SECURITY
BY
DESIGN

WHITEPAPER

CODE MANAGEMENT

INDUSTRIENS FOND

SbD

SECURITY
BY
DESIGN

Code Management

Simon Dieterle René Rydhof Hansen
Christian Gram Kalhauge

Abstract

Code management as an explicit task or process is easy to overlook. A misconception is that code, when written, will remain operational forever. In practice, code degrades over time, and if it is not managed, it might go from being the thing that gives us the edge in the market to the thing that holds us back. This document is a gentle introduction to the area of code management, detailing common problems and essential processes that other companies have found useful in managing their code.

Contents

1 Introduction

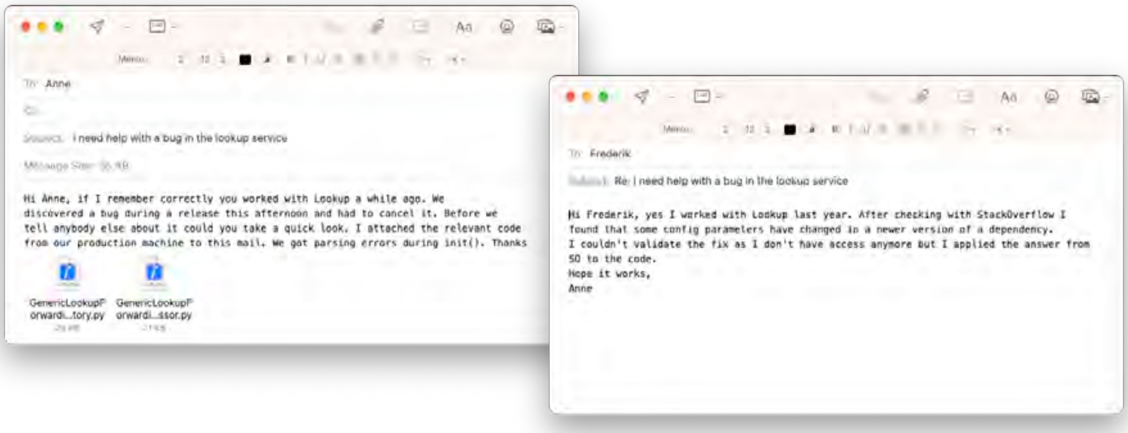
Code Management is ensuring and improving code quality over time. When we have met all the functional requirements, code quality is the only thing differentiating a good implementation from a bad one. Bad code is slow, insecure, and hard to debug. Worst of all, bad code is hard to change, which means that we cannot iterate as quickly as our competitors. Code quality quickly degrades when unmanaged, even when untouched by developers.

Managing code is hard. Software is extremely versatile: the fundamental processes that build the code that runs your dish-washer also run nuclear reactors and hospitals. When we range a gap this large, it can be hard to know what is expected and what we can do without. We have written this whitepaper for you, an Engineering Manager at a small or medium-sized company, as a guide to the processes that should go into your development cycle, which larger companies take for granted. We will describe each process, why we do them, and what we can do to optimise them. Optimisation is always costly, but it can be worth it in the economies of scale.

In this whitepaper, we will focus on code in the implementation and verification phases. There is also the exciting topic of code in production, how to track and debug failures, but we will leave that for another whitepaper.

If the following email chain reminds you of your work week, this is the whitepaper for you.

A tough week always starts with an email over the weekend...


To: Anne

Cc:

Subject: I need help with a bug in the lookup service

Message Size: 56 KB

```
Hi Anne, if I remember correctly you worked with Lookup a while ago. We
discovered a bug during a release this afternoon and had to cancel it. Before we
tell anybody else about it could you take a quick look. I attached the relevant code
from our production machine to this mail. We got parsing errors during init(). Thanks
```

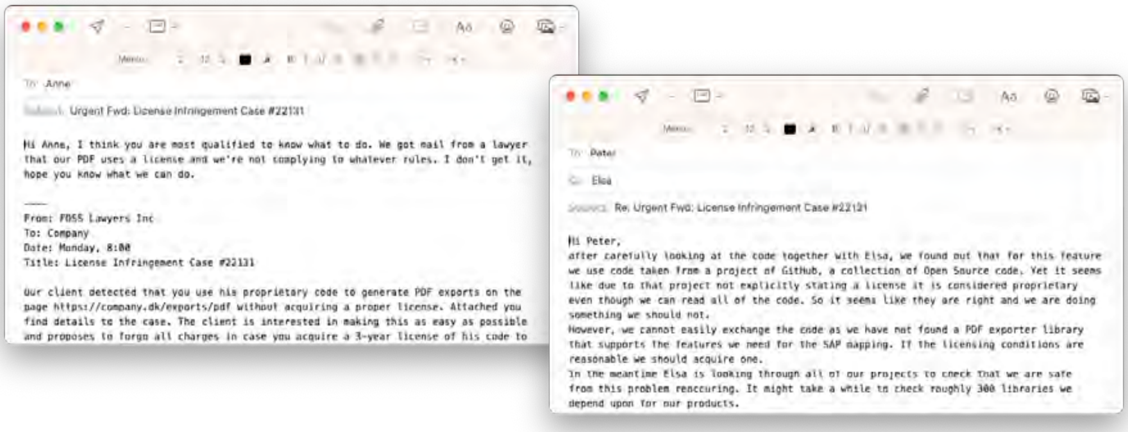
GenericLookupForwardi...tory.py GenericLookupForwardi...ssor.py

