WHITE PAPER

Best Practices for Version Management

Abstract

Software version management and version control tools are at the heart of many companies, but the value they provide is often misunderstood, leading to poor implementations, bad practices and missed opportunities.

Additionally, as developer preferences and enterprise requirements have evolved, there is a greater need to balance distributed workflows with centralized visibility, security and support for DevOps.

This white paper promotes some high-level best practices that reflect our experiences with software version management.

PERFORCE

Contents

In the age of "software is eating the world[1]," just about every business is a software business. The way that software is developed, built and deployed has never been more critical to any company's success. After all, that software represents valuable intellectual property and so should be managed and protected as such. However, that treatment isn't always the case.

Introduction

There are many challenges for users of version management systems. Teams have inherited version management tools and processes that have been used for years without serious review. Many different tools may be in use for historical reasons or because of different team or project needs. Products are now more complicated than ever—often including media (video, graphics, audio), open source components, collaboration across teams or businesses—in ways that didn't exist when tools and processes were originally implemented. In some cases, best practices have been compromised due to lack of functionality in tools.

All companies have a need and duty to continually improve their processes to speed up delivery, improve customer satisfaction and, ultimately, deliver value to stakeholders.

As providers of software version management tools and as consultants to software companies, we are often asked for sound advice on software version management best practices—that is, how to deploy software version management for maximum advantage. Answering these requests has created a bounty of direct and indirect version management experience from which to draw. The direct experience comes from being developers and codeline managers ourselves; the indirect experience comes from customer reports of successes and failures with our product (Perforce Helix) and other version control tools.

This white paper reviews the key areas to evaluate when considering version management and offers recommendations for successful deployments. Starting with the basics of good version management through such advanced topics as distributed versioning and security, it supplies a guide to assessing and deploying high-quality, productive and robust software version management.

[1]"Why Software is Eating the World" Marc Andreesen, Wall Street Journal, Aug 2011 http://www.wsj.com/articles/SB10001424053111903480904576512250915629460

Version Management Basics—The Good Check-In

The check-in (often called a commit or a submit) is the central operation of any software version management. It stores a new version of one or more files in a repository. The sum of all check-ins constitutes the history of all content within the repository, which can be used to roll back to a previous state or to trace the origin of a bug or feature within a project.

Bad check-ins slow down productivity—developers will have difficulty understanding how and why changes were made, and partial check-ins may mean a product cannot be built successfully because some changes may have been missed. Good check-ins enable better collaboration and furnish more usable audit trails.

To make the best use of a repository's history, a check-in should follow some or all of these best practices:

- **A check-in is an atomic operation.** All files in a check-in are either committed together or not at all, and no other user can see partial or incomplete changes. As such, a check-in is similar to a database transaction described by its ACID properties: atomic, consistent, isolated and durable. Commit all files that belong to a task in a single operation to keep the project consistent at all times.
- **A check-in should have a single intent.** Each check-in should have a single purpose— for example, fixing a bug or adding a new feature. If a single change makes several independent modifications to your project, it can become difficult to read and to review this change. It will also become complex or time-consuming to back out one of these modifications. A check-in is not a backup of your current state of your local files, even if it occurs at the end of the day.

- **Keep the scope of a check-in narrow.** By breaking down a larger task into smaller chunks, you can more readily understand and review the intent of this change. For example, you could break a task into infrastructure and refactoring tasks before making user-visible changes. Keeping the scope narrow also makes it easier to back out a check-in.
- **The description should communicate the reason for the change.** Each check-in should have a description that explains the why, but not necessarily the how, regarding the change. How is usually deducible by comparing the file contents before and after the change. A good description makes it easier for a reviewer and for you later to understand the purpose of the check-in.
- **Each check-in should be consistent.** The project should be able to build and pass its test cases before and after the check-in. If you notice a bug and want to track down the change that introduced the bug, you usually reset your working environment to a previous time (either by hand or through some bisect facility) to verify the bug is still there. If previous changes do not even build, tracing down a bug becomes a lot more difficult.
- **Each check-in should be complete.** By providing test cases and at least stubs for new APIs, every check-in should be usable by any other member in the team without breaking his or her build. A complete check-in is easier to propagate between branches. An incomplete check-in of an API, for example, might build locally in your work area and pass all tests but could break in another team member's work area.
- **Use pre-flight reviews.** A good check-in is often reviewed before committing it to a shared repository, either through a review system or a pull-request. Reviews are a great way to get another perspective on a change and to improve code quality. Code reviews are also useful to increase code awareness within the team, enhancing the team's productivity through reuse and quality of output.
- **Ensure a check-in is traceable.** For security and auditing, you must store the author of the change as well as additional information such as the reviewer's comments. A check-in is also often associated with a job or issue tracker item.
- **The ideal check-in is reversible.** Following the best practices highlighted here will ensure that each check-in can be backed out again if necessary. The best pre-check-in reviews and build tests cannot always prevent unintended side effects that appear in later testing. In such cases, it might be necessary to back out a check-in, which returns the state of the project to an earlier time. This operation usually preserves history as well, so that the change can later be re-applied or analyzed and fixed as necessary.

