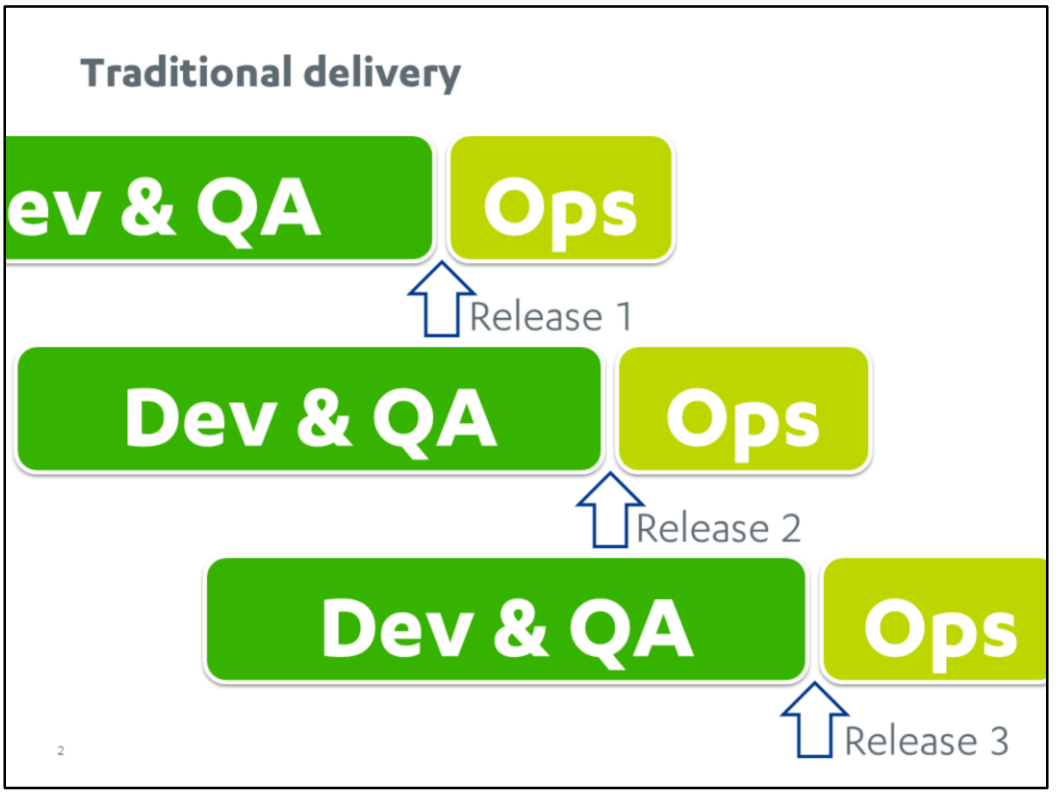
Software Security & DevOps

BSIMM Conference 2013

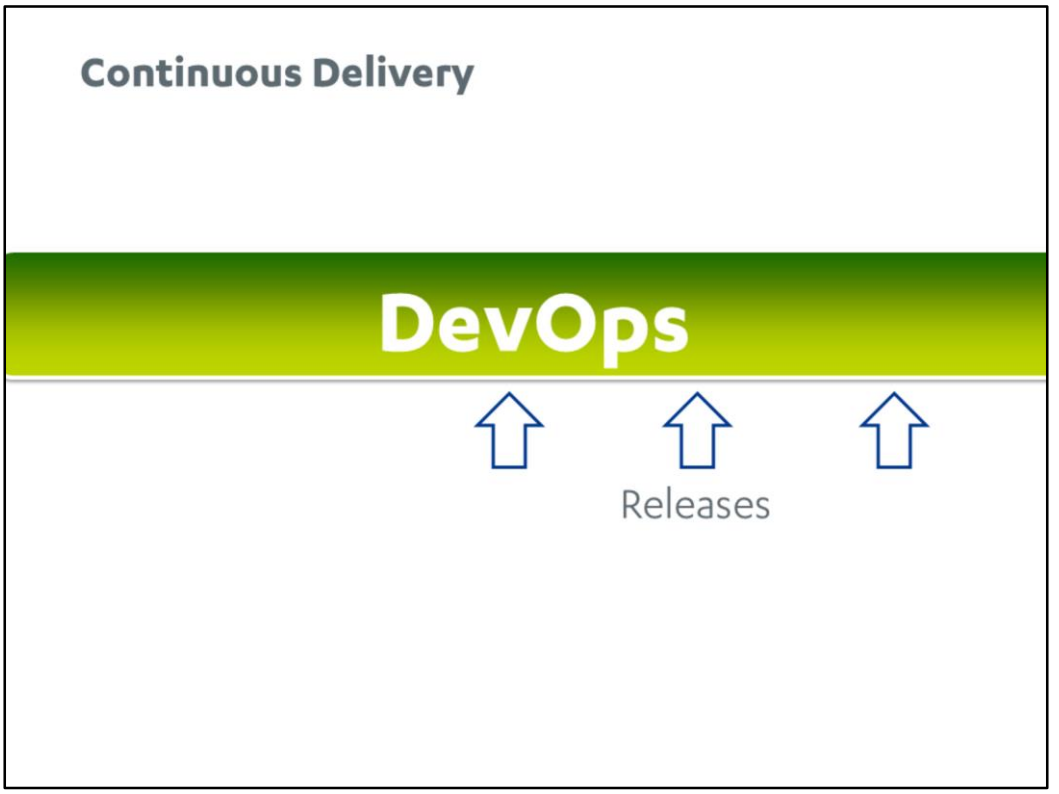
Antti Vähä-Sipilä
antti.vaha-sipila@f-secure.com
@anttivs



This is how I understand the “traditional” delivery to production. Developers and testing (which, in some cases, can even be further segregated) provide a release, and some sort of operations / hosting organisation owns it thereafter.

The main points are:

- There is a handover to production
- Development & QA may be working in parallel on several different releases (potentially branches)
- Maintenance may be segregated from development, sometimes even a different team
- In the picture above, it is assumed that new releases replace existing production release when released. Specifically in a case where there are several instances of a service, or the end result is a product that has its own lifecycle, each release may end up having a lifecycle that is much longer.



Now, the “DevOps” means that there is a very tight coupling between development and IT operations and hosting. I usually think of them being in the same team, or at least very, very close.

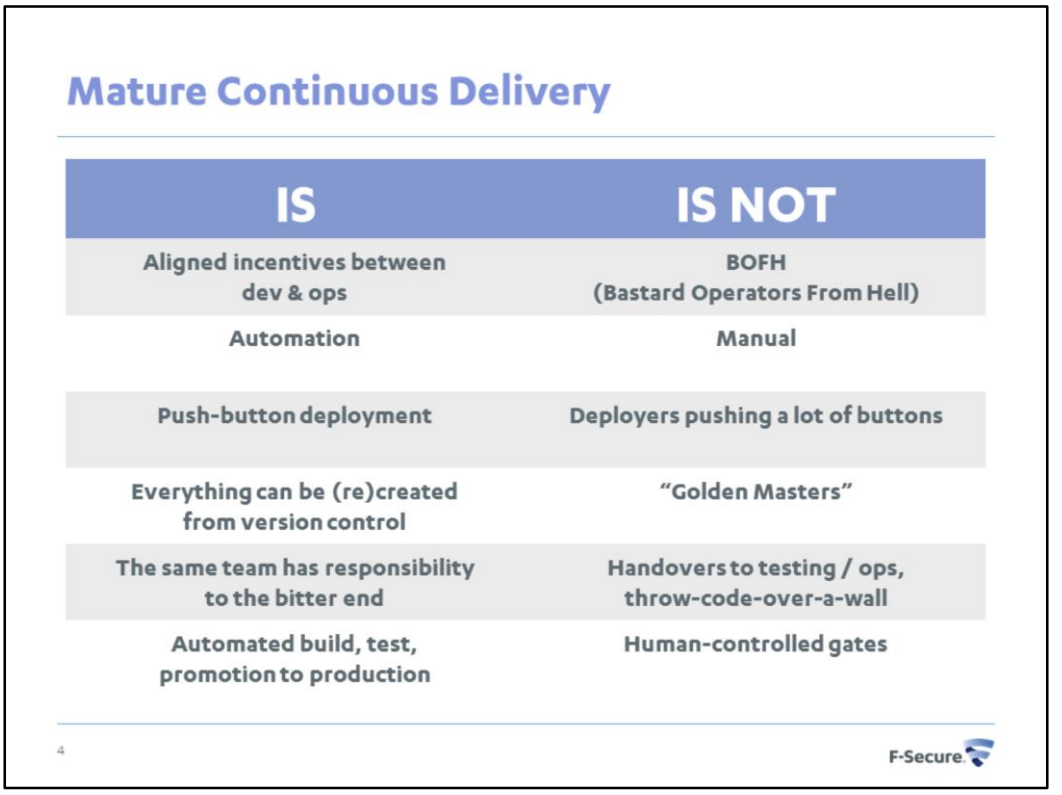
DevOps-type team composition, tooling, and a certain set of practices implement something called *Continuous Delivery*. Continuous Delivery enables rapid releases.