To: Frederik

Subject: Re: I need help with a bug in the lookup service

```
Hi Frederik, yes I worked with Lookup last year. After checking with StackOverflow I
found that some config parameters have changed in a newer version of a dependency.
I couldn't validate the fix as I don't have access anymore but I applied the answer from
SO to the code.
Hope it works,
Anne
```

...then it gets worse...


To: Anne

Subject: Urgent Fwd: License Infringement Case #22131

```
Hi Anne, I think you are most qualified to know what to do. We got mail from a lawyer
that our PDF uses a license and we're not complying to whatever rules. I don't get it,
hope you know what we can do.

——
From: FOSS Lawyers Inc
To: Company
Date: Monday, 8:00
Title: License Infringement Case #22131

Our client detected that you use his proprietary code to generate PDF exports on the
page https://company.dk/exports/pdf without acquiring a proper license. Attached you
find details to the case. The client is interested in making this as easy as possible
and proposes to forgo all charges in case you acquire a 3-year license of his code to
```

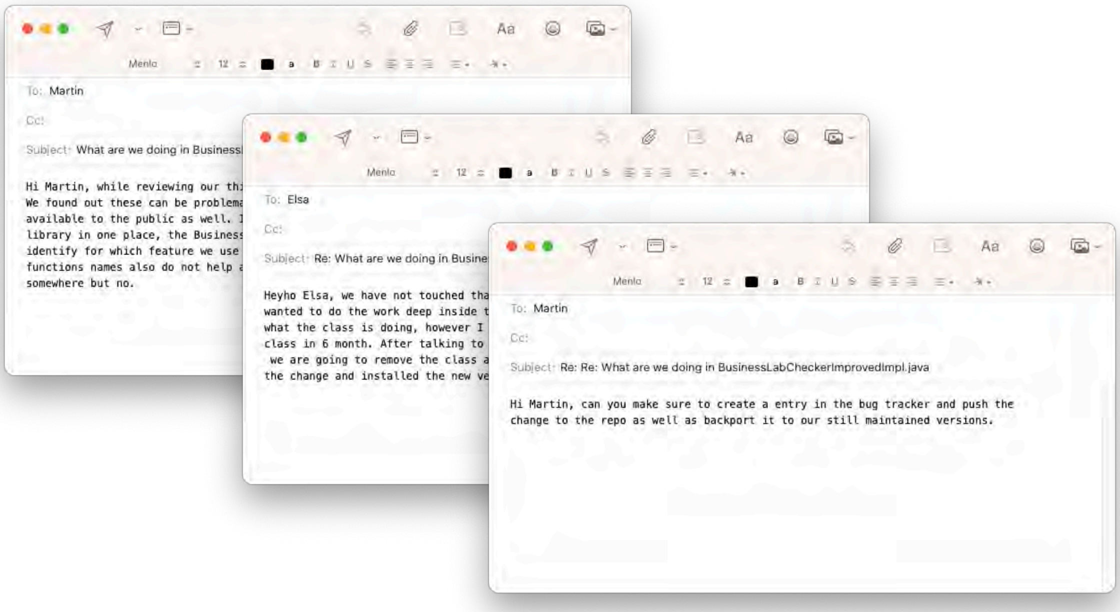
To: Peter

Cc: Elsa

Subject: Re: Urgent Fwd: License Infringement Case #22121

```
Hi Peter,
after carefully looking at the code together with Elsa, we found out that for this feature
we use code taken from a project of GitHub, a collection of Open Source code. Yet it seems
like due to that project not explicitly stating a license it is considered proprietary
even though we can read all of the code. So it seems like they are right and we are doing
something we should not.
However, we cannot easily exchange the code as we have not found a PDF exporter library
that supports the features we need for the SAP mapping. If the licensing conditions are
reasonable we should acquire one.
In the meantime Elsa is looking through all of our projects to check that we are safe
from this problem reoccuring. It might take a while to check roughly 300 libraries we
depend upon for our products.
```

...and worse...


