Measuring and Optimizing Patch Management: an Open Model

Findings from the Project Quant patch management project

Version 1.0
Released: July 27, 2009

Author's Note

Securosis was approached by Jeffrey Jones of Microsoft in late 2008 to develop an open, independent patch management metrics model. Unlike most of our research, Jeff (the sponsor) wanted to participate and contribute directly to the project. To maintain objectivity, Microsoft participants agreed to follow the Totally Transparent Research process and contribute all content in the open, just like any other member of the public.

To support the project we created a dedicated Project Quant landing site and forums. All research was produced in the open, and no public comments were restricted or moderated (other than spam). All members of the public, and even the vendor community, were invited to participate and contribute. We hope this process has removed any potential bias in the model, and produced a truly objective, community-driven result.

Special thanks to Chris Pepper for editing and content support.

Contributors

This research was primarily conducted by Rich Mogull of Securosis and Jeffrey Jones of Microsoft.

The following individuals also contributed significantly to this report through comments on the Securosis blog and follow-on review and conversations (listed by their display names from the Project Quant site, to protect privacy):

Daniel
DS
Mark
Amrit WIlliams
Dutch
Lonervamp

Additional contributors from non-public mailing lists or anonymous accounts can contact us for inclusion on this list in future versions of the report.

The following organizations and communities assisted in promoting Project Quant research and survey efforts (without paying for any promotional considerations or other financial contributions):

- The Institute for Applied Network Security (IANS)
- BigFix
- Qualys
- Tenable Network Security
- The securitymetrics.org community
- The patchmanagement.org community
- The New School of Information Security

Copyright

This report is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 license.

http://creativecommons.org/licenses/by-nc-nd/3.0/us/

Project Quant Executive Summary

Developing an Open Patch Management Metrics Model

This report includes the findings of the Project Quant patch management research project. Project Quant is dedicated to the development of a refined, unbiased patch management metrics model. Our goal is to provide organizations with a tool to better understand their patching costs, and to guide improvements through an operational efficiency model capable of capturing accurate and precise performance metrics. Project Quant was developed through independent research, community involvement, and an open industry survey.

The Project Quant Survey

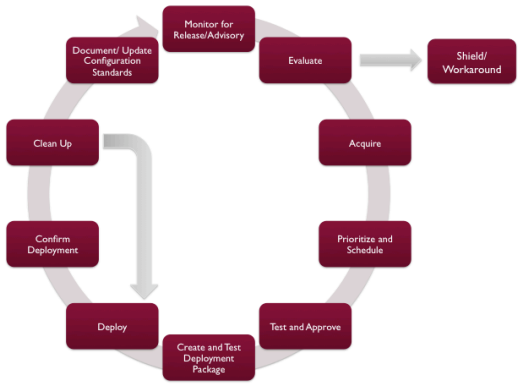
Project Quant conducted an open, industry-wide survey. 100 organizations responded, representing a broad range of organizations from under 10 to over 100,000 employees. The survey results and raw data (for your own analysis) are available at the Project Quant site.

Key Findings

- There is no public platform-independent, industry-standard patch management process framework. As a result, Project Quant developed a superset framework to encompass most patching activities within any organization, regardless of technology assets under review. It includes ten phases with forty steps.

Based on survey responses, organizations are generally mature in terms of managing desktop and server operating system patches, but process maturity quickly falls off for other technologies and platforms.

- Staff time dedicated to patch management activities represents the majority of patch management costs, and thus the model was designed to focus heavily on granular patching activities.
- Patching across multiple platforms and business activities is a very complex process, and although the Project Quant model is extremely detailed, most organizations should focus on their key metrics, identified through the model.



The Patch Management Cycle

Summary and Next Steps

•This release includes a detailed patch management process framework and metrics model to enable organizations to quantify and optimize their patch management processes.

•This is Version 1.0 of the model; future work will continue refinement, add sample use cases, and assess its functionality in various user environments.

•The next step is to interview end-user organizations to determine how their processes and maturity align with the model and survey results.

•The model can then be adapted for industry benchmarking.

Table of Contents

Introduction 1

An Intractable Problem 1

Project Quant Definitions and Goals 2

Problem Definition 2

Objective 2

Additional Detail 2

Assumptions, Process, Status, and Background 2

Research Process 3

Project Status 3

The Patch Management Process 4

The Patch Management Cycle 4

Patch Cycle Phases 5

Shielding and Workarounds 5

The Deploy Through Clean Up Sub-Cycle 6

Detailed Phases 7

Introduction 7

Monitor for Release/Advisory 8

Evaluate 10

Acquire 12

Prioritize and Schedule 13

Test and Approve 14

Create and Test Deployment Package 16

Deploy 17

Confirm Deployment 18

Clean Up 19

Document and Update Configuration Standards 20

The Metrics Model 21

Introduction 21

How to Use the Model 21

Define the Asset Type (or Program) 22

Define Roles 22

Determine Non-Phase Program Costs 23

Determine Individual Phase Costs 23

Phase 1: Monitor for Release/Advisory 24

Step 1: Identify Asset Types 24

Step 2: Identify Advisory Sources 24

Step 3: Monitor for Advisories 24

Phase 2: Evaluate 24

Step 1: Match to Asset Type 24

Step 2: Determine Nature 25

Step 3: Determine Relevance and Priority 25

Step 4: Determine Dependencies 25

Step 5: Workarounds and Shielding 26

Phase 3: Acquire 26

Step 1: Locate 26

Step 2: Acquire 26

Step 3: Validate 26

Phase 4: Prioritize and Schedule 27

Step 1: Prioritize 27

Step 2: Match to Assets 27

Step 3: Schedule 27

Phase 5: Test and Approve 28

Step 1: Develop Test Criteria 28

Step 2: Test 28

Step 3: Analyze Results 28

Step 4: Approve 29

Test/Analyze Cycle 29

Phase 6: Create and Test Deployment Package 29

Step 1: Identify Deployment Tool 29

Step 2: Consolidate Patches 29

Step 3: Build Deployment Package 29

Step 4: Test Deployability 30

Step 5: Test Functionality 30

Step 5: Approve Package 30

Test/Analyze Cycle 30

Phase 7: Deploy 30

Step 1: Prepare 30

Step 2: Deliver 31

Step 3: Install 31

Step 4: Clean Up 31

Phase 8: Confirm Deployment 31

Step 1: Test Deployment 31

Step 2: Test Functionality 32

Step 3: Document 32

Phase 9: Clean Up 32

Step 1: Identify Failed Deployments 32

Step 2: Determine Deployment Failure Cause 32

Step 3: Adjust Deployment Parameters 33

Step 4: Re-Deploy 33

Phase 10: Document 33

Step 1: Document Patch Deployment 33

Step 2: Determine and Document Configuration Standard Changes 33

Step 3: Approve Configuration Standard Changes 34

Program Metrics 34

The Center for Internet Security's Consensus Metrics 34

Combine and Analyze Costs 34

Adapting the Model for Measuring a Complete Program 35

Full Evaluation 35

Key Metrics Evaluation 35

The Costs of Maintenance Windows and Predictable Patches 36

Conclusions and Next Steps 38

Patch Management Is Still Difficult... Mostly 38

Next Steps 38

Who We Are 40

About the Authors 40

Introduction

An Intractable Problem

Patch management is not only one of the single most important and frequent activities in any information technology organization, it's also one with which we have decades of practice and advancement. Yet despite our extensive collective experience, IT managers frequently cite patch management as one of their primary concerns in terms of costs, efficiency, and effectiveness. While information is often available on techniques for patching specific platforms, very little detailed work on overall patch management processes and efficiencies is publicly available. Given its far-reaching impact on security, reliability, compliance, and performance, it's astounding how immature the practice actually is. We, as a community, lack the detailed frameworks, models, and performance metrics we need to understand if we're managing patches appropriately, using an optimal process.

This is largely due to the scope of the problem; patch management affects every single tool in the technology arsenal — from our most critical servers down to the phones in our pockets. Each platform comes with its own set of requirements, procedures, processes, and dependencies. Even the philosophical predilections of the product vendor affect how we manage their software or hardware. While some areas are more or less mature, and a variety of first and third party solutions are available to help with different aspects of patch management, organizations lack independent frameworks and metrics to help them determine if they are doing an effective job, or where they can improve their processes.

Although it's impossible to completely standardize patch management across the entire spectrum of industries, organizations, and technologies, Project Quant was established to assist organizations in better understanding and optimizing their processes. The initial goal of the project was to build a basic metrics model, but it has since expanded to include a full patch management framework, detailed metrics, and an open survey to better understand the current maturity of patching processes.

By providing a detailed performance metrics model we hope to help organizations improve their internal processes, as well as improve overall efficiency and effectiveness. The model should help identify specific areas of inefficiency, and guide users towards specific improvements. Project Quant is also a quantified cost model, and provides a way to measure patch management costs in different areas across their entire programs. We have used surveys and interviews to inform and support our findings, and (as with the model) all data is being made completely public. We hope this helps organizations better understand the state of patching in the industry, and where they fit in terms of maturity.

It's time to remove the guesswork, begin understanding the real costs of patch management decisions, and provide the open frameworks, models, metrics, and data to optimize our processes.

Project Quant is an ongoing project, and although this document reflects the current state of the research, it should still be considered a work in progress. We will update and re-release this report as the model, surveys, and other findings evolve.

Project Quant Definitions and Goals

We established the following goals when we launched the project:

Problem Definition

Based on our research, there is no independent, objective model to measure the costs of patch management, nor any comprehensive proprietary models. There are also no operational efficiency models/metrics to assist organizations in optimizing their patch management processes. Finally, in general, the security industry lacks operational metrics models as seen in other areas of IT and business.

Objective

The objective of Project Quant is to develop a cost model for patch management that accurately reflects the financial and resource costs associated with the process of evaluating and deploying software updates (patch management).

Additional Detail

As part of maintaining their technology infrastructure, all organizations deploy software updates and patches. The goal of this project is to provide a framework for evaluating the costs of patch management, while providing information to help optimize the associated processes. The model should apply to organizations of different sizes, circumstances, and industries. Since patch management processes vary throughout the industry, Project Quant will develop a general model that reflects best practices and can be adapted to different circumstances. The model will encompass the process from monitoring for updates to confirming successful rollout, and should apply to both workstations and servers. The model should be unbiased and vendor-neutral. Ideally, the model should also help advance the field of information technology metrics, particularly information security metrics.

Assumptions, Process, Status, and Background

Microsoft contacted Securosis to develop an open patch management metrics model. One of the primary goals was to involve the larger community in order to create an effective, accurate, and unbiased model.

