Insecurity about Git: Myths and Solutions about SCM Security

Randall S. Becker >> President >> Nexbridge Inc.

Preamble

Jan woke up with a start. Her smart phone was beeping loudly. The CIO was demanding answers to why some of the company's critical ATM driver code had shown up in something called GitHub Enterprise. Wasn't access to GitHub blocked by the firewall rules? What was the exposure? Who had done it? A war room was being set up and she was expected to join the bridge immediately.

This was the sum of all fears for Jan and her team – not that security had been compromised, but that confusion about roles, tools, and access would have to be explained at 3am instead of during a weekly security briefing with the CSIO.

This was going to be a long night, with very little value to the company other than clearing up some misconceptions. Again.

What you have read is fortunately not based on real events, but on the fears that the author has had to explain whenever the subject of git comes up in conversation around NonStop. While git itself has been generally available since 2007 and available for NonStop since early 2014, adoption has been extremely slow. The primary cause is the perception, right or wrong, is that git is somehow less secure than Subversion, RMS, and CVS. This article discusses the reality of git's strengths and weaknesses, techniques to secure your code properly, and addresses the fundamental questions about the security of workstation-based development.

Disclaimer: Many products will be named in this article. None of the mentions are an explicit or implicit endorsement by the author or Connect for the suitability of the product in your environment and should not be interpreted as advice.

The Chaos of Terms

The name git derives from UK English slang meaning "unpleasant person". It was coined by Linus Torvalds, who had a penchant for egotistically naming things after himself, in some way. The name itself was intended humorously and possibly ironically as "The Stupid Content Tracker". Git itself refers exactly and only to the Distributed Version Control System software itself. You may have heard the terms EGit and JGit, which refer to the ECLIPSE Plug-in for git and the Pure Java implementation of git, respectively, that come bundled with the NSDEE ECLIPSE configuration.

What was inevitable in the world of Cloud services, and with the word *distributed* in it, is that people will almost automatically come to the conclusion that you can make a SAAS-like service out of it. Up popped services like BitBucket, Atlassian, GitHub, and a whole bunch of other service providers where you could host your git repository *for free*, for Open Source projects, or for reasonable fees for private secure repositories. This was the first bit of confusion in the git world.

The second bit of confusion came when GitHub decided to create a private Cloud service called GitHub Enterprise. Atlassian did the same with their Stash service. Both provide self-managed secure environments for hosting your repositories entirely inside your own firewall. It might have been better for our intrepid CIO had GitHub called their offering the "Private Really Secure Internal Cloud Git Thing", so that the hypothetical 3am call would have been avoided.

This article will go into the differences in security models for some of these layered services, but the most important thing to take away is that you have different domains for git repositories, and you need to know exactly what you are discussing when panic sets in:

Domain	Description	Examples
Clone	A working repository where a developer will be interacting directly with git to create activity and history.	Repositories created by a clone operation via git clone, ECLIPSE EGit, Atlassian's SourceTree.
Upstream	The Repository of Record (RoR) or mirror where changes are integrated. This may contain the sum of all history from all clones. Developers usually manage the merging of activities. Officially known as 'origin'.	Usually on a private server, including NonStop, Linux, and Windows.
Enterprise	A set of repositories or mirror backbones (See Article 3 in the series) that are managed in a secure environment. Changes are typically integrated into the mainline histories through merge operations by repository managers.	GitHub Enterprise and Atlassian Stash are examples of this type of structure.
Hosted	A Cloud SAAS environment where your upstream repository is outside your own network. You may have control of the security rules and policies, depending on the level of openness of your repository.	GitHub, BitBucket, and Atlassian are examples of these providers.

As you can see from the terminology, there is a lot of overlap in the brand names.

When talking about git, be clear about what domain you are discussing.

The biggest differentiation for corporations between the different layers comes at the Enterprise domain. Off-the-shelf products like Stash and GitHub Enterprise provide security that you would otherwise have to script yourself when in a simple upstream environment. There are many products to do just that, including branch-level security, from products like gitolite. With the enterprise-class products, you get a lot of structure, process, and security enforcement. With git alone, you must depend on OSS and SSH security rules. Publishing git through HTTP without any other authentication can leave you exposed.

Software at Rest

An important concern for all intellectual property managers and Corporate Security Information Officers (CSIO) is how to deal with a situation where your software could be stolen while sitting on someone's laptop, desktop, or jump-drive. This question has been present since large-scale software development on platforms like NonStop moved from EDIT/VS or TEDIT to workstations with the Enterprise ToolKit (ETK) back around 2002 and ECLIPSE NonStop Development (now NSDEE) around 2010. In fact, this has little to do with git, because software on a workstation, whether in CVS, ClearCase, PrimeCode, Control-CS, or Subversion, is still resident on a hard drive that can leave your data centre/development office. The preferred solution for this has been, and probably will continue to be, drive encryption – regardless of which Version Control System you have, whether distributed or not. Another option taken by some companies has been to do all development on virtual thin clients where the code only resides on in a VM environment. Other organizations have taken control of the hardware on desktops and laptops to prevent jump-drives and other means of storing source code in a portable fashion, but that gets into the next question.

