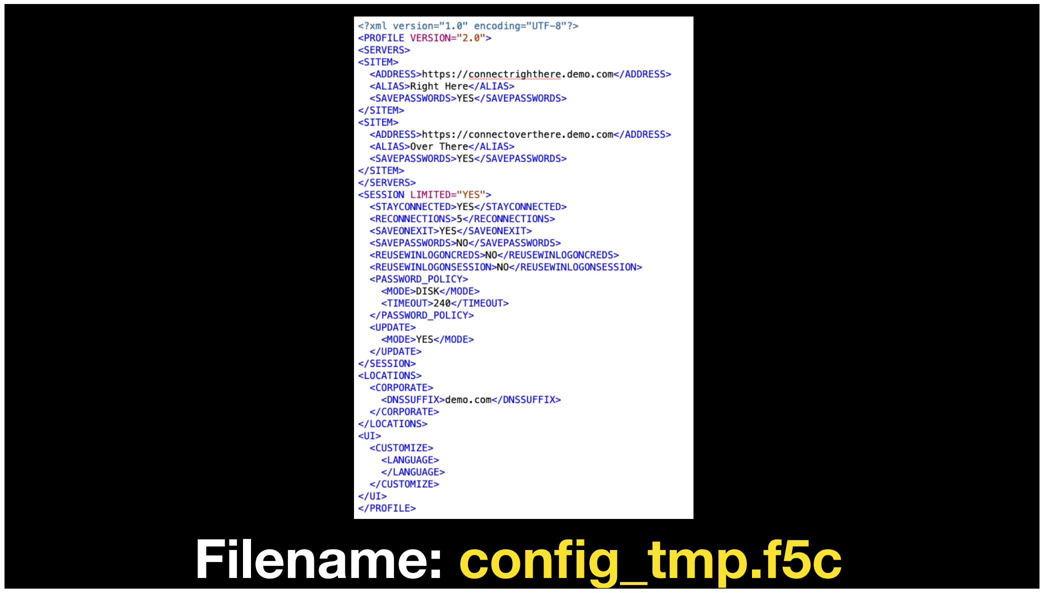
https://github.com/autopkg/rtrouton-recipes/blob/master/AdobeCreativeCloud/AdobeCreativeCloudInstaller.pkg.recipe

For those who want to automate this process using AutoPkg, I have written a Adobe Creative Cloud Desktop app package recipe to create a package just like the one we've been talking about. It's available on GitHub via the address shown on the screen and uses the same postinstall script that I've described.

Deploy Custom Configurations

You can also use scripts to help deploy a custom software configuration. In many cases, vendors will support placing a configuration file in the same directory as their installer.

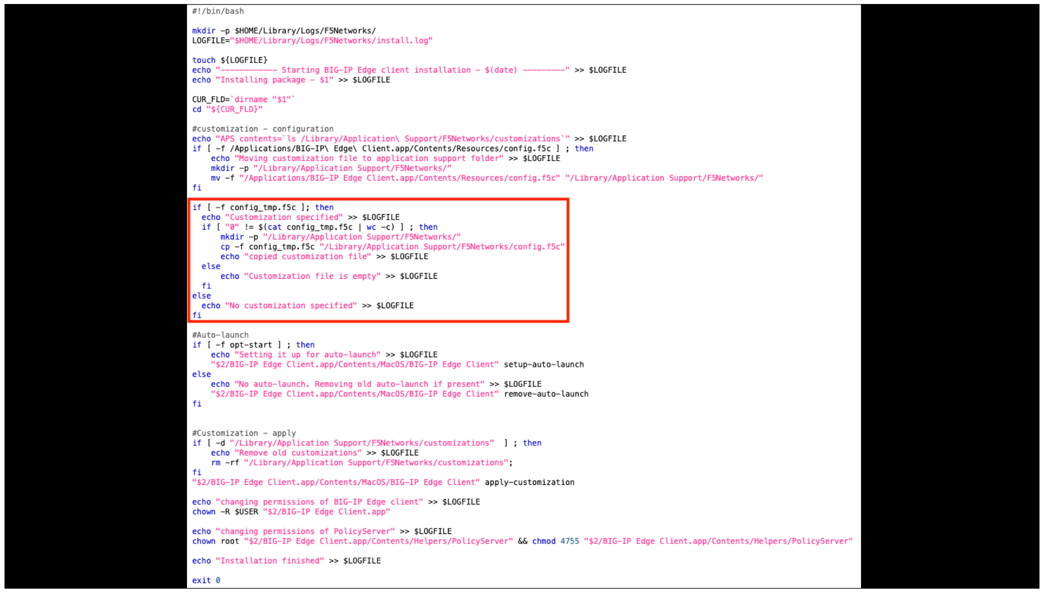
An example of this is F5 Network’s VPN client. When the F5 installer detects a file with a certain name in the same directory as the installer, it uses the contents of that file to install the VPN configuration along with the software.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<PROFILE VERSION="2.0">
<SERVERS>
<SITEM>
  <ADDRESS>https://connectrighthere.demo.com</ADDRESS>
  <ALIAS>Right Here</ALIAS>
  <SAVEPASSWORDS>YES</SAVEPASSWORDS>
</SITEM>
<SITEM>
  <ADDRESS>https://connectoverthere.demo.com</ADDRESS>
  <ALIAS>Over There</ALIAS>
  <SAVEPASSWORDS>YES</SAVEPASSWORDS>
</SITEM>
</SERVERS>
<SESSION LIMITED="YES">
  <STAYCONNECTED>YES</STAYCONNECTED>
  <RECONNECTIONS>5</RECONNECTIONS>
  <SAVEONEXIT>YES</SAVEONEXIT>
  <SAVEPASSWORDS>NO</SAVEPASSWORDS>
  <REUSEWINLOGONCREDS>NO</REUSEWINLOGONCREDS>
  <REUSEWINLOGONSESSION>NO</REUSEWINLOGONSESSION>
  <PASSWORD_POLICY>
    <MODE>DISK</MODE>
    <TIMEOUT>240</TIMEOUT>
  </PASSWORD_POLICY>
  <UPDATE>
    <MODE>YES</MODE>
  </UPDATE>
</SESSION>
<LOCATIONS>
  <CORPORATE>
    <DNSSUFFIX>demo.com</DNSSUFFIX>
  </CORPORATE>
</LOCATIONS>
<UI>
  <CUSTOMIZE>
    <LANGUAGE>
    </LANGUAGE>
  </CUSTOMIZE>
</UI>
</PROFILE>
```

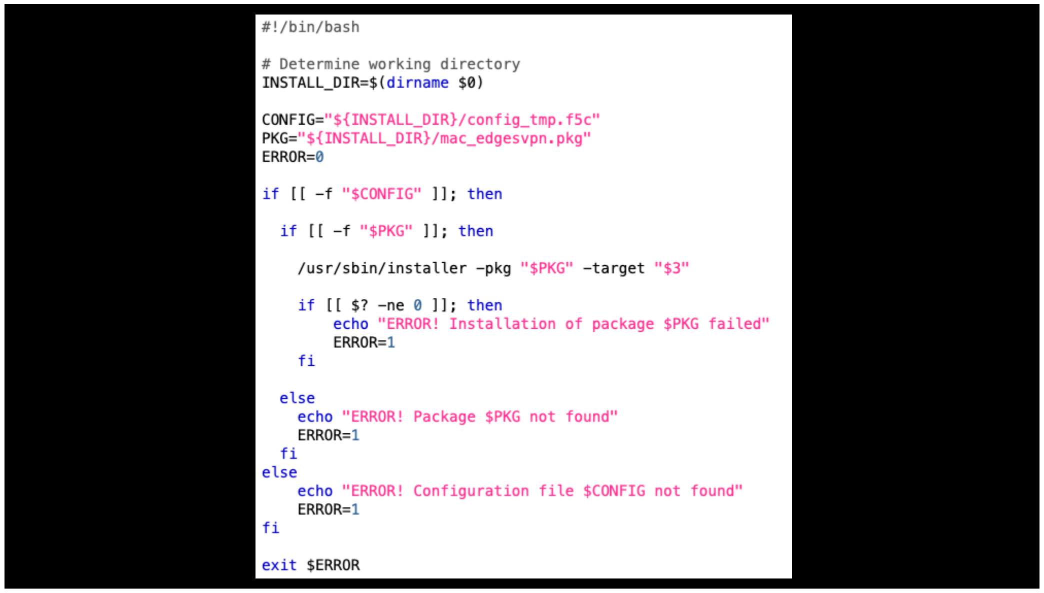
Filename: config_tmp.f5c

The file in question is an XML document configured and named as shown on the screen. You normally shouldn't have to worry about creating this file, as your VPN admin should be able to provide it to you.



```
#!/bin/bash

mkdir -p $HOME/Library/Logs/F5Networks/
LOGFILE="$HOME/Library/Logs/F5Networks/install.log"

touch ${LOGFILE}
echo "------------ Starting BIG-IP Edge client installation - $(date) ---------" >> $LOGFILE
echo "Installing package - $1" >> $LOGFILE

CUR_FLD=`dirname "$1"`
cd "${CUR_FLD}"

#customization - configuration
echo "APS contents=`ls /Library/Application\ Support/F5Networks/customizations`" >> $LOGFILE
if [ -f /Applications/BIG-IP\ Edge\ Client.app/Contents/Resources/config.f5c ] ; then
    echo "Moving customization file to application support folder" >> $LOGFILE
    mkdir -p "/Library/Application Support/F5Networks/"
    mv -f "/Applications/BIG-IP Edge Client.app/Contents/Resources/config.f5c" "/Library/Application Support/F5Networks/"
fi

if [ -f config_tmp.f5c ]; then
  echo "Customization specified" >> $LOGFILE
  if [ "0" != $(cat config_tmp.f5c | wc -c) ] ; then
      mkdir -p "/Library/Application Support/F5Networks/"
      cp -f config_tmp.f5c "/Library/Application Support/F5Networks/config.f5c"
      echo "copied customization file" >> $LOGFILE
  else
      echo "Customization file is empty" >> $LOGFILE
  fi
else
  echo "No customization specified" >> $LOGFILE
fi

#Auto-launch
if [ -f opt-start ] ; then
    echo "Setting it up for auto-launch" >> $LOGFILE
    "$2/BIG-IP Edge Client.app/Contents/MacOS/BIG-IP Edge Client" setup-auto-launch
else
    echo "No auto-launch. Removing old auto-launch if present" >> $LOGFILE
    "$2/BIG-IP Edge Client.app/Contents/MacOS/BIG-IP Edge Client" remove-auto-launch
fi


