Continuous Integration

DevSecOps

Agenda

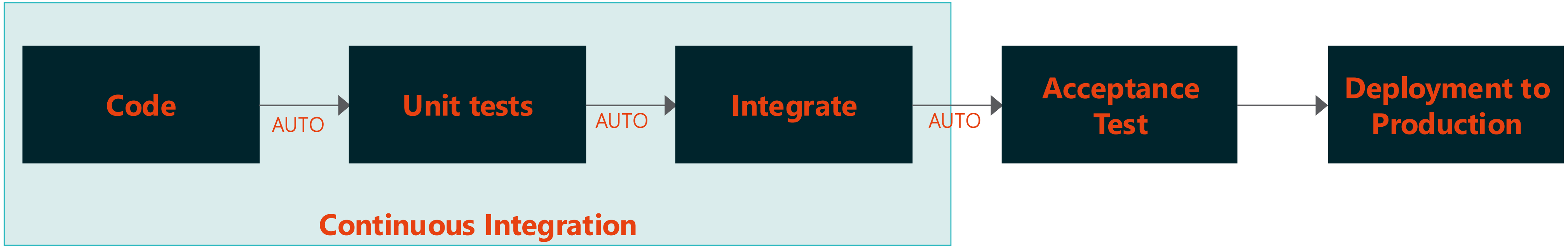
- Continuous Integration
- Security at Dev Time
- Git Hooks
- Handle Sensitive Data
- Credential Scanning

Continuous Integration

Secure DevOps

What is Continuous Integration

"… a **software development practice** where members of a team **integrate their work frequently**, usually each person integrates **at least daily** – leading to multiple integrations per day. Each integration is **verified by an automated build** (including test) to detect integration errors as quickly as possible. Many teams find that this **approach leads to significantly reduced integration problems** and allows a team to **develop cohesive software more rapidly**." (Martin Fowler)



Goals

1. Leverage team collaboration
2. Enable parallel development
3. Minimize integration debt
4. Act as a quality gate
5. Automate everything!

Security at Dev Time

Secure DevOps

What is “Development Time”

- When you mention “Development” means when you’re creating your code
- On Everything as Code approach can be anything
- During authoring phase, you interact with your IDE/Code Editor and your source control system
- Current IDE/Code Editors uses the concept of extensions to leverage your experience
- With this approach you can shift-left several automation to you IDE

Source Control

- On Everything as Code source control is the center of all activities around our projects
- Nowadays the de facto source control tool is git
- Git adoption is probably the biggest within tech communities compared with every other tool

Git: Remote vs Local

- As a distributed source control, git uses the concept of remote and local vs server and client
- After remote repository events we can trigger several processes to start work upon our code
    - Pull requests
    - Code analysis
    - SCA

Git: Local events

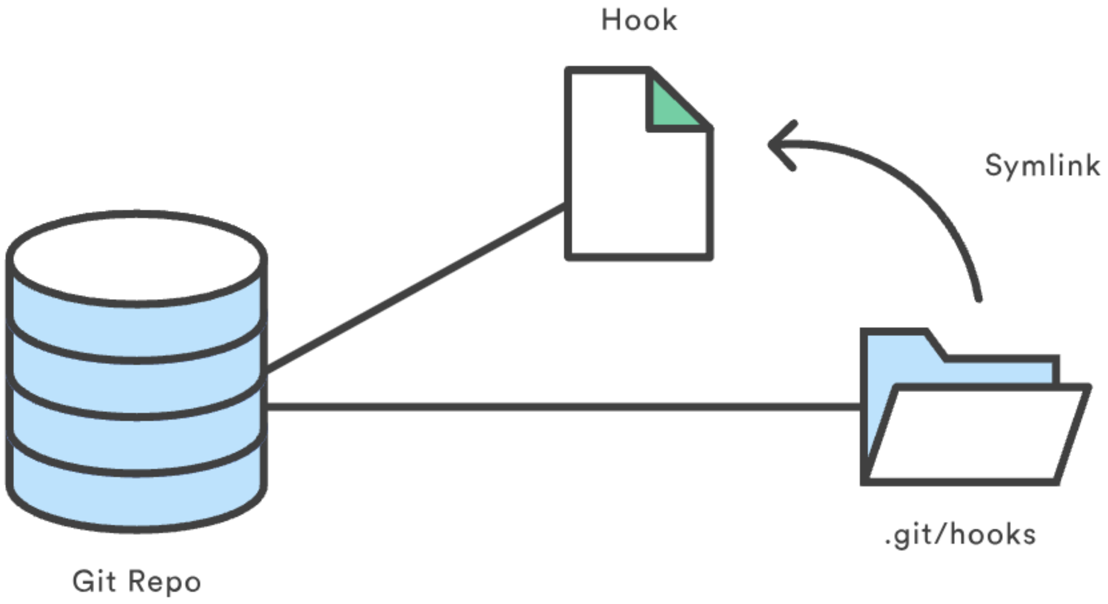
- But since git is full featured on local side we can leverage its event triggering capabilities to make validation of our code
- This validation can bring additional security guardrails to the code we add to our local repo (and in sequence, push to remote repo)
- This can be implemented using Git Hooks

