An NCC Group publication

Securing the continuous integration process

Prepared by:
Irene Michlin

Contents

1 Introduction

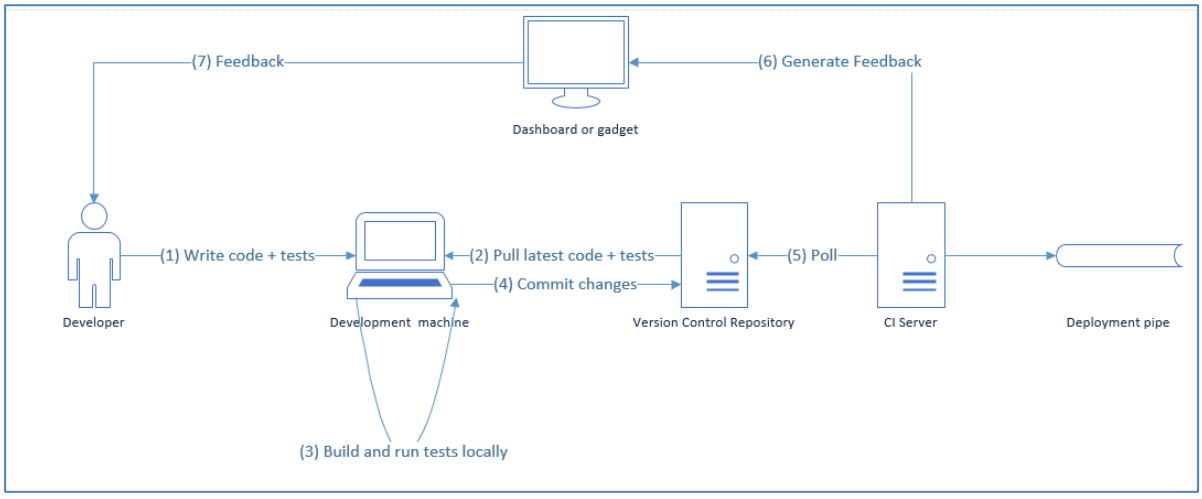
Continuous integration (CI) has long left the stage of experimental practices and moved into mainstream software development. It is used everywhere from start-ups to large organisations, in a variety of technology stacks and problem domains, from web applications to embedded software.

However, the security implications of introducing CI are often overlooked or underestimated. There are whitepapers and advisories on this topic, but most are sales-oriented, promoting a specific tool or another form of silver bullet that will solve all of your security problems.

This paper intentionally avoids recommending a specific solution or vendor. Instead, it focuses on technology and process changes involved in setting up a CI environment and aims to provide best practice guidance for introducing CI in to your secure software development life cycle (SDLC).

The choice of tools in various steps of CI is enormous (see: *The Ultimate DevOps Tools Ecosystem Tutorial* [14] for the illustration of the choice available). This paper does not discuss their relative merits from a functionality point of view, but specifies which features are necessary to allow secure integration of the tools.

After considering the requirements for building CI in a way that does not compromise your organisation's security, this paper lists the techniques for leveraging the CI setup to improve the security of the developed software.



2 Basic continuous integration cycle

Specifics of CI can differ between various implementations, but the basics can be summarised in the following flowchart:

(1) A developer writes code or modifies existing code, usually with corresponding tests.

(2) The developer pulls the latest version to the development machine in case it has changed since the development starting point.

(3) The developer builds the code on the development machine and runs the tests locally. This step is optional, but popular in many CI implementations. If there are any conflicts or broken tests, the

developer repeats steps (1) and (2) until everything is resolved.

(4) The developer commits the changes to version control.

(5) The continuous integration server periodically polls the version control repository, and picks the change from step (4). The CI server builds the code at the very least. Most implementations of CI will also include running a test suite as part of this step and a variety of additional tools can be integrated here.

(6) The CI server generates feedback which is visible on a shared dashboard. Many teams get creative with the dashboard and there are gadgets which visually reflect the state of the build for users and admins to view.

(7) The developer receives the feedback. This can be implemented as a *Pull* pattern, where developers are expected to observe the dashboard after committing a change, but the *Push* pattern is more reliable. In the case of negative feedback (broken build), the process restarts from step (1). Depending on the overall length of the cycle and other factors, some teams choose to implement an automatic revert or rollback to the last good state. If the feedback is positive, the CI server can be connected to the deployment pipe and trigger the next steps in continuous deployment (CD). The deployment pipe and CD in general are out of scope for this paper.

The exact details of the implementation depend on your technology stack, SDLC and other organisational specifics. If you need guidance on the choices available, *Continuous Integration: Improving Software Quality and Reducing Risk* [1] is an excellent overview book.

The following chapters of the paper look at the steps above in detail, providing recommendations on securing them.