#Customization - apply
if [ -d "/Library/Application Support/F5Networks/customizations"  ] ; then
    echo "Remove old customizations" >> $LOGFILE
    rm -rf "/Library/Application Support/F5Networks/customizations";
fi
"$2/BIG-IP Edge Client.app/Contents/MacOS/BIG-IP Edge Client" apply-customization

echo "changing permissions of BIG-IP Edge client" >> $LOGFILE
chown -R $USER "$2/BIG-IP Edge Client.app"

echo "changing permissions of PolicyServer" >> $LOGFILE
chown root "$2/BIG-IP Edge Client.app/Contents/Helpers/PolicyServer" && chmod 4755 "$2/BIG-IP Edge Client.app/Contents/Helpers/PolicyServer"

echo "Installation finished" >> $LOGFILE

exit 0
```

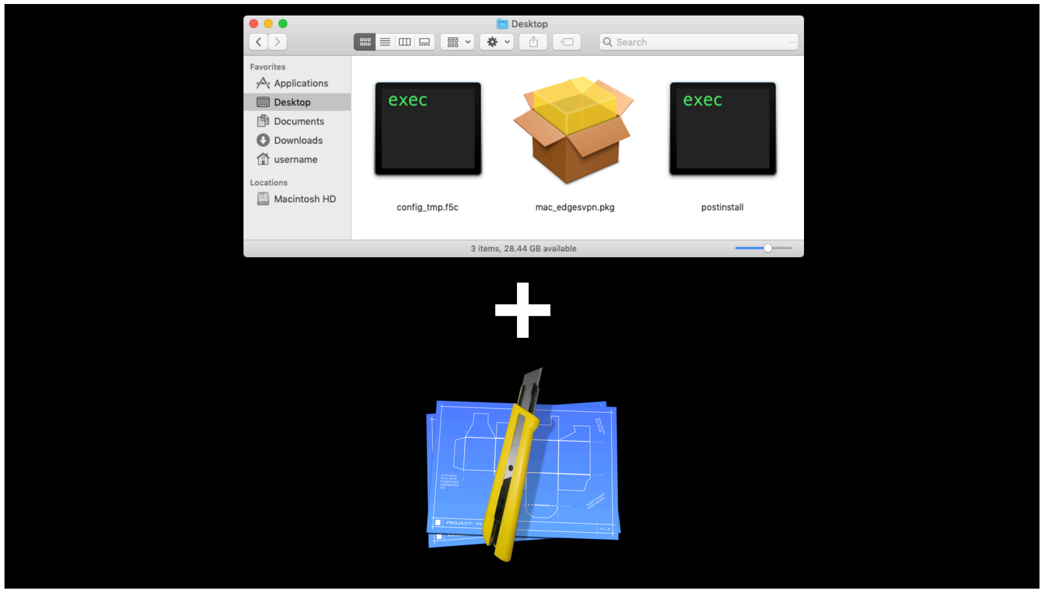
A check for a file with the specified name is included as part of a postinstall script included with the F5 VPN installer. The script assumes that the installer and configuration file are located in the same directory. If the configuration file is found, the installer copies it into Library Application Support F5Networks so that the VPN software can detect and use it.



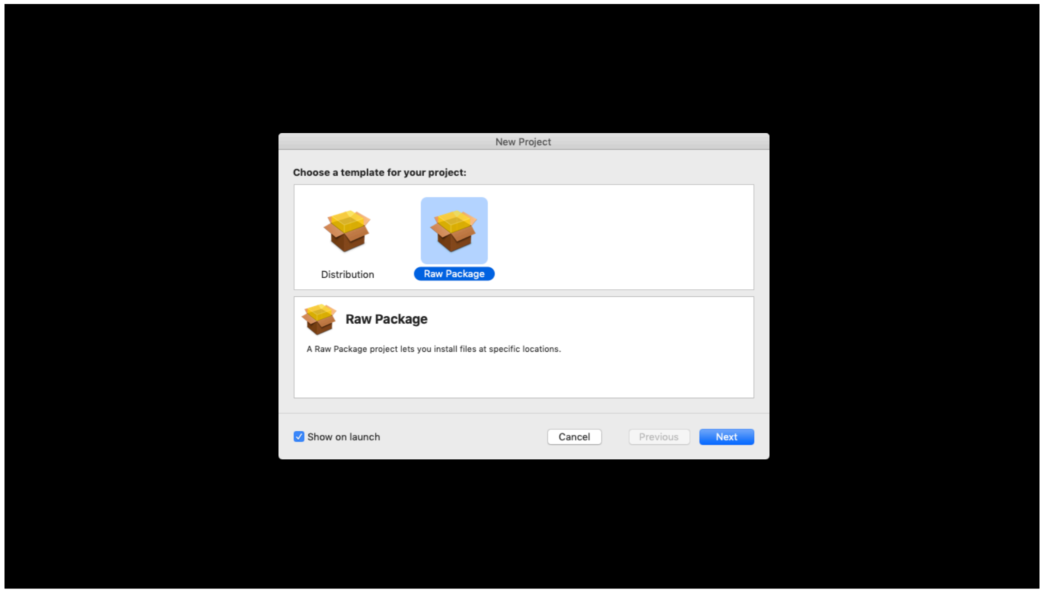
```
#!/bin/bash

# Determine working directory
INSTALL_DIR=$(dirname $0)

CONFIG="${INSTALL_DIR}/config_tmp.f5c"
PKG="${INSTALL_DIR}/mac_edgesvpn.pkg"
ERROR=0

if [[ -f "$CONFIG" ]]; then

  if [[ -f "$PKG" ]]; then

    /usr/sbin/installer -pkg "$PKG" -target "$3"

    if [[ $? -ne 0 ]]; then
        echo "ERROR! Installation of package $PKG failed"
        ERROR=1
    fi

  else
    echo "ERROR! Package $PKG not found"
    ERROR=1
  fi
else
    echo "ERROR! Configuration file $CONFIG not found"
    ERROR=1
fi