Early on we established certain parameters to achieve the project goals, as well as some background assumptions:

- *This should be a quantified metrics model, focused on costs:* All the metrics or variables in the model should be measurable with accuracy and precision. “Qualified” metrics, such as risk and threat ratings, are not included. This model is designed only to measure the costs of patch management, and to identify operational efficiencies or deficiencies in specific process areas. It relies on measurable, quantifiable inputs, rather than assessments or other qualified values based on human judgement.
- *The model should apply to all potential patching activities and asset types:* The model was developed to apply to any asset — from fixed hardware like multifunction printers, to desktops, to major application servers. Due to this design, certain portions of the model will need to be tuned, or even dropped, depending on the specific patching activity under consideration.
- *The model should apply to organizations of any size or vertical:* The model is not designed only for large organizations in particular vertical markets. Although smaller organizations work with fewer resources and different processes, the model will still provide a functional framework.
- *The model thus represents a superset of patching activities:* To achieve the dual goals of applying to any potential patching activity, and to organizations of differing sizes and verticals, the model was designed as a superset of any one organization’s patching activities. *We do not expect all users to utilize all portions of the model, and you are encouraged to adapt the model for your own particular needs.*

- *The model will not be limited to only security patches:* The model should apply to general patches, not just security patches.
- *The model cannot measure the costs of not patching:* Clearly, the easiest way to reduce your patching costs to zero is to avoid patching. While there are many potential ways to measure the business impact of not patching, they are not part of this model. In this phase of Project Quant we are concerned only with measuring the costs when you do patch. In large part this is due to our strict focus on quantified metrics; addressing the impact of not patching would require us to include predictive and more subjective elements.
- *While the model can measure total program costs, it is focused on measuring patching for single asset types:* Since there is such a high degree of patching variability within organizations, if we focused on measuring total costs throughout the organization, we would have to reduce the number and quality of variables under consideration. We instead decided to focus the model on measuring patching costs for specific asset types and platforms, such as workstation operating systems, specific database management systems, and so on. When used for a particular platform, the model should provide reasonably accurate results. At the same time, we recognize the value of measuring the costs for an entire patching program, and have identified key metrics to support this. While this approach isn't as refined, and the inputs won't have the same degree of precision or accuracy, it should still provide useful information.
- *The model should break out costs by process to support optimization:* One reason for the extensive detail included on the patch management process is to support identification of specific operational efficiencies or problems. Our goal is to help organizations identify, and then correct, problem areas. For example, the model will help identify reasons for failed patch deployments requiring more effort, or managerial/sign off problems due to unresponsive personnel.
- *Not all users will use all parts of the model:* This is a complex detail-oriented model that could cost more than patching itself if it's manually completed with full detail. We purposely erred on the side of greater specificity, with the full understanding that very few users will dig in at such a low level. We strongly encourage you to adapt the model to your own needs, and have identified key metrics to assist with prioritization. Over time we hope that more and more of these metrics will be obtainable through automation and inclusion in support tools.

Research Process

All materials are being made publicly available throughout the project, including internal communications (the Totally Transparent Research process). The model was developed through a combination of primary research, surveys, focused interviews, and public/community participation. Survey results and interview summaries will be posted on the project site, but certain materials may be anonymized to respect the concerns of interview subjects. All interviewees and survey participants are asked if they wish their responses to remain anonymous, and details are only released with consent. Securosis and Microsoft use existing customers and contacts for focused interviews and surveys, but also release public calls for participation to minimize bias due to participant selection.

Project Status

This document represents version 1.0 of the model. It includes the detailed patch management cycle and framework, initial identification of key metrics, and a first pass at relating the metrics.

We have also completed the initial Open Patch Management Survey. Some of those results are included in this report, and a full survey analysis and the raw data are being released separately.

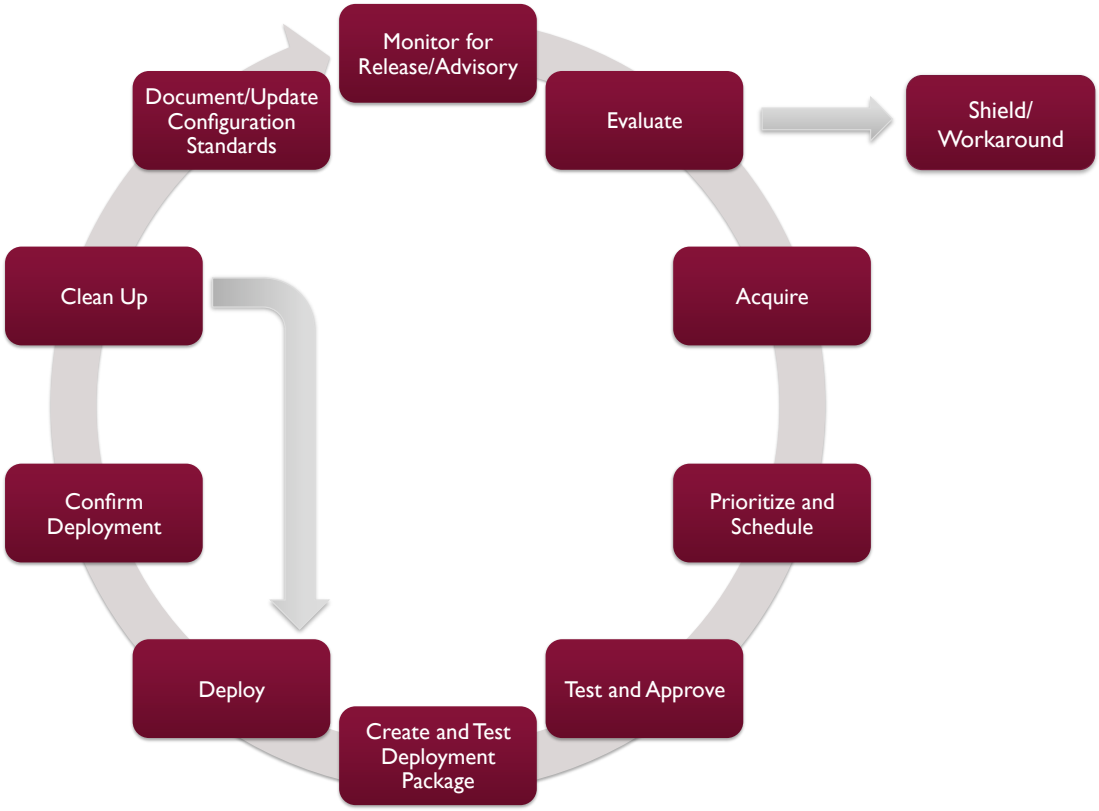
This version of the model does not include a detailed spreadsheet tool or use cases. Our goal is to complete those with ongoing work. We encourage anyone interested in participating visit the Project Quant site.

The Patch Management Process

The Patch Management Cycle

During our initial research we were unable to find any documented patch management processes that met the needs of Project Quant. Existing processes were either too high-level or specific to a limited number of technology platforms, and couldn't support the detailed metrics we required to meet the project goals. Thus we developed a new process framework, starting with a high-level cycle, and then detailed steps for each phase of the process.

This cycle represents a superset of potential patch management activities across any technology platform. Not all organizations follow each step in this exact order, but we feel this captures most of the potential patch management phases in sufficient detail and a relatively intuitive order.



Patch Cycle Phases

1. **Monitor for Release/Advisory:** Identify the asset types (platforms) you need to maintain, identify patch sources for those platforms, and then monitor on an ongoing basis for new patch releases. Since asset types, patch sources, and the patches themselves are changing on a constant basis; it's important to follow an ongoing process.
2. **Evaluate:** Perform the initial evaluation of the patch to determine if it applies within your organization, what type of patch it is, and if it's relevant to your environment. This is the initial prioritization phase to determine the nature of the patch (*e.g.,* security fix vs. reliability improvement), its relevance and general priority for your organization, and any possible shielding or workarounds. (Shielding/workaround is a separate process outside this model, but would be initiated during this phase).
3. **Acquire:** Locate the patch, acquire it, and validate the integrity of the patch files. Since most patches are downloaded these days, this is to ensure the download completed properly, but could also apply to patches on physical media.
4. **Prioritize and Schedule:** Prioritize based on the nature of the patch itself and your infrastructure/assets. Then build out a deployment schedule based on your prioritization, scheduled maintenance windows, and other factors. This usually involves the participation of multiple stakeholders, ranging from application and system owners, to business unit representatives if any downtime or feature-change training is involved.
5. **Test and Approve:** Develop test criteria, perform any required testing, analyze the results, and approve the patch for release once it meets your requirements. Testing should include patch installation, operation, and performance.
6. **Create and Test Deployment Package:** Identify the proper deployment tool, consolidate patches and build a deployment package, then test that package for deployment, installation, operation, and performance. Based on your earlier scheduling you may be combining a variety of patches for the same platform into a single package, such as application and operating system patches for a desktop.
7. **Deploy:** Prepare the target assets for deployment, deliver the patch, install, and then clean up any patch residue such as temporary files.
8. **Confirm Deployment:** Verify that patches were properly deployed, including successful installation and operation. This might include use of configuration management or vulnerability assessment tools.
9. **Clean Up:** Identify any failed deployments, determine the reason for the failure, adjust the deployment parameters, and reinstall the patch or deployment package. It's rare to have a patch rollout without any failures, particularly when deploying to multiple simultaneous assets (like desktops) as opposed to a single update on a single server.
10. **Document and Update Configuration Standards:** Document the patch deployment, which may be required for regulatory compliance, and update any associated configuration standards/guidelines/requirements. Since updates to a new version may change all your configuration standards, it's important to both document the patch installation and then update your standards.

Shielding and Workarounds

For many patches, especially security updates, you may need to employ workarounds or implement tactical security controls (shielding). For example, if there's a new vulnerability in a database server that allows unauthenticated remote code execution over a network port, your first step will be to block that at the network. If a feature of your application server is behaving inappropriately, you may employ some sort of a functionality workaround, such as a scheduled batch process to clean up temporary files or incorrect database entries.

Shielding and workarounds are critical to any effective patch management process, since even when a patch is available, you may not be able to install it immediately due to testing requirements, scheduling downtime, or a simple lack of resources.

During the research phase of this project we determined that, while shielding and workarounds are a critical adjunct to patch management, determining detailed metrics was beyond the scope of this project. With such high variability — having to account for everything from a simple firewall rule change to complex structural application changes — shielding and workarounds come with their own sets of processes and costs. We consider this a high priority area for future research, even though it is beyond the scope of Project Quant.

The Deploy Through Clean Up Sub-Cycle

Even when deploying patches on a single system, it's extremely common to encounter situations resulting in installation or operation failure. This problem is exacerbated with larger deployments involving anything from a few dozen, to hundreds of thousands of systems. We've represented this with a sub-cycle covering deployment, confirmation, and clean up, and costs vary based on the number of cycles to achieve complete deployment.

You might also encounter other situations forcing you to break out of the cycle and repeat steps. For example, if the vendor provides a bad patch and you identify problems during testing, you'll need to repeat back from the beginning of the cycle. If that same patch passes testing, but breaks functionality during deployment, you'll need to repeat more phases, and deal with the associated costs.

Since failed deployment is fairly common, we formalized it as a sub-cycle. The other cases generally occur far less often and are far less predictable, so we didn't include them in the model. If you have a vendor that consistently provides bad patches, or you encounter other failures on a regular basis, you can adjust the cycle and the model to account for these costs.