In Summary

Each team tends to have rules and guidelines about what a check-in should look like. Compare these guidelines against the best practices we have highlighted here and see if there is room for improvement.

Good-quality check-ins can improve your project, making you more productive and successful.

Define a standard process for check-ins that includes these steps

- Have all check-ins be atomic, complete, consistent, traceable and with a single intent
- Make changes visible through frequent check-ins
- Consider how you would use the comments in the future
- Review code before committing to the mainline
- Make check-ins reversible

Branching

In an ideal world, no branches would be necessary. Every check-in is good, there are no bugs, you don't get delays or slipups, there is only one release and customers always upgrade immediately to the latest version.

The reality in software development is normally different. One release is rarely enough, so you have to make regular releases. In the real world, these releases have bugs, so you need to stabilize individual releases. These bugs delay your releases, so you need to consider staggered releases by introducing stages. In the meantime, development of the next release continues, but portions of it might not get finished on time, so you have to find a way to run several development threads in parallel.

You can address all these issues in software development by using branches; this is why almost all version management systems support branches. But branches can introduce another problem: Changes in one branch often have to flow to other branches. Without rules and guidance, propagating changes can lead to instability through merge conflicts, lost updates and unintentional overwriting of existing changes.

Let's look at different branching models to understand the issues with propagating changes and how to address them.

Codeline

The set of files required to produce a piece of software is called a codeline. In our ideal world, there is a single codeline for all developers, often called trunk, main or master. The codeline consists of a range of individual changes.

The head is the latest change checked in (see Figure 1). Individual releases can, in our ideal world, be identified through a simple label on a change.

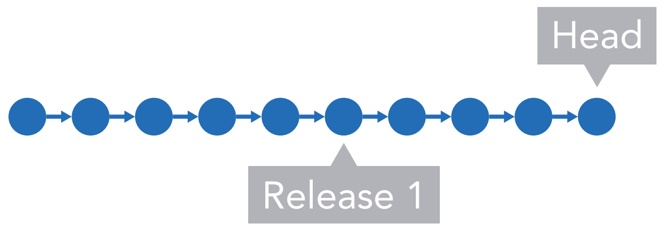

Figure 1: A simple codeline where each dot represents a check-in

Best Practices For Codelines

- ***Define a set of policies for each codeline.***
  Policies could cover, for example, who can make changes, whether each change will be built and verified and what kind of tests are run and when. To prevent confusion, these policies should not change during the lifetime of a codeline. An example of a possible policy for the main codeline would be to ensure its stability so that you will always be able to deliver the product if required.
- ***Each codeline should have an owner.***
  The owner should look after and enforce the policies. The owner is also responsible to resolve ambiguities if there are special cases, for example how to handle an emergency change.

Branch for Releases/Milestones

In the real world, releases are not stable immediately. Fixing a release requires a code freeze to prevent new features from derailing the stabilization efforts. To avoid holding up future development and to avoid changing the policy on your codeline, branch a release codeline, typically at the time of the code freeze (see Figure 2).

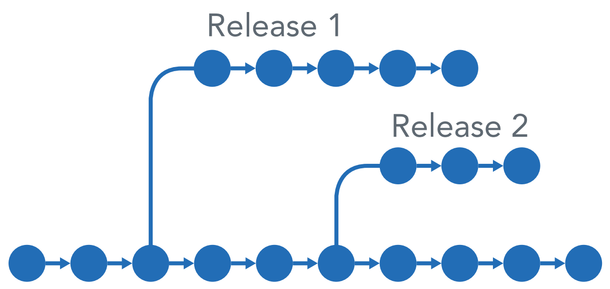

Figure 2: Creating branches for each release

Branching for a release has an additional advantage: It can support several releases at the same time. A bug fix for release 1 does not necessarily apply to a future release, which might have been refactored.