Git Hooks

Secure DevOps

Git Hooks

- Git hooks are scripts that run automatically every time a particular event occurs in a Git repository
- They let you customize Git's internal behavior and trigger customizable actions at key points in the development life cycle



Installing Git Hooks

- As everything in Git, the installation is a basic file operation
- You only need to add your scripts to **.git/hooks** folder
- If you look to the folder, you will get a list of samples scripts
- To enable it you only need to remove the .sample extension
- You can use any scripting language you like as long as it can be run as an executable

Authoring Git Hooks

- Sample hooks are developed in bash or PERL scripts
- You can use any scripting language you like as long as it can be run as an executable
- Python can be used giving you a common programming language to implement the hooks

Scope of Git Hooks

- Hooks are local to any given Git repository, and they are not copied over to the new repository when you run **git clone**
- And, since hooks are local, they can be altered by anybody with access to the repository.
- You can have server-side hooks but they manage different events

Local Git Hooks

- pre-commit
- prepare-commit-msg
- commit-msg
- post-commit
- post-checkout
- pre-rebase

Server-side Git Hooks

- **pre-receive** is executed every time somebody uses **git push** to push commits to the repository
- **update** is called after **pre-receive**, and it works much the same way
- **post-receive** gets called after a successful push operation, making it a good place to perform notifications

Local Git Hooks: pre-commit

- The **pre-commit** script is executed every time you run **git commit** before Git asks the developer for a commit message or generates a commit object
- You can use this hook to inspect the snapshot that is about to be committed
- For example, you may want to run some automated tests that make sure the commit doesn’t break any existing functionality
- No arguments are passed to the pre-commit script, and exiting with a non-zero status aborts the entire commit

Local Git Hooks: prepare-commit-msg

- The **prepare-commit-msg** hook is called after the **pre-commit** hook to populate the text editor with a commit message

- This is a good place to alter the automatically generated commit messages for squashed or merged commits.

- One to three arguments are passed to the prepare-commit-msg script:
    - The name of a temporary file that contains the message. You change the commit message by altering this file in-place.
    - The type of commit. This can be message (**-m** or **-F** option), template (**-t** option), merge (if the commit is a merge commit), or squash (if the commit is squashing other commits).
    - The SHA1 hash of the relevant commit. Only given if **-c**, **-C**, or **--amend** option was given.

Local Git Hooks: commit-msg

- The **commit-msg** hook is much like the **prepare-commit-msg** hook, but it's called after the user enters a commit message
- This is an appropriate place to warn developers that their message doesn't adhere to your team's standards.
- The only argument passed to this hook is the name of the file that contains the message
- If it doesn't like the message that the user entered, it can alter this file in-place (just like with **prepare-commit-msg**) or it can abort the commit entirely by exiting with a non-zero status

Local Git Hooks: post-commit

- The **post-commit hook** is called immediately after the **commit-msg** hook
- It can’t change the outcome of the git commit operation, so it’s used primarily for notification purposes.
- The script takes no parameters and its exit status does not affect the commit in any way.

Git Hooks on a team

- Local hooks are not pushed to remote repos so cannot be directly shared
- You need to make always local configuration to activate them
- You may find any way to make this happen but there are two recommendations: symlinks or git template directory

Git Hooks on a team: Symlinks

- This approach uses the repo to store the hooks and then any developer need to do an initial configuration
- You store your hooks scripts on a folder called .hooks (as any other folder on you repo)
- Then you need to create symlink on **.git/hooks** folder to use the scripts on **.hooks** folder
- You may have a script that can be executed by any developer as soon as clone the repo

Git Hooks on a team: Git Template

- You can set a git configuration to point to a folder using this command:
  git config --global init.templateDir ${DIR}
- All the content of this folder ${DIR} will automatically copied to **.git** folder of your repo

Git Hooks: Solutions

- Apart from having you authoring your own git hooks there are some community-driven solutions
- Talisman, from ThoughtWorks focused on security
- Husky, a git hooks ”package manager”
- Pre-commit.com, another git hooks ”package manager”
- Detect-secrets, started as a scanning tool but now can be installed as pre-commit hook as well

Talisman

- Open source project created and maintained by ThoughtWork
- Available on GitHub: https://thoughtworks.github.io/talisman/docs
- Talisman is a tool that installs a hook to your repository to ensure that potential secrets or sensitive information do not leave the developer’s workstation
- It validates the outgoing changeset for things that look suspicious - such as potential SSH keys, authorization tokens, private keys etc
- Talisman can also be used as a repository history scanner to detect secrets that have already been checked in, so that you can take an informed decision to safeguard secrets.