To: Martin

Cc:

Subject: What are we doing in Business

```
Hi Martin, while reviewing our thi
We found out these can be problema
available to the public as well. I
library in one place, the Business
identify for which feature we use
functions names also do not help a
somewhere but no.
```

To: Elsa

Cc:

Subject: Re: What are we doing in Busine

```
Heyho Elsa, we have not touched tha
wanted to do the work deep inside t
what the class is doing, however I
class in 6 month. After talking to
 we are going to remove the class a
the change and installed the new ve
```

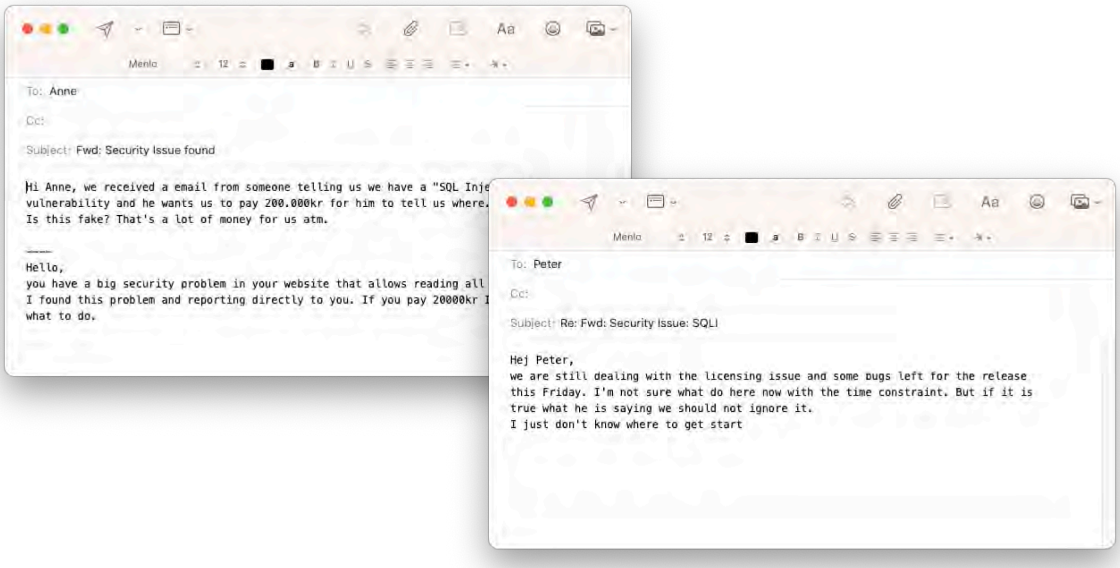
To: Martin

Cc:

Subject: Re: Re: What are we doing in BusinessLabCheckerImprovedImpl.java

```
Hi Martin, can you make sure to create a entry in the bug tracker and push the
change to the repo as well as backport it to our still maintained versions.
```

...and worse...



To: Anne

Cc:

Subject: Fwd: Security Issue found

Hi Anne, we received a email from someone telling us we have a "SQL Inje
vulnerability and he wants us to pay 200.000kr for him to tell us where.
Is this fake? That's a lot of money for us atm.

———

Hello,
you have a big security problem in your website that allows reading all
I found this problem and reporting directly to you. If you pay 20000kr I
what to do.

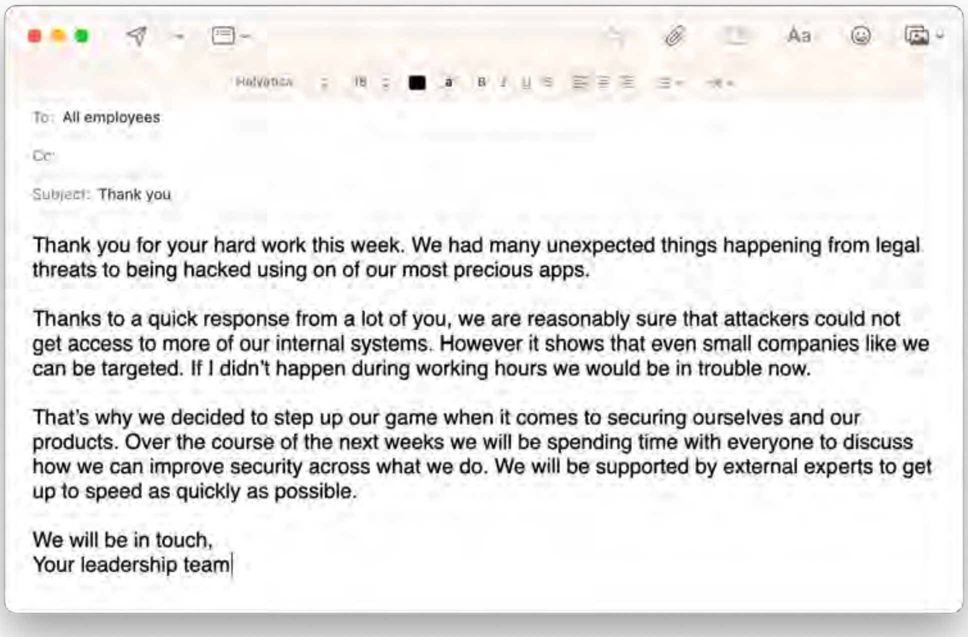
To: Peter

Cc:

Subject: Re: Fwd: Security Issue: SQLI

Hej Peter,
we are still dealing with the licensing issue and some bugs left for the release
this Friday. I'm not sure what do here now with the time constraint. But if it is
true what he is saying we should not ignore it.
I just don't know where to get start

...and may end with some lessons and more work.