Branching is also useful for milestone branches to create a demo or pre-release version of the product. Milestones often require short-term fixes that should not go into a final product, and a milestone branch is the ideal place to store these fixes.

Best Practices For Release Branches

- ***Create a release branch for each release.***
  Separate release branches make it easier to identify and compare releases and to support several releases in parallel.
- ***Make a release branch instead of imposing a code freeze***
  This step opens up the main branch for future development, speeding up the development cycles.
- ***Define a separate policy for a release branch.***
  Compared to the mainline, release branches have a higher emphasis on tests to ensure stability and often limit permissions to a small group of developers dedicated to stabilization.
- ***Do not reuse release branches for multiple releases.***
  Create a release branch for each distinct product release. This avoids confusion about the location of a particular release.

Staging Branches

For products that have no distinguishable releases or where releases change rapidly, the best practice of creating a separate release branch is too rigid. This is particularly true for busy websites that change often. For these cases, it makes sense to have several reusable stages to allow verification before publishing any updates (see Figure 3).

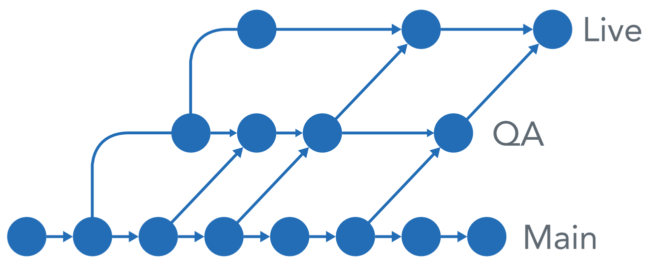

Figure 3: Rapidly changing code can use stages, in this example there's only "QA," for review of changes from the mainline prior to inclusion in the "Live" branch which will be run in production.

Staging decouples QA from the development effort and allows for faster updates as changes can be promoted from the mainline even while verification continues.

Best Practice For Staging Branches

- ***Use staging branches for projects that are continuously updated.***

Development Branches

Developing products in the same shared codeline has advantages. Your colleagues immediately see changes, which improves communication within the team and reduces late surprises before delivery time.

For larger and more complex changes or for experimental work, a single codeline with its policy of always being able to build and pass tests can be too restrictive. Instead of destabilizing the shared codeline, create a separate development branch, which gives developers a space to make incremental changes (see Figure 4).

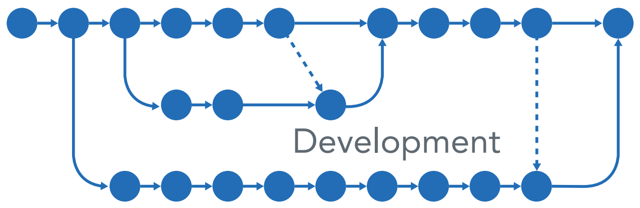

Figure 4: Using team, feature or task branches to enable parallel development

Development branches typically come in two types:

- shared by a small team or used by individuals
- used for a single task or reused for a group of tasks or features

Either model can be useful, and they are often intertwined. For example, you can use a shared reusable integration branch for a larger piece of work from which individual task branches can be created.

Best Practices For Development Branches

- ***Branch on incompatible policy.***
  Instead of imposing a code freeze or disabling builds or tests for a larger development, create a separate development branch.
- ***Avoid long-lasting development in a separate branch.***
  If the development branch diverges too far from the parent branch, integrating the changes into the parent becomes a disruptive and lengthy process.
- ***Rebase development branches regularly.***
  Integrating changes from the parent into the development branch (a "rebase") on a regular basis prevents painful merge conflicts and increases the stability of the development branch.

Mainline Model

An important consideration for these branching strategies is the existence of a main codeline from which all branches are derived. This codeline is often known as the mainline. Having a common mainline allows all contributors to see all changes as early as possible to avoid any last-minute "merge hell" where the mainline has diverged so much from the development branches that merging becomes a long, difficult task.

Ideally the mainline should always be buildable and, in a continuous delivery environment, releasable. To do so requires that all commits to the mainline are high-quality—you don't want unstable development code to break the mainline build. Strategies such as frequent rebasing of development branches from the mainline and code review prior to committing to the mainline are often used.