Detailed Phases

Introduction

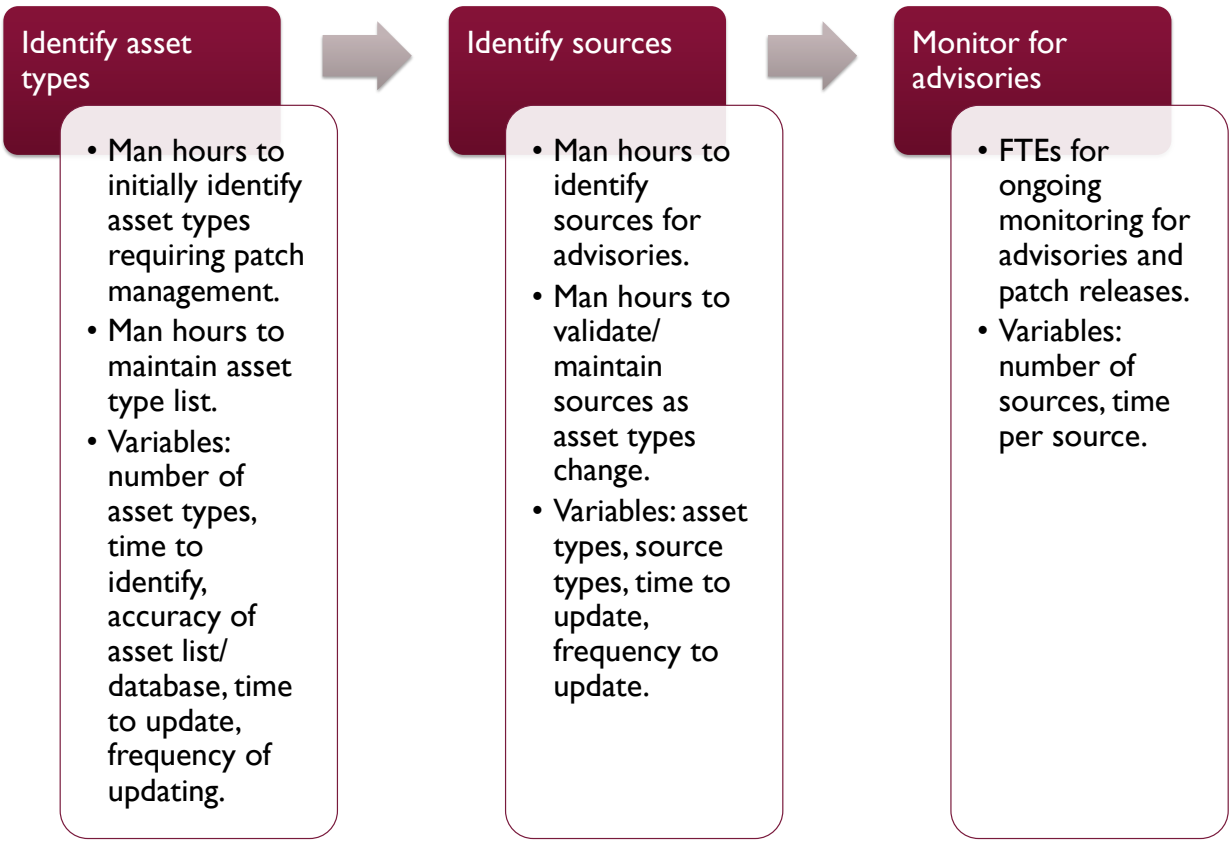
For each phase in the patch management cycle, we developed a series of steps — each with specific variables to feed the model. In this section we will describe each of the phases at a macro level, while in the next section we will call out the variables in more detail.

These steps should encompass the vast majority of potential patch management activities, and many of the key variables. Although the variables listed in this section of the report correlate directly to the metrics portion of the model, they are not presented with the same level of detail for space and brevity's sake. Although still fairly detailed, these are generalized variables included more to provide a sense of the factors involved in each step than to precisely represent the in-depth metrics of a phase. Please see the corresponding phases in the metrics portion of the model for specific metrics and identification of key metrics.

In most cases the variables are in terms of staff hours (for a single step) or FTEs (for an ongoing activity).

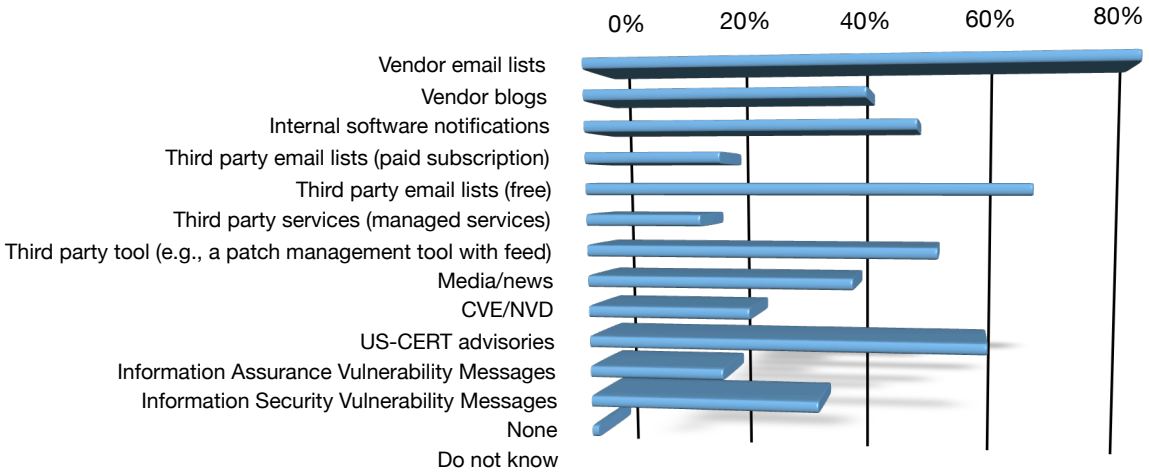
Monitor for Release/Advisory

The three steps in this phase are to identify asset types, identify sources for the advisory, and put a monitoring process in place. We define an advisory as notification that a patch for a platform is available.



1. **Identify asset types:** Before you can monitor for potential advisories, you need to identify all the asset types (platforms) that require patches. There are two sub-steps — first any initial identification activity if you don't already have an asset type list, and second maintaining that list over time. For some platforms, such as certain server applications, a single asset may compromise an entire asset type. A highly-customized asset of a particular type may also be considered an entirely new asset type if it requires special handling (*e.g.,* a legacy server beyond normal maintenance and support).
2. **Identify advisory sources:** After identifying which asset types you need to maintain, you then need to identify potential sources for advisories. In most cases this will be the software vendor, but any given vendor may release advisories through a variety of channels. There is also effort to keep this list up to date and matched to any changes in your asset types.
3. **Monitor for advisories:** This is the ongoing process of monitoring your various sources for any updates. It varies greatly for different asset types and software providers, and is likely to be broken out by who is responsible for the various assets.

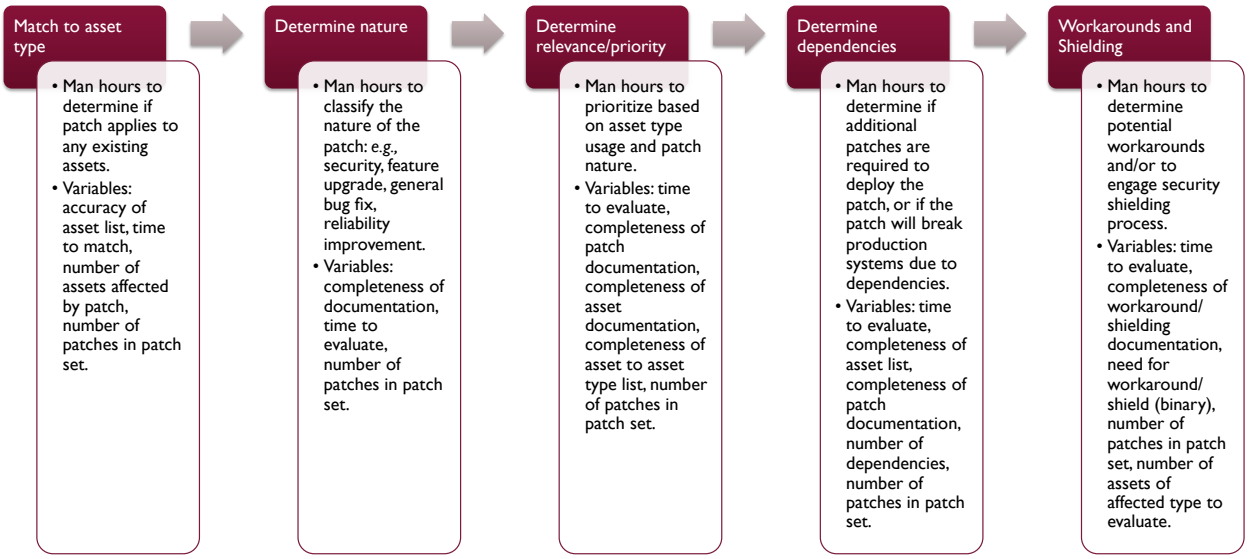
We developed a generalized list of advisory sources for the Project Quant survey, and the chart below shows the responses:



A number of respondents also cited the OSVDB as a data source.

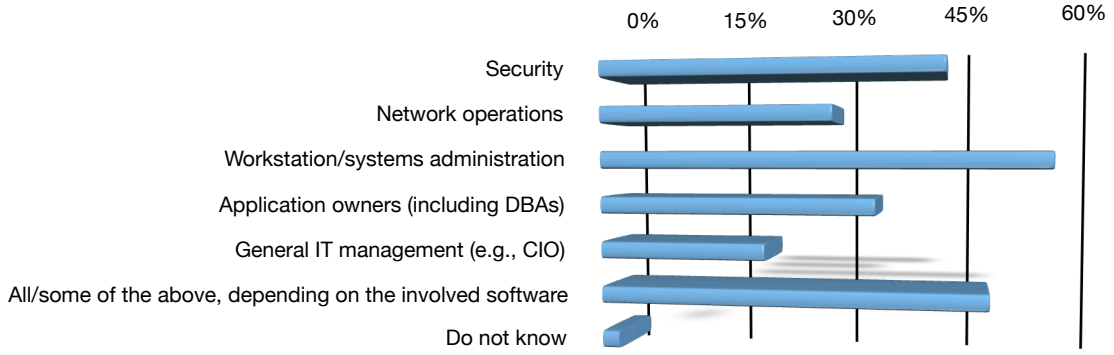
Evaluate

This phase includes the initial evaluation of an advisory in order to determine relevance, initial priority, and general nature. This is also the phase where any potential shielding or workaround processes are initiated. It includes five steps:



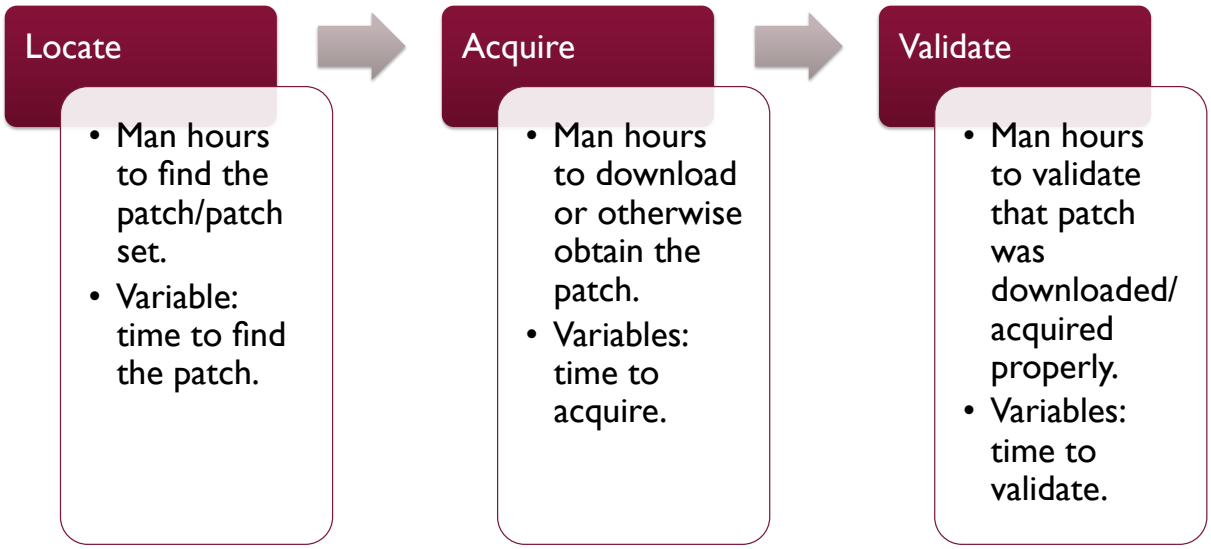
1. **Match to asset type:** When an advisory is released, the first step is to determine if it matches any of your asset types. The speed of this process clearly varies based on how up to date your asset type list is, the documentation quality of the advisory, and how many platforms are covered by the advisory. This is why it's important to have an up to date list of asset types with current version numbers. Also, don't assume that all assets of that type are at the current version, especially when dealing with servers and applications.
2. **Determine nature:** Most organizations manage different types of patches differently. A security patch may initiate a rapid response process, while general feature improvements and bug fixes are managed more slowly.
3. **Determine relevance and priority:** Now that you know if the patch matches a platform in your environment, and the nature of the patch, you can determine its initial priority. This may also vary based on the importance of particular assets, not merely whether or not they exist in your environment. For example, a medium priority update (per the vendor's definition) with a bug fix may be a high priority update if it's for a critical server experiencing ongoing performance issues.
4. **Determine dependencies:** Many patches require certain dependencies to function properly, and these aren't always included in the issued patch, particularly on servers. The amount of time required to determine any dependencies will depends on the quality of documentation and any asset and asset types lists.
5. **Workarounds and shielding:** As discussed in the previous section, workarounds and shielding are a critical part of an effective patch management process. In this step, determine any potential workaround and/or shielding requirements, then kick off separate processes (outside the scope of this model) to implement such options.

A number of roles are typically involved in evaluating patches, ranging from security to system owners. The chart below from the Project Quant survey shows the roles involved in respondent organizations when evaluating a patch for possible deployment:



Acquire

While it can be simple, acquiring a patch still involves multiple steps:

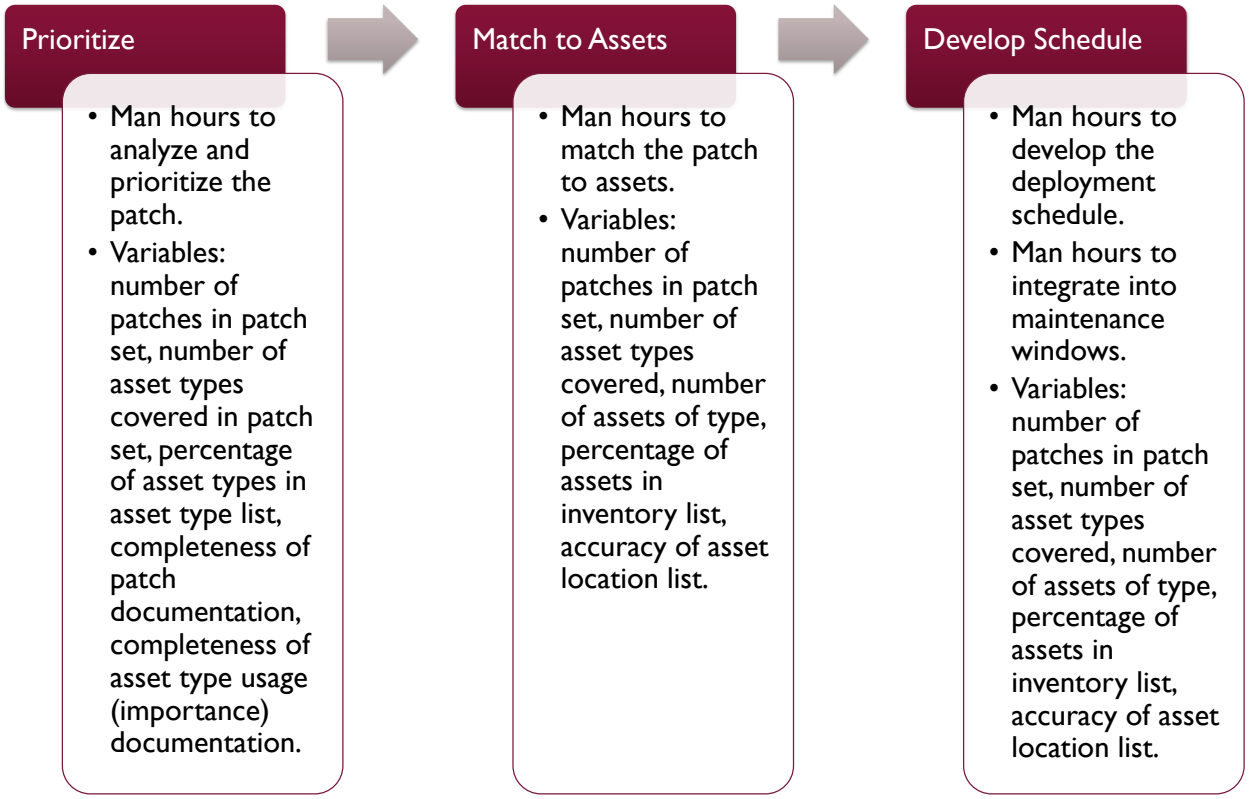


1. **Locate:** Determine the location of the patch/set. This may involve access to a subscription-only support site, or even physical media.
2. **Acquire:** Download or otherwise obtain the patch.
3. **Validate:** Determine that the patch was acquired properly — for example, by checking its hash against a published hash.

Most patches today are downloaded, but there are still occasions where physical media are used. Also, the current status of any maintenance or support licenses, and tracking down the license holder and patch acquisition method, can add considerable costs to this phase of the process if not managed well.

Prioritize and Schedule

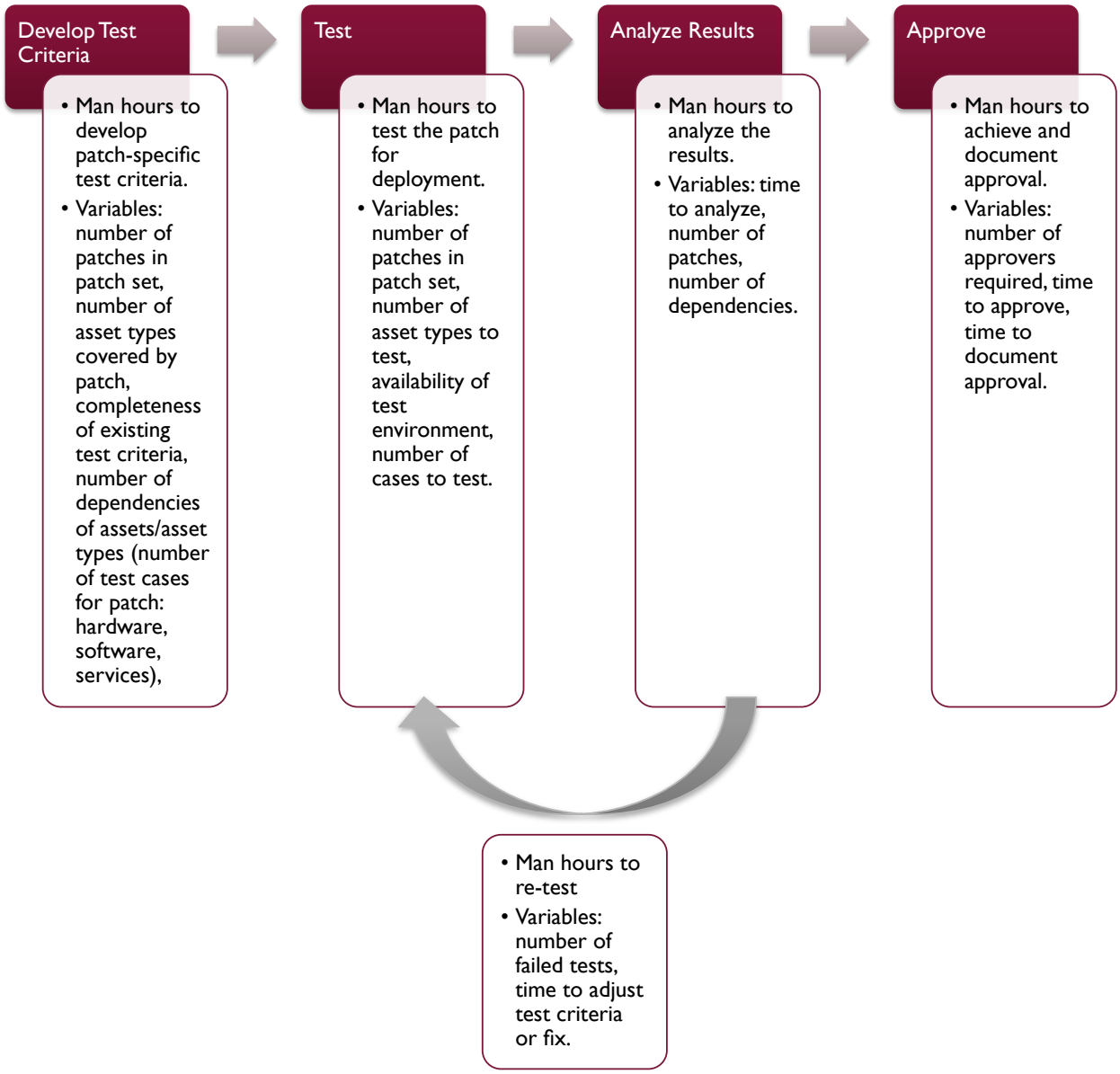
This phase includes three steps to complete prioritization of the patch, match it to existing assets, and schedule deployment.



1. **Prioritize:** Determine the overall priority of the patch. This will often involve multiple teams, especially for security related patches. Priority is usually a combination of factors, including the criticality of the patch, availability of mitigating options (workarounds/shielding), business needs or constraints, and importance of assets covered by the patch. The costs involved vary based on the quality of patch and asset documentation. For example, a highly critical database security flaw may translate into a lower priority for deployment if the specific configuration of the database server is less vulnerable, the server is of low importance, or it is highly protected with alternative security controls.
2. **Match to assets:** After determining the overall priority of the patch, match it to specific assets to determine deployment priorities. This will directly affect the deployment schedule. Again, poor documentation of assets will result in higher analysis costs.
3. **Develop the schedule:** Now that the priority of the patch is established and matched to specific assets, build out the deployment schedule. As with the other steps, the quality of documentation is extremely important. The schedule also needs to account for any maintenance windows, and may involve multiple stakeholders as it is coordinated with business units or application/platform owners.