Continuous Delivery is what I think is the actual target, and DevOps is just a way (and a prerequisite) of getting there.

Now, let’s say you’re making mobile phones. In this case, you wouldn’t probably be pushing out releases a dozen times a day, and also “ops” means something else to you. But there are ways in which you can utilise Continuous Delivery to streamline your development, too.

Contrast with the previous slide:

- No specific handover for a different production team, or at least production is run in tight integration with the development
- There is just one codebase that preferably has no branches, all releases are made out of it
- The original developers do all the maintenance (because “maintenance” is just fixing bugs to the current codebase)
- At least with services, the production service is typically the latest snapshot of the codebase



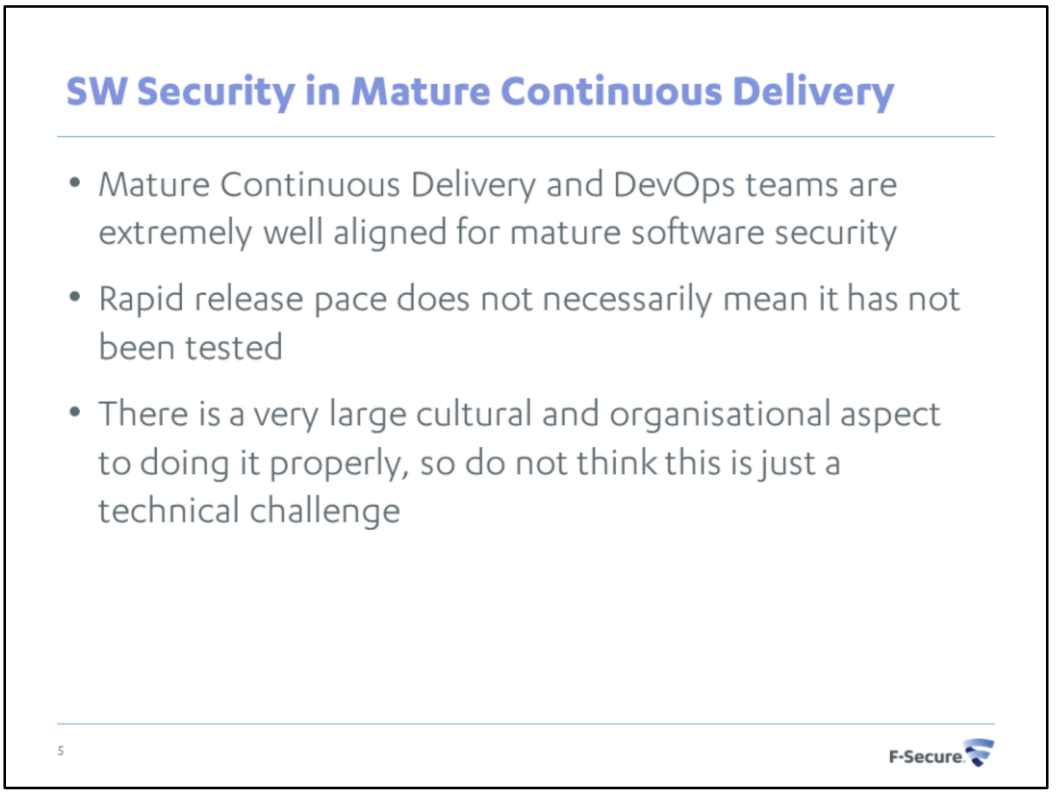
Like with Agile and Cowboy Agile, there's a notion of *mature* Continuous Delivery. To be frank, I don't think very many companies are on a very mature level at this moment, but it doesn't prevent consultants from waving hands.

If you like consultants, you can read about Continuous Delivery Maturity Models. The following slides are based on a synthesis of both of these.

http://www.infoq.com/articles/Continuous-Delivery-Maturity-Model
http://www.infoq.com/presentations/continuous-delivery-model

I guess you could build a BSIMM-like thing out of those if you measured them.

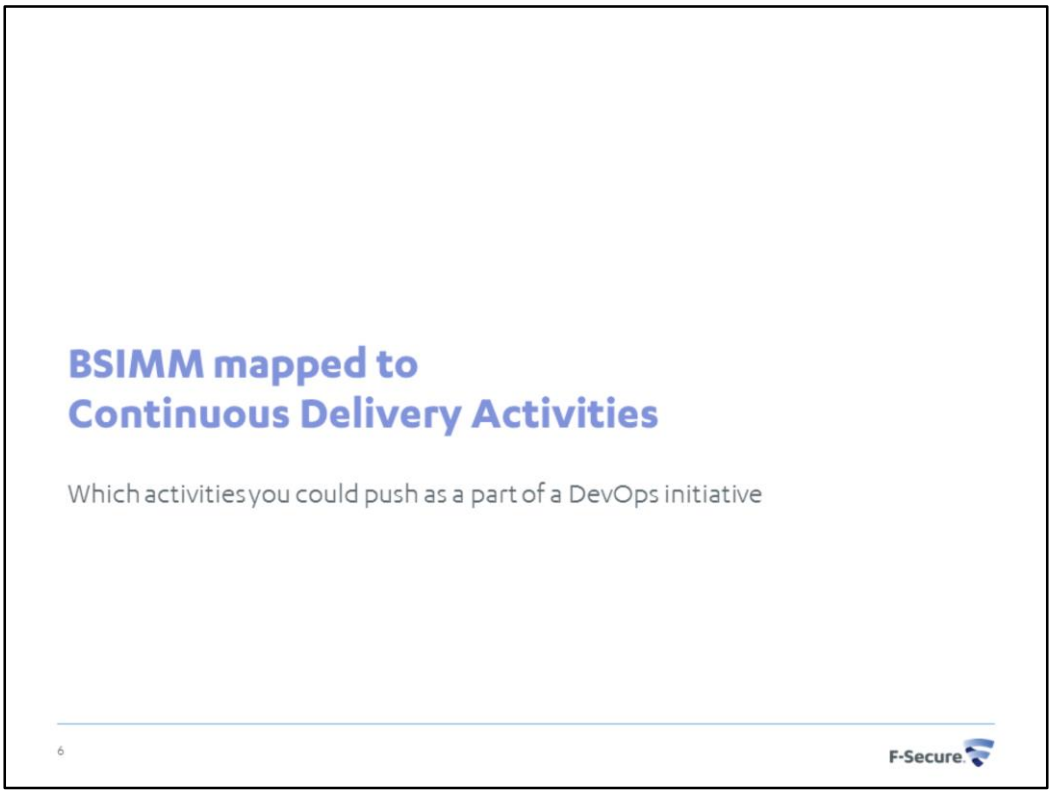
(The guys behind those models aren't very software security centric so software security activities turn up in odd places. Never mind about that. You know where they ought to be.)



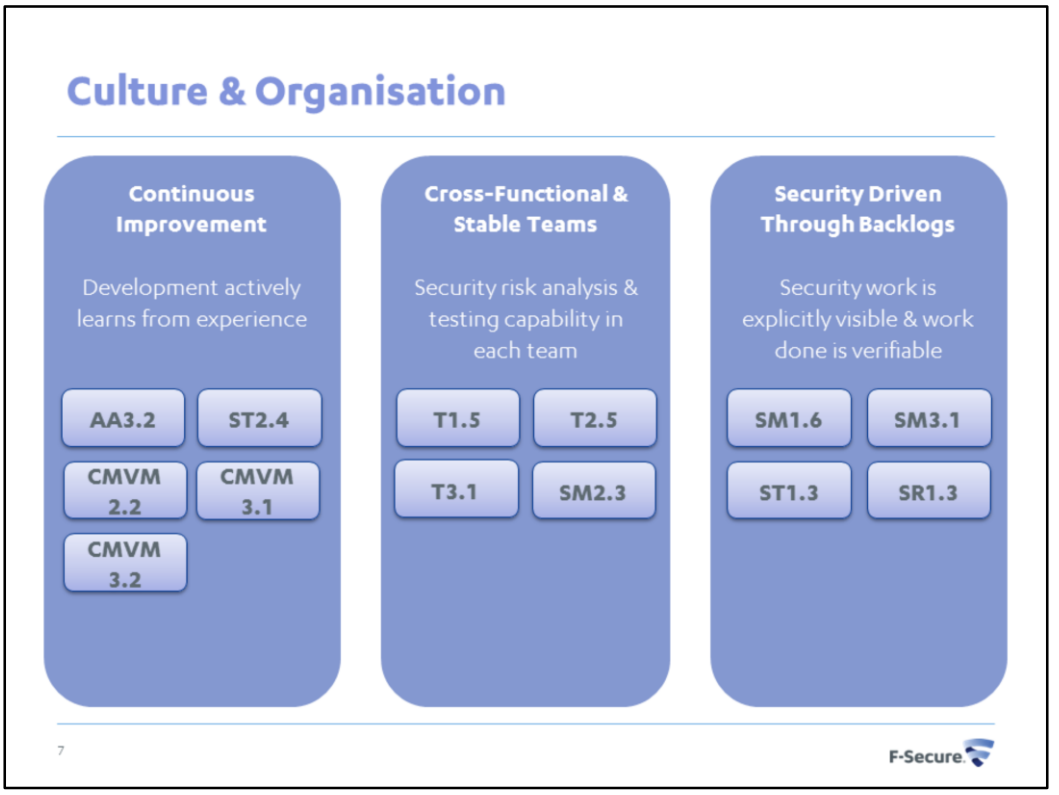
The good thing is that if your company is going to take on DevOps or Continuous Delivery (whichever term will catch on at your company), I think there's much to be gained **if** you work within the initiative.