Best Practices For Mainline Model

- ***Use the mainline model to organize your branches.***
  Each branch should have one role and purpose and a fixed set of policies. All changes flow to the mainline, which should have the latest stable version of a project in a releasable state.
- ***Ensure high-quality changes to the mainline.***
  Use code review and pre-commit build and test runs before changes are committed to the mainline.
- ***Keep development or task branches short-lived.***
  Once a piece of work has been completed and merged into the baseline, remove the development branches to keep the code base as clean as possible.

Streams

Branches do not live on their own; they define a natural hierarchy based on the stability or firmness of their codelines. A release branch is more stable than a development branch because it is more thoroughly tested and only accepts bug fixes, not new features. Changes should flow from firmer codeline to softer codelines to increase the stability and quality of the code. The hierarchy is based around the main codeline to which all changes should eventually flow—bug fixes from the release branches and new features from the development branches.

The relationship between the different branches in many version management systems is not explicit but only defined by naming convention and external rules. This factor can make it difficult to determine how changes should flow and whether there are missing changes that are not propagated yet (which could lead to regressions).

Some version management systems make the relationship between branches explicit by recording the nature of the branch (such as mainline or development), its relationship with other branches and the rules for the propagation of changes. Such branches are called streams (see Figure 5).

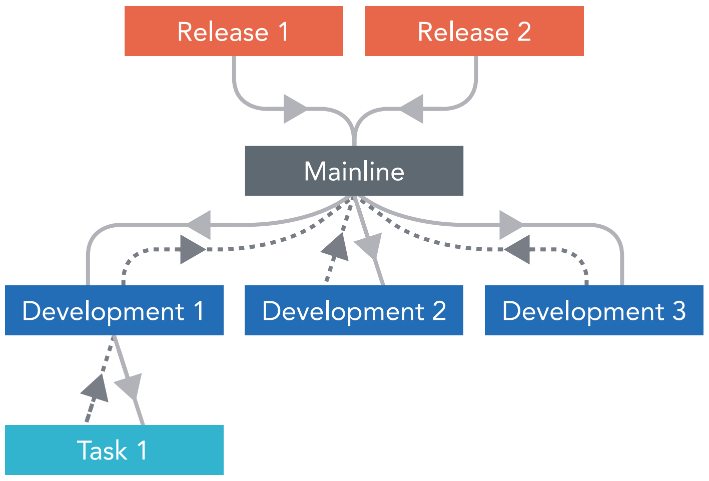

Figure 5: A typical streams pattern showing propagation of changes between release and development streams

Streams manifest many of the best practices for software version management and simplify the use of branches for developers and codeline owners.

Best Practices For Branch Relationship and Change Propagation

- ***Use merge-down/copy-up protocol to propagate changes.***
  Changes flow from firmer to softer codelines. Instead of merging from a development branch into a firmer codeline (which would introduce instability), merge down first and verify and then copy up to the parent.
- ***Make relationships between branches explicit.***
  Streams are an excellent way to visualize how branches are related and how changes should flow.

Distributed vs Centralized Version Control

It's easy to forget, in the Internet age, that the precursors of modern version control systems (VCSs) were purely local. The question then wasn't how to unite teams across the world but rather how to manage file revisions and roll back if needed from awful mistakes or other problems. Often, in this pre-client/server world, development was taking place on mainframes or mainframe-style servers. The working files as well as the repository of previous versions were on the same server. Developers had lightweight clients connecting to a high-power server, and simple version control systems, such as ISPF, SCCS and RCCS, were used because they were bundled with the operating system (see Figure 1).