Test and Approve

This phase is more complex, depending on the degree of testing performed, which also varies greatly based on asset and asset type. It consists of four steps with a sub-cycle to account for different test cases and failed tests:



1. **Develop test criteria:** Determine the specific testing criteria for the patches and asset types. These should include installation, operation, and performance. The depth of testing varies, depending on the value of the platform and the nature of the patch. For example, test criteria for a critical server/application environment might be extremely detailed and involve extensive evaluation in a lab. Testing for a non-critical desktop application might be limited to installation on a standard image and basic compatibility/functionality tests.
2. **Test:** The process of performing the tests.
3. **Analyze results:** Review the test results. In most cases, you will also want to document the results in case of problems later.

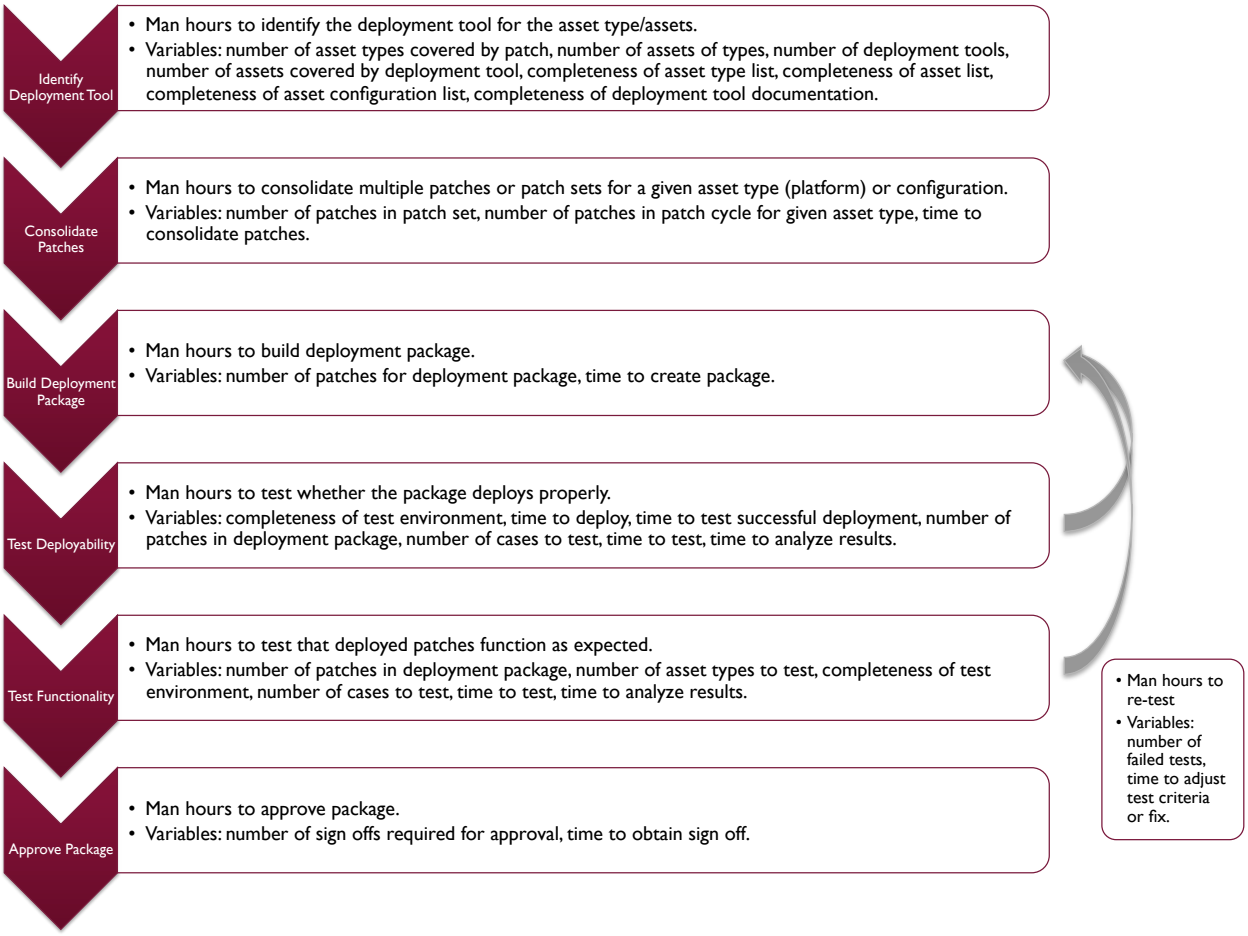
4. **Approve:** Formally approve the patch for deployment. This often involves multiple individuals from different teams, and the time to gain approval may be a cost factor.

This phase also includes another sub-cycle if additional testing is needed due to a failed test, or a test that reveals other issues. This may involve adjusting the test criteria, testing environment, or other factors to achieve a successful outcome.

There are a number of other factors that affect testing costs and effectiveness. The availability of proper test environments and tools is obvious, but proper documentation of assets, especially servers and applications with complex dependencies and functional requirements, is also clearly important.

Create and Test Deployment Package

While some patches are deployable out of the box or manually, others need to be built into deployment packages for distribution. Organizations use a wide variety of deployment tools to distribute patches, each with different requirements and capabilities. In these six steps we identify the proper tool, build the deployment package (if needed), and test the package for expected behavior. Not all patches involve deployment tools, so this phase might be completely skipped for manual deployments.

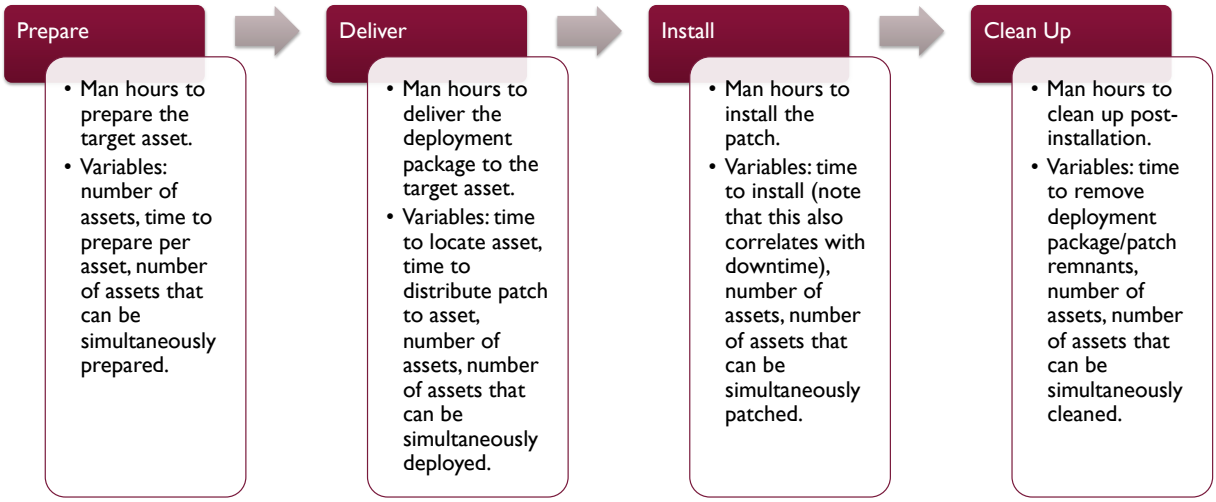


1. **Identify deployment tool:** Determine which tool (or tools) will be used to deploy the patch. Usually this is based on platform, but there are often exceptions, especially for servers and multi-patch bundles.
2. **Consolidate patches:** Pull together individual patches that will be bundled into a single deployment.
3. **Build deployment package:** Create the deployment package. The effort/cost of this task varies based on the tools involved, platforms covered, number of patches, and overall complexity.
4. **Test deployability:** Using the deployment tool, install the package on test systems to confirm it deploys properly.
5. **Test functionality:** Make sure the patch still functions correctly. This is not as in-depth as testing in the *Test and Approve* phase, but is to confirm that the patch still functions properly after being deployed in a package.
6. **Approve package:** Gain formal approval to proceed with deployment.

As with the other phases that involve testing, there is a sub-cycle if any of the tests fail.

Deploy

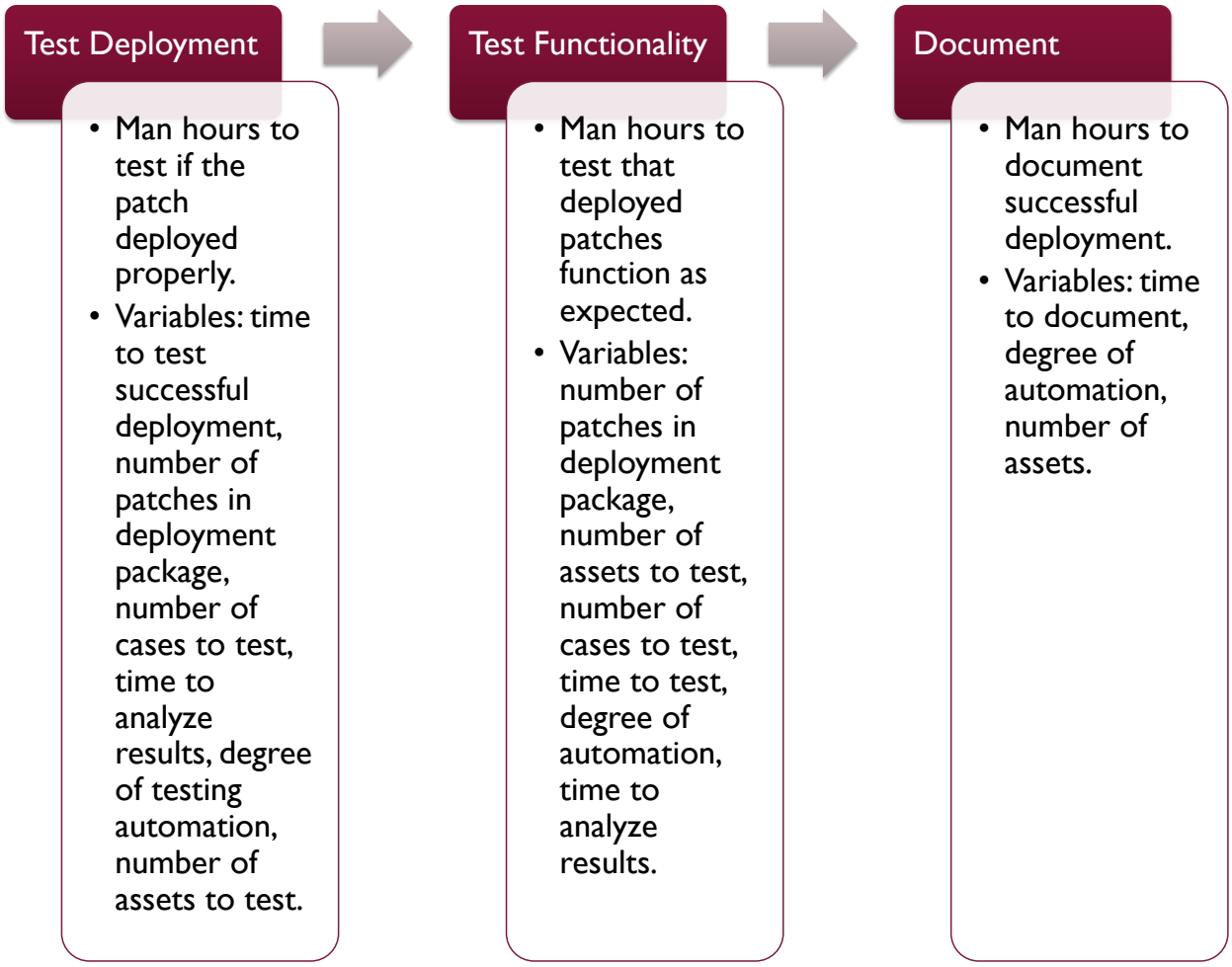
There are four steps to deploy a patch, which include preparing the target system, delivering the patch, installing it, and cleaning up.