I really, really think that Trojanising your methods/processes team pays off for software security. Or perhaps you want them to infect your SSG. Whichever way, in change initiative, ensure SW security is done inline with it. Don't wait for the change initiative to be ready.

Many BSIMM activities map almost directly to many of the Continuous Delivery activities, or at least can support them very well.



I took the Continuous Delivery Maturity Models mentioned earlier and cross-referenced them with BSIMM... And we will have a couple of real-life examples.



Continuous improvement:

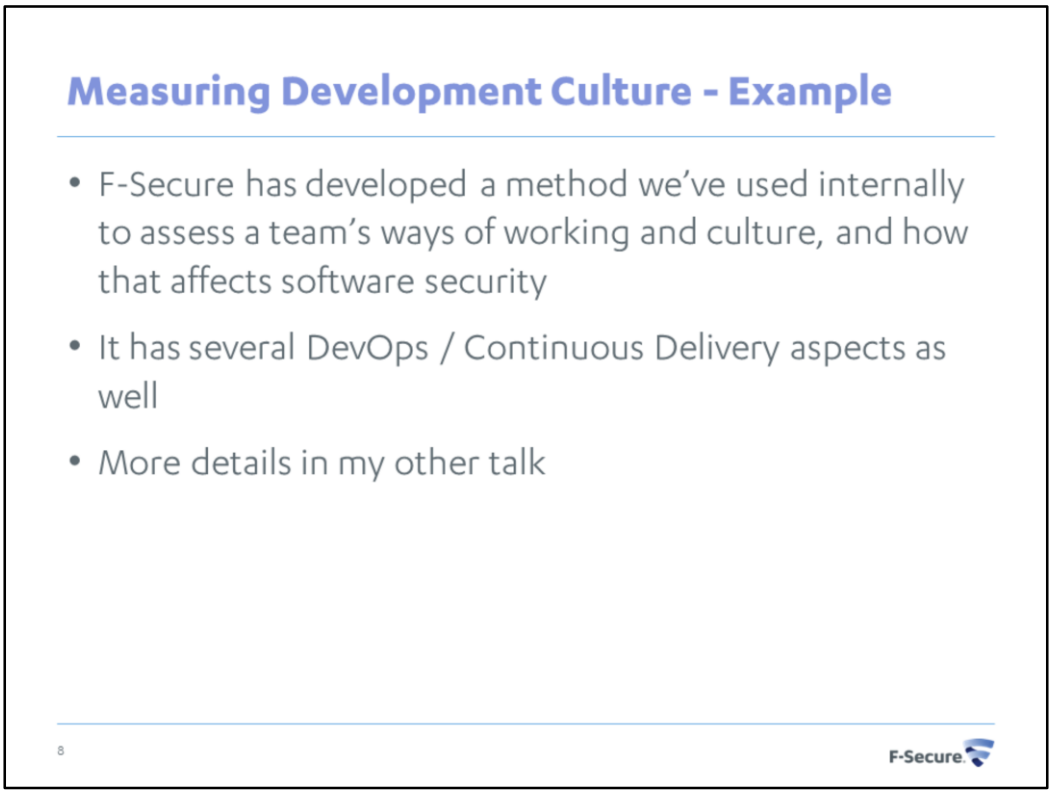
- Information about security bugs, incidents, and risks is fed back on multiple levels
    - Build new tests
    - Organise feedback trainings / root cause analyses
    - Alter architectural decisions
    - Alter our perception of attack model

Cross-functional & stable teams:

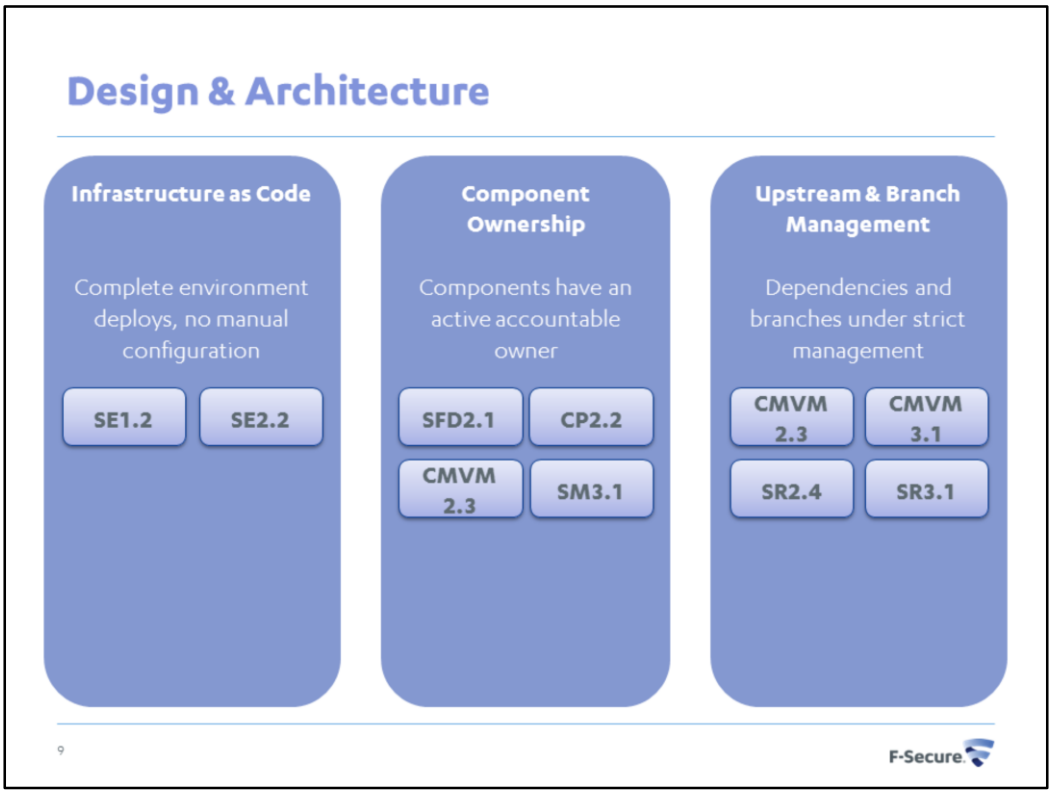
- Teams can autonomously perform security activities including architectural security risk assessment (threat modelling) and security testing
- Teams will carry ‘silent knowledge’ of their attack model

Security driven through backlogs:

- Security work is made explicit instead of being just assumed to be done
- The effect of security work on velocity becomes apparent and rose-tinted glasses go away
- Rejection of security work becomes an explicit business decision where someone is accountable for that decision



See the talk on software security & agile development, BSIMM Conference 2013.