exit $ERROR
```

Once we have both the configuration file and the VPN installer, we can write a script that does three tasks:

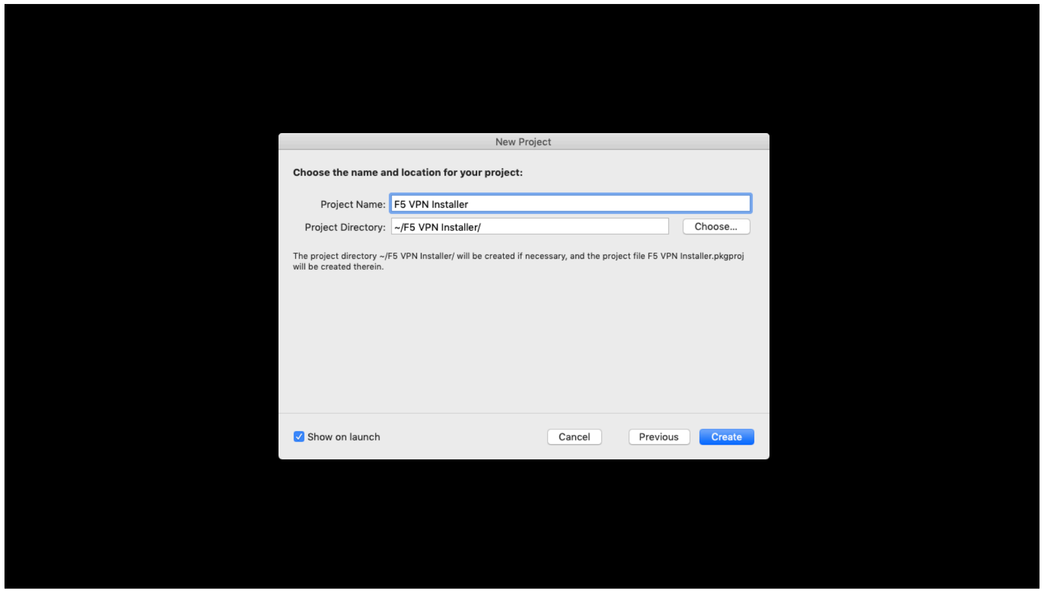
1. Verifying that the configuration file is available at a location defined by the script.
2. Verifying that the VPN installer is available at the same location.
3. Running the VPN installer and verifying it ran successfully.



Now that we have that, let's put all of this together into a package. To do this, I'll be using the script I just described as a postinstall script, a copy of the the latest F5 VPN installer, the configuration file and the Packages app to create the package.


New Project
Choose a template for your project:
Distribution
Raw Package
Raw Package
A Raw Package project lets you install files at specific locations.
Show on launch
Cancel
Previous
Next

The process is going to be pretty much identical to our previous examples. First step is to open the Packages app and set up a raw package project.



Choose the name and location for your project:

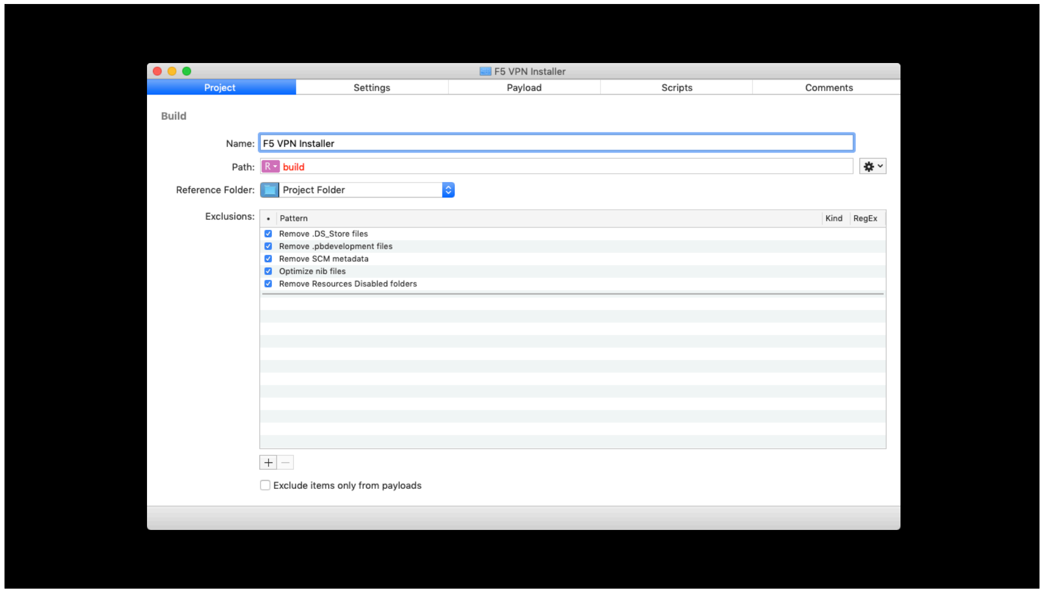
Project Name: F5 VPN Installer

Project Directory: ~/F5 VPN Installer/

The project directory ~/F5 VPN Installer/ will be created if necessary, and the project file F5 VPN Installer.pkgproj will be created therein.

☑ Show on launch

Next, we name it and define where the Packages project file will be stored. In this case, I’m naming the package as F5 VPN Installer and storing the project files in a F5 VPN Installer directory in my home folder.


F5 VPN Installer

Project | Settings | Payload | Scripts | Comments

Build

Name: F5 VPN Installer

Path: build

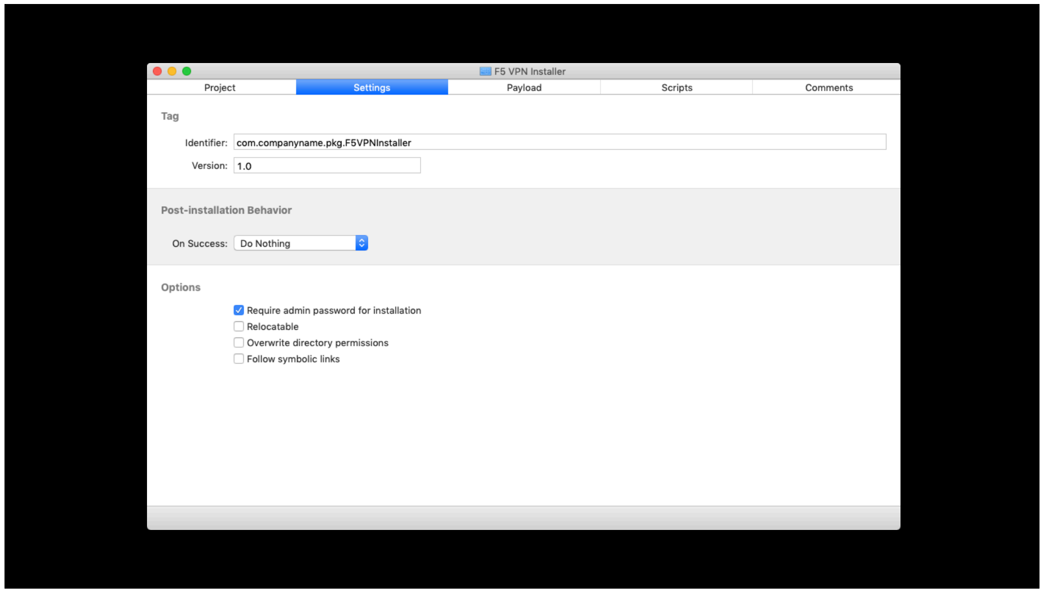
Reference Folder: Project Folder

Exclusions:

•	Pattern	Kind	RegEx
✓	Remove .DS_Store files		
✓	Remove .pbdevelopment files		
✓	Remove SCM metadata		
✓	Optimize nib files		
✓	Remove Resources Disabled folders		

☐ Exclude items only from payloads

In the Project pane, the main thing I'm checking here is the name to make sure it's right. As long as that's correct, there's no need to change anything here from the defaults.



Tag

Identifier: com.companyname.pkg.F5VPNInstaller

Version: 1.0

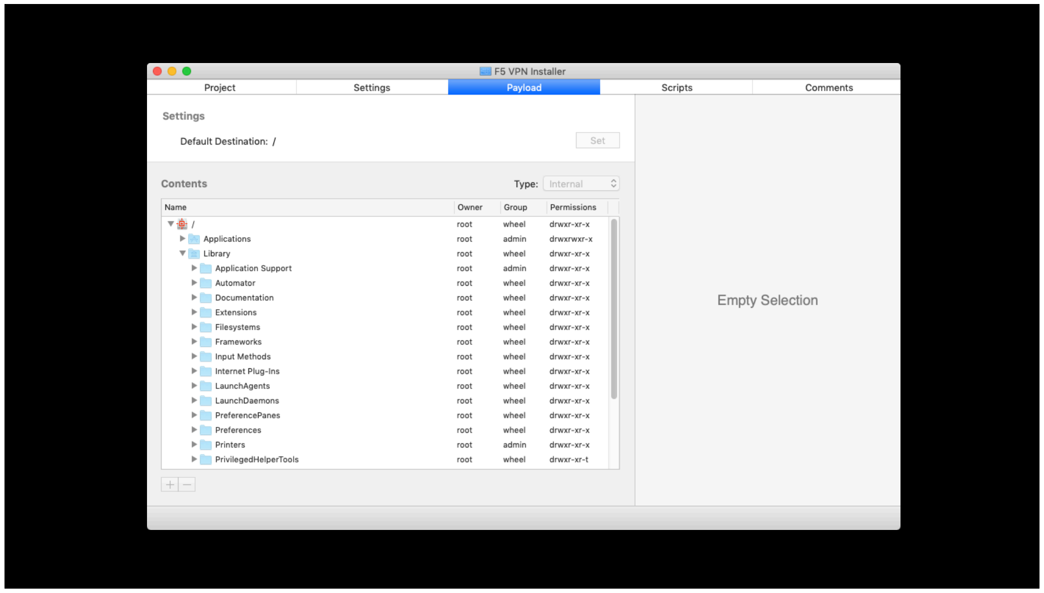
Post-installation Behavior

On Success: Do Nothing

Options

- [x] Require admin password for installation
- [ ] Relocatable
- [ ] Overwrite directory permissions
- [ ] Follow symbolic links

In the Settings pane, we want to require the admin password for installation and on successful installation, we don't want to do anything else.