Software in Flight

Moving software is a more relevant concern for workstation development and DVCS systems like git. With git, being about to modify the upstream repository – the place where you got your copy of the code originally – to point to an open environment, is a very real concern. Preventing developers from emailing patches, or pushing your code to their private repository is a serious and very relevant security consideration. Firewall rules and email policies and filters are invaluable here – whether or not you have git. It applies equally to any code that resides on a network. Lock it down. Do not let your code get pushed up to a hosted facility, unless it is your own. Do not even let patches or code fragments get emailed to "friends".

Don't be confused between GitHub Enterprise and GitHub. They may run the same software, but the former is inside your network, the latter is not.

The above concerns are always with you when developers have their own devices, workstations, and laptops. But for git itself, as well as other DVCS facilities, there are some core questions around which you need to establish some solid policies around very real questions.

The Fundamental Questions

The questions of security in a DVCS world, come down to four basic areas:

1. Who has read and/or modify access to the code?
2. Are historical records secure and how visible should they be?
3. How is the Repository of Record managed?
4. Which branches need to be kept secure and protected?

It is a given that audit is a major concern – who is looking at your code, who has access to it, who is modifying it. But the fundamental questions will drive a lot of your decisions and are not really different from traditional centralized VCS systems. The core difference is whether history visibility represents a vulnerability. If you have code that needs to be protected using different access rules, put that in its own repository and lock it down, or add security management software like Stash.

Who has Access to Code

The biggest question is who can see your code. If you are an Open Source participant, the answer may be everyone. This is unlikely in our community, although if you participate in the ITUGLIB structure, some of your components may have external connections, relationships, and license requirements.

There are many mechanisms of security for your code within git. At the simplest, you can use OSS access control lists (ACL) or basic group ownership to define roles on a team. Developers code clone repositories for projects for which they are a part. If you have something like Stash, you can do this on a per-user basis by adding people to a project. Using SSH, you can designate a functional user id with separate audit capabilities for known public key pairs – check that out, seriously.

When it comes to modifying code, the basic best practice out there is to designate a repository manager. Developers would make changes on their own topic branch – you should be familiar with this by now, and will request that the repository manager merge their changes into the main integration branch. This is known as a Pull Request. Have a look at the GitFlow Process for details on this. It is a useful reference. This structure gives companies a lot of control over who actually can contribute changes to production. It groups staff into two roles: developers and contributors – Open Source terms for people who make changes, and people who approve the changes.

More confusion: the git stash function and the Atlassian Stash product are completely different things. Don't confuse the two.

Stash provides a means of setting up a discussion for these Pull Requests (requests to merge my changes), and Gerrit takes it further by providing explicit code reviews on a change-by-change basis.

Visibility of Historical Records

A conundrum for git, perhaps its greatest value and its significant concern, is how visible is the sum of historical records of a project. In git, every commit made by the team in a project's repository can be visible to any other developer – this is not a requirement, but is a general practice. The capability allows developers to see the origin of any change, including how it was merged and who made the change. Some organizations view this as highly valuable, while others consider it a vulnerability. This is one area where you need to decide whether the change audit trails of source code modification are themselves subject to separate security rules. It is a serious question and one that needs to be answered early during the implementation of your improved software development process.

Management of the Repository of Record

In any DVCS, you will have a definitive copy of the code located in one or more repositories. This will include:

- The changes that come from developers;
- Release packages containing the commits being installed and built code;
- Hot fixes originated from development; and
- Production fixes originated from production environments.

Management of these repositories should be kept separate from development. It may be part of a Quality Assurance group or Production function, and your security policies will decide that. However, feeding production fixes back to development is really important or you will lose critical fixes.

Establishing a separate role for managing definitive repositories is really important, particularly if you have a requirement for separation of duties. This role will be responsible for pulling changes from development repositories – developers should not have access to the repository of record – and pushing changes back to development to ensure visibility of production fixes. This role will also maintain archives of supported releases and will clean up archives that are no longer necessary – repositories containing complete images can be very large.

Managing Branches

An early perception of vulnerability of git was its lack of protection of branches. For example, anyone on a project could merge code into any branch. This was done under the assumption that developers were essentially good people of conscience. This notion was quickly dismissed as risky despite very positive conduct overall. Separation of duty into developers, contributors, and reviewers, was physically divided into separate repositories to allow UNIX security rules to govern who did what. Products later evolved, including gitolite, Stash, and GitHub Enterprise to formalize branch security while simplifying and reducing the number of repositories companies needed to have. For ITUGLIB, as an example, there is a separation of physical access to the repository of record so that only contributors may merge developer changes into the official code.

Git branch management has evolved rapidly and effectively in the last few years. If you haven't looked recently, go look again.

Conclusion

Git has a standard set of security concerns that are common to most DVCS and central VCS systems. When looking at git as a choice of SCM solution, take into account the capabilities and consider carefully the security rules you need to have in place in your organization for workstation-based development. And importantly, try to get past the very confusing git nomenclature; whether you are trying to convince your management to embrace the git DVCS or having that dreaded 3am conference call.

[i] Documentation on the Gitflow process can be read at both Github and Atlassian websites and is widely searchable.

Randall S. Becker is a speaker, author, and consultant on Policy and Process that delivers continuous availability. He is an expert in Software Configuration and Change Management since 1989 and has spoken at many NonStop and community events.

Randall can be contacted at: +1.416.984.9826 or rsbecker@nexbridge.com.