Infrastructure as code:

- Deployment environment and configuration is brought to code level
    - Reduces human error
    - Allows development of security tools (like “code review for infrastructure”)

Component ownership:

- Components have an accountable owner, who looks after them throughout the lifecycle
- Software security considerations get explicitly into design decisions

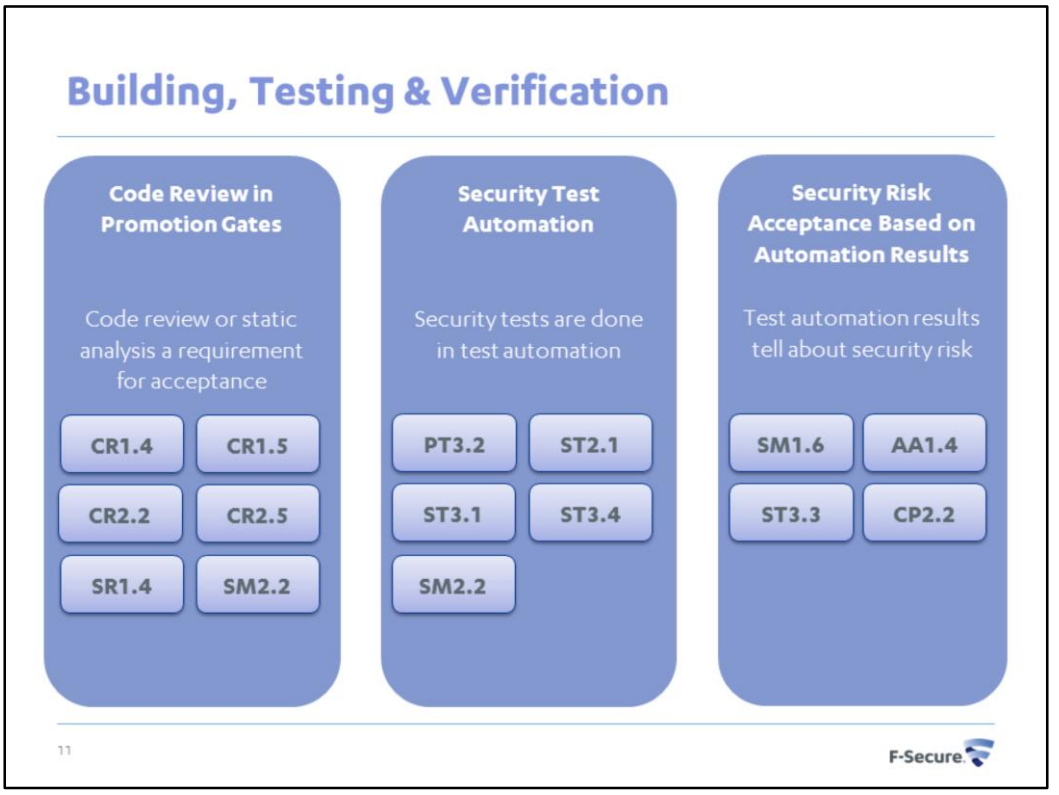
Upstream & branch management:

- Know what software we are running and where, for reactive fixing
- Know where the software comes from (including Open Source)
- Have less variability which speeds up vulnerability management and fixes

"Infrastructure as Code" - Example

- Assume various different back-end nodes
- Not every node needs to connect to every other node
- Knowledge of who needs to connect where is defined by developer
- This connection graph is calculated at deploy time
- Routing tables & firewalls configured according to the graph automatically

F-Secure



Code review in promotion gates:

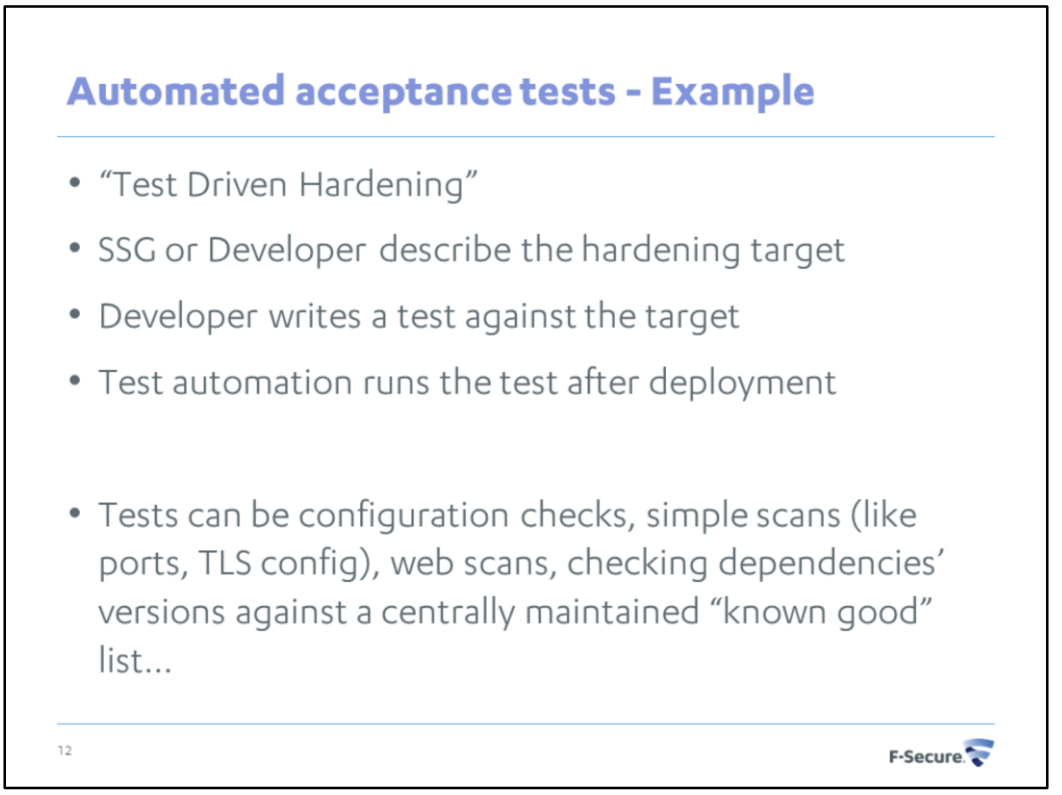
- Once code is promoted towards production, it is forced to go through code review
- Target for as automated solution as possible

Security test automation:

- Target is to do manual and exploratory security testing only for those areas that are very risky or very difficult to automate
- Increase the level of security testing done in-house (vs. outsourced to consultants)

Security risk acceptance based on automation results:

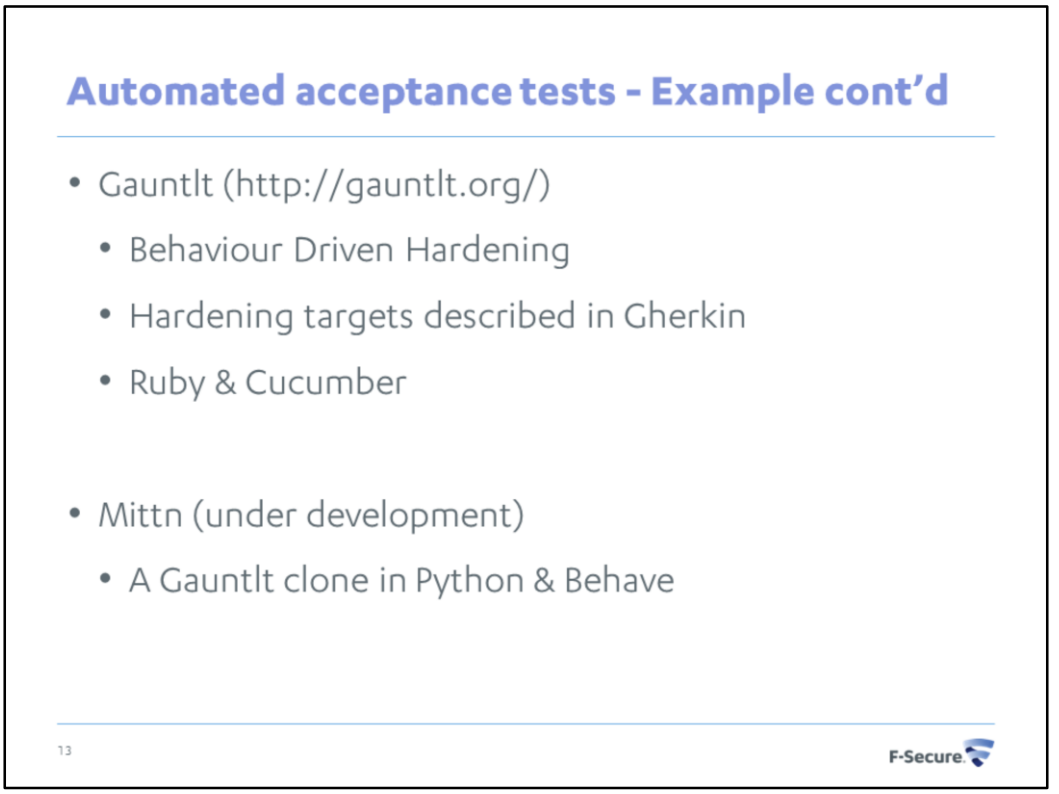
- Target is to get meaningful results from the build & test process that can be used to actually make decisions on whether the software is good enough from security perspective



One way to use test automation and Test Driven Development is for the SSG or developers to create "hardening targets" which are then run in test automation.

Mainly scans, checks, and this sort of stuff.

If your company doesn't yet have a good test automation platform, then a Continuous Delivery initiative may result in building it, and getting a large number of this sort of "safety net" tests is then just a few lines of code, and a return code to Jenkins CI!