Settings

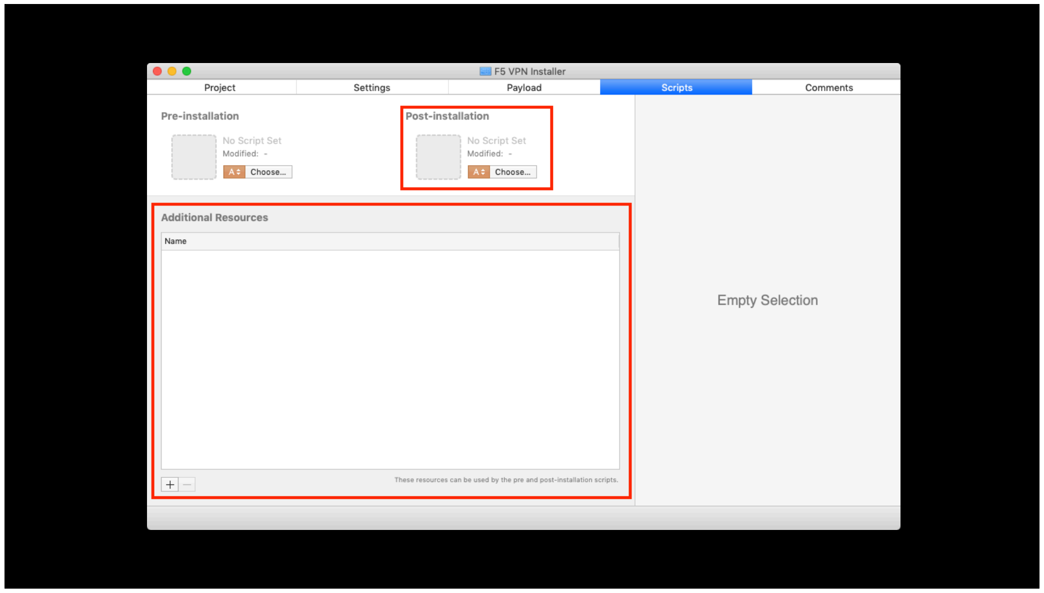
Default Destination: /

Contents

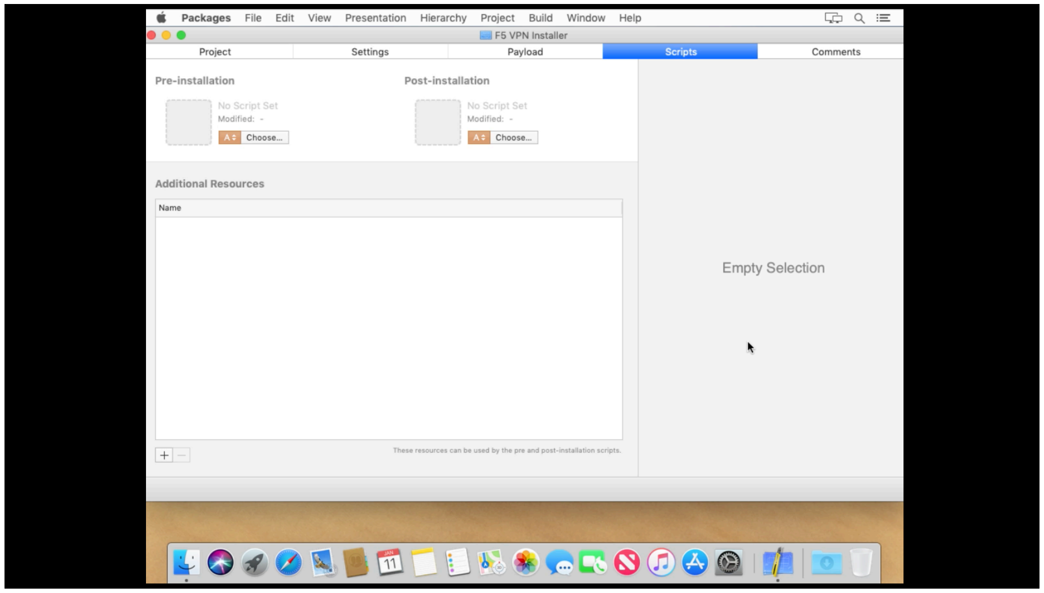
Type: Internal

Name	Owner	Group	Permissions
/	root	wheel	drwxr-xr-x
Applications	root	admin	drwxrwxr-x
Library	root	wheel	drwxr-xr-x
Application Support	root	admin	drwxr-xr-x
Automator	root	wheel	drwxr-xr-x
Documentation	root	wheel	drwxr-xr-x
Extensions	root	wheel	drwxr-xr-x
Filesystems	root	wheel	drwxr-xr-x
Frameworks	root	wheel	drwxr-xr-x
Input Methods	root	wheel	drwxr-xr-x
Internet Plug-Ins	root	wheel	drwxr-xr-x
LaunchAgents	root	wheel	drwxr-xr-x
LaunchDaemons	root	wheel	drwxr-xr-x
PreferencePanes	root	wheel	drwxr-xr-x
Preferences	root	wheel	drwxr-xr-x
Printers	root	admin	drwxr-xr-x
PrivilegedHelperTools	root	wheel	drwxr-xr-t

In the Payload pane, we're not changing anything from the defaults.



In the scripts pane, we need to add the F5 installer and the configuration file under additional resources and add the postinstall script to the post-installation section. As mentioned during the previous examples, placing things in additional resources means they can be referenced by pre and post installation scripts.



Packages File Edit View Presentation Hierarchy Project Build Window Help

F5 VPN Installer

Project | Settings | Payload | Scripts | Comments

Pre-installation
No Script Set
Modified: -
Choose...

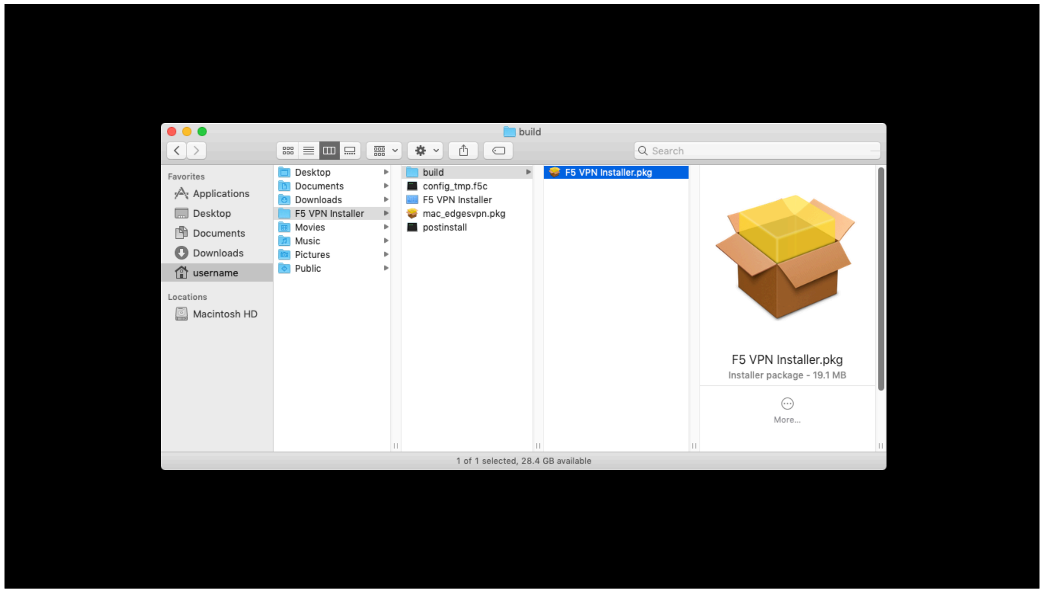
Post-installation
No Script Set
Modified: -
Choose...

Additional Resources

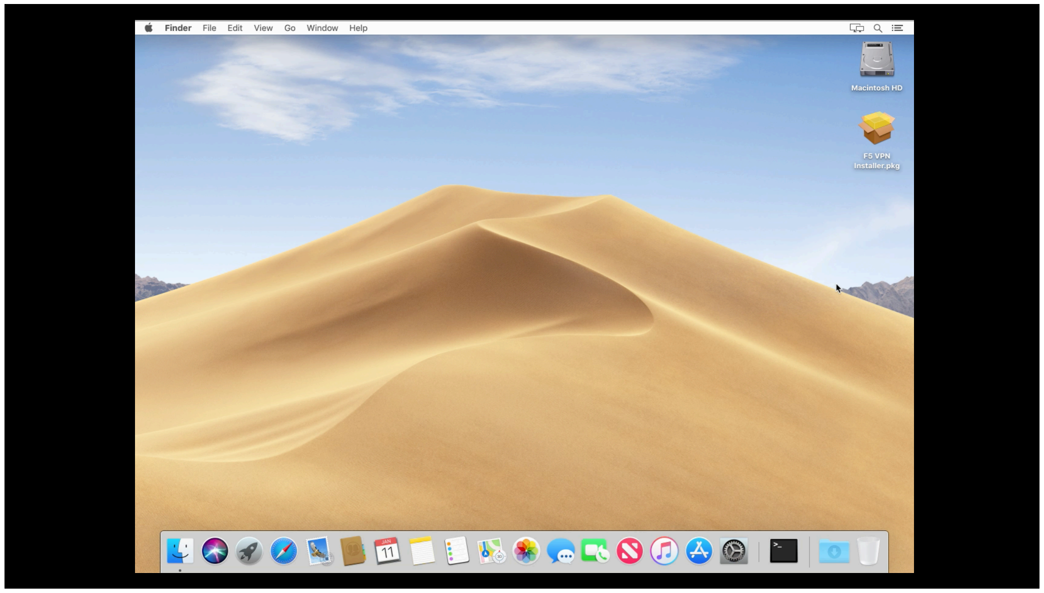
Name

These resources can be used by the pre and post-installation scripts.

Empty Selection



Once the package is built, you can test it by deploying it onto a Mac that doesn't have the F5 VPN installed and verify that the VPN installs correctly and is configured with the desired setup.


Finder
File
Edit
View
Go
Window
Help
Macintosh HD
F5 VPN Installer.pkg

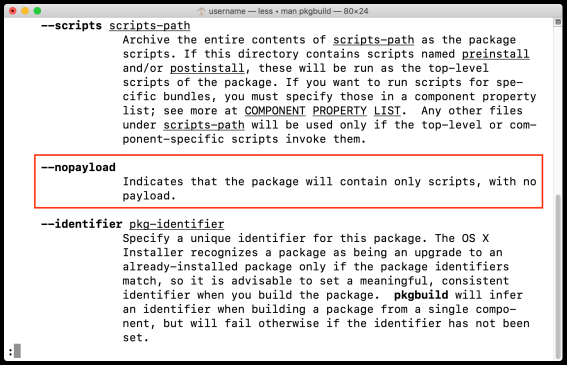
Run scripts without installing files

The final area I want to talk about is one of my favorites, where you can use an installer package as a delivery mechanism for scripts. No files get installed in this case because the only thing in the installer package is the script.

Payload-free packages

Payload-free packages is Apple's term to describe installer packages that install no files and which have been built only to run scripts. As with other installer packages, there are two kinds of payload-free installer packages, bundle-style and flat.

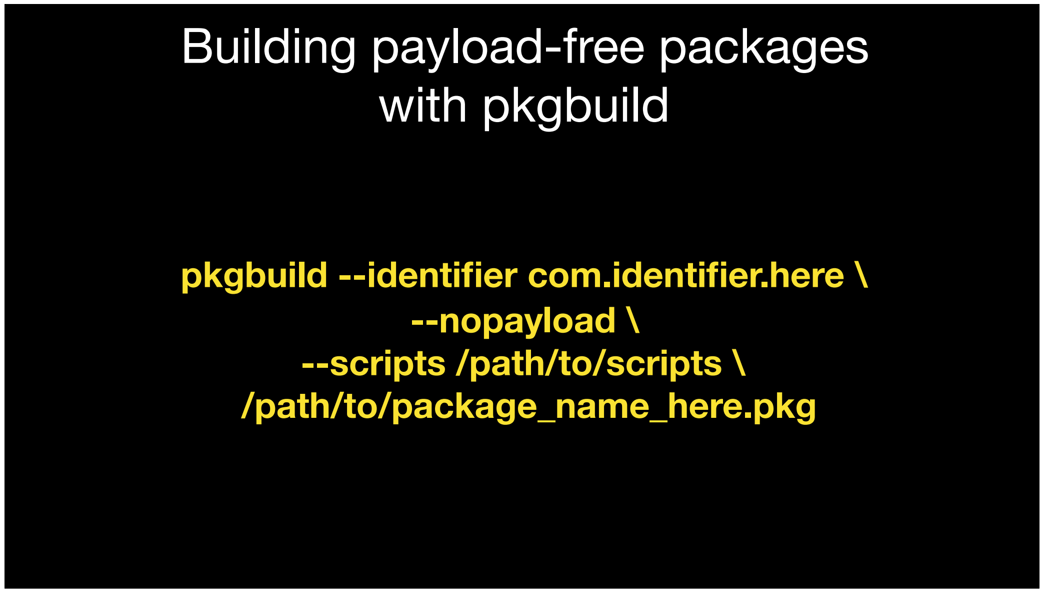
Building payload-free packages with pkgbuild