3 Security implications – technology

3.1 Development environment

Whether the developers work on physical or virtual machines, individually or using pair programming, these machines need to be isolated from the rest of the corporate network. Many compliance standards mention segregated environments, but even if your organisation does not have a compliance obligation, you should still do it.

3.1.1 Protecting development from the corporate network

Presumably, the organisation develops software to affect the bottom line either directly, by selling the products, or indirectly, if internal applications support other business processes. Regardless, it is an asset - an intellectual property - and needs to be protected. Access control to the development machines can be much more efficient if the machines reside on a dedicated network.

Corporate applications such as email can be used for social engineering, for example, sending phishing links which, if clicked on, will result in the installation of a keylogger or remote access tools. Not having these applications on the development machines reduces the likelihood of an attacker stealing the intellectual property.

3.1.2 Protecting the corporate network from development

Depending on the problem domain, it is possible that the software under development can adversely affect the network it resides on. For example, traffic patterns involved in testing distributed systems will add unnecessary complexity to firewall and intrusion detection system rules.

Many IT departments exercise a high level of control over applications that can run on a machine. Making global exemptions for the software under development and various development tools will weaken these controls across multiple departments, reducing the overall security posture of the organisation. Keeping the environments separate removes the need for global exemptions.

3.1.3 Remote work

Many organisations provide an option of working from home. If the developers use their own computers, it is difficult to impose any security standard on their development environment. A popular solution is to provide remote access to a computer on the development network. In this scenario, it is usually possible to achieve an acceptable level of assurance on the security of the remote access. Using a virtual private network (VPN) is a popular approach for securing this access.

Takeaway	Keep individual development machines on a segregated network. If allowing remote work, validate the security of remote access.

3.2 Version control

The choice of the version control pattern depends on multiple factors, such as frequency of releases, number of teams, structure of the code base etc. Generally speaking, patterns that require frequent commits to a shared branch are more suited for implementing CI. This paper does not discuss various version control strategies in detail. A good tool-agnostic book is *Software Configuration Management Patterns* by Stephen Berczuk and Brad Appleton [2].

Each of the main security properties (confidentiality, availability, and integrity) is important when choosing a version control tool. In a large and distributed organisation, a tool that does not allow multiple local repositories will be a constant drain on the productivity as developers lose time on each operation. The convenience of cloud-based solutions needs to be balanced against the potential of the down-time if the tool does not support remote work.

Modern tools, whether cloud-based or hosted, allow role-based access controls (RBAC). In a large organisation, many roles will require access to the version control repository: developers, testers, database administrators, DevOp engineers, user interface (UI) team and potentially other roles depending on the organisational structure.

Whatever tool is in use has to be configured to provide granular controls on a “need-to-access” basis. The accounts need to be individual, as shared accounts prevent meaningful logging and auditing. If the tool has a generic administrator account, investigate whether it is possible to disable it and instead give administrative privileges to an individual account. If that is not possible, ensure that the default password on the administrator account is changed to a suitably secure password.

For projects with distributed contributors, consider signed commits. Git, the most popular choice of distributed version control system (DVCS), supports this functionality (see [20] and [21] for details).

Proliferation of accounts for various development and test tools is a known security problem and a coherent account lifecycle management is required. The problem becomes even more acute if the development uses short-term contractors. It cannot be solved purely by technological means, and sometimes the business process needs to be set up across multiple departments. Generally speaking, tools that support single sign-on are preferable in this respect to the tools that require a standalone account.

A well-implemented RBAC policy can prevent unauthorised access to the source control repository and assure code confidentiality.

Another of the security concerns around source control repositories is code integrity, in particular the prevention of backdoors. As is the case with many security issues, it cannot be solved with a single administrative measure or technical tool. One of the components required to mitigate this threat is a comprehensive code review, in conjunction with chain of custody in the release process. Chain of custody is discussed in detail in section 4.3.

In terms of technical controls, consider installing an application whitelisting solution. The version control server has one job only and it needs to run a restricted set of programs. Therefore the server can be hardened by removing or disabling everything else that is not required.

Takeaway	Pick a tool that has RBAC, and configure it with enough granularity. Do not forget to remove/adjust the accounts when people leave or move teams.

3.3 Integration build server

This is unsurprisingly the most important piece of the CI puzzle. As your CI practices mature, more and more work will be delegated to this server and as a consequence more tools will need to be installed and maintained (see section 6 for more details). But before the integration server does anything else, it needs to be able to retrieve source files from version control and produce binaries based on the latest version. Do not give the integration account more privileges on the version control server than required – it should be provisioned with read-only access.