One example of an approach is Gauntlt, which uses a language called Gherkin to describe the hardening targets.

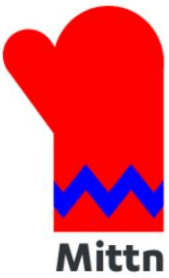
I have my own side project that is a clone of Gauntlt called Mittn, with a difference that it aims to be simpler and Python-based (Gauntlt being Ruby).

Gherkin is a language that aims to make writing user stories and acceptance criteria (~ use cases and test cases) easy. The Behave (Python-based BDD framework) explanation is a good one: http://pythonhosted.org/behave/philosophy.html

The idea here is that the developers (or security specialists) can write a target for system hardening or security checks in plain language; and that is interpreted into actual test cases; the tests are run, and the build fails if they pass.

You could do that with a simple script without the BDD stuff, too. But this way (arguably) you have up-to-date documentation and reusable tests.

Automated acceptance tests - Example cont'd




```
Scenario: Test that weak cipher suites are
      disabled

      Given sslyze is correctly installed

      When a "TLSv1" connection is made to
      "www.gtn" port "443"

      Then the following cipher suites are
      disabled

        | cipher suite |
        | EXP-         |
        | ADH          |
        | AECDH        |
        | NULL         |
```

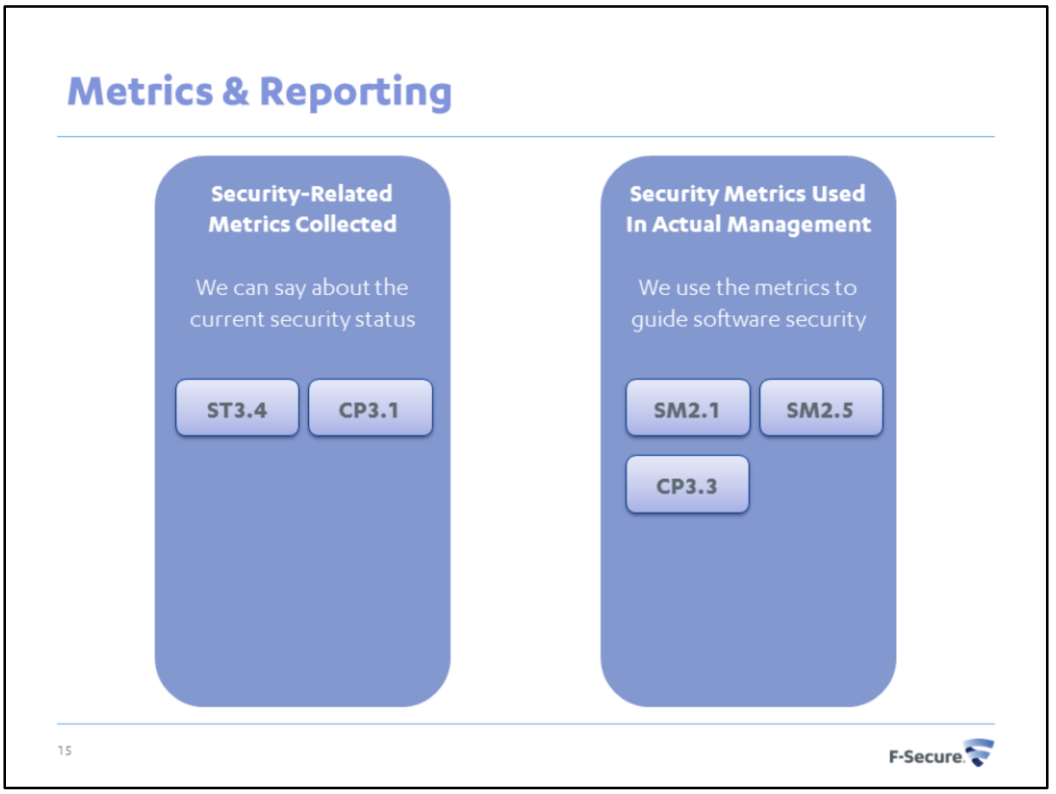

F-Secure

Here is an example of a Mittn hardening target written in Gherkin. This is one test scenario for testing TLS configuration at a server. It runs sslyze (a TLS configuration tester tool) according to the Gherkin instructions.

The Mittn slogan is “for that warm and fluffy feeling”.

You can see that the hardening test is fairly easy to read, and you can create new test cases just by copy-pasting the “Given”, “When” and “Then” lines in various configurations.

My plan is to integrate a fuzzer and injectors for REST API testing next.



Security-related metrics collected:

- The Continuous Delivery process provides information about the security level and activities performed

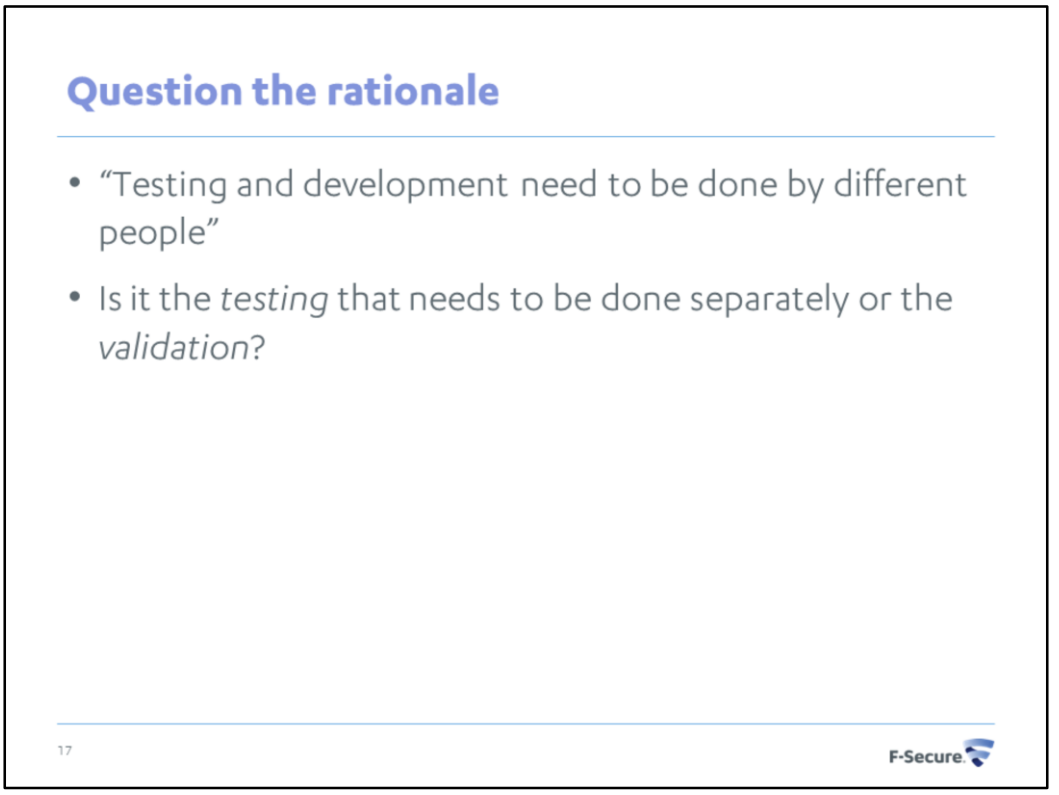
Security metrics used in actual management:

- We can decide to do / cease to do certain activities based on the metrics
- We can supply evidence (e.g., to a customer or regulatory body) on activities we've done

At the BSIMM Conference 2012, an audience question was posed after the Agile SW security workshop about segregated acceptance testing.

I could not answer immediately. I'll now try again, because Continuous Delivery is just about impossible if you need a segregated validation / acceptance testing requirement.