```
username — less • man pkgbuild — 80×24
     --scripts scripts-path
                 Archive the entire contents of scripts-path as the package
                 scripts. If this directory contains scripts named preinstall
                 and/or postinstall, these will be run as the top-level
                 scripts of the package. If you want to run scripts for spe-
                 cific bundles, you must specify those in a component property
                 list; see more at COMPONENT PROPERTY LIST.  Any other files
                 under scripts-path will be used only if the top-level or com-
                 ponent-specific scripts invoke them.

     --nopayload
                 Indicates that the package will contain only scripts, with no
                 payload.

     --identifier pkg-identifier
                 Specify a unique identifier for this package. The OS X
                 Installer recognizes a package as being an upgrade to an
                 already-installed package only if the package identifiers
                 match, so it is advisable to set a meaningful, consistent
                 identifier when you build the package.  pkgbuild will infer
                 an identifier when building a package from a single compo-
                 nent, but will fail otherwise if the identifier has not been
                 set.
:
```

http://www.manpagez.com/man/1/pkgbuild/

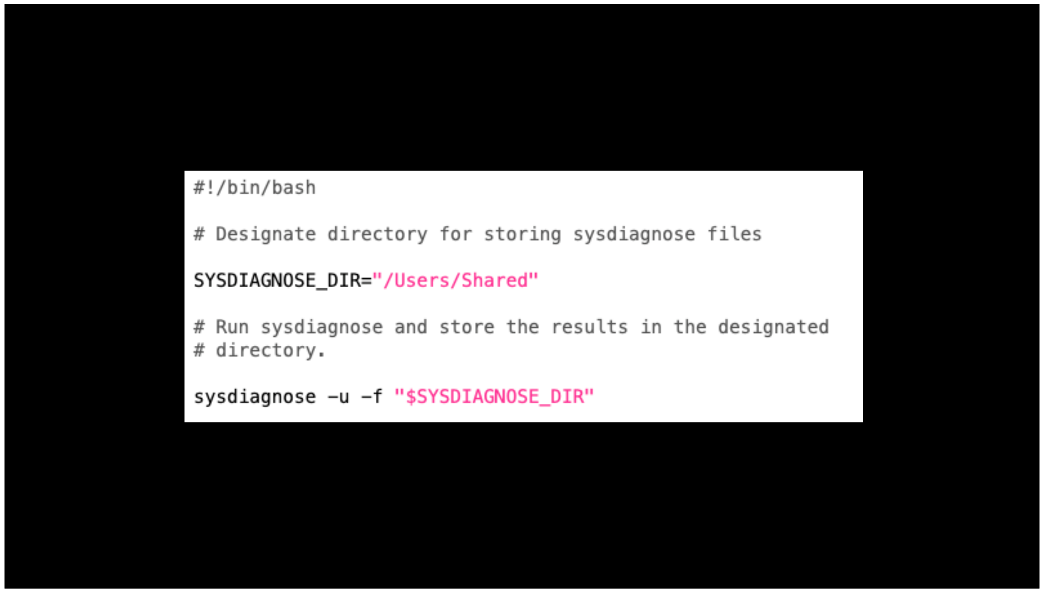
Apple has built support into its command line pkgbuild tool, which is used to build flat installer packages.



Building payload-free packages with pkgbuild

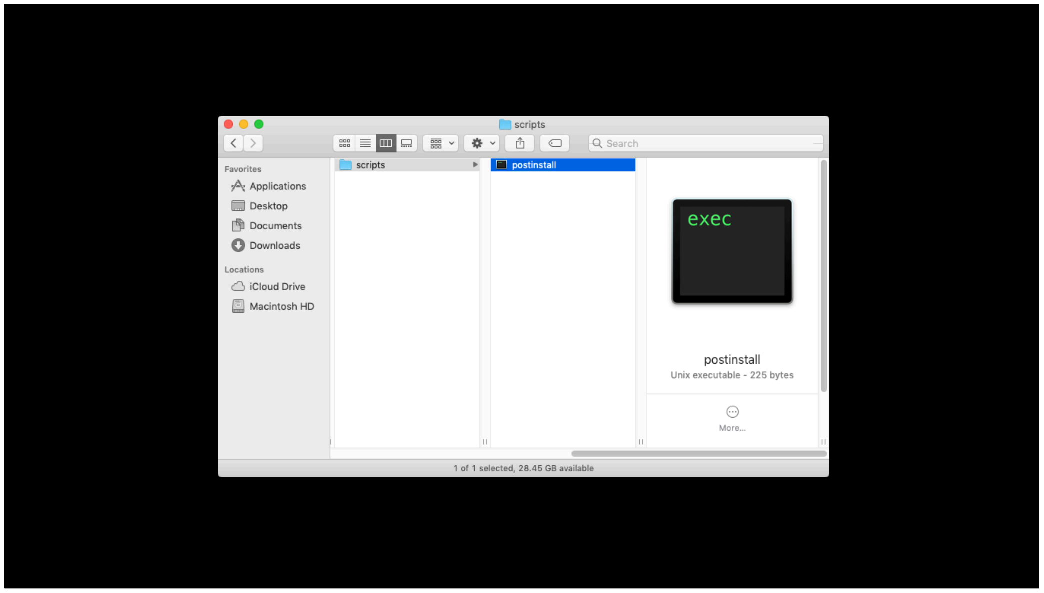
```
pkgbuild --identifier com.identifier.here \
--nopayload \
--scripts /path/to/scripts \
/path/to/package_name_here.pkg
```

The no payload option tells pkgbuild that the package being built will contain only scripts



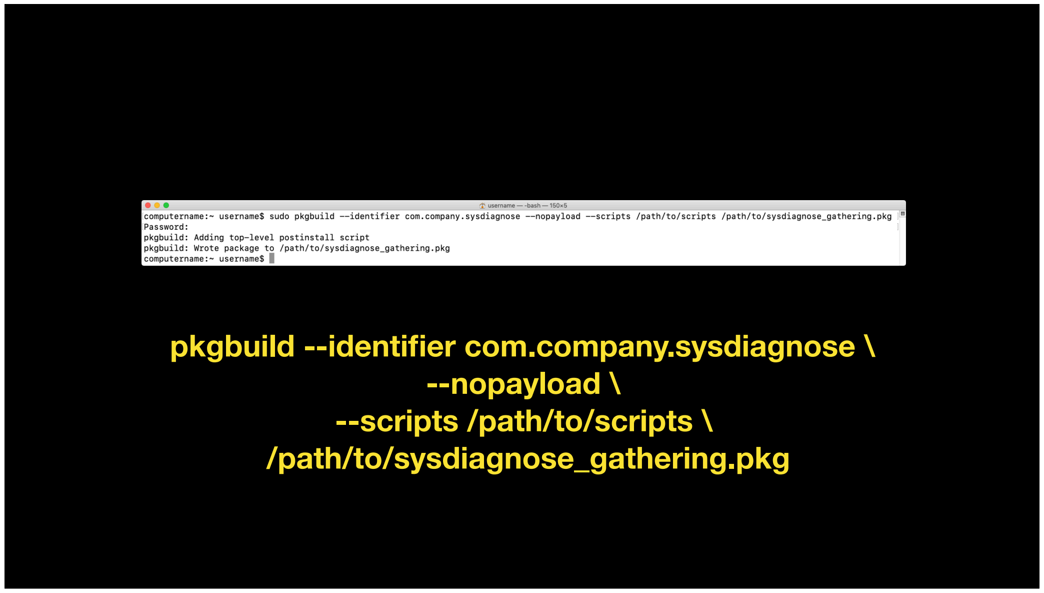
```
#!/bin/bash

# Designate directory for storing sysdiagnose files

SYSDIAGNOSE_DIR="/Users/Shared"

# Run sysdiagnose and store the results in the designated
# directory.

sysdiagnose -u -f "$SYSDIAGNOSE_DIR"
```

Let’s take a look at how this works by turning the following script into a payload-free package.



We’ll start by saving the script inside a directory named scripts as an executable file named postinstall. Not postinstall dot sh, just postinstall. The name must be right or Installer won’t recognize it.

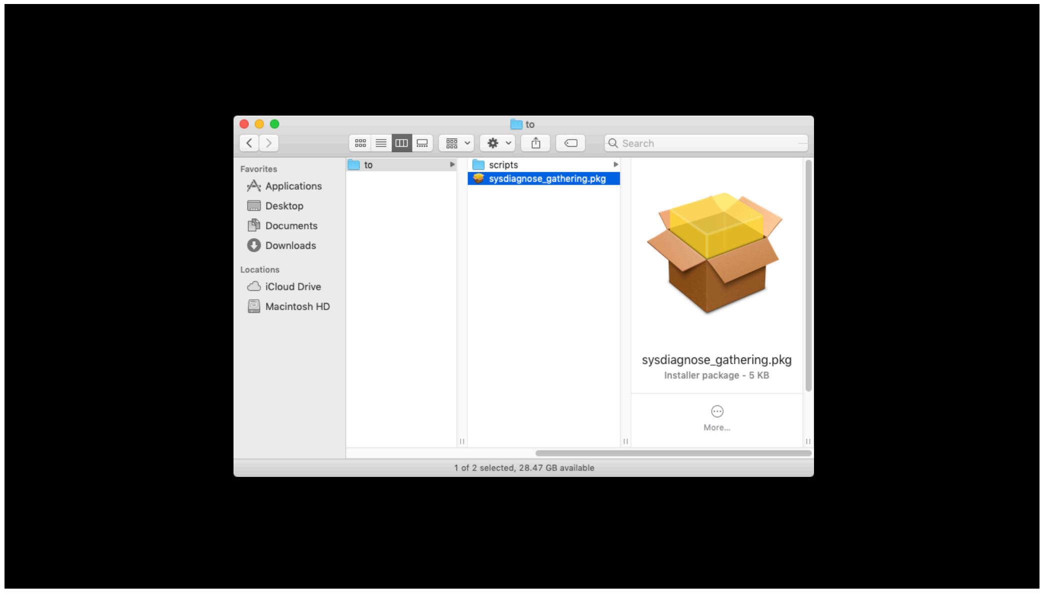
The reason for the scripts directory is that pkgbuild's scripts option is set to look for a directory with scripts inside, rather than specifying the scripts themselves.



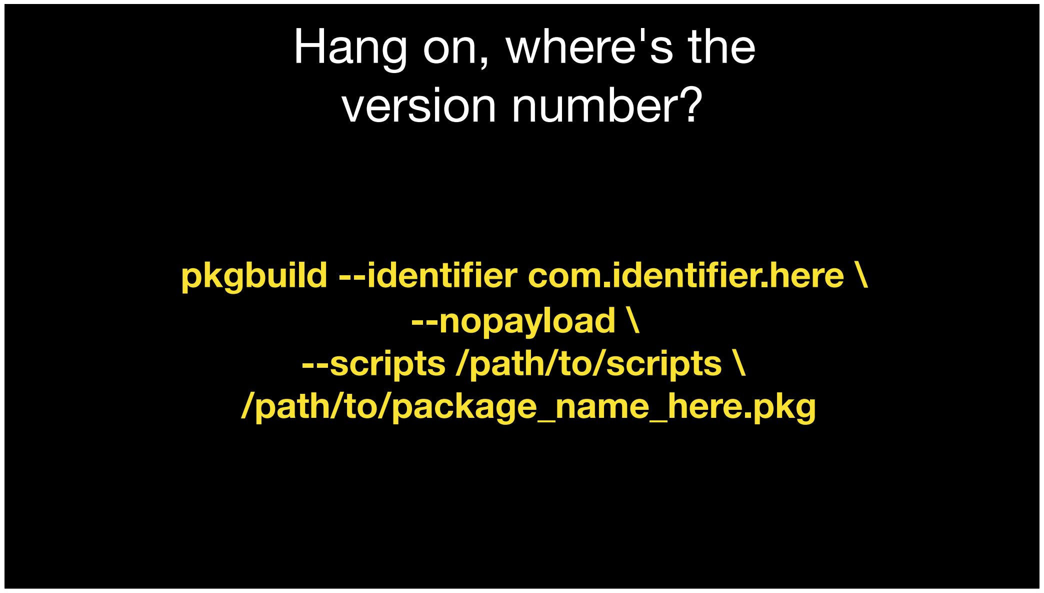
```
computername:~ username$ sudo pkgbuild --identifier com.company.sysdiagnose --nopayload --scripts /path/to/scripts /path/to/sysdiagnose_gathering.pkg
Password:
pkgbuild: Adding top-level postinstall script
pkgbuild: Wrote package to /path/to/sysdiagnose_gathering.pkg
computername:~ username$
```

```
pkgbuild --identifier com.company.sysdiagnose \
--nopayload \
--scripts /path/to/scripts \
/path/to/sysdiagnose_gathering.pkg
```

Once the scripts directory is set up and the postinstall script saved inside it, running the following command with root privileges will build a payload-free package and store it in the designated location.



Once the package is built, you can hand it off to whatever or whoever needs it and they'll be able to execute the script without knowing anything more than how to install a package.

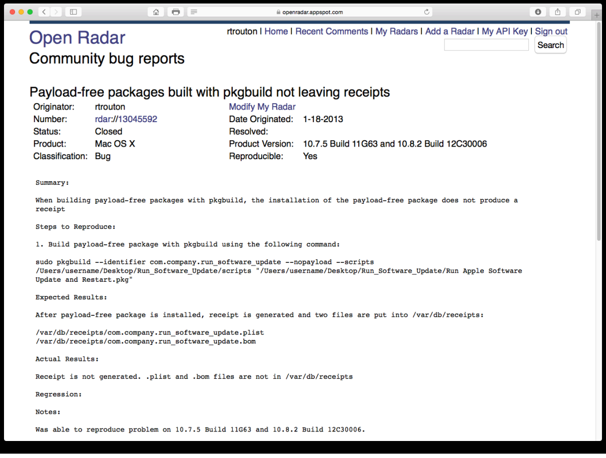


Hang on, where's the version number?