If your software relies on external libraries and components - either commercial or open-source - it is good practice to keep everything you need in an internal repository instead of fetching components at build time. Each external component can introduce a security issue and if a repository was compromised, your build will fetch potentially malicious components.

The practice of keeping internal copies has benefits beyond security, as it gives you control over the transitions to new versions of the components.

It is difficult to provide language-agnostic recommendations for secure options when compiling and linking, but most vendors of compilers provide appropriate security recommendations. It is important to define a standard set of compiler and linker flags and recommend (or even enforce) this set to be used for local builds as well. This practice will save the team from chasing broken CI builds which "worked on my machine".

Depending on the team maturity and project complexity, the integration build server could be one physical machine on the development network, or a remote server farm. From a security point of view, a larger set of servers only means one thing: a larger attack surface.

If you use a cloud-based solution, review the authentication and authorisation available. For hosted solutions you are also responsible for keeping the operating system (OS) and any installed software up-to-date. Containers are becoming popular technology in this area. See [6] and [7] for our research on Linux containers security.

The rest of this section considers additional activities which can be done by the CI server, in the order of increasing maturity required for their implementation.

3.4 Dashboards & gadgets

One of the major functions of a continuous integration server is to provide quick feedback to the developers. It is best practice to provide the output of all the tools installed on the CI server on a unified dashboard, or information radiator.

As the number of tools installed on a CI server increases, the complexity of the dashboard grows with it. Many teams choose to use a programmatically controlled gadget that visually shows a current state of the build. Gadgets are more fun than physically carrying a mop or a wooden spoon to the desk of the build-breaker. However, electronic toys which are part of the Internet of Things (IoT) are notoriously insecure. Before installing one, make sure it cannot be used by an attacker as an entry point into your network.

Dashboards often require custom integration scripts. Make sure that these scripts are treated with the same respect as the rest of the code: they need to be reviewed and stored under version control.

3.5 Feedback mechanism security

Unless it is a low-tech mechanism - all the developers are in the same room and they look at a physical dashboard when they feel like it - you have to consider the security of the mechanism.

Can the mail application or SMS-gateway be abused? Does it run with higher privilege than necessary? Does the feedback mechanism leak information and allow bypassing of access controls?

3.6 Summary

Continuous integration environments are rarely planned upfront. Tools are most likely introduced one by one as the team's practices mature and the technology landscape is usually heterogeneous, with ad hoc scripts that hold everything together. In addition, development organisations frequently need to move at a significantly faster pace than the overall enterprise architecture, so the tools and servers added would be exempt from the risk assessment usually performed by an IT department.

A responsible development organisation has to take care of the following:

- **Assessment of the new components.** The process can be as informal as your overall risk appetite dictates, but at the very least it has to capture the decision to introduce a new application or infrastructure component. Many organisations only discover what they have as a result of an audit.
- **Installation and configuration.** Some tools will have lots of optional features but do not install and enable features that are not required. Review all the default accounts and keep only those that are necessary, protecting them with passwords that conform to your overall password policy.
- **Integration with existing components.** If new scripts are required, they need to be added to source control. Any adjustments to internal firewall rules or IDS configuration need to be documented and reviewed to ensure that the isolation of the development network is not

broken. The scripts used for the integration need to be reviewed to the same standard of scrutiny (or even higher) than the production code. A widespread problem, for example, is hardcoding of passwords to privileged accounts.

- **Maintenance.** For hosted tools, it is your responsibility to follow the vendor security advisories and install patches as necessary.
- **Disposal.** Hardware that was used to store sensitive data requires secure disposal. If cloud storage was used, then you do not control media sanitisation directly and can only enforce secure disposal contractually.
- **Account lifecycle management.** Developers leaving the company may retain access to multiple accounts, which is especially dangerous in case of cloud-based tools. Even if everyone associated with the company behaves ethically and never tries to access GitHub or JIRA after leaving, these dormant accounts increase the attack surface for password brute-forcing or other attacks by external threat agents.

Even if the system was deliberately designed to some degree, often the design makes an implicit assumption that all the interactions in a CI system happen between trusted components and therefore no input validation or other controls are necessary. This assumption can prove damaging if an attacker bypasses perimeter controls and gets a foothold on the development network.

Takeaway	CI introduces additional applications and infrastructure components into the development environment. Each one presents an additional attack surface. Choose appropriate security controls to mitigate the new threats.

4 Security implications - beyond technology

All security programs have three components: people, process, and technology. CI is not an exception and not all security risks can be mitigated with technology alone.

Many management books are written about securing the piece of the puzzle called *"people"*. But the best advice can be summarised as follows: *"Hire the right people and treat them right."* [8]

Even well-meaning individuals can make a mistake that leads to a security breach and the controls will only be effective if the team perceives them as a safety net that protects them, rather than surveillance that treats them as potential criminals.