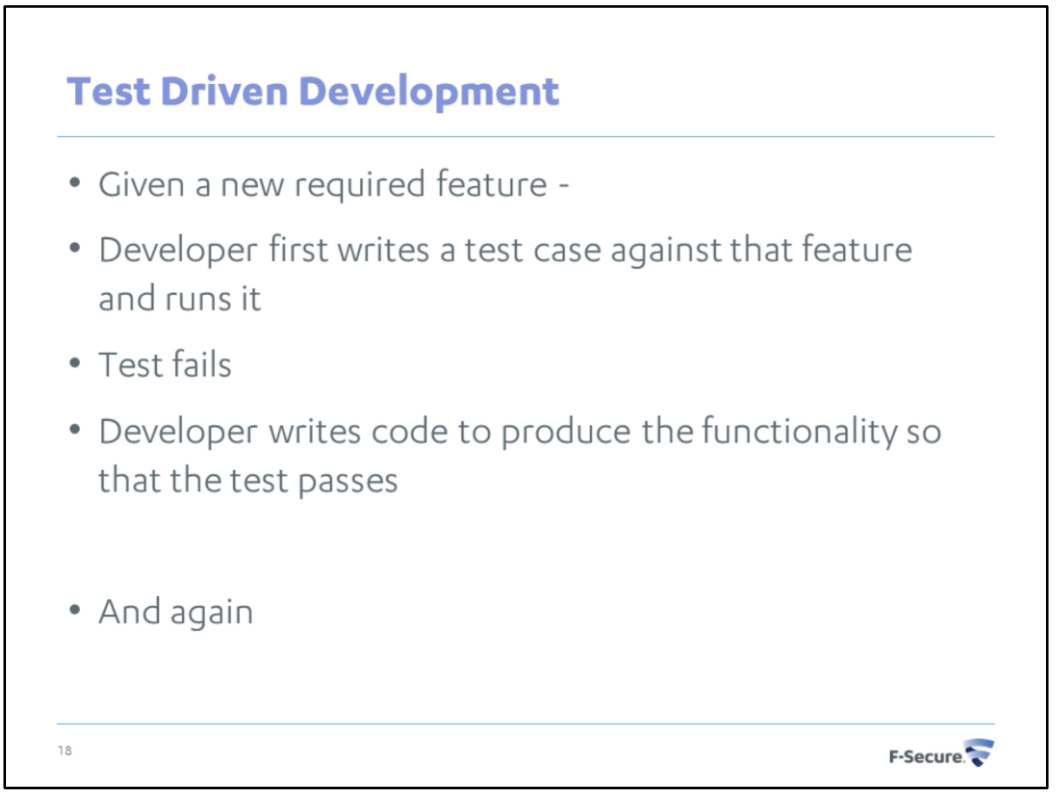
(Segregated = Different people do the implementation from those who test)



At least one financial institution that I talked to in detail explained that they really don't need *testing* to be done by someone else, but they need *validation* for the code.

This is markedly different because validation is (in my vocabulary) ensuring this do what they are supposed to do, whereas testing is the actual work of running test cases.

If this is true, can we do validation without testing? Kind of.



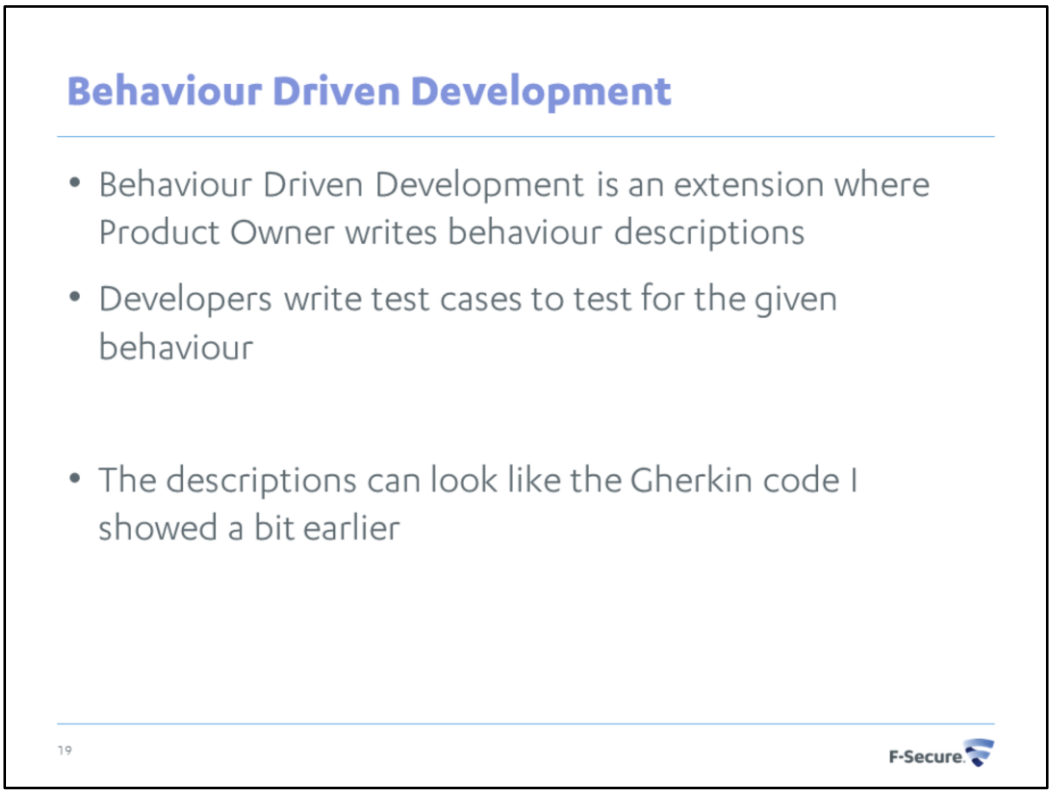
The solution would come in the form of two buzzwords:

BDD, or Behaviour Driven Development
TDD, or Test Driven Development

TDD is a way to develop code where the *test* for the feature is written first. It should be an automated test.

The tests (all the tests) are then run, and obviously the newly added test fails.

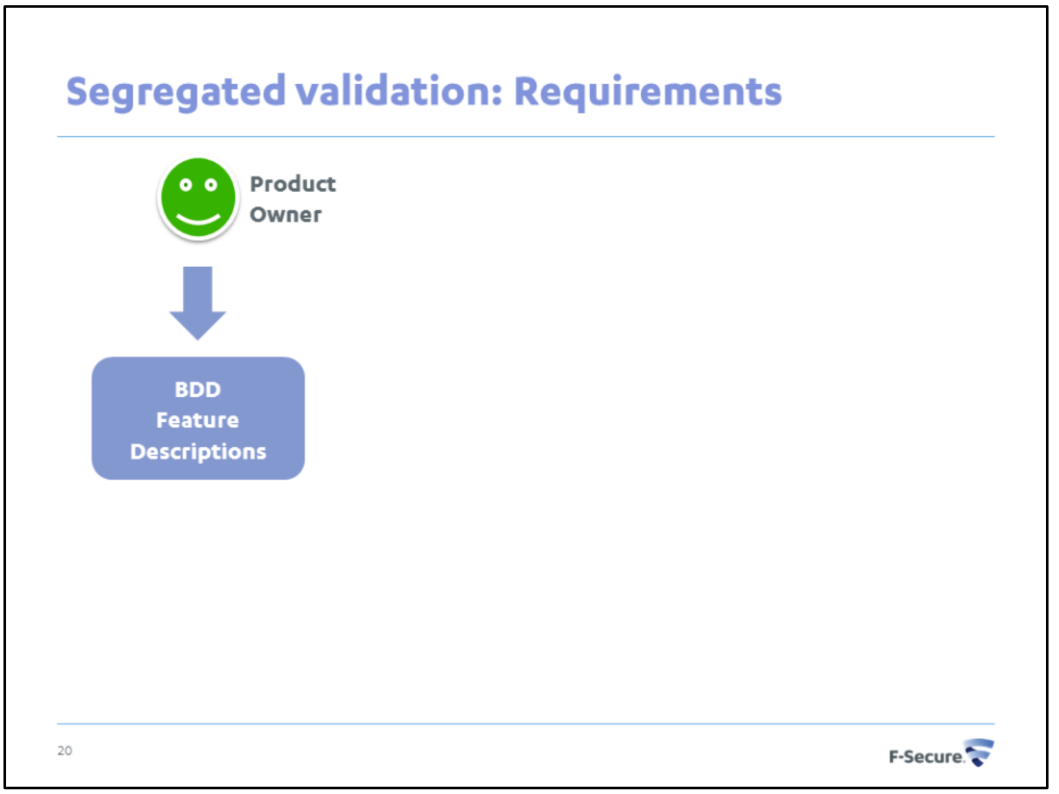
New feature is then developed so that the test passes. The code is then refactored, if necessary, and a new cycle begins.



In BDD, Behaviour Driven Development, the business owner (e.g., the Product Owner) writes out the business level descriptions of what should happen, scenarios, and their acceptance criteria (what is the acceptable outcome).