To: All employees

Cc:

Subject: Thank you

Thank you for your hard work this week. We had many unexpected things happening from legal threats to being hacked using on of our most precious apps.

Thanks to a quick response from a lot of you, we are reasonably sure that attackers could not get access to more of our internal systems. However it shows that even small companies like we can be targeted. If I didn't happen during working hours we would be in trouble now.

That's why we decided to step up our game when it comes to securing ourselves and our products. Over the course of the next weeks we will be spending time with everyone to discuss how we can improve security across what we do. We will be supported by external experts to get up to speed as quickly as possible.

We will be in touch,
Your leadership team

2 Categories of Code Mismanagement

Mismanaged code often falls into a simple set of categories. For us to avoid these five categories, it is essential to learn about what they look like and what they entail.

2.1 Unspecified Code

Unspecified code is code the developers no longer know how is supposed to work.

(Software is cheap, specifications are expensive) Software is cheap. Most software engineers are incredibly productive if they know what they are coding. The expensive part is figuring out what to code: the specification of the code.

(What is a specification?) A software specification is a description of what a piece of software should do. They exist both at a high level, as presented to the customer; mid level, as design documents specifying the architecture of the software; and low level, detailing how an individual piece of code should work. Specifications range in complexity from a nice comment describing the code, illustrative usage examples, or thorough mathematical models. We will focus on the low-level specifications.

(Specifications are important) All kinds of specifications are hard-earned, either by long conversations or prototyping with the customer or by continuous optimisation and debugging of code. But good specifications allow you to make better decisions and iterate faster on a product without fear of breaking the system. Therefore, these specifications, and the domain-specific knowledge attached to them, should be considered the most valuable part of any software company.

(Low-level specifications are lost over time) Sadly, if left unmanaged, low-level specifications are seldom written down by developers. Of course, you can hope that other developers can read the code and understand the underlying specification, but this is rarely the case. In the extreme, the specification of the code is stored in one or two individuals' brains. And, if they leave, the code is suddenly unspecified.

(Code without specifications is hard to use and change) The specification is the interface between the users of the code and the implementers. The problems with unspecified code therefore affect both. First, code without specifications is hard to build upon and use. Second, it is hard to change the existing code without breaking some application which depends on it.

(Three sides to a specification) Specifications come in three flavours. Functional requirements which is about what the system should be able to do (also known as liveness properties); quality requirements which is about what the system should never do (also known as safety properties), and finally developmental requirements which is about how it is supposed to be extended and changed to maintain the other requirements.

Signs that code is unspecified:

- Code is "unreadable" for some developers, making collaboration hard.
- Out-of-date or non-existing documentation.
- Onboarding of new developers might take a long time since there are no good ways to learn about the structure and intent of the code.
- Code changes often break the system because code invariants (things believed to be true about the code) are invalidated.

To ensure that code remains specified over time, we recommend looking into the following processes:

- To maintain the specification of a system, the most important process is Documentation (3.4). Code comments and detailed design documents can get you a long way.
- The choice of Code Style (3.5) and Language (3.9) is important as it can make the code easier to read and reason about. This enables the specification to remain in the code for longer.
- Code Review (3.1) is a crucial process, not only to check that the code works but also to check that the intention of the code is readable by others.
- Finally, an overlooked source of specifications is Testing (3.7). When writing tests, we actually give examples about how to use the code.

2.2 Unverified Code

Unverified code is code that the developers no longer know works.

(Verifying code is convincing yourself and others that your code works) Verifying code is the process of ensuring that the code lives up to its specification. Assuming the specification is still known to the developers, there is still work left to ensure that the code works correctly. Unverified code might work, but we do not know for sure. When developing a new feature on an unverified codebase, the developers will constantly be checking not only their own code but also the codebase. This makes bug isolation a slow and painful process. A developer who works on a codebase they do not know and understand will develop slower, be less agile, and use more time debugging.

(Making something that works is different from proving that something works) The goal of a developer is not only to write code that works but to convince any future developer that it does. This can be done with documentation, tests, static analysis, types, or mathematical proofs.

(The developer shapes the tools and the tool shapes the developer) When writing code that can be verified, you need to be mindful of the tools you will use to verify it. Code can quickly become untestable or have many false positives when analysed with a static tool. When using a tool over a long time, the developer will write code which the tool accepts, greatly improving the value of the tool.

(Verification is not a one-time thing) The code that worked on a developer's machine yesterday might not work on the production server tomorrow. Code is rarely used in isolation but depends on other pieces working correctly, and a seemingly small bug in a library can cascade quickly in a codebase. Verifications should, therefore, be done often and possibly automatically.

Signs of unverified code:

- Code contains a lot of bugs, which need to be fixed after deployment.
- Components integrate poorly, and internal errors are common in production.
- Time to market is slowed as new features await bug fixes.

To ensure that code remains verified over time, we recommend looking into the following processes:

- The primary process for maintaining verified code is the Code Review (3.1); here another developer can inject that they have not been convinced the code works. Some of this work can be done automatically using Static Analysis (3.6).
- Another good source of verification is Testing (3.7). The good thing about testing is that it can often be done automatically.
- To increase the productivity of developers and guarantee that a piece of code gets verified the same way every time, consider Continuous Integration and Delivery (3.3).
- For specific critical systems that cannot fail, we recommend looking into Formal Methods (3.8).