Training is important, and the right training for the CI environment is not compliance training, with contrived video clips and multiple choice tests. New developers joining the team should not be left to figure out the security policy around CI by trial and error, or expected to learn it by osmosis.

When putting together developer-specific training (which could be as simple as a wiki page; content is more important here than presentation), it is important to assess whether your culture undermines your policies (see NCC Group blog *"Does your organisational culture undermine your resilience to social engineering"* [9] for a detailed example).

The rest of this section looks at the security of some important CI processes.

4.1 Review all things

To get the best return on investment in security activities, start them as early in the lifecycle as possible.

A feature that did not have secure requirements, was not designed with secure design principles in mind, skipped threat modelling and attack surface reduction, will be prohibitively expensive or even impossible to secure at the integration point.

These pre-development activities are out of scope of this paper. However, it is worth noting that they do not have to be expensive or slow down rapid development lifecycles. Even within extreme programming (XP) practices, a pair that picks up a story with the goal of completing it the same day, starts with having a conversation about this story. Developers with secure design training would be able to catch violation of secure design principles or changes to the overall threat model as part of this conversation.

4.2 Code review

Manual code review normally happens after step (1) and before step (4) in the flowchart.

A well-disciplined team can get away with doing a code review after step (4) if there are additional gateways on the way to continuous deployment.

Assuming that trivially-automated coding rules are checked before investing the expensive human effort, the following principles are useful for security-oriented code review:

- Coding standards need to be kept up to date with security recommendations specific to the technology stack. If there is an industry-accepted secure coding standard for the languages used in development, it is best to adopt it (see CERT standards for an example [15]).
- No change is too small to be reviewed. Sometimes, teams have a policy to only manually review risky changes, but you do not know if something is risky until it has been reviewed.
- Checklists derived from the coding standard can be very useful.
- In Agile development, do not have a separate task for code review of a story. Instead, add code review to your “definition of done” (see *Agile Alliance* definition [10]).
- Continuous integration is all about quick feedback – do not save up code reviews until the end of the project/iteration/week.
- Have a defined process for the “reject and rework” cycle – it is part of normal development, not somebody’s exceptional personal failure.
- If tools allow (e.g., JIRA or TFS), make code review an explicit mandatory step of a story workflow.

It is important to record evidence of the code review. Some tools used for code review management can be integrated with version control and the name of the reviewer is recorded as part of the commit information. Alternatively, the evidence can be kept in a workflow management/storyboard type of tool.

Pair programming can serve as a continuous code review if there is a safety net to catch and review the cases where a “trivial” change was done without using pair programming.

4.3 Chain of custody

Whether you release every change that successfully passes integration checks, or collect multiple changes together to be released on a different cadence, it is extremely important to have a precise control over what is going to be released.

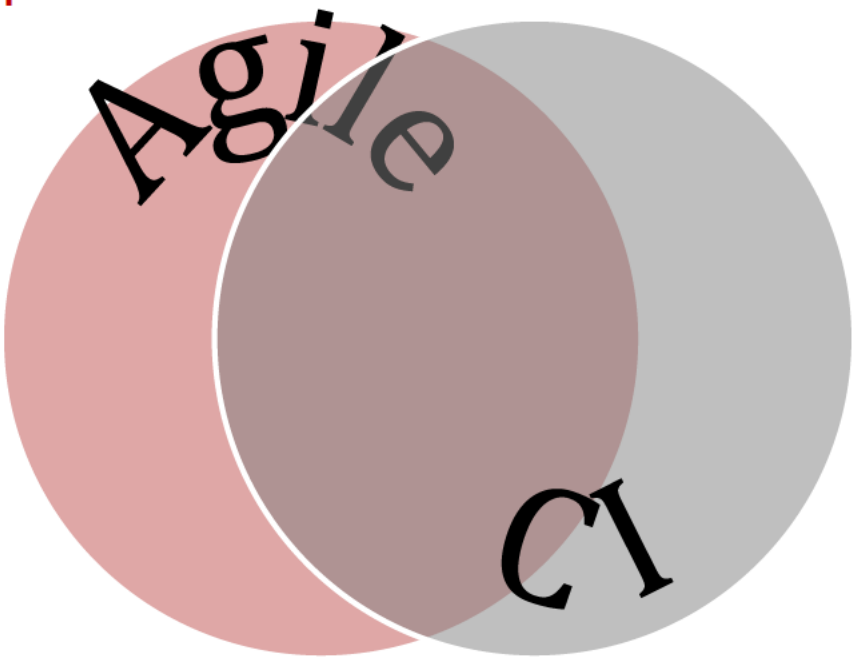
Your process and your tool chain must support the ability to inspect exactly which units of work (such as tasks, stories, change lists etc.) are going into the release and to validate that each of these units did not bypass any of the CI checks and was not modified since passing the checks. Without

implementing this security concept your CI environment is vulnerable to the threat of backdoors.