1. **Prepare:** Prepare the target asset for the patch. This could involve activities such as rebooting, logging in with administrative credentials, backing up, putting applications into maintenance mode, and so on.
2. **Deliver:** Get the patch or deployment package on the system for installation. This could range from pushing a deployment package from an administrative console, to physical delivery of installation media with a technician to a remote location with low bandwidth connections.
3. **Install:** Install the patch or deployment package.
4. **Clean up:** Remove any temporary files or other remnants from the patch installation, and return the system to functional status.

Confirm Deployment

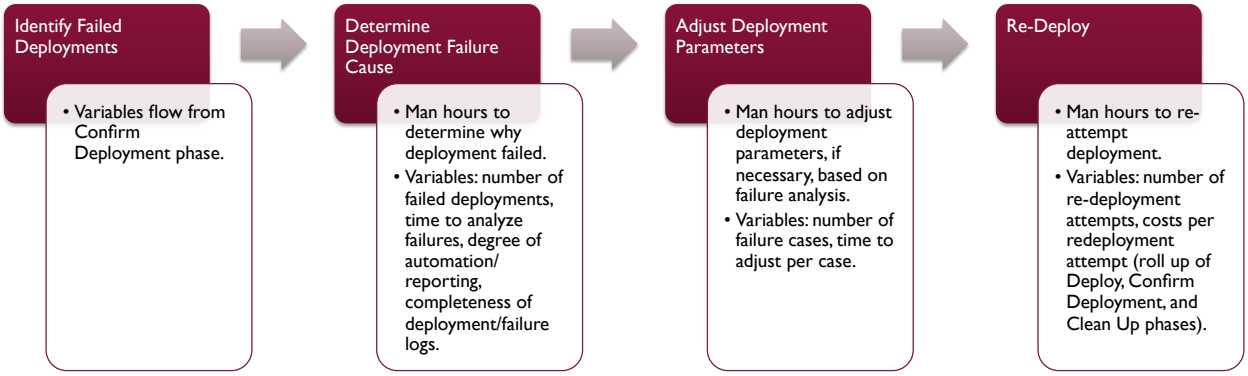
This is the last testing phase, where you confirm that the patch successfully deployed and is functioning properly. For simple updates this could be very superficial, while for major application updates it might involve nearly as much effort as initial testing.



1. **Test deployment:** Test to confirm that the patch deployed. Could involve use of external tools, such as configuration or vulnerability scanners.
2. **Test functionality:** Test that the target asset is functioning properly in general, and that the patch delivered expected functionality.
3. **Document:** Document successful deployment. This is important for compliance reasons, as well as for keeping your asset configuration lists current.

Clean Up

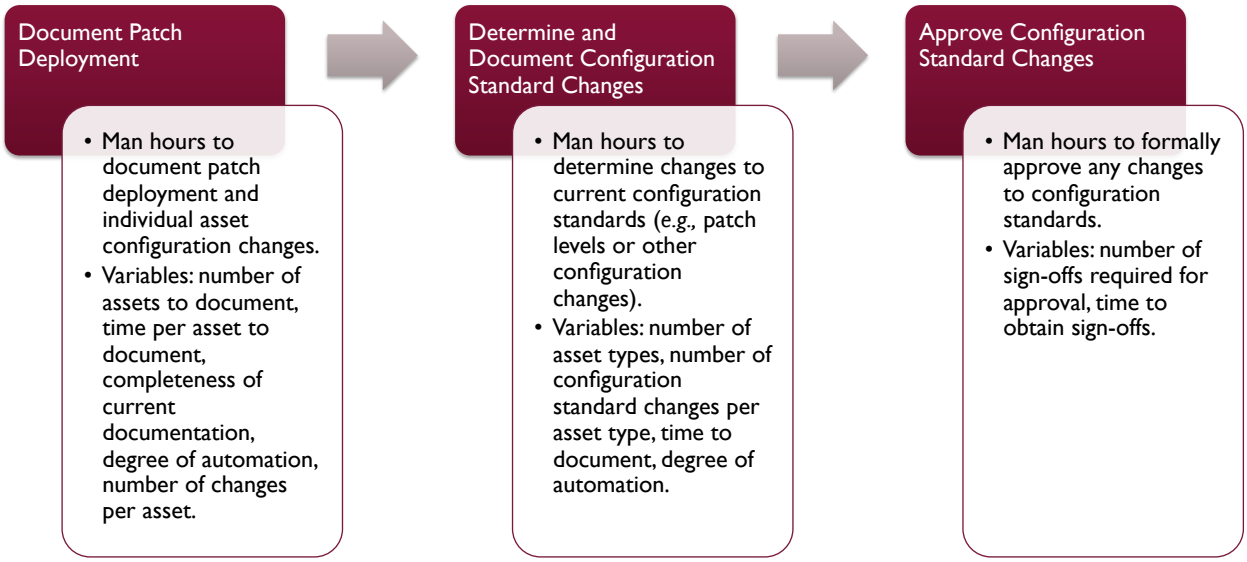
This phase consists of four steps involved with handling failed deployments.



1. **Identify failed deployments:** This flows directly from the *Confirm Deployment* phase.
2. **Determine deployment failure cause:** Perform an analysis to determine why the patch or deployment package didn't install properly. This could be as simple as a system being shut down during installation, or as complicated as undocumented dependencies and unexpected configuration parameters.
3. **Adjust deployment parameters:** Determine what's needed to achieve a successful deployment. Although it's not reflected here, additional testing may be required.
4. **Re-deploy:** Attempt to reinstall the patch. This initiates the installation sub-cycle which returns to the *Deploy* phase.

Document and Update Configuration Standards

This phase includes documentation of successful patch deployment, any specific system configuration changes, and updates to configuration standards.



1. **Document patch deployment:** Document the details of which systems were patched, and the patches applied. This is increasingly important for compliance reasons, especially with security related patches. For large numbers of assets, such as desktops, some level of automation is clearly important here.
2. **Determine and document configuration standard changes:** Some patches, such as those which increase version levels or fix security flaws, will affect any configuration standards for the asset or asset type being patched. For systems such as workstations which are frequently deployed from standard images you may need to also update the image, or update deployment processes to apply the patch on new deployments before distribution.
3. **Approve configuration standard changes:** Obtain the appropriate approval for any updates to configuration standards.

The Metrics Model

Introduction

We've designed this model to be as intuitive as possible while still capturing the necessary level of detail. The model collects an inclusive set of potential patch management metrics, and as with the patch management process we strongly encourage you to tune your usage to fit your own environment.

Because the model includes so many possible metrics, we've color coded key metrics to help prioritize:

Key	The most important metrics in a given category. Using only key metrics will provide a rough but reasonably accurate overview of costs. These are the important useful metrics for determining costs and operational efficiency, and can be reasonably collected by most organizations.
Valuable	Metrics that are valuable but not critical for determining costs and efficiency. They provide greater accuracy than key metrics alone, but take more effort to collect.
Standard	Detailed metrics to help with deep quantification of a process, but these are either less important or more difficult to quantify. They may be more difficult to collect, or might involve complex interdependencies with other metrics.

Using key metrics alone will provide a reasonable picture of your patch management costs, operational efficiency, and program effectiveness, but factoring in valuable metrics, or valuable & standard metrics, will provide greater detail.

How to Use the Model

For most organizations, we recommend you first identify the platform/asset type/process to evaluate, and then match it against the patch management process before delving into collecting individual metrics. This serves two goals:

- First, it helps document your existing process. Since all the metrics in the model correlate with steps in the patch management process, you'll need this to begin quantifying your costs.
- Second, you may find that this identifies clear deficiencies in your current process, even before evaluating any metrics.

While the model may be used to evaluate patch management for all patching activity within an organization, it's really designed to focus on a specific patching process. Most organizations follow very different processes for different platforms and asset types, such as desktop operating systems vs. database servers vs. network hardware. They are also typically managed by different teams using different tools.

For the remainder of the model's description we will assume you are measuring a specific process, and not evaluating an entire program. The metrics we include are far too detailed to apply across multiple processes, teams, and platforms in any single evaluation. Although one way to measure total costs for all patching activities is to make detailed measurements for all the individual processes and then combine them (subtracting out overlapping efforts), this isn't

realistic for most organizations. Since measuring total program costs is also important, we have included a section on adapting the model after the detailed description.

After identifying the patching process to measure, you'll identify the roles/people involved in the process, as well as initial fixed costs. Then walk through each step in the process, quantifying the individual metrics. Metrics vary for any given patch, even on a single asset type, so *enter the average cost for any given step*. Then multiply that by the number of patches over a given time period to determine total cost.

If you do have the ability to fully quantify costs for individual patches you'll get a more accurate result, but this isn't realistic for most organizations. That said, with the right tools and automation you may be able to come extremely close for certain processes. You may also find it useful to individually quantify certain patching activities for efficiency spot checks, or after process changes.

Most of the metrics in this model are in terms of staff hours or ongoing full-time equivalents; others are hard costs (e.g., licensing fees, test equipment, etc.). Throughout the model we also collect counts for certain activities — *e.g.,* the number of patches, or the number of assets to patch. While these are not used to generate cost values, we find them extremely useful in examining process efficiency and effectiveness.

Define the Asset Type (or Program)

The first step is to determine which asset type/platform to measure. Alternatively, you may decide to estimate overall program costs using the recommendations that follow the detailed model description.

It's important to understand your goals before using the model; the more granular your definition, the more accurate your cost metrics. For example, measuring the costs for an entire database platform will be less accurate than for a single database/application stack, since testing requirements, maintenance windows, and other factors vary greatly as you move from instance to instance. Platforms that are more standardized, such as desktop operating system deployments, can be more accurately combined since they are managed collectively.

Choose the level of granularity that best meets your goals. In some cases, you may want to measure overall program costs and efficiency. It might not be as precise as a single platform analysis, but as long as you measure consistently, with measurements you can track over time, you will be able to compare the maturity of different areas. In other cases, you might want to compare patching costs of specific applications in order to estimate their Total Costs of Ownership. For major asset types, you might run an ongoing program to measure detailed patching costs in different phases to identify efficiency issues and problem hot spots over time.

We've developed the model with extreme granularity to give you the flexibility to adapt it for a variety of needs within your own organization.

Asset type/platform	
Description	
Number of assets of type	

Define Roles

Costs in the model are primarily measured by the staff hours involved in patch management. To correlate this with financial costs, we need to define the various roles involved with different stages of patching.

Roles (and titles) vary greatly between organizations, so we list some representative examples.

