Software security in agile product management

Antti Vähä-Sipilä, avs@iki.fi
2011

Preface

Intended audience

I have written this e-pamphlet for two audiences:

- Security managers and specialists who one day get a visit from a young guy with a title of a 'coach', wearing horn-rimmed glasses, telling them that we're agile now;

- Product owners and developers who would like to bring in security as one of the "qualities" of their product, but are afraid of the impact that security practices would cause to their agile projects.

The first part of this document tries to convey the ideas behind agile and lean development that I find most important to understand when bringing security into agile. The second part delves into the actual agile security practices. The latter part builds heavily on the concepts from the first part, and in order to ensure that our vocabularies are synchronised, I would recommend reading the sections in this order.

Some history

We started to look into software security engineering in agile product development as the company I was working for at the time was adopting agile methods, specifically Scrum and agile requirements management – product owners, epics, user stories, and so on.

Our existing security development lifecycle had some challenges that I wanted to rectify. Granted, our model dated from sometime in 2002 when this topic wasn't really yet on every conference agenda. Anyway, there was now an opportunity to polish the existing model, and at the same time, map the practices into agile software development.

After about two years of discussions and small trials, I believe there is now a plausible story of how security (and many other non-functional qualities) could be brought into an agile development environment without breaking the leanness properties. In the past events where I have been speaking, a member of the audience has proposed that writing a book would be a good idea.

So, I thought – why not? But instead of a book (which, I think, should have at least 200 pages and I would be sweating over typography for longer than the actual text) I decided to publish an e-pamphlet, which I hear is nowadays fashionable. This will hopefully free me from having to use bullet points in the presentations and just project inspiring microstock images behind me.

The ideas in this e-pamphlet have been collected and built upon from discussions with several people. I would like to credit, in no specific order: Martin von Weissenberg for the initial start; Vasco Duarte and Camillo Särs for the first public shot at it with Agile

Finland; the participants of various events where I've talked about it; Heikki Mäki from my previous team; Lauri Paatero for some very valuable commentary; and finally, an agile guru from Nokia's Mobile Phones organisation who gave excellent feedback