The exact implementation depends on the tools used and on the branching strategy, but to validate it you can ask yourself the following question: Can I trust my release notes to contain the truth, the whole truth, and nothing but the truth?

Takeaway	"Hire the right people and treat them right". Provide training, tools, and processes to support the development team in achieving their goal of releasing secure software.

5 Detour: How continuous integration relates to Agile development



The overlap between Agile development and CI is massive, but it is not 100 per cent. It is worth having a closer look at the non-overlapping areas.

First of all, the area on the right: CI in a non-Agile environment. What is it?

It could be some version of a Lean process, for example, or any other iterative but not strictly Agile environment. These types of SDLC frequently use CI, especially Lean. Or, it could even be a waterfall process.

The author of this paper once worked in a waterfall environment with CI. As a team, we could get away with not being Agile as the releases were every 12-18 months. We had the luxury of working directly on the main branch (sometimes called *master* or *trunk*, depending on the version control tool), as each version was released before any development work on the next version was started.

In this - by modern standards - slow environment, CI was extremely useful: continuous build, automated tests, SAST and fuzzing rig all ensured that the quality of software was very high even

before the Quality Assessment (QA) handover. Therefore, QA only ever found a handful of bugs which could be fixed in time for the release, without descending into the infinite loop of rework-retest which kills most waterfall projects.

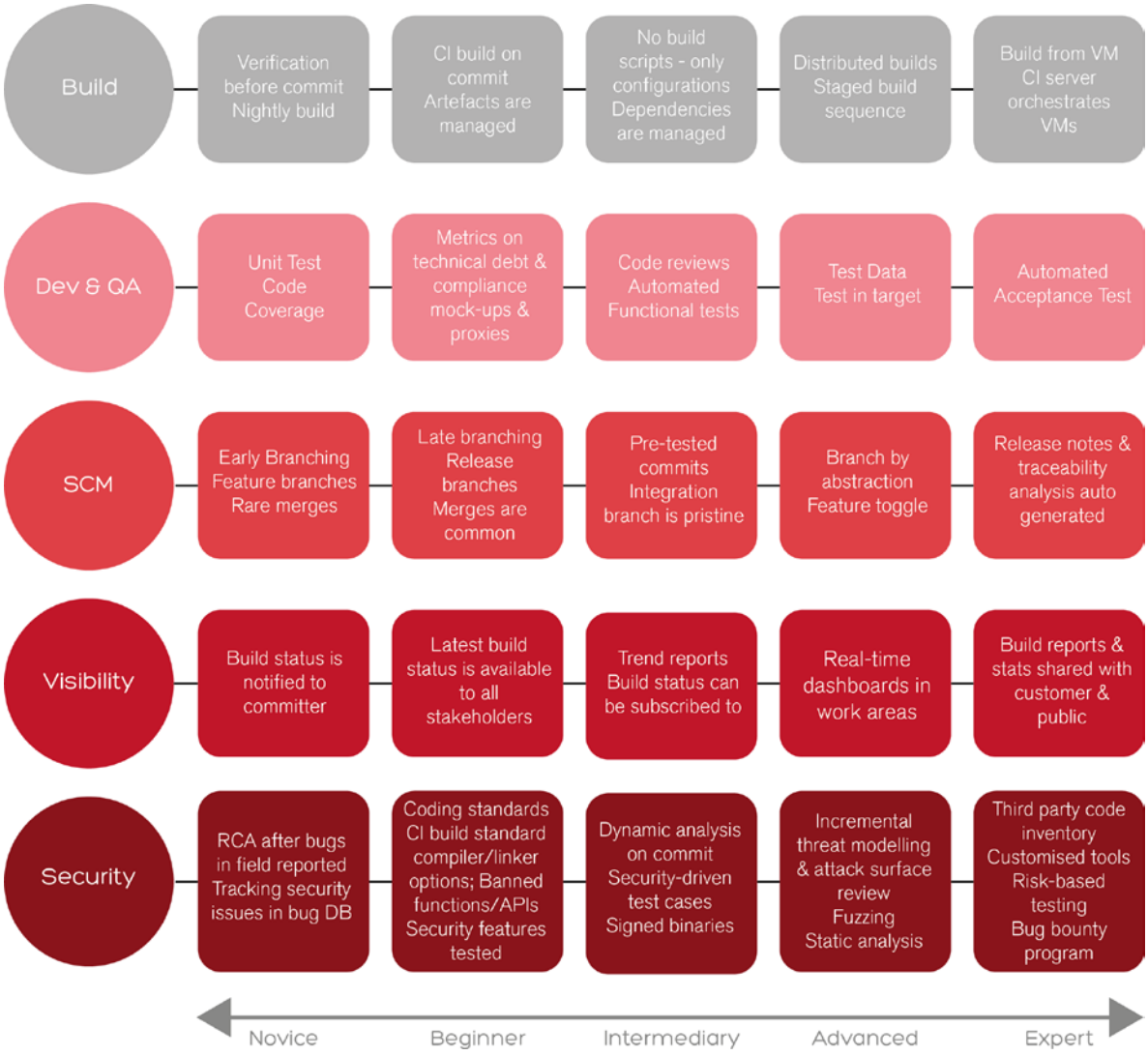
So, to summarise – CI has a place in non-Agile lifecycles as it has a positive impact on the software quality, regardless of the frequency of the releases.