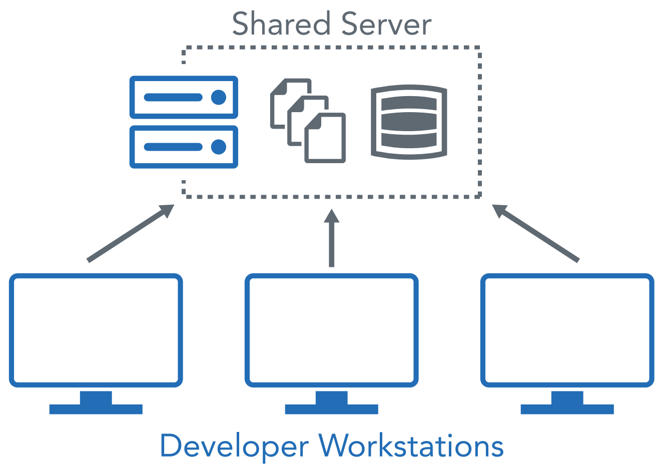

Figure 1: First generation: Local, server-based version management

With the evolution to client/server computing, VCSs quickly went beyond the local, establishing a new paradigm based almost entirely around file locking to prevent users from stepping on each other's toes. (Many developers today believe file locking to be negative; however, for many others, especially those working on images, video, chip designs or other files that can't easily be merged, file locking is still a critical requirement to prevent conflicts and lost work.) But even with that limitation, the benefits of a centralized model with a single, shared server were clear. Administrators and other interested parties could get a clear view of the state of all the assets, typically source code—what was changing, by whom and why. They could also easily manage backups and ensure high-availability or disaster recovery so development teams could keep working even if their local machines had problems. Security across all projects was managed in a single place (see Figure 2).

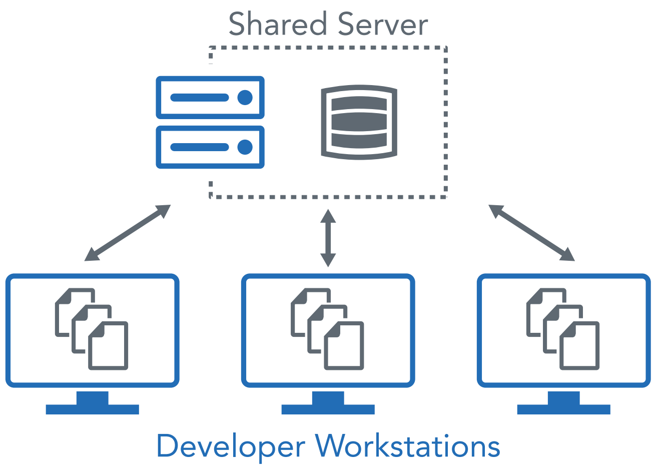

Figure 2: Second generation: Centralized version management

Yet the gains of a centralized model in no way eliminated the need for local capabilities, and thus the distributed version control system (DVCS) movement was a historical inevitability. Features such as lightweight, in-place branching and local-file-system performance were as a siren's song, luring developers away from central servers toward Git and other, similar DVCS tools on their desktops (see Figure 3).

Being able to share files directly between team members was also attractive. Instead of lengthy waits for IT administrators to set up shared servers, developers could share what they wanted only when they were ready to do so. The adoption of these tools by high-profile projects such as the Linux kernel was seen as a beacon for the way modern development and version management should be.

But the advantages of DVCS did not address organizations' needs for centralized capabilities—especially in the enterprise when working on larger projects with bigger teams. The plethora of Git management and similar solutions vying for space in the market today testifies to this fact.

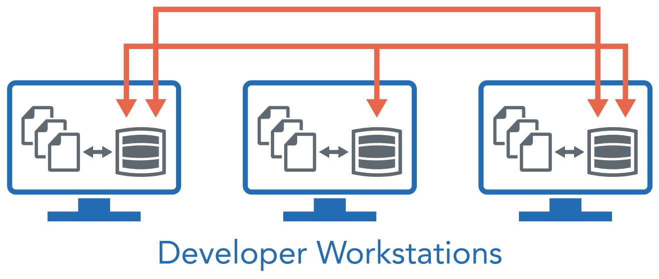

Figure 3: Third generation: Distributed version management

DVCS users almost universally adopted a centralized-like model with a shared master repository hosted somewhere on an internal server or hosted in the cloud. However, these tools were never built to be used in this manner, so they skipped all those features that actually made centralized VCS useful—such as flexible security policies and being able to lock files when needed.

History aside, today's developers want local branching, local speed, rewriting of local history and other capabilities. At the same time, other contributors need support for large binaries, file locking and other advanced features. And DevOps and IT need powerful automation, integration, scalability, reliability and all the other abilities that define enterprise-class software. The necessary conclusion is as obvious as it is overlooked: *Only a true hybrid can satisfy the needs of all relevant stakeholders in modern product development* (see Figure 4).