2.3 Unlawful Code

Unlawful code is introducing legal challenges.

(So easy to break the law) There are many legal pitfalls to be aware of when developing code, especially when using third-party code. This can have huge consequences for the product.

(Unlicensed or invalidly licensed code) While a lot of source code is freely available to look at and read or study, it still can be impossible to use. Public source code repositories like GitHub contain a multitude of projects where the license is not explicitly stated. This makes them proprietary and unusable for others to build upon even it is just copying 3 lines of code for whatever small problem it solves.

(Licensing challenges) Most projects on GitHub, however, use licenses from a common pool of Free and Open Source licenses. Not all of these licenses allow the use of their code without restrictions and may introduce challenges for other projects. Even Open Source licenses can restrict usage in countries or in industries or require projects that use code under these licenses to publish the source code.

Signs that code might be unlawful:

- Unorganised use of third-party libraries.
- Use of libraries with restrictive or conflicting libraries.

Solutions to maintain lawful code:

- First and foremost Dependency Management (3.2). Documenting the allowed libraries and licenses will enable the developers to make better decisions when including new libraries.
- When new libraries are included, a Code Review (3.1) carried out by more experienced developers can catch many of these problems. There also exist Static Analysis (3.6) tools that will automatically check for problems.
- Finally, training developers in best practice is invaluable.

2.4 Unprotected Code

Unprotected code is code that is not protected after development ends.

(Bad configurations and misuse of libraries) Security vulnerabilities come in three kinds: exploiting bad design, implementations, and configurations. The design and implementation should be okay if the code is specified and verified. This leaves us with bad configurations. Many libraries and frameworks require very precise configurations to be used securely. And this configuration might even be different from development to production. Configuration requirements might also change over time. Therefore, it is important to keep track of the libraries used to ensure that they are all configured correctly.

(Supply chain mistakes and attacks) The quality of a piece of software does not end when the developer commits code to a repository. After a commit, the software goes through building and packaging before being shipped to the consumer. Any of these points have potential for misconfiguration, mistakes, and potential attacks. The more complex and manual it is to deploy, the longer it will take to iterate over a product.

(Source leaks) Furthermore, most systems for secure communication rely on secrets, which should not be shared with the public. Sometimes these are accidentally leaked to the public. Taking extra care to protect keys and developer passwords should always be a priority.

Signs that code is unprotected:

- Code in production is pulled back to fix security problems.
- Passwords and keys are stored in repositories or on machines.
- Debug mode is active in production.
- It takes days to produce a new deliverable because some steps need to be done manually.

Solutions to protect unprotected code:

- To better protect code, it is important to use Documentation (3.4) to describe the steps needed to publish it.
- A particularly effective kind of documentation is "Infrastructure as Code", which is part of Continuous Integration and Delivery (3.3). When we write down our publish steps, we can also check them in a Code Review (3.1) or with a Static Analysis (3.6).

2.5 Unmaintained Code

Unmaintained code is code that no developer is currently responsible for.

(Somebody needs to be in charge) Code is a living thing, even when it looks feature complete. Suddenly, a vulnerability in a library is found, or an online service on which the code depends is taken down. Changes need to be made, but if nobody is maintaining the code, who is responsible? A common problem with unmaintained code is that bugs and feature requests fall through the cracks with no one in charge.

(Maintenance is required) Like all other infrastructure, code needs to be maintained. If not, it will quickly turn into legacy code. The code specifications change over the lifetime of the code. The code then needs to be refactored to allow for faster changes in the future. With no resources allocated for maintenance, the code will rot and gradually become outdated, accruing technical debt that will later need to be paid, or the project will have to be abandoned.

Signs that code is unmaintained:

- Bugs and feature requests are lingering for months or dropped completely.
- Code uses increasingly obsolete libraries, and over time, it becomes harder to make changes.

The main process for maintaining code is to prioritise it. We can write down who is responsible in Documentation (3.4) and then have a process in place for maintaining the overview of all the products in production.

3 Essential Processes

Now we know what mismanaged code looks like, what can we do to avoid it? Here is a list of processes that support quality code development. Most of the processes are manual but with a little work, and a lot of caution, we can automate some of the work with tools.

Finally, there also exists a process, Software Process Improvement (3.10), which is about how to adopt processes.

3.1 Code Review

The code review is the process of having another developer look through the changes to a project before accepting it into the code repository. Reviewing code can protect against many different varieties of threats from external to internal and from mishaps to intentional malicious behaviours.

To get started with code review, the first task to tackle is to make sure that enough control can be achieved so that all new code undergoes a review before it reaches the code repository. This can be done by introducing organisational and technical controls. On the technical side, we can use Branch Protection to ensure no changes can be directly pushed to a branch in the repository without going through e.g., a pull request or other forms of peer review. The second task is to improve the review itself. Here something as simple as a checklist can dramatically increase the coverage and consistency of the review.