```
pkgbuild --identifier com.identifier.here \
--nopayload \
--scripts /path/to/scripts \
/path/to/package_name_here.pkg
```

For those familiar with pkgbuild, you may think I've left something out - The version number. Actually, I didn't leave it out. When you're building payload-free packages using the nopayload option, the version number isn't required. How come?

Payload-free flat packages may not leave installer receipts


openradar.appspot.com
rtrouton | Home | Recent Comments | My Radars | Add a Radar | My API Key | Sign out
Search

Open Radar

Community bug reports

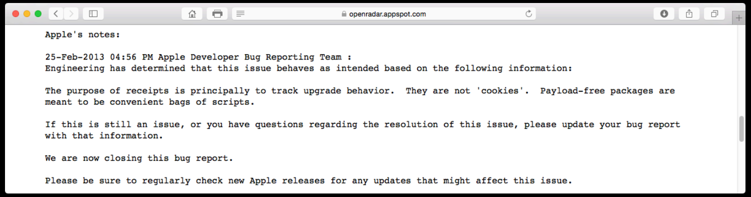
Payload-free packages built with pkgbuild not leaving receipts

Originator:	rtrouton	Modify My Radar	
Number:	rdar://13045592	Date Originated:	1-18-2013
Status:	Closed	Resolved:	
Product:	Mac OS X	Product Version:	10.7.5 Build 11G63 and 10.8.2 Build 12C30006
Classification:	Bug	Reproducible:	Yes

```
Summary:

When building payload-free packages with pkgbuild, the installation of the payload-free package does not produce a
receipt

Steps to Reproduce:

1. Build payload-free package with pkgbuild using the following command:

sudo pkgbuild --identifier com.company.run_software_update --nopayload --scripts
/Users/username/Desktop/Run_Software_Update/scripts "/Users/username/Desktop/Run_Software_Update/Run Apple Software
Update and Restart.pkg"

Expected Results:

After payload-free package is installed, receipt is generated and two files are put into /var/db/receipts:

/var/db/receipts/com.company.run_software_update.plist
/var/db/receipts/com.company.run_software_update.bom

Actual Results:

Receipt is not generated. .plist and .bom files are not in /var/db/receipts

Regression:

Notes:

Was able to reproduce problem on 10.7.5 Build 11G63 and 10.8.2 Build 12C30006.
```

When building a payload-free package with Apple's pkgbuild tool using the nopayload flag, no receipt is left behind. No receipt, no need for a version number.

Payload-free flat packages may not leave installer receipts

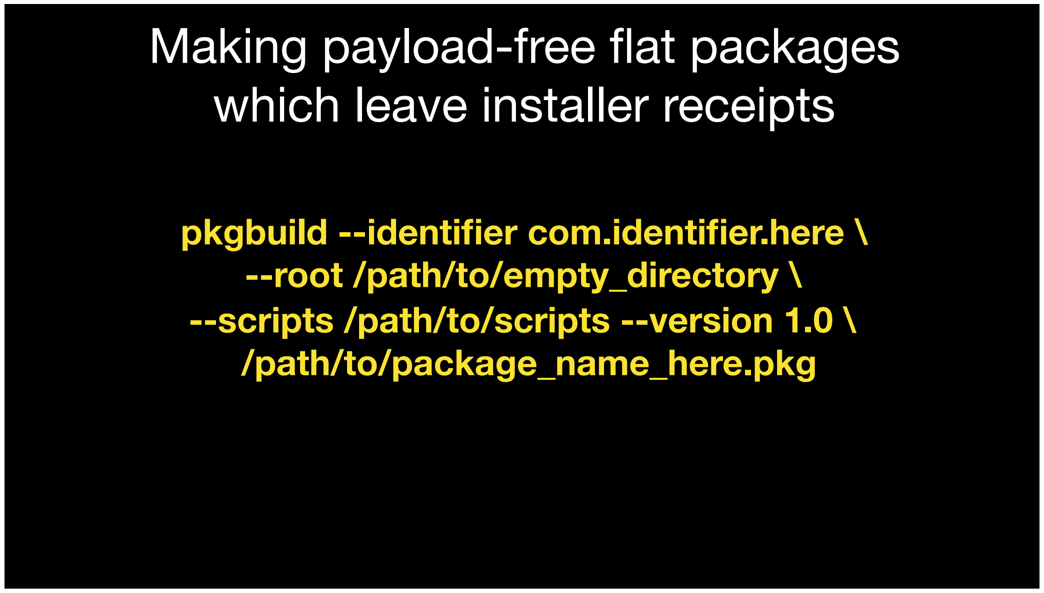


```
Apple's notes:

25-Feb-2013 04:56 PM Apple Developer Bug Reporting Team :
Engineering has determined that this issue behaves as intended based on the following information:

The purpose of receipts is principally to track upgrade behavior.  They are not 'cookies'.  Payload-free packages are
meant to be convenient bags of scripts.

If this is still an issue, or you have questions regarding the resolution of this issue, please update your bug report
with that information.

We are now closing this bug report.

Please be sure to regularly check new Apple releases for any updates that might affect this issue.
```

When I filed a bug on this, Apple said that this was intended behavior for payload-free packages built using pkgbuild's nopayload option. In Apple’s opinion, payload-free packages are a convenient bag for scripts.

Any other type of package will leave behind a receipt, including a bundle-style payload-free package.

Why is this important? If your systems management tools relies on receipts to tell whether a payload-free package has been installed, a payload-free package that leaves no receipt behind means that your management tool won’t be able to tell that it’s been installed. This may result in the payload-free package and its associated script being run repeatedly on your managed machine.



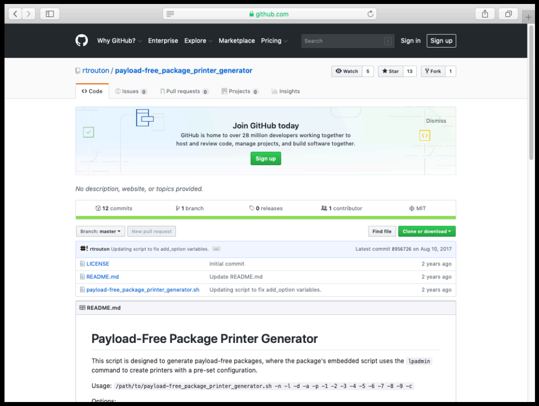
Making payload-free flat packages which leave installer receipts