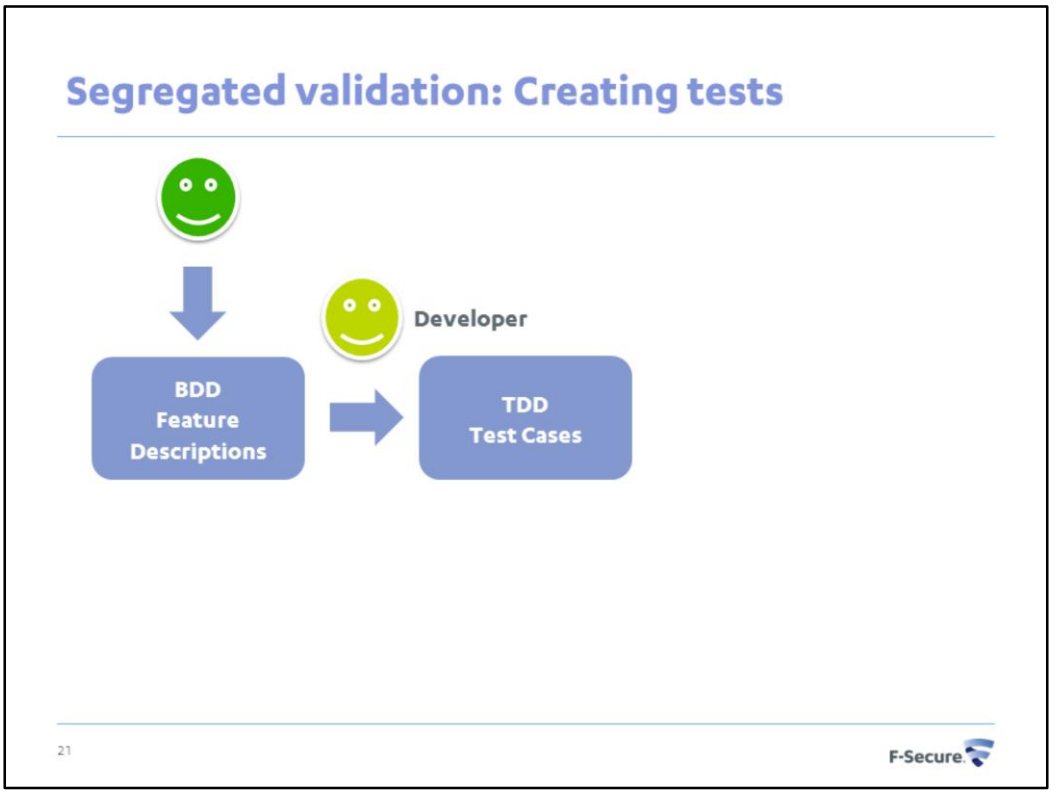
These are then transformed into TDD-type test cases that are actual code, and can be run automatically. They are run, and they fail. Then the actual implementation is written so that the tests don't fail any more.

This creates a "chain of custody" for all the code from the Product Owner's requirements to the code. All code has a reason to be there, it has tests, and some sort of contextual documentation.

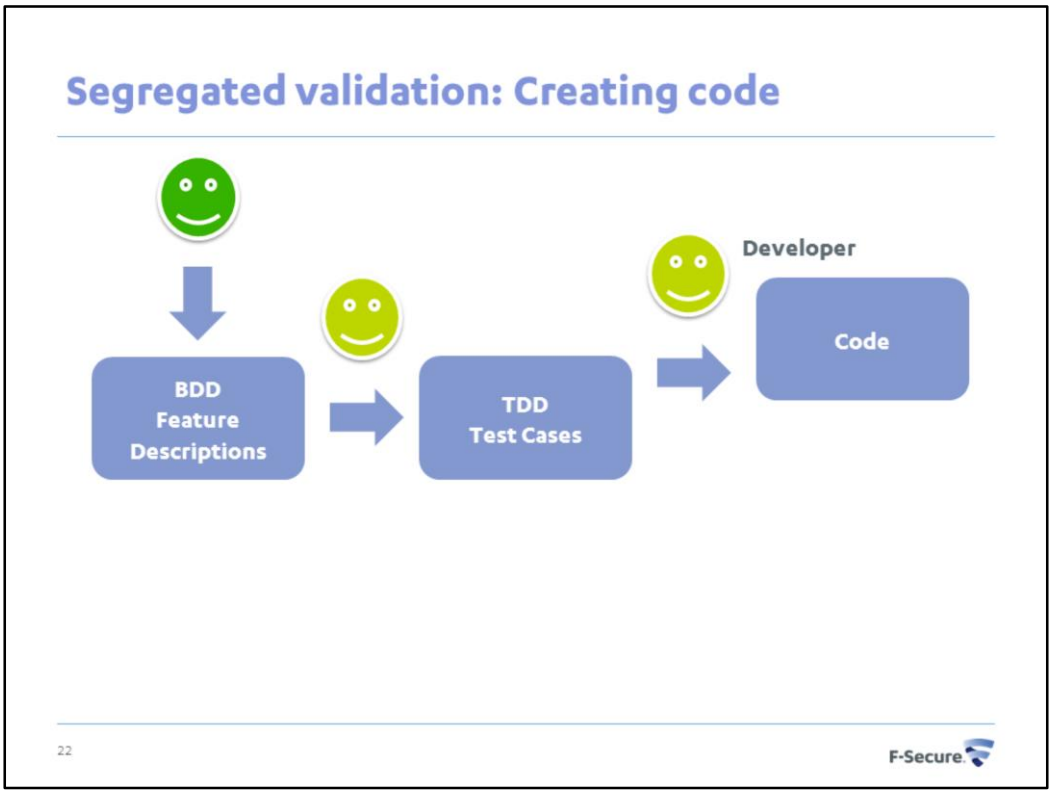


So the idea would be to run everything under a strict BDD/TDD framework. Because we do Continuous Delivery, we should have a very good automation-supporting workflow anyway, and we might be doing TDD anyway.

It all would begin with the Product Owner providing the features, or even just security and compliance requirements, in BDD scenarios.

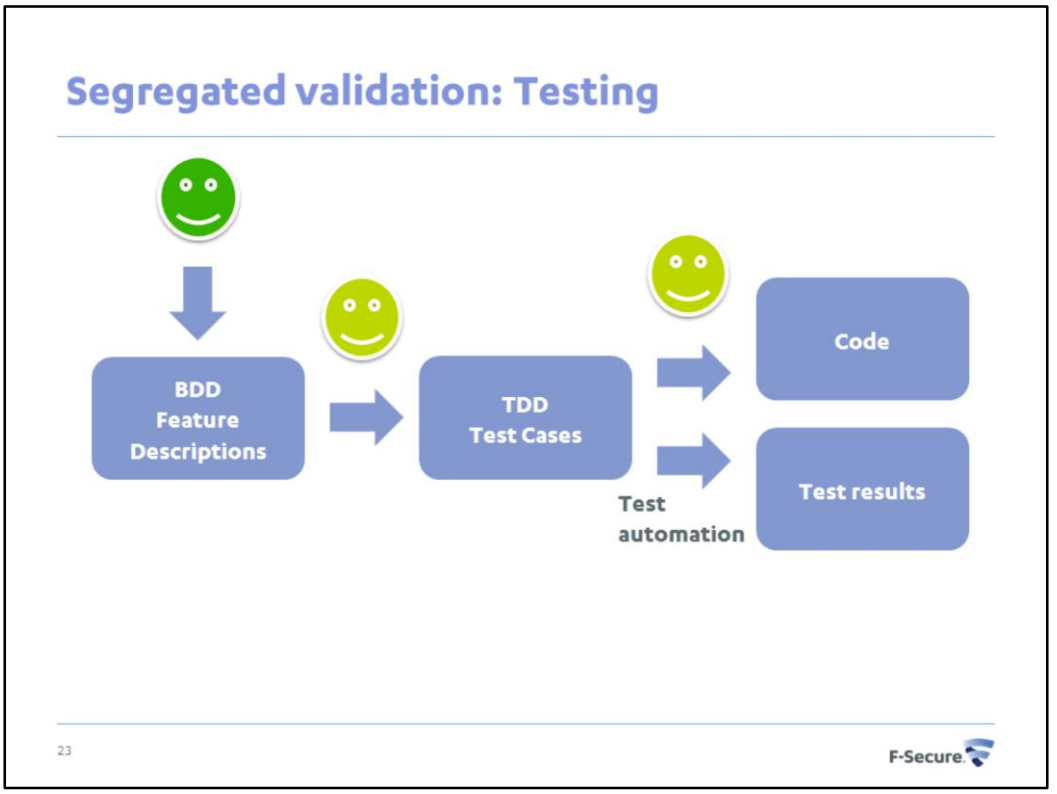


The Developer would take the requirements and write tests (which, as we now know, are failing, because there is no implementation).