A true hybrid would furnish all the benefits of DVCS features with all the benefits of centralized servers and management. Fortunately, this description exactly fits Perforce Helix. It offers more DVCS power than any other system, providing its own native DVCS features alongside seamless support for Git and its entire ecosystem of tools. Yet Helix also provides all the centralized features and enterprise abilities underneath, scaling to meet any load from the tiniest project to the largest enterprise.

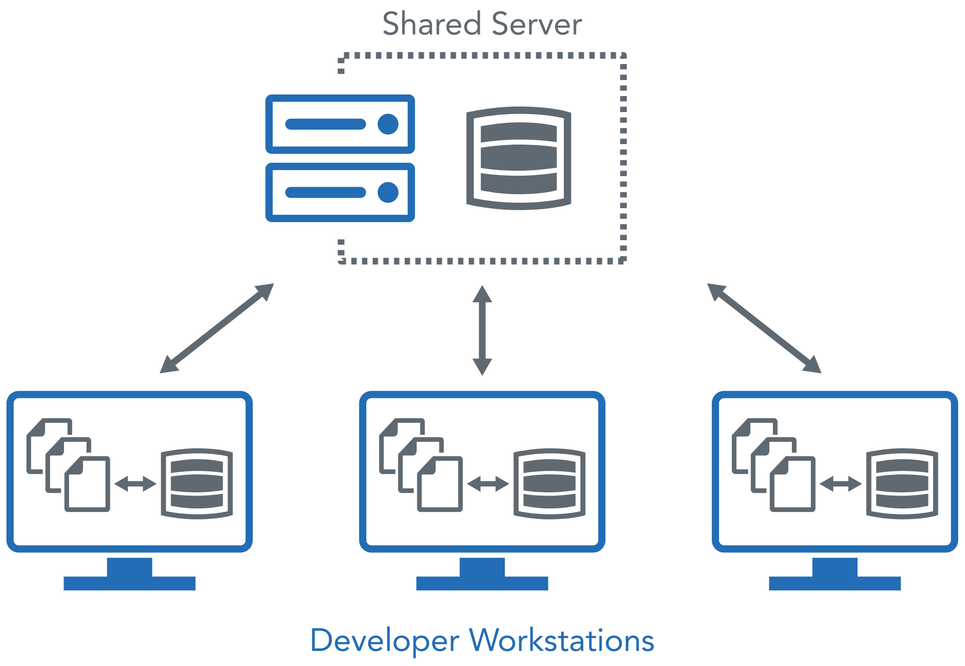

Figure 4: Fourth generation: Hybrid version management

Protecting Your Assets

Your version management system is a key repository for your organization. It stores and manages some of the most valuable assets in the company: your intellectual property (IP), which includes source code for applications used internally or by your customers; your product designs; export or compliance documentation; your videos, graphics or images; business documents and much more. Consider the value of these assets or the time and effort needed to recreate them after any potential disaster or the possible risk if they were leaked to a competitor, and you get some idea of why security should be a major consideration when choosing a version management tool and planning its implementation.

Version management protection of digital assets is different from simple backup or replication tools. Of course, backup and failover need to be part of the overall security environment, but they are not a complete solution by themselves. A good version management system will ensure that all the contributors to a product have the necessary access at the right time and allow for parallel working. It will need to understand the history of changes made to a file or collection of files and help contributors to understand what changed between a previous release and their current work. It will also provide insight into the kinds of changes being made and spot when suspicious activity may be occurring.

Levels of Protection

There are many different security levels that you need to consider.

User Access

Ensure that only approved team members have access to the version management repository. This level should, at least, guarantee robust password authentication and also integrate with enterprise identity management tools (e.g., LDAP or Active Directory) for improved security and simplified administration.

User Authorization

Beyond identifying that a user is the person he or she claims to be, the system should also control what operations that user is allowed to perform. You may need to allow a broad range of users to have read access while being unable to make changes, whereas other users can have full access to add, change or delete files. Advanced tools also allow controls over when someone can use the system (e.g., prevent access outside normal 9-to-5 workday hours) or where (from only certain locations). This level of security should also enable you to manage authorizations at a group level rather than always having to manage permissions at the individual user level.

File-Level Protection

You need the ability to further restrict access at the individual file level. Some tools—for example, Git—only provide access control at the repository level: All files have all the same access rights. Repository level protection requires that files need to be split between different repositories each with different protection settings. This makes administration of protections expensive and risky: with no central view, administrators cannot be sure of what protections are in place. For developers, managing multiple repositories to build their projects is also complex and slows down productivity.