What is the area on the left, then? An Agile environment without CI? It has to be a shrinking space, no doubt. Any organisation that is serious about being Agile and actually applying Agile principles - rather than just decorating the walls with cards and charts - will find itself moving fast towards CI. This would, at the very least, be through self-organising teams improving their process, if not through upfront organisational decision.

Therefore, if you are working in an Agile environment, but not yet doing continuous integration, you should be aware that it is coming your way, ready or not. It would be better to be ready and to pay attention to the security of your CI from the beginning.

For more information on securing Agile SDLC, see Microsoft guidance [17] or OWASP resources ([18] and [19]).

6 Putting it all together



The figure above is based on several popular maturity matrices for continuous integration, as well as NCC Group experience with a variety of clients.

There is now an understanding in the software development industry that security features are not enough for building a secure product. To achieve an acceptable level of overall security, the product needs all features to be secure; the whole attack surface of the product must be considered from a security point of view, not only features that are directly security related, such as login functionality.

The same is true for securing continuous integration. In the matrix above, the dedicated '*Security'* row lists explicit security related activities that must happen at some point towards CI maturity. However, every cell on the other rows needs to be evaluated for the security of its tools and processes as well.

This paper has so far concentrated on securing the CI environment, rather than adapting the environment to contribute to the security of the developed software. In this section the focus will be on the practices which can help to deliver secure software, i.e. the dedicated *'Security'* row.

6.1 Root-Cause Analysis (RCA)

If you are at the very beginning of your security journey, the best feedback loop you can introduce at this stage is root cause analysis for externally reported security issues. Each of these should be discussed from the point of view of the progress gap: what was missing in our process to allow this issue to slip all the internal checks and be released?

After this is understood, you can close the gap straight away, or add it to your security roadmap if it is impossible to fix on the current maturity level. In many cases, closing the gap will involve adding tests to the ones running as part of CI.

6.2 Unified bug tracker

In the early days of a security roadmap when your CI server does nothing apart from the build - it is a good idea to get into the practice of handling all the bugs in one place. Start adding security issues to the same tracker where your functional bugs are, instead of keeping them in a separate tool or, worse, a spreadsheet. Otherwise, when other CI practices start to mature and additional tools show up, your team will find itself not only juggling multiple bug sources, but also having separate triage meetings for each source.

If it looks like the current bug tracker will not cope, consider moving to a better one. A useful feature in a bug tracker is an ability to define custom fields. At the very least, you will want to be able to select all the security-related issues. Other custom fields might become useful later, such as a specific class of security bug, or method of discovery. Do not take a shortcut of trying to express these through a keyword in the issue description, or a phantom software component, as whatever time will be saved by it you will have to repay with interest later when it becomes impossible to use the tracker for deriving quality metrics or just searching older issues.

6.3 Coding standards

Many languages have tools that allow automated coding standards checks to be performed. The tools usually come with a pre-defined rule set and allow the addition of custom rules. The automated check does not replace code review, but it leaves human reviewers more time to deal with complex and non-trivial issues by automating the trivial checks.

CERT has secure coding standards for several popular languages [15].

Code complexity checks are possible for most languages and can be complementary to the coding standard. Another family of tools that can be applied here is searching for duplicate code; on the basis that code reuse through copy/paste is typically a breeding ground for unfixed security vulnerabilities.

6.4 Unsafe functions blacklist

Sometimes, a popular library or function dates from before the modern era of security awareness. The most famous example is Microsoft's CRT library, but there are lists for Java, JavaScript and other languages. It is good CI practice to keep an up-to-date blacklist on the integration server, to ensure the code base stays in line with the latest recommendations.

Note that this advice is certainly applicable to newly developed code, but a legacy codebase needs to be approached with care. There were cases where automated (or almost automated) replacement of insecure functions with their recommended modern versions caused security bugs, as the replacement did not preserve the intended semantics of the original code.