```
pkgbuild --identifier com.identifier.here \
--root /path/to/empty_directory \
--scripts /path/to/scripts --version 1.0 \
/path/to/package_name_here.pkg
```

However, you can make a package with pkgbuild that, while not technically payload–free, will act just like one. The key is to create an empty directory and set pkgbuild's –root option to look there for files. pkgbuild's –root option is used to tell pkgbuild which files to package, but since there will be no files in an empty directory, the package will install no files on the destination Mac. However, it will leave behind a receipt.

Finder
File
Edit
View
Go
Window
Help
Macintosh HD
SHELL
sysdiagnose.sh

Payload-Free Package Printer Generator



rtrouton / payload-free_package_printer_generator

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

No description, website, or topics provided.

12 commits | 1 branch | 0 releases | 1 contributor | MIT

rtrouton Updating script to fix add_option variables. ...		Latest commit 8956726 on Aug 10, 2017
LICENSE	Initial commit	2 years ago
README.md	Update README.md	2 years ago
payload-free_package_printer_generator.sh	Updating script to fix add_option variables.	2 years ago

README.md

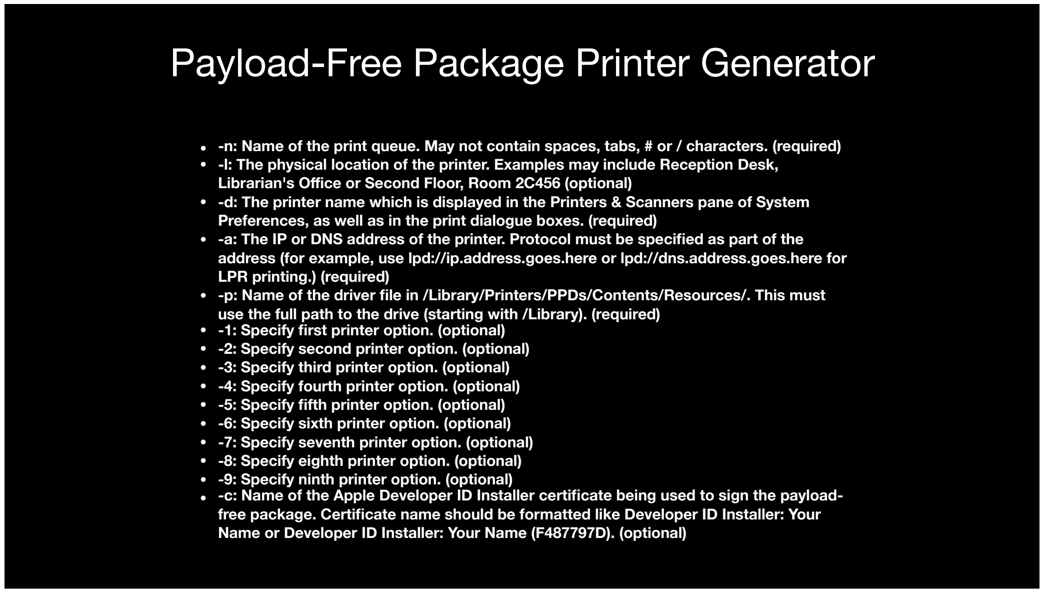
Payload-Free Package Printer Generator

This script is designed to generate payload-free packages, where the package's embedded script uses the `lpadmin` command to create printers with a pre-set configuration.

Usage: `/path/to/payload-free_package_printer_generator.sh -n -l -d -a -p -1 -2 -3 -4 -5 -6 -7 -8 -9 -c`

https://github.com/rtrouton/payload-free_package_printer_generator

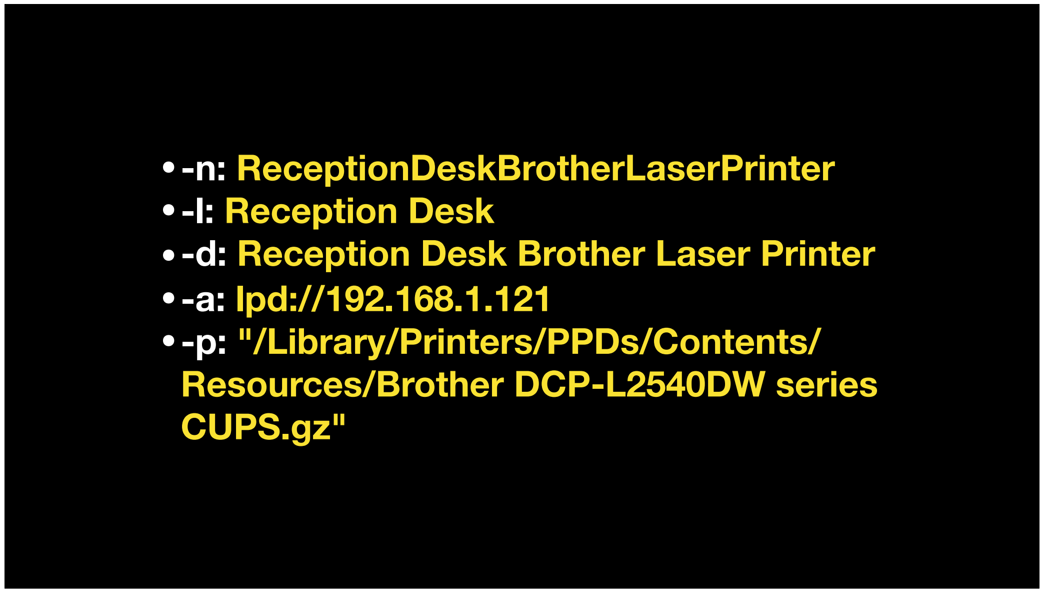
Another tool I've developed is a script used to build payload-free packages, where the package sets up a printer with a desired configuration.



Payload-Free Package Printer Generator

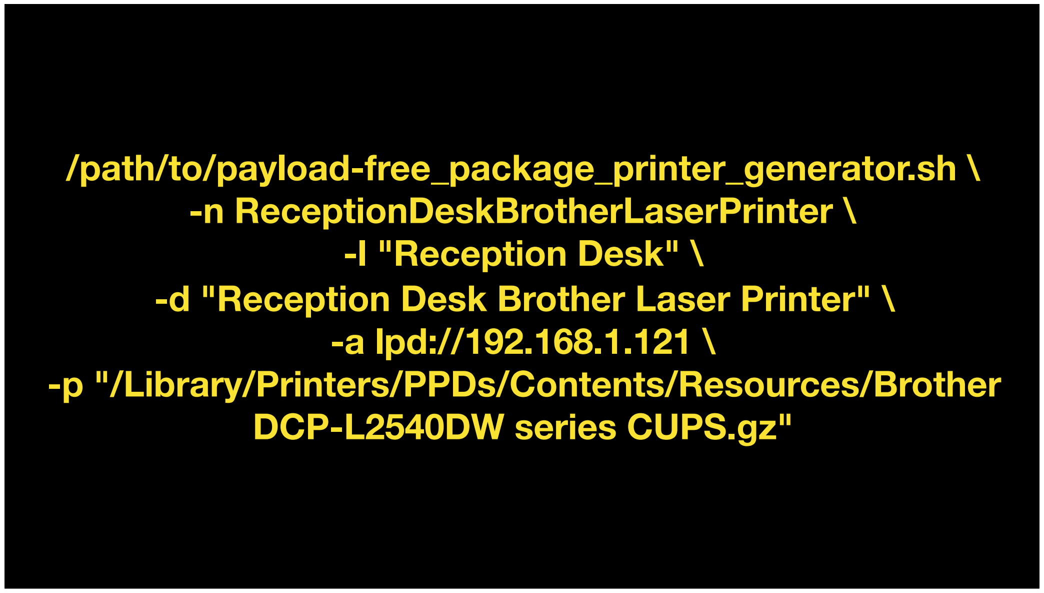
- -n: Name of the print queue. May not contain spaces, tabs, # or / characters. (required)
- -l: The physical location of the printer. Examples may include Reception Desk, Librarian's Office or Second Floor, Room 2C456 (optional)
- -d: The printer name which is displayed in the Printers & Scanners pane of System Preferences, as well as in the print dialogue boxes. (required)
- -a: The IP or DNS address of the printer. Protocol must be specified as part of the address (for example, use lpd://ip.address.goes.here or lpd://dns.address.goes.here for LPR printing.) (required)
- -p: Name of the driver file in /Library/Printers/PPDs/Contents/Resources/. This must use the full path to the drive (starting with /Library). (required)
- -1: Specify first printer option. (optional)
- -2: Specify second printer option. (optional)
- -3: Specify third printer option. (optional)
- -4: Specify fourth printer option. (optional)
- -5: Specify fifth printer option. (optional)
- -6: Specify sixth printer option. (optional)
- -7: Specify seventh printer option. (optional)
- -8: Specify eighth printer option. (optional)
- -9: Specify ninth printer option. (optional)
- -c: Name of the Apple Developer ID Installer certificate being used to sign the payload-free package. Certificate name should be formatted like Developer ID Installer: Your Name or Developer ID Installer: Your Name (F487797D). (optional)

This script has a number of options, including an option to sign the payload-free package using an Apple Developer certificate. Signing the package would enable the package to be posted somewhere for download and get past Gatekeeper.



- -n: ReceptionDeskBrotherLaserPrinter
- -l: Reception Desk
- -d: Reception Desk Brother Laser Printer
- -a: lpd://192.168.1.121
- -p: "/Library/Printers/PPDs/Contents/Resources/Brother DCP-L2540DW series CUPS.gz"

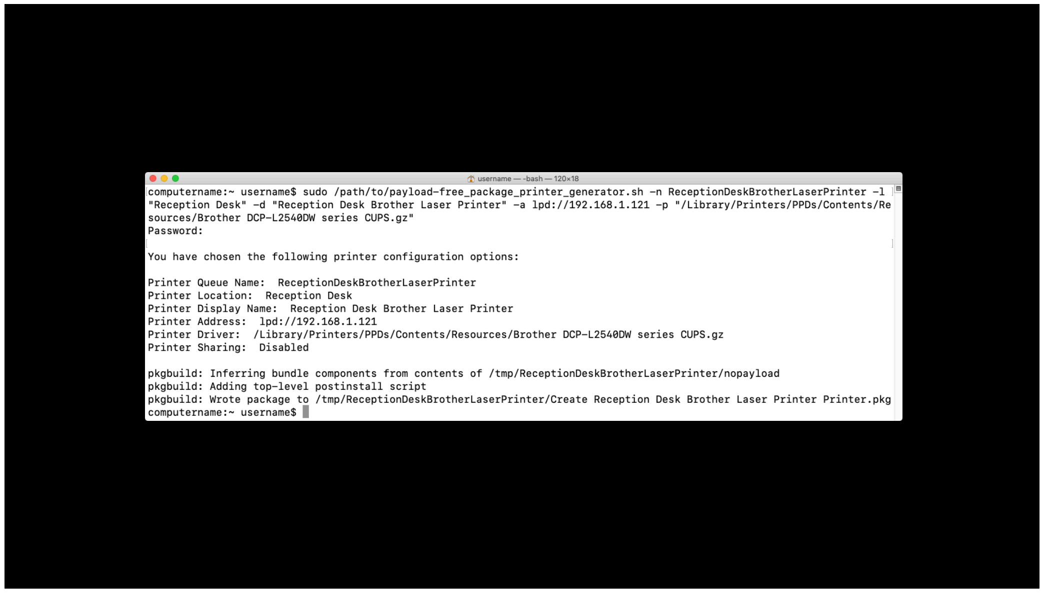
While this tool has a lot of options, you may only need to use a few of them to set up a printer. Let's take a look at how this works using only these options.



```
/path/to/payload-free_package_printer_generator.sh \
-n ReceptionDeskBrotherLaserPrinter \
-l "Reception Desk" \
-d "Reception Desk Brother Laser Printer" \
-a lpd://192.168.1.121 \
-p "/Library/Printers/PPDs/Contents/Resources/Brother
DCP-L2540DW series CUPS.gz"
```

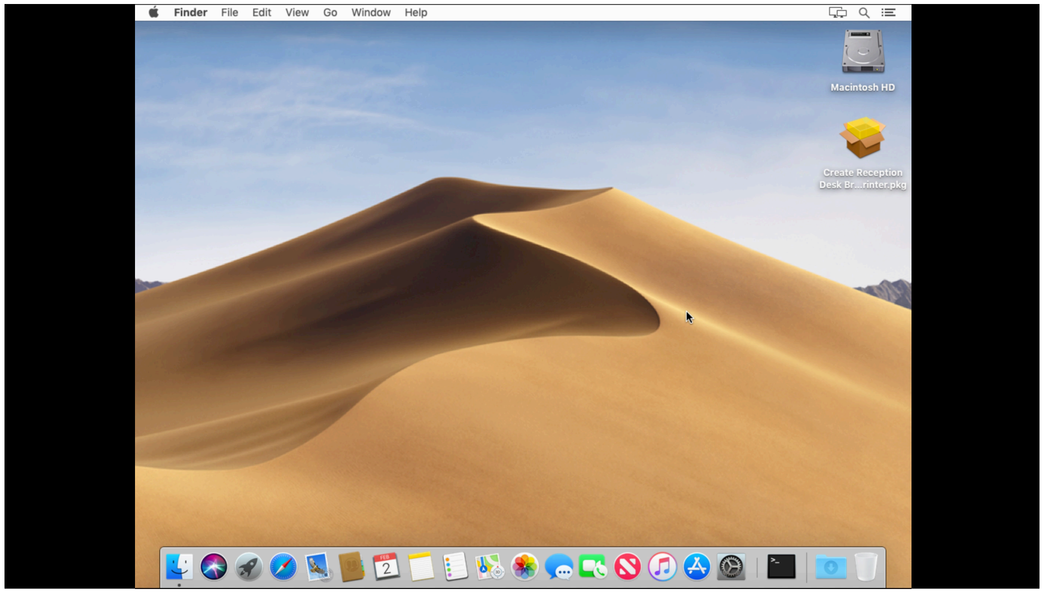
Here's how the command to run the script would look. I have it broken up into separate lines for clarity, normally this would all be on one line.

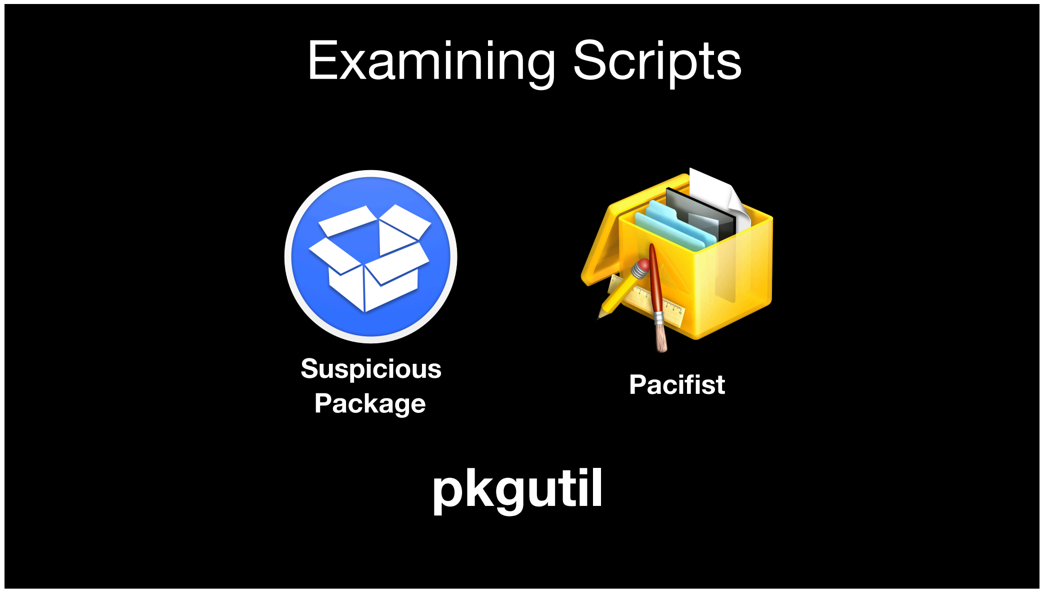
Finder
File
Edit
View
Go
Window
Help
Macintosh HD