(Check for logical errors) The most important part of a code review is to check that the logic is correct. Logic bugs cannot be found automatically in a suitable way for every scenario. In general, a reviewer should spend most of their energy reviewing logic.

(Then for the rest of the processes) After this, the goal of the reviewer is to check that all the other processes adopted by the company (see the other processes) have been done correctly. A good code review should check that the code is well documented, tested, and maintains the style of the company.

(The code review is customisable) The code review should focus on the risks to be avoided. Based on the team and organisation's history of bugs or incidents, new review steps can be introduced. E.g., if there have been problems with third-party libraries breaking your code, a task to review newly added dependencies makes sense.

(Automate with caution) Many checks in the code review can be automated by a computer as part of CI/CD, which can remove some of the work from the reviewer. Ultimately, the reviewer should have the final word and the responsibility.

(Reviewing is valuable time) Lastly, for code reviews to work inside an organisation, their value has to be recognised across the metrics of how performance is evaluated. Reviewing code is and should be as valuable as writing it.

3.2 Dependency Management

Dependency management is the act of maintaining a list of all the libraries and tools that we use in the development of software. This is critical both for checking for vulnerabilities and for legal reasons.

To get started with dependency management, you can write down a list of accepted tools and libraries in a shared spreadsheet. It is important to include versions and licenses to this list. There are now tools as well that can automatically extract the libraries from a project and check them up against the list or against publicly available and known vulnerabilities.

3.3 Continuous Integration and Delivery

Continuous Integration and Delivery (CI/CD) is the automation of the build and packaging process. This serves two purposes: The first, and most obvious, is faster and more frequent releases, which works well with agile development. The second, and possibly more valuable, is the complete documentation and strict adherence of the source code to customer pipeline. The more we automate, the more we can check and validate automatically.

To get started with CI/CD, we first recommend switching to a modern version control system (VCS) like Git. Most of the commercial tools like GitHub and GitLab integrate best with it. The next step is to set up tasks to compile and run tools as part of the code review phase. We can later try to move towards deploying automatically. However, the stakes and complexity are higher here.

(Infrastructure as Code) There are many tools that try to make development and operations (DevOps) into a joint success. To get the most bang for your buck, we can change to infrastructure as code. Here we can specify what the deployment server or router configurations should look like, and how they can perform smoothly with different cloud solutions[1]. By having everything in code, we can potentially check that it is done correctly, or at least that it is done consistently. This includes security related configurations like access to storage, databases or things like management of secrets and keys.

3.4 Documentation

Code Documentation is writing down the intention of a piece of code. While good documentation is important for any system, it is *essential* for a secure system. Here documentation covers both the technical description of specific details, choices, and things to pay attention to in the (source) code of the system, as well as the more abstract and general system documentation.

To get started with documentation, start by using a documentation system. Most languages have built-in standards for documentation and also solutions that can generate websites. This allows developers to quickly look up the important intentions of the system.

(Well-documented code is easy to review) Among other things, well-documented code makes it easier to review the code, especially for reviewers that are not familiar with the code base, e.g., external auditors or security consultants. Through the use of specialised tools, it is possible to automatically extract certain documentation directly from (comments and annotations in) the code, e.g., API specification or documentation of library functions. This provides a simple way to keep critical documentation up to date with relatively little overhead.

(Documentation can go stale) Like all other techniques and processes in this document, documentation is not free. The biggest threat to a documented code base is that the documentation goes stale over time. It therefore requires manual inspection from time to time to ensure that the documentation is up to date.

(Code comments can be checked) Code comments or annotations may also be used to document the "code contracts" that source code functions abide by, i.e., what must

[1]But remember "Cloud" only means "run on somebody else's computer"

be assumed of the input and what is guaranteed of the output. Such "design by contract" can substantially raise the assurance level of code and has been formalised and incorporated into languages and development environments for high-assurance platforms, e.g., SPARK ADA that allows code contracts to be formally verified (see Formal Methods (3.8) for more).

(It is important to document the development as well) System documentation provides a means for communicating the policies and organisational processes that must be in place to ensure secure deployment, operations, and end-of-life of a secure system.

To go further, consider **code annotations** which allow you to specify contracts of code, and **doctests** where you can give examples in the documentation and mechanically ensure they work over time.

3.5 Code Style

A code style is an agreement on how to write readable and correctly formatted code. We can write a single piece of software in many ways, and even code that fundamentally works the same way can still look different. This can be problematic as it can be hard to read code if it uses formatting that the developer does not recognise. A code style guide is a description of how code developed by the company should look like.

To get started, know that code style can be very hard to get people to agree on. First start out by figuring out the format used in the projects and document patterns and styles used in the code so far. Then move towards definition. Once a suggested style guide is in place, the developers' IDE's to conform with the guide can be set up. Finally move towards checks in the CI/CD.

(Not only whitespace) Code style is more than whitespace: we can require that the variable names are formatted to make intent clear, we can ban some features in a language, we can require that we only use some specific patterns.