Role	Description	FTE	Cost per FTE (Hourly)	Annual Cost
Monitor				
Risk/Security Assessment				
Test				
Patch Management/ Deployment				
System/Asset Management				
Audit/Compliance				
Other				
Other				
Other				

Since the entire cost model relies on understanding the hourly costs of manpower, these are all key metrics. In a small organization these roles might all be filled by a single individual.

Determine Non-Phase Program Costs

These are costs associated with patch management, which aren't specific to any particular phase of the process. Some of these are for third party tools which need to be prorated by how much of their usage is for the asset under evaluation. Not all organizations use all of these tools, and some may use additional tools, so we've presented some common options.

Variable	Cost	% Dedicated to Asset
Support or maintenance license		
Patch management/deployment system		
Time to configure and deploy patch management/deployment tool on target assets		
Vulnerability assessment tool		
Configuration management tool		
Test environment/tools		
Documentation management		
Patch notification service		
Other		

Determine Individual Phase Costs

We now advance through each phase and step in the patch management process, collecting individual metrics.

Phase 1: Monitor for Release/Advisory

Step 1: Identify Asset Types

Variable	Notes	Responsible Role
Number of asset types in program	The number of different hardware/ software platforms in the patching program being evaluated. This could be everything in the organization, or a subset, such as application servers.	NA
Initial time to identify asset types		
Time to update asset type list		
Asset list updates per year		NA

Step 2: Identify Advisory Sources

Variable	Notes	Responsible Role
Number of advisory sources in program		NA
Initial time to identify sources		
Initial time to match sources to asset types		
Time to update source list and match to asset types	The time dedicated to keeping the source list updated.	
Source list updates per year		NA

Step 3: Monitor for Advisories

Variable	Notes	Responsible Role
Time to monitor sources for advisories (per release)	The average time per advisory to look for, acquire, and read the advisory.	
Number of advisories per year		NA
Time to identify asset type per patch	The time it takes, on average, to identify the asset type affected by a patch on release.	
Number of patches per year		NA

Phase 2: Evaluate

Step 1: Match to Asset Type

Variable	Notes	Role
Number of patches in set	Most patch sets include a number of patches. Since the number of patches strongly affects testing and deployment, it's important to track.	NA

Variable	Notes	Role
Number of assets matching patch set		
Time to match patches in set to asset types	The time to determine if the patches affect your organization at all.	
Number of patches in set matching assets	Carries through to all steps in this phase.	

Step 2: Determine Nature

Variable	Notes	Role
Time to determine nature from patch documentation	For example, "security update" or "general bug fix". Throughout the model we track the time to find information in various documentation to determine the effectiveness of the documentation and its impact on time/ resources.	
Time to validate/evaluate nature	The total time to evaluate and determine the nature of the patch.	

Step 3: Determine Relevance and Priority

Variable	Notes	Role
Time to determine priority and criticality from patch documentation		
Time to determine priority and criticality of matching assets or asset type in asset list	This is the time to check internal documentation (the asset list) and determine the importance of each asset being patched. For widely distributed platforms this is generally by asset type, while it may be per asset for critical systems, such as the customer transaction system.	
Time to determine overall priority (based on priority of patch and priority of asset)	This is the initial assessment of the patch to determine if it's a priority for further acquisition, evaluation, and deployment. The detailed security and priority analysis occurs in phase 4.	

Step 4: Determine Dependencies

Some patches, especially for enterprises software, have complex sets of child dependencies (software that relies on them) or parent dependencies (software they rely on). Since patches may affect upstream and downstream functionality, or may require other patches of components on the same system, we determine that here.

Variable	Notes	Role
Time to identify dependencies in patch documentation		
Time to match dependencies to asset types		
Time to evaluate dependencies		

Step 5: Workarounds and Shielding

Although workarounds and shielding are part of a separate process, it's still important to evaluate the amount of time spent on these activities by the patch management team.

Variable	Notes	Role
Time to identify workarounds and shielding in patch documentation		
Time to identify undocumented/ alternative workarounds and shielding		
Time to match workarounds and shielding to assets		
Time to initiate workaround and shielding processes	This is the time to initiate any external process, although it can also be used if you already integrate shielding and workarounds into your core patch management process.	

Phase 3: Acquire

Step 1: Locate

Variable	Notes	Role
Time to locate patch		

Step 2: Acquire

Variable	Notes	Role
Time to acquire	The staff hours to acquire the patch.	
Acquisition channel costs	Any special bandwidth or physical delivery charges.	

Step 3: Validate

Variable	Notes	Role
Time to validate	Confirm all components were acquired properly, and verify hashes (if available).	

Phase 4: Prioritize and Schedule

Step 1: Prioritize

Variable	Notes	Role
Time to determine number of patches in set documentation	Flows from Phase 2	
Number of patches in set	Flows from Phase 2	NA
Time to determine asset types patched from patch documentation	Flows from Phase 2	
Number of asset types matching patch set	Flows from Phase 2	NA
Time to determine asset types matching patch set within organization	Flows from Phase 2	
Number of patches in set matching organization asset types		NA
Time to determine priority of patch from patch documentation		
Time to determine priority of asset types from organization documentation		
Time to perform security assessment/ prioritization (if required)	Included since security assessments of patches are often performed separately from the platform manager's assessment.	
Time for internal evaluation and determination of patch priorities	Overall time to determine the priority of the patches, based on importance of the asset types patched.	

Step 2: Match to Assets

Variable	Notes	Role
Number of assets of asset type		NA
Time to match assets to patched asset types	Completeness/currency of asset list documentation is key.	
Time to determine network and/or physical locations of assets to be patched	Completeness/currency of asset list documentation is key.	

Step 3: Schedule

Variable	Notes	Role
Number of assets to be scheduled		NA
Time to develop and document patching schedule		

Phase 5: Test and Approve

Step 1: Develop Test Criteria

Variable	Notes	Role
Number of patches in set	Flows from Phase 2	NA
Number of assets/asset types requiring individual test cases	For some asset types, general testing is possible (*e.g.,* standard desktop operating system images). For others, such as application/database stacks, each individual asset will require its own test cases.	NA
Time to identify and locate existing test cases/criteria		
Time to develop new test cases/ criteria	This includes functional and performance testing, and may include installation testing. More robust deployment/installation testing is performed in the next phase.	
Time to identify test-related dependencies	For some patches, there are requirements to test dependent applications/functions.	
Total number of test cases/criteria		
Total time to document test criteria		

Step 2: Test

Variable	Notes	Role
Time to establish test environment and assemble testing resources	Building a test environment may take considerable time and involve material costs.	
Time to perform tests		
Time to document test results		

Step 3: Analyze Results

Variable	Notes	Role
Number of test results to analyze		NA
Time to analyze test results	This is the total time, which may involve multiple staff/roles.	

Step 4: Approve

Variable	Notes	Role
Number of approvers required	There is generally a correlation between the number of people involved in an approval process, and the time it takes to approve.	
Time to approve		
Time to document approval		

Test/Analyze Cycle

Variable	Notes	Role
Number of failed tests		NA
Time to adjust test criteria, asset, or patch		
Number of re-test cycles	When re-testing is required, costs are cumulative across all cycles.	

Phase 6: Create and Test Deployment Package

Step 1: Identify Deployment Tool

Variable	Notes	Role
Number of asset types to patch	Flows from Phase 2	NA
Number of deployment tools		NA
Time to identify correct deployment tool for asset and patch	Complete documentation is important for minimizing this time.	

Step 2: Consolidate Patches

Variable	Notes	Role
Number of patches to consolidate for asset types	For scheduled patching, this could involve multiple patches and patch sets.	NA
Time to consolidate patches		

Step 3: Build Deployment Package

Variable	Notes	Role
Number of patches for deployment package		NA
Time to create package	The time to create the actual deployment package.	

Step 4: Test Deployability

Variable	Notes	Role
Number of deployment packages to test		NA
Number of deployment conditions/ targets to test		NA
Time to prepare test environment		
Time to test		
Time to analyze test results		

Step 5: Test Functionality

Variable	Notes	Role
Number of deployment packages to test		NA
Number of deployment conditions/ targets to test		NA
Time to prepare test environment		
Time to test		
Time to analyze test results		

Step 5: Approve Package

Variable	Notes	Role
Number of sign-offs required for approval		NA
Time to obtain approval		
Time to document approval		

Test/Analyze Cycle

Note that these apply to both functional testing and deployability testing:

Variable	Notes	Role
Number of failed tests		NA
Time to adjust test criteria, asset, or patch		
Number of re-test cycles	When re-testing is required, costs are cumulative.	

Phase 7: Deploy

Step 1: Prepare

Variable	Notes	Role
Number of assets to prepare for patch		NA

Variable	Notes	Role
Number of assets that can be simultaneously prepared		NA
Time to locate target assets	This time tends to be considerable in less mature organizations, and should be promoted to a key metric in such environments.	
Time to prepare target for patch	This includes gaining access to the machine, performing backups, or other pre-patch activities.	

Step 2: Deliver

Variable	Notes	Role
Number of assets to which patch can be simultaneously deployed		NA
Time to deploy	The time to deliver the patch to the target.	

Step 3: Install

Variable	Notes	Role
Time to install	This is the total time to install the patch/package on all target assets, not just a single asset. If deployment time per asset is consistent, you can take the average time per asset and multiply by the number of assets.	NA

Step 4: Clean Up

Variable	Notes	Role
Time to remove deployment package and patch remnants	Costs associated with anything beyond cleaning up the patch components are included in the Clean Up phase.	

Phase 8: Confirm Deployment

Step 1: Test Deployment

Variable	Notes	Role
Number of deployment packages to test		NA
Number of assets to test		NA

Variable	Notes	Role
Number of assets that can be simultaneously tested		NA
Time to test		
Time to analyze test results		

Step 2: Test Functionality

Variable	Notes	Role
Number of deployment packages to test		NA
Number of assets to test		NA
Number of assets that can be simultaneously tested		NA
Time to test		
Time to analyze test results		

Step 3: Document

Variable	Notes	Role
Number of failed deployments		
Time to document	It's especially important to document the number (and identification) of failed deployments for the next phase.	

Phase 9: Clean Up

For unsuccessful deployments, this phase kicks off a re-deployment cycle that will normally include the Deploy, Confirm Deployment, and Clean Up phases until all target assets are patched.

Step 1: Identify Failed Deployments

Variable	Notes	Role
Number of failed deployments	Use an annual average unless you are measuring a single deployment.	NA
Time to identify failed deployments		NA

Step 2: Determine Deployment Failure Cause

Variable	Notes	Role
Time to locate information source or log with failure information	Poor logging will significantly increase investigative time to identify the cause of the failure.	
Time to determine failure cause		

Step 3: Adjust Deployment Parameters

Variable	Notes	Role
Number of different failure modes		NA
Time to adjust deployment parameters	The time to change the asset or the deployment to achieve a successful deployment.	

Step 4: Re-Deploy

Variable	Notes	Role
Number of re-deployment attempts		NA
Total cost per re-deployment	This is a combination of the Deploy, Confirm Deployment, and Clean Up phases.	

Phase 10: Document

In some cases, tools generate reports that meet many documentation requirements.