6.5 Security-focused testing

This is a large area of software development and covering it fully is out of scope for the paper. OWASP provide a good overall guide (see [11]).

A popular entry-level practice, especially in Agile environments, is to introduce an *"evil user"* role and add some stories about stopping this user from achieving a compromise. It may also be easier to concentrate new testing practices on security specific features initially (e.g., authentication) and extend to all the features as QA engineers get more proficient.

At the highest level of maturity you will extend the testing to third party code, because an attacker does not care whether the weakest link in your product was written by you, or just included from a supplier or open source.

No matter what your test suite composition in terms of unit tests/functional tests/component tests/system integration tests, as much as possible it needs to be automated. Automation allows mutating unit tests and functional level tests to support dynamic analysis.

Many teams have a tool that measures percentage of code covered by the automated tests and after achieving a reasonable overall coverage, they introduce a threshold for minimum test coverage of the new code commits. A useful threshold depends on the actual code base, but tends to be between 60 and 80 per cent. It might be that in your circumstances it is possible to aim for coverage closer to 100 per cent, but for many teams chasing that last per cent would not be worthwhile.

A frequent action point to come out of RCA (6.1) is creating a regression test case. This should also be used to apply some proactive analysis to find additional issues belonging to the same class of vulnerabilities. Repeated occurrences of related vulnerabilities may signal systemic weaknesses in the SDLC and require actions beyond what is required to deal with a single instance, e.g. introducing additional training or a specialised tool.

6.6 Signed binaries

All externally released binaries need to be digitally signed (see Wikipedia *"Code signing"* [12] for available tools on various platforms).

If your software needs to be installed by your customers, and especially if it can self-update, it is

essential that all the components are signed and the signature is validated before using an update.

Some specialised use cases require source code to be signed. ISO/IEC standard [16] describes a language-neutral methodology.

6.7 Key management

Introducing the practice of digital signing will require the organisation to manage the keys in a secure manner, otherwise the benefit is lost. Another use case that will bring the problem of key management into a CI environment is the software itself having encryption/decryption functionality.

OWASP has key management guidance [23], and additional information can be found in tech target report [22].

Guidance that is particularly relevant to CI:

- Do not use production keys in any other environment
- Do not embed keys in code or scripts
- Have a plan to deal with compromised keys

An excellent overview of best practices for secret management [24] considers pros and cons of several tools providing this functionality.

6.8 Incremental threat modelling & attack surface review (ASR)

This practice is often perceived as time consuming and therefore unsuitable for a CI process. However, it is the most efficient way to find design-level security issues, which, if missed at this stage, are very unlikely to be caught by any other means before the software is released. After the investment in training and building the initial model, each feature or story can go through incremental modelling and ASR without slowing down the development process.

An example of a successful adoption of this practice in an aggressively Agile environment was a team dedicating 15 minutes in one hour-long planning meeting to incremental threat modelling of the features going into the next iteration. Achieving good familiarity with the overall model and proficiency in the modelling techniques allowed the team to concentrate on the changes required by each feature and identify associated threats and increased attack surface in this timeframe.

6.9 Dynamic analysis

Whole books are written on the automated testing strategy and it would be out of scope for this paper to cover the various testing approaches.

From a security point of view, the following points need to be considered when introducing dynamic analysis to the CI environment:

- Usage of production data in tests
- The configuration of the installed tools
- Access privileges of the cloud-based tools

6.10 Fuzzing

Fuzzing is used to search for vulnerabilities in input processing by providing randomised data across

the entry points (network, file I/O, simulated user interaction etc.)

Fuzzing is a very useful technique and will almost always deliver some great initial results when introduced. However, it is subject to the problem of diminishing returns: as you find and fix the bugs found by fuzzing, the average time to find the next bug goes up, but the time for a motivated human attacker to find this next bug does not grow at the same scale.

To keep it useful beyond the initial quick wins from dumb fuzzing, you will have to invest time into providing better input templates and configuring more sophisticated fuzzing tools. Randomisation of initialisation values for the fuzzer is more likely to find additional issues, but care must be taken to preserve the data used on each test as it may be required to understand why a build failed testing.

It is computationally intensive and almost always will be done on a separate server. As you have probably realised by now, it means additional attack surface: how are the binaries transferred between the servers? Is sensitive information used in data seeds? Will the amount of data generated create self-inflicted DoS conditions?

6.11 Static analysis

Using static analysis in a way that aids rather than impedes development is not a trivial problem.

NCC Group has a whitepaper with recommendations on effective integration of Static Application Security Testing (SAST) tools [4].

Keep in mind that an installation can be effective in terms of helping the teams to improve the security of developed software and yet it may be insecure due to configuration mistakes or weak access controls.

6.12 Third party code inventory & vulnerability management

This practice requires the highest level of maturity and involves maintaining an up-to-date inventory of all third party inclusions (commercial and open source) and monitoring their security status.

When a vendor or security researcher releases a security advisory, you need a rapid process of evaluating the versions affected, making a risk-based decision and mitigating if necessary. This practice is usually implemented at a high level of process maturity.

As a starting point, make a list of what components are currently in use and introduce a process for keeping the list up to date with the correct version of each third party code item. Investigate if there is a practice of modifying or customising third party code. Even if the licence allows such customisations, it is not recommended from security point of view, as it prevents rapid transition to the latest version.

If your architecture requires third party code to be modified, it is better to build a wrapper layer around this code, therefore retaining the ability to perform drop-in upgrades when the vendor releases a critical fix.

The good news is that a mature CI environment, with high coverage by automated tests, allows quick and reliable validation of such updates.

If various components are used across multiple projects, a central repository of dependencies is vital; otherwise, a forest of dependency trees per project will get out of control. This does not preclude different projects using different versions or different upgrade processes.

Takeaway	Do not become overwhelmed by the variety of available security practices, you do not have to introduce everything at the same time. Assess your current maturity level and pick the logical next steps.

7 Conclusion

Continuous integration provides many well-documented benefits for software development teams, such as schedule predictability, code quality and team communication improvements.

However, when introducing various tools and practices, care needs to be taken not to reduce your organisation's security posture.

As you build up your CI capabilities, you will discover that beyond the direct benefits provided, it can also be extended to improve the security of the released software.

8 References & further reading

[1] *Continuous Integration: Improving Software Quality and Reducing Risk,* Martin Fowler Signature Books

[2] *Software Configuration Management Patterns*, Stephen Berczuk and Brad Appleton

[3] *Fuzzing*, OWASP https://www.owasp.org/index.php/Fuzzing

[4] *Best Practices for the use of Static Code Analysis within a Real-World Secure Development Lifecycle*, Jeremy Boone https://www.nccgroup.trust/uk/our-research/best-practices-for-the-use-of-static-code-analysis-within-a-real-world-secure-development-lifecycle

[5] *Security considerations in the system development life cycle*, NIST http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-64r2.pdf

[6] *Understanding and Hardening Linux Containers*, Aaron Grattafiori https://www.nccgroup.trust/uk/our-research/understanding-and-hardening-linux-containers

[7] *Abusing Privileged and Unprivileged Linux Containers*, Jesse Hertz https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers

[8] *Security Laboratory: Methods of Attack Series*, Stephen Northcutt http://www.sans.edu/research/security-laboratory/article/log-bmb-trp-door

[9] *Does your organisational culture undermine your resilience to social engineering?*, Irene Michlin https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/september/does-your-organisational-culture-undermine-your-resilience-to-social-engineering/

[10] *Definition of Done*, Agile Alliance https://www.agilealliance.org/glossary/definition-of-done/

[11] *OWASP Testing Guide v4* https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

[12] *Code signing*, Wikipedia https://en.wikipedia.org/wiki/Code_signing

[13] *Building a Vulnerability Management Programe – A project management approach*, Wylie Shanks https://www.sans.org/reading-room/whitepapers/projectmanagement/building-vulnerability-management-program-project-management-approach-35932

[14] *The Ultimate DevOps Tools Ecosystem Tutorial*, Noga Cohen https://www.blazemeter.com/blog/ultimate-devops-tools-ecosystem-tutorial-part-1

[15] *SEI CERT Coding Standards* https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards

[16] *ISO/IEC 17960:2015* http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=61133

[17] *SDL for Agile* https://www.microsoft.com/en-us/SDL/Discover/sdlagile.aspx

[18] *Secure SDLC Cheat Sheet*, OWASP https://www.owasp.org/index.php/Secure_SDLC_Cheat_Sheet

[19] *Secure Development Lifecycle*, Eoin Keary & Jim Manico https://www.owasp.org/images/7/76/Jim_Manico_(Hamburg)_-_Securiing_the_SDLC.pdf

[20] *Signing commits using GPG* https://help.github.com/articles/signing-commits-using-gpg/

[21] *GPG signature verification* https://github.com/blog/2144-gpg-signature-verification

[22] *How to manage encryption keys* http://searchdatabackup.techtarget.com/report/How-to-manage-encryption-keys

[23] *Key Management Cheat Sheet*, OWASP https://www.owasp.org/index.php/Key_Management_Cheat_Sheet

[24] *Secrets and LIE-abilities: The State of Modern Secret Management (2017)*, Jeff Nickoloff https://medium.com/on-docker/secrets-and-lie-abilities-the-state-of-modern-secret-management-2017-c82ec9136a3d#.tkcdd8hf3