(Automating the style) There exist tools that can both check and to some extent refactor the code to match a specific style. Code formatters check and verify the formatting of a document. Code linters check more complicated problems, and some even use some Static Analysis (3.6).

To go further with code style and formatting, you can read and possibly adopt Google's code styles[2], which are also supported by many automatic tools.

[2]https://google.github.io/styleguide/

3.6 Static Analysis

Static analysis is a technique for automatically analysing the full source code of an application in a comprehensive and systematic manner to find potential security vulnerabilities and general code quality issues. Static analysis works at *build time*, e.g., as part of a CI/CD pipeline. Examples of bugs found include determining potential use of uninitialised variables, accessing memory that has been de-allocated (also known as "use after free"), possible buffer overflows, and many more.

To get started with static analysis, the simplest way is to enable all warnings in your compiler. Many modern compilers already include sophisticated static analyses, including security specific analyses. Enabling these analyses *by default* is a quick and easy way to get started. We can use it as a risk free first step to learn how to incorporate hints and warnings into our developmental cycle. Do note that compiling with all warnings will typically generate a *lot* of warnings – at least the first few times, especially when used on older code bases. We recommend starting on small well-defined modules and building from there.

(Reducing false positives) To get the most utility from static analysis, and to reduce the number of *false positives*, i.e., analysis alerts that do not correspond to a real bug, the analysis tool must be configured, or "tuned", to take into account specifics of the code base under analysis.

(Coding for verification) In addition to "tuning" the analysis tools to the specific code base, it is also recommended to "code for verification", i.e., to adopt a code style that facilitates analysis and verification and avoids or minimises code patterns and language features that are hard to analyse. This will typically also make manual code review easier and more thorough.

(Sound analysis) Some analysis tools called *sound* analysis tools can be used not only to find potential bugs but also to *prove* that certain bugs cannot occur in the program under analysis. Such tools typically produce more false positives but give stronger guarantees, see Formal Methods (3.8).

3.7 Testing

Testing a piece of software is running it and observing the behaviour. This can be done by a human or by a machine. Testing is a great way to increase the quality of the code: it serves as a straightforward verification tool and as a crude specification. Actually, good tests are examples of how to use the software you have built.

To get started with testing, the easiest way is to write down our acceptance tests, i.e., the tests that ensure the product works as indented. Make sure they get performed for every release by a Quality Assurance (QA) officer. Moving forward, it is important to design systems so they *can* be tested. It can be very hard to write tests for systems after they have been written. Requiring tests from the beginning of a project can both make it easier to test and improve the quality of the software.

(Run the tests, often) When we have written a lot of tests, it is important to actually run them. This can be done either by the developers, as part of the CI/CD system, or by QA in development. The closer the developer is to a test that fails, the faster they can detect the mistake and fix it.

(Finding a testing balance) Tests can never prove that the code works, but using a code coverage tool gives us a metric for how well we have tested the code. Striving for a high code coverage score is great but can lead to over-testing. Over-testing is when the developers use more time on writing tests than writing code. A rule of thumb is to only test visible effects. For example, in a login system, we should not test that creating a user creates a user row in the database. Instead, we should test that after creating the user, we can login as that user. Treating the database as an invisible part of the system will enable you to change that part of the code without breaking the test.

There exist many tools and techniques for making testing easier and more exhaustive. To learn more, look into **random testing**, **fuzzing**, **mocking**, and **test driven development**.

3.8 Formal Methods

Formal methods is an overarching term for the application of mathematics to different aspects of software development, including specification, design, and implementation. By using mathematical tools, it is possible to model and predict certain software behaviour with a high degree of accuracy and, e.g., prove that this is within specification or that entire classes of errors and bugs are not possible. Due to their cost, formal methods are mostly applied for systems of *high criticality* requiring a high degree of assurance.

It is not easy to get started with formal methods. As a rule of thumb, formal methods are costly, time consuming, and challenging to apply, especially after the code has been produced. To get full value from formal methods, they should be used across the development life cycle of an application. However, using a *sound* static analysis tool can be a good entry point for understanding some of the potential (and resources required) of formal methods. You can also move to strongly typed programming language, e.g., TypeScript, to get a sense of benefits and discipline needed to use formal methods.

(Other benefits of formal methods) In addition to proving correctness or absence of bugs, formalising (parts of) a system also enables more powerful analysis of a system to be performed using state-of-the-art mathematical tools and methods, e.g., analysing possible race conditions or what happens if a rare fault occurs.

(Different approaches to formal methods) Although formal methods are a broad concept, they typically involve one or more of the following approaches:

Types One of the most widespread and easiest formal methods to get started with is also one of the most overlooked and under-appreciated: *types*. Most modern (compiled) programming languages are typed, meaning the compiler ensures that operators are only used on operands that "make sense", e.g., integer addition is only applied to integers. Programming languages with strong(er) type systems allow programmers to enforce a wide variety of security and correctness properties through the built-in type system, e.g., that only authenticated users have access to areas of a web application.