Demo: Using Talisman

Secure DevOps

Lab 01: Enable GitHooks with Talisman

Secure DevOps

Handle Sensitive Data

Secure DevOps

Sensitive Information

- Sensitive information can be seen in several perspectives
- Can be passwords or other type of information that can give you access to data that you should not have access
- Can be tokens or keys that can give access to another servers or services
- Can be sensitive and personal data (PII) that can make you on a position to take advantage from another person

How to store that information

- Knowing that you need to keep that information secure you can always store it using some cypher method
- But in some place in time you need to have that information in clear text to be used accordingly
- Keys, tokens or password need to be used in clear text when you interact with expected server or machine
- Personal data needs to be in clear text for the correct person see her/his information
- Here we’ll focus on data used by systems, like passwords and tokens

How to store that information

- These information should not be in any case on your repo
- First, that information will give you access to something that you should not have access directly
- Second (and unfortunately), is still too common that the same password can be used to login in several servers (remember the lateral move attack)
- Same techniques arise on last years to help on this: password-less architectures, infra as code deployment, usage of Vaults, and credential scanning

Password-less architectures

- On modern architectures, password-less approach is the recommended way
- Passkeys for user logins instead of passwords and MFA
- For system integration, usage of Managed Identities concept allow you to not use user+pass or tokens for authentication
- These approach needs a deeper architectural change and make it harder to implement in a faster way

Infra as code deployment

- Infra as code is a great open to minimize passwords or tokens exposure
- If you manage your infra using this approach, provisioning and configuration is done on an automated way
- All sensitive information that need to be shared between several components can be done by code
- Another benefit is the secret rotation that can be done on a easy, automated and faster way
- More details later in this training

Usage of Vaults

- However there still are needs to keep secrets on a secure place
- Using CI/CD pipelines can help since you can store the secrets on the platform and then apply during automation process
- On last years, mainly with clouds, the concept of Vaults took a crucial place on this need
- All cloud providers have a PaaS solution, like Azure Key Vault
- There are cloud-agnostic options like Hashicorp Vault (with self-hosted option too)

Usage of Vaults

- The concept consists of a centralized solution where all data is ciphered using strong keys
- That keys are managed by the platform or even directly by the user
- When you use platform provided keys you can have auto-rotation process out of the box
- Then only allowed users (ideally services or service accounts) could have access to the decrypted data
- This integration can happen on automation processes or even directly on the application code

Credential Scanning

- Credential Scanning is the least protective measure to handle sensitive information
- Although is the easiest and faster to implement and should be a no-brainer decision
- Can be a complement to the git hooks approach that we saw before
- This approach consists on running during CI process on your automation platform and scan your code to check for possible sensitive information

Credential Scanning vs Git Hooks

- Centralized solution: You don’t need to rely on configuration on each developer machine (imagine on a open source project)
- More complete scanning: Git Hooks solution are local and smaller on their scope. Credential scanning solution are integrated solutions that are constantly updated
- Auto-revoke: Some solutions in the market (like GitHub Advanced Security) have an integrated feature that can contact directly the provider of the token to revoke automatically. This feature can minimize a lot the impact of a pushed token

Implement CredScan into CI

Secure DevOps

Credential Scanning: Solutions

- gitleaks, open-source secret scanner for git repositories, being one of the most used. You need a free license for Enterprise repos
- Trufflehog. For Enterprise they are moving to a more complete (and paid) solution - trufflehog-enterprise
- GitHub Secret Scanning. Free for all public repos (personal or Enterprise). Part of GitHub Advanced Security pack for private repos
- GitGuardian. Paid solution with full integration with main Git providers like GitHub, Azure DevOps, BitBucket, …
- SpectralOne, from CheckPoint. Another paid solution (quite expensive) but with the best reporting system by far.

Credential Scanning: Where to implement on CI?

- When you’re working on a collaborative approach, your repo should always have defined a protection rule on your main branch
- With this, you’re only capable to merge changes to main branch using a Pull Request
- Credential scanning is a mandatory part of your Pull Request approach
- When the scan send you an alert, you must perform two tasks: remove the sensitive information from your branch and review the repo history to cleanup that information too

Demo: GitHub Secret Scanning

Secure DevOps

Lab 02: Enable Secret Scanning

Secure DevOps