```
computername:~ username$ sudo /path/to/payload-free_package_printer_generator.sh -n ReceptionDeskBrotherLaserPrinter -l
"Reception Desk" -d "Reception Desk Brother Laser Printer" -a lpd://192.168.1.121 -p "/Library/Printers/PPDs/Contents/Re
sources/Brother DCP-L2540DW series CUPS.gz"
Password:

You have chosen the following printer configuration options:

Printer Queue Name:  ReceptionDeskBrotherLaserPrinter
Printer Location:  Reception Desk
Printer Display Name:  Reception Desk Brother Laser Printer
Printer Address:  lpd://192.168.1.121
Printer Driver:  /Library/Printers/PPDs/Contents/Resources/Brother DCP-L2540DW series CUPS.gz
Printer Sharing:  Disabled

pkgbuild: Inferring bundle components from contents of /tmp/ReceptionDeskBrotherLaserPrinter/nopayload
pkgbuild: Adding top-level postinstall script
pkgbuild: Wrote package to /tmp/ReceptionDeskBrotherLaserPrinter/Create Reception Desk Brother Laser Printer Printer.pkg
computername:~ username$ 
```

The build process covered up the Terminal window, but here's the output you should see.


Finder
File
Edit
View
Go
Window
Help
Macintosh HD
Create Reception Desk Br...rinter.pkg



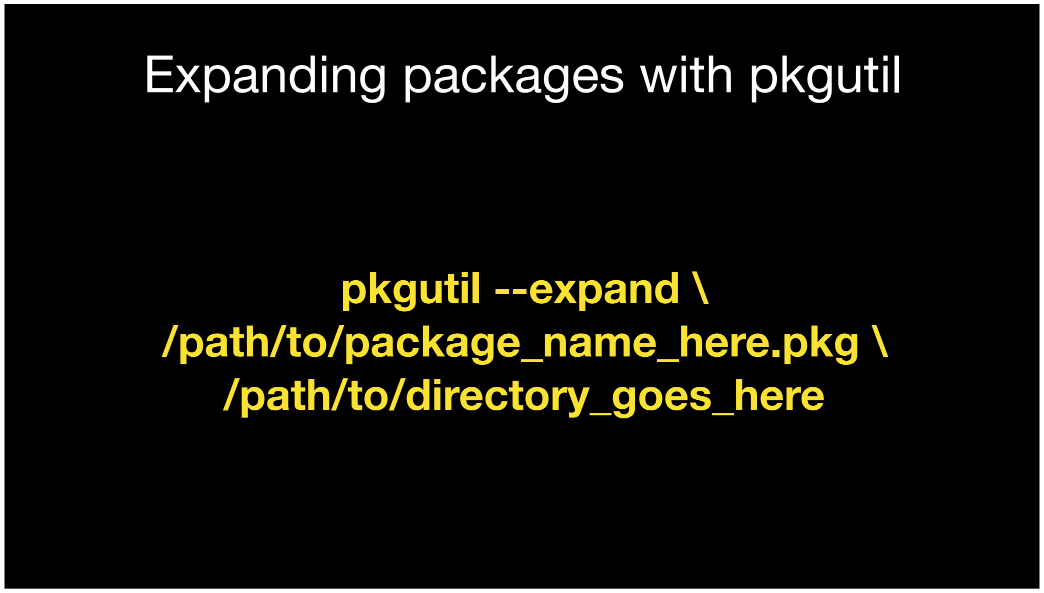
Examining Scripts

Suspicious Package

Pacifist

pkgutil

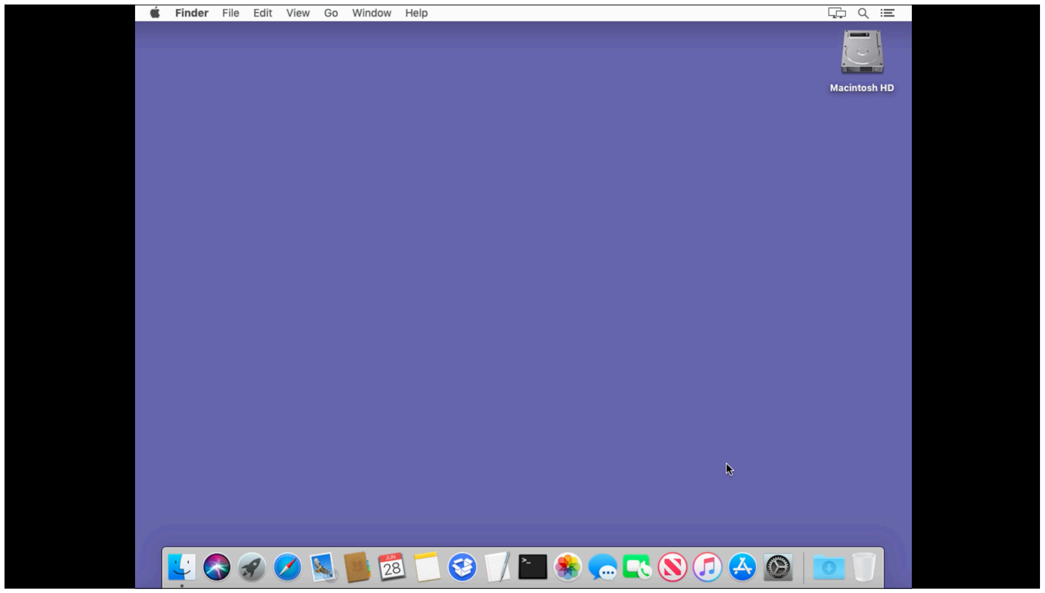
Now that you’ve seen how you can use scripts, how can you check out existing installer package scripts to see what they’re doing? There’s a few tools that can help you out. There’s Apple’s pkgutil command line tool, Pacifist and Suspicious Package. pkgutil and Suspicious Package are both free while Pacifist is shareware with a free trial period. Since both are free with no strings attached, let’s look at pkgutil and Suspicious Package.



Expanding packages with pkgutil

```
pkgutil --expand \
/path/to/package_name_here.pkg \
/path/to/directory_goes_here
```

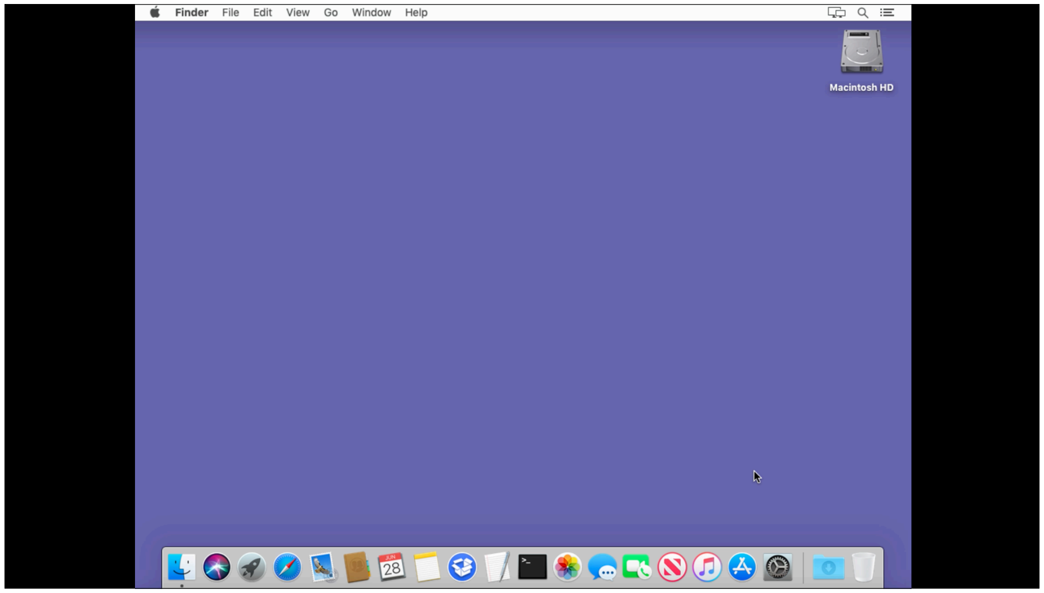
pkgutil doesn’t directly examine scripts for you, but it expands the contents of an installer package into a directory. From there, you can examine the scripts included with the package.



pkgutil doesn't directly examine scripts for you, but it expands the contents of an installer package into a directory. From there, you can examine the scripts included with the package.

Suspicious Package

Suspicious Package is a phenomenally useful tool, for reasons that will be shown in a second. It'll check out an installer package for you, show you its contents and even warn you if there's problems.


Finder
File
Edit
View
Go
Window
Help
Macintosh HD

Useful Links

Understanding Payload-Free Packages: https://derflounder.wordpress.com/2014/06/01/understanding-payload-free-packages/

Creating payload-free packages with pkgbuild: https://derflounder.wordpress.com/2012/08/15/creating-payload-free-packages-with-pkgbuild/

Apple Developer Software Delivery Legacy Guide: http://tinyurl.com/hze8pr8

Flat Package Format - The missing documentation: http://s.sudre.free.fr/Stuff/Ivanhoe/FLAT.html

Useful Links

Preparing EndNote X8 for deployment using AutoPkg: https://derflounder.wordpress.com/2016/11/15/preparing-endnote-x8-for-deployment-using-autopkg/

Creating a DNAStar Lasergene 13.x installer: https://derflounder.wordpress.com/2016/03/17/creating-a-dnastar-lasergene-13-x-installer/

Deploying a pre-configured Junos Pulse VPN client on OS X: https://derflounder.wordpress.com/2015/03/13/deploying-a-pre-configured-junos-pulse-vpn-client-on-os-x/

Repackaging the LabVIEW 2013 Pro installer: https://derflounder.wordpress.com/2013/12/06/repackaging-the-labview-2013-pro-installer/

Useful Links

Pacifist: https://charlessoft.com

Suspicious Package: https://mothersruin.com/software/SuspiciousPackage/

Iceberg: http://s.sudre.free.fr/Software/Iceberg.html

Packages: http://s.sudre.free.fr/Software/Packages/about.html

The Luggage: http://luggage.apesseekingknowledge.net

Useful Links

MunkiPkg: https://github.com/munki/munki-pkg

Jamf Composer: https://www.jamf.com/products/jamf-composer/

Payload-Free Package Creator: https://github.com/rtrouton/Payload-Free-Package-Creator

Payload-Free Package Printer Generator: https://github.com/rtrouton/payload-free_package_printer_generator

Downloads

PDF available from the following link:

https://tinyurl.com/PSUMacAdmins2019PDF

Keynote slides available from the following link:

https://tinyurl.com/PSUMacAdmins2019Keynote