Step 1: Document Patch Deployment

Variable	Notes	Role
Number of assets, asset types, and patches to document		NA
Time to document	This is a key aspect, as it is often required for compliance audits or reporting.	NA

Step 2: Determine and Document Configuration Standard Changes

Variable	Notes	Role
Number of asset types requiring configuration standard documentation changes		NA
Number of configuration changes		NA
Time to identify and analyze current configuration standards.		
Time to determine configuration standard changes		
Time to document	This is a key metric due to its role in compliance. Although still important in organizations without compliance mandates, it can be reduced to Valuable in such situations.	

Step 3: Approve Configuration Standard Changes

Variable	Notes	Role
Number of sign-offs required for approval		NA
Time to obtain sign-offs		

Program Metrics

These metrics don't correlate directly to costs, but are useful in evaluating overall efficiency and effectiveness.

The Center for Internet Security's Consensus Metrics

The Center for Internet Security maintains a list of consensus metrics for benchmarking entire security programs at http://www.cisecurity.org/securitymetrics.html. These include a section on patch management, and we recommend using these metrics for both security and non-security evaluations:

Metric	Definition/Notes
Patch Policy Compliance	The number of assets patched to current policy or configuration management standards, divided by the total number of assets. This helps evaluate the overall effectiveness of your program, and can be measured with vulnerability and configuration scanning tools.
Patch Management Coverage	The number of assets included in a formal patch management process or automated system, divided by the total number of assets. This helps evaluate both the efficiency of your program (under the assumption systems under a process are more efficiently managed), as well as its effectiveness (how well the organization identifies assets and asset types and incorporates them into a management process).
Mean Time to Patch	The time from the release of an advisory until successful patch installation. This measures how effective the organization is at updating systems.

Combine and Analyze Costs

The final step is to combine, then analyze, the various costs and metrics. Convert time-based metrics into dollar values by correlating back to the responsible roles, which should have per-hour costs assigned from the roles section. The non-time/cost metrics are included to help with efficiency and effectiveness analysis, while the program metrics provide a high-level overview of your program.

You will then have financial costs for each phase of the patch management process, personnel costs, and resource/tools costs. You will also have the total time spent on each phase of the process, and for the steps in each phase.

Your analysis will then vary depending on your goals. Some possibilities include:

- Tracking patching costs for a specific program/asset type over time for trending.
- Analyzing a patching process to identify specific inefficiencies, such as one part of the process dominating time and resources (*e.g.,* learning that the lack of a dedicated test environment costs more over time than building an environment).
- Comparing potential costs of different software platforms based on historical modeling (number of patches, frequency of patches, failed/bad patches, completeness of documentation, and so on).

- Evaluating process changes for their potential cost, as well as possible efficiency changes.

Adapting the Model for Measuring a Complete Program

There are two approaches for measuring patching across the entire inventory of asset types. In one, you perform a full metrics evaluation in different areas and then roll up the results. In the other, you choose only one or two key metrics for each patching phase, measure those, and then roll up the totals.

Full Evaluation

This is essentially using the complete model in different areas (desktops, servers, etc.), and then rolling up the total costs. This is likely only practical in mature organizations with a high degree of automation to assist in metrics collections. Here are a few suggestions for adapting the process:

- Focus heavily on identifying roles and the amount of time they dedicate to patching. Break this out by different patching processes (servers vs. desktops). This alone will give you a good idea of the resources dedicated to patching different asset types.
- When collecting tool and licensing costs, also inventory which asset types are covered by the various tools. You may find that you've already licensed a tool that could work in other areas, which isn't being used due to lack of communication between IT groups.
- Issue guidelines on how to collect the metrics. Although they are fairly self-explanatory, different teams almost certainly use different process and will fit (or interpret) the model differently.
- Use automation to collect the metrics as much as possible (for example, reports from tools). This model is very granular, and it will be difficult to collect this volume of metrics without some degree of automation.

Key Metrics Evaluation

This method focuses more on key metrics for those organizations without the resources for a full evaluation. It's more practical across a wider range of organizations, but far less granular.

- As with a full evaluation, pay particular attention to your role analysis and costs, since these alone will give a good picture of your program.
- Also pay close attention to tools and licensing costs and usage.
- Then estimate the metrics for each phase of the process, using only 1-2 key metrics. We suggest:

Phase	Cost Metrics	Other Metrics
Monitor for Release/Advisory	Time to monitor sources for advisories	Number of advisories per year
Evaluate	Time to determine overall priority (based on priority of patch and priority of asset)	None
Acquire	None	None
Prioritize and Schedule	Time to develop and document patching schedule, Time to perform security assessment/ prioritization (if required), Time for internal evaluation and determination of patch priorities	None
Test and Approve	Time to perform tests	Number of re-test cycles

Phase	Cost Metrics	Other Metrics
Create and Test Deployment Package	Time to create package	None
Deploy	Time to install	Number of assets to prepare for patch
Confirm Deployment	Time to Test (Deployed)	Number of failed deployments
Clean Up	Time to identify failed deployments, Total cost per re-deployment	Number of re-deployment attempts
Document and Update Configuration Standards	Time to document	None

The Costs of Maintenance Windows and Predictable Patches

Although some patches are released on a predictable schedule, many more appear somewhat randomly as vendors fix the inevitable flaws that appear after products are released. For non-security patches, unless a system is currently experiencing problems, these updates can be scheduled for installation during a set maintenance window. As we discovered in the project survey, most organizations establish formal maintenance windows for assets to apply major updates and perform regular maintenance tasks while minimizing any disruption to operations.

On many platforms the bulk of unscheduled updates are security patches to fix open vulnerabilities, and occasionally we also see updates for functionality or reliability flaws that aren't initially apparent, but require immediate attention. These patches, depending on their risk and priority, often can't wait until the next scheduled maintenance window. While patching during scheduled maintenance windows clearly disrupts operations (both business and IT) less than unscheduled patches, it's difficult to measure the potential additional costs of patching outside the schedule.

During the review process, multiple contributors noted the lack of any metrics around maintenance windows. Since this model is focused on measuring the costs associated with patching, it's beyond its current scope to measure the costs of downtime or productivity loss. Also, total downtime may be equal for a scheduled or unscheduled patch, and can equally affect productivity in terms of the time involved in the patching activity, so it's extremely difficult to accurately model. An IT administrator may need to put off other projects to deal with an unexpected security patch, but if this doesn't impact the total hours worked, there's no way to measure the associated financial costs.

For organizations interested in understanding the role of maintenance windows, we suggest the following approaches:

- Overtime for off-hours updates are normally included in project costs for scheduled updates and maintenance windows, but will not be formally budgeted for unscheduled patches. Even for salaried employees, these hours can be tracked and costs calculated for unbudgeted vs. budgeted hours. Although total salary costs don't change, you are still calculating the costs associated with unplanned patches that could otherwise be dedicated to other activities.
- For unplanned patches, many of the individual metrics may rise. For example, if a test environment isn't prepared, or personnel are not available, this could add to both the direct time/cost metrics (such as time to perform activities in the phases), or the program metrics (time to patch). This allows you to compare the costs of various kinds of patches, including scheduled vs. unscheduled.
- The model will also help improve efficiency in handling patches. You'll have a better idea of how many scheduled vs. unscheduled patches to expect over time, and gain insight into efficiency issues in handling the different kinds of patches.

Although anyone with experience in patch management intuitively understands the difference in disruptions between scheduled and unscheduled patches, and the value of maintenance windows, this model doesn't make any blanket

assumptions or judgements as to advantages or disadvantages. Instead, the model is designed to allow you to measure and compare these costs yourself, with the caveat that it doesn't measure business disruptions, productivity costs, or user frustration levels.

Conclusions and Next Steps

Patch Management Is Still Difficult... Mostly

Patch management is one of the most fundamental functions of IT departments, yet in our research we discovered it remains one of the biggest pain points for many organizations. Despite decades of experience, a combination of vendor inconsistency, conflicting priorities, and a lack of industry standards make patch management more difficult than it needs to be for many IT practitioners. We want to constrain our patch management costs, but lack the tools to measure them, or standard processes to guide our efforts.

There is also wide variance in maturity between technology platforms. Workstation operating systems, likely in large part due to security issues, are generally patched more consistently and effectively than other platforms, such as enterprise applications. But even on workstations, based on our survey results, desktop applications and device drivers are patched far less frequently and effectively than the operating system itself (despite these being major vectors for security exploits).

By providing a granular process framework and metrics model we hope to help organizations better drive process improvements, reduce costs, and increase both efficiency and effectiveness.

Next Steps

This report, and the patch management survey results hosted on the Project Quant site, are the result of months of community research and effort, but are only the beginning. Our goal is to continue this effort to improve both the state of patch management specifically, and the collection of IT metrics in general. Specific next steps include:

- Conduct focused interviews with survey respondents who indicated interest in additional discussions, and publish the (anonymous) results.
- Incorporate the public feedback that we anticipate on release of this document, and use it to refine and improve the model.
- Publish example use cases for the model, covering different kinds of technology assets (servers, workstations, databases) for organizations of various sizes and natures.
- Expand into adjacent research areas, such as building out a model for shielding and workaround costs.
- Continue to engage heavily with the patch management community and solicit their direct involvement in future revisions of the model.
- Research the possibility of benchmarking to better enable organizations to compare their costs to their peers.
- Develop a standard taxonomy for enhanced communication and automation of metrics collection, and work with the vendor community to include these in future products.
- Build a sustainable community dedicated to the improvement and advancement of patch management metrics.

Finally, the authors of this report would like to encourage additional open, independent, community research and analysis projects in IT and security metrics. Utilizing a transparent research process enables new kinds of collaboration capable of producing unbiased results. We are investigating other opportunities to promote open research and analysis, particularly in the areas of metrics, frameworks, and benchmarks.

In closing, we want to encourage readers to help drive further progress on Project Quant goals by visiting the project site at http://www.securosis.com/projectquant and becoming an active community participant. One key way you can help is to provide your own experience to the community by taking the survey featured in this report at:

http://www.surveymonkey.com/s.aspx?sm=SjehgbiAl3mR_2b1gauMibQw_3d_3d

Who We Are

About the Authors

Rich Mogull, Securosis

Rich has over 17 years experience in information security, physical security, and risk management. Prior to founding Securosis, Rich spent 7 years as one of Gartner's leading security analysts, where he advised thousands of clients, authored dozens of reports, and was consistently rated one of Gartner's top international speakers. He is well known for his work on data security technologies and has covered issues ranging from vulnerabilities and threats, to risk management frameworks, to major application security. Rich is the Security Editor of *TidBITS*, a monthly columnist for *Dark Reading*, and a frequent contributor to publications ranging from *Information Security Magazine* to *Macworld*.

Jeffrey Jones, Microsoft

Jeff Jones is a Director in Microsoft's Trustworthy Computing group. In this role, Jeff draws upon his years of security experience to work with enterprise CSOs and Microsoft's internal security teams to drive practical and measurable security improvements into Microsoft process and products. Prior to his position at Microsoft, Jeff was the vice president of product management for security products at Network Associates where his responsibilities included PGP, Gauntlet, and Cybercop products, and prior to that, the corporate McAfee anti-virus product line. These latest positions cap a 20 year hands-on career in security, performing risk assessments, building custom firewalls, and being involved in DARPA security research projects focused on operating system security while part of Trusted Information Systems. Jeff is a frequent global speaker and writer on security topics ranging from the very technical to more high level, CxO-focused topics such as Security TCO and metrics.