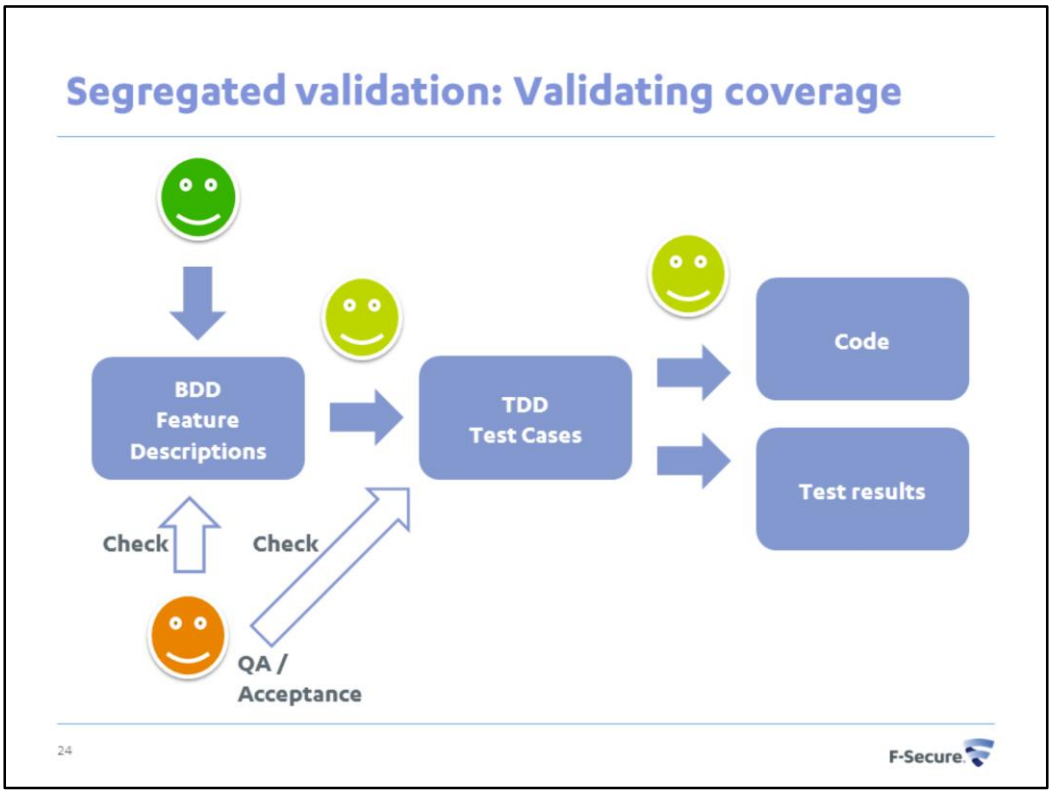


Code would also be written by the developer.

Notice how in this phase the development can be done in an integrated (i.e., non-segregated) team where testing and development are done by the same people.

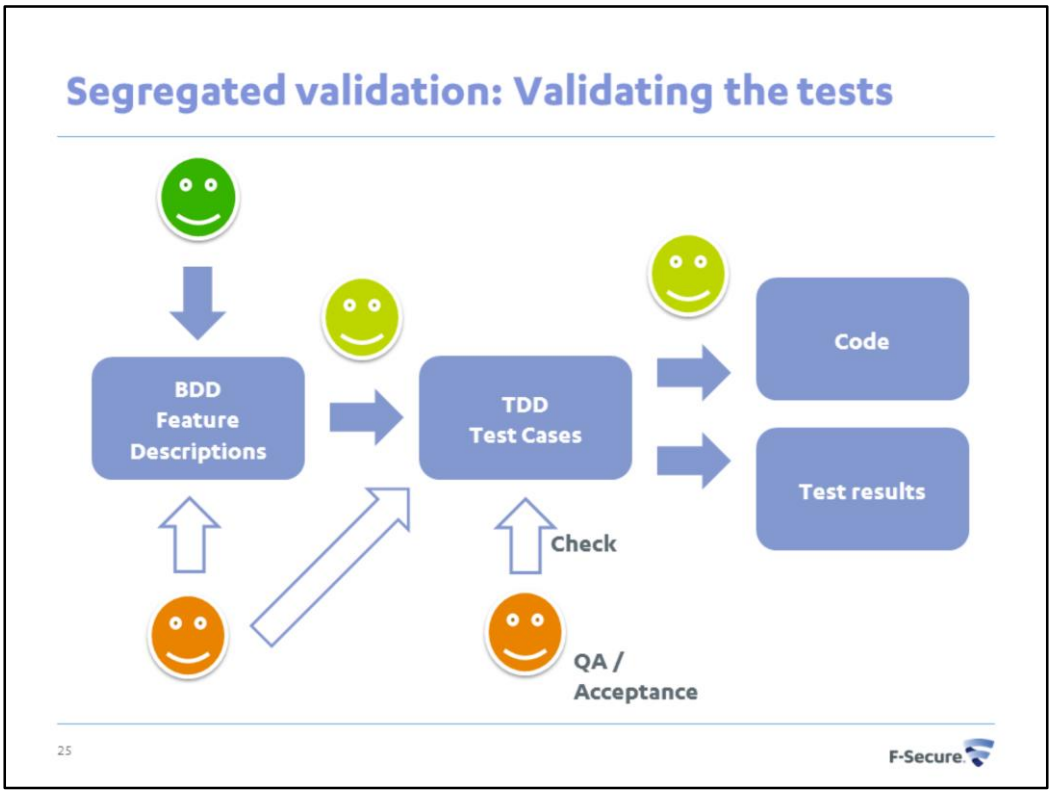


Tests are actually run by test automation, so that is an impartial way to do it. The developers aren't actually "doing" testing, they just tell the computer how to do the testing.

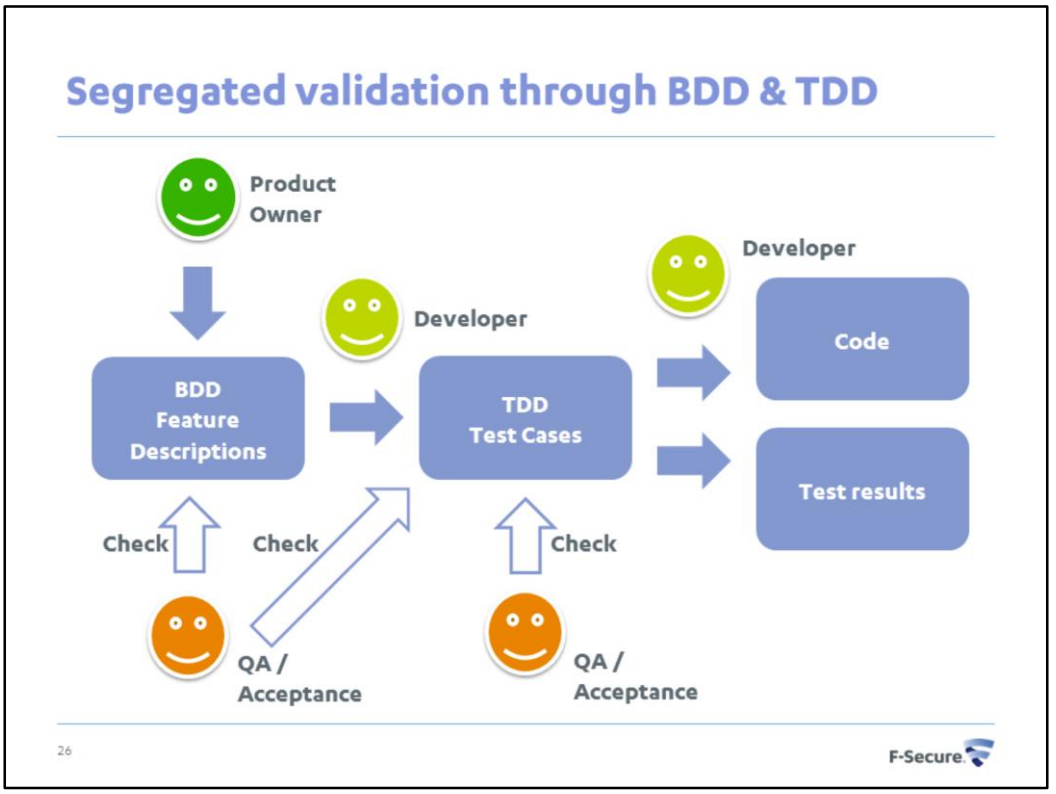


Now, enter the segregated Quality Assurance or Acceptance Testing person. These folks, separate from the dev team, would have two distinct tasks:

1) Ensure that the TDD tests written by the developer actually cover all the test requirements for the BDD feature description written by the Product Owner, and...



2) Ensure that the test cases are correctly written.



Because BDD descriptions and TDD cases are long-lived, once validated, they can be run indefinitely without re-validation. Computers, being deterministic, run the tests over and over again for every code commit.

Test cases would therefore form a “proxy” between development and validation, and developers could still write code and perform Continuous Delivery - the validation folks just need to review new scenarios and test cases for any new functionality.

Comments welcome

antti.vaha-sipila@f-secure.com (work)

avs@iki.fi (personal)

Twitter: @anttivs

Protecting the irreplaceable