For ease of management, you also should be able to manage access at the file-type level. For example, developers might have full access to source file types (e.g., C++ or Java source code) but have read-only or no access to the built executables generated from the continuous integration system. However, operations staff who are responsible for deploying applications to production servers will need access to those executables but should not be able to change source code.

Branch- or Stream-Level Protection

In a similar way, teams frequently need to be able to control access for particular branches or streams (see Chapter 3). For example, a developer may have full access when working on a development stream while only senior staff can update release streams, reflecting the current production system.

Encryption

In secure environments, it may also be necessary to encrypt data. You need to consider both the data at rest (i.e., stored in the repository) and in transit (i.e., when moving across the network between the user's client and the repository). Choosing the appropriate level of encryption will depend on your particular needs and industry or organizational standards.

Recording Activity

One of the key features of a version management system is the ability to maintain a history of changes. A history is a convenient feature for contributors such as programmers to see who made what changes over time. Comments associated with each check-in provide the narrative about why a particular change was made (see Chapter 2 for recommendations on comments). But this history is also important to the business as an audit trail.

Some regulated industries—for example, healthcare or financial services—have legal requirements for recording changes to systems. As part of an audit, they may need to furnish evidence to show what changes were included in a release, who made those changes and why. An audit trail that cannot be changed (known as "immutable history") can make these audits much easier than trying to work out what files were in each release and what changed in each revision of those files. This record can be an issue with some version management tools such as Git, which allow the audit trail or change log to be rewritten so some history may be lost.

Insider Threat Detection

Another use of the audit trail is to spot if any risky behavior may have occurred. Advanced security tools can use behavioral analytics to process the audit trail to understand what is normal behavior for members of the team. For example, most employees only access a small number of projects, so if they start taking files from other projects, that may indicate suspicious activity. Similarly, if they take many more files than other team members, put those files onto removable storage or start taking files outside normal working hours, they may be preparing to steal the files to take them to a competitor company. Beyond the obvious risks of competitors gaining access to intellectual property and trade secrets there may be a reputational risk to the organization if it becomes public that sensitive data has been compromized. Because audit trails may contain many thousands or millions of records even in relatively small teams, the automation of analytics is necessary to process the vast amounts of data involved.

In Summary

The version management system contains valuable IP. The loss of any files or changes either by accident or deliberate malicious activity could be expensive in terms of rework, reputation damage or loss of competitive advantage. A good version management security environment should include all levels of protection from file encryption through user- and file-level protections to advanced analytics to detect potentially dangerous behavior.

Your security plan must consider multiple levels

- **Data:** encryption at rest and in transit; specific file and file-type access controls
- **Users:** authentication and authorization; integration with enterprise tools
- **Branches and streams:** partitioning access control according to the intent of a change—development or release
- **Audit trails:** immutable history of all changes
- **Threat detection:** using data collected to warn of accidental or malicious risks

Best practices in version management, as with best practices anywhere, always seem obvious once you've used them. The practices discussed in this white paper have worked well for us and our customers, but we recognize that no single, short document can contain them all. So we have presented the practices that offer the greatest return and yet seem to be violated more often than not.

Conclusion

Version management tools and practices need constant review to ensure that they are being used correctly and have kept pace with changes in your organization and goals. What appears to be working may not be making use of modern branching capabilities or may not be suited to high-performance Continuous Delivery. Also, as projects become more complex (e.g., with new asset types such as product designs and with less technical contributors) or DevOps practices are adopted, then traditional tools and old habits may not be suitable. As a result, we recommend regular reviews of the tools and processes you use.

If you have other recommendations for best practices, please let us know and join the conversation via the Perforce Forums and social media channels.

North America
Perforce Software Inc.
2320 Blanding Ave
Alameda, CA 94501
USA
Phone: +1 510.864.7400
info@perforce.com

Europe
Perforce Software UK Ltd.
West Forest Gate
Wellington Road
Wokingham
Berkshire RG40 2AT
UK
Phone: +44 (0) 845 345 0116
uk@perforce.com

Australia
Perforce Software Pty. Ltd.
Suite 3, Level 10
221 Miller Street
North Sydney
NSW 2060
AUSTRALIA
Phone: +61 (0)2 8912-4600
au@perforce.com

PERFORCE

perforce.com