Program Logics Taking the idea of types a bit further, a dedicated *program logics*, also called Floyd-Hoare logics, allows developers to annotate their code with formulae in a specialised logic enabling reasoning about program states and execution. This is related to **design by contract** and can achieve a high degree of assurance.

Testing Testing (see Section 3.7), in general, is not considered a formal method because it does not require a formal model of the target program and therefore only gives limited, if any, guarantees about program behaviour not explicitly covered by a test. Nevertheless, testing methodologies like property- and model-based testing incorporate aspects of formal methods and can thus achieve higher assurance.

Sound static analysis Static analysis is covered in more detail in Section 3.6, but we briefly mention it here, as static analysis is also considered a formal method – at least when the analysis is **sound**, i.e., correctly captures (through conservative approximation) the behaviour of the target program. Due to the nature of sound static analysis, they can be used to formally (mathematically) prove desirable properties of a program.

Model checking In principle, model checking is concerned with exhaustive exploration of a system's state space, verifying that all reachable states satisfy a user-specified condition, e.g., that there is no deadlock. Since even the smallest software applications may exhibit a very large number of states, model checking typically works on an abstract state model of a system. Anything that can be modelled can be model checked, e.g., entire systems, protocols, algorithms, or indeed code.

Interactive theorem proving Interactive theorem proving is a very general approach to formal methods for software verification, in which a system is formalised using a *proof assistant* and then manually proven mathematically correct. The proof assistant ensures the correctness of every step in a proof, resulting in a very high degree of assurance that the proofs are correct. It is typically resource intensive to develop such models and proofs and also requires significant training and expertise.

3.9 Language

As with all other communication, the choice of media is essential for the message. When writing code, our choice of language can fundamentally change the way we develop and design software. Different languages allow you to think differently, and some languages can greatly improve the security, quality, and productivity of the development.

To get started with newer languages, the easiest solution is simply to upgrade to a newer version of the language you are already using, or to use a new framework within the language you are already using. We can also use a newer language that can interoperate with the existing code base, e.g., Rust and Python can call C code, and Kotlin and Scala can call Java code. When adopting a new language, it is best to start on a small project and test if the new language has the intended effect, without risking complete failure.

(Developmental Upsides and Downsides) There are, of course, both upsides and downsides to choosing new and existing languages. The downsides are that new languages require new expertise and tools. It can therefore be a massive investment to change ship. Maintaining different languages also reduces the effect of an economy of scale and makes the company less productive. By having many languages in one stack, we risk making it harder for a single developer to get an overview of the code. However, not keeping up with modern technology is a risk in itself. New companies without existing language bases will choose the most efficient language for the task. The older technologies get, the harder it will be to find employees excited about them.

(*Memory Safety*) Over the past ten years, 70% of security vulnerabilities in Microsoft Windows have been memory issues. So if you do not need the explicit memory control or performance potential, consider using a memory-safe language like C# or Java. If speed is critical, consider Rust.

(*Types*) Types are a great way of explaining the intent and removing clear interface errors, as described in Formal Methods (3.8). It does sometimes require more code to convince the type checker that the code is correct, but that also often makes the code more readable by humans. Extreme examples are languages like Haskell or Scala that have types that can very precisely express the programmer's intentions. A joke in these languages is that they do exactly what you expect **IF** you can get them to compile. If you produce code in a non-typed language, you often upgrade gradually. Python has annotations, Javascript has Typescript, and PHP has Hack. Even language, like Java, can be improved by a few more types. Tools like the Checker Framework or the IntelliJ IDEA can statically detect null-pointer exceptions using code annotations.

To learn more about programming languages, consider looking at the Learn X in Y minutes[3] website, and look at the different programming paradigms like **procedural**, **object-oriented**, **functional**, **logical**, and **reactive**.

3.10 Software Process Improvement

Software process improvement, or SPI, is about continuously improving the software process. The fundamental idea can be summed up as having a dedicated group of people with ambassadors from both the management and the developers that can continuously solve problems in the developmental process and collect knowledge about state-of-the-art processes and tools.

The best way to get started with SPI is by creating a small team of people who want to see change. Start with a simple problem in a limited setting, so that the team can get some initial success and hopefully support from developers and management. SPI can be included in other approaches like the retrospective.

To learn more, a good start is the work by Mathiassen et al. (2007): "Learning SPI in practice"[4].

[3]https://learnxinyminutes.com/

[4]https://www.researchgate.net/publication/238785794_Learning_SPI_in_Practice

PROJEKTPARTNERE

ALEXANDRA INSTITUTTET

AALBORG UNIVERSITET

DTU

DI Dansk Industri

Erhvervshus MIDTJYLLAND

DANSK ERHVERV

DECEMBER 2022

Sb3D er et samarbejde mellem DTU Compute, Institut for Datalogi på Aalborg Universitet, Dansk Erhverv, Dansk Industri, Erhvervshus Midtjylland og Alexandra Instituttet. Projektet er finansieret af Industriens Fond.

INDUSTRIENS FOND

Sb3D SECURITY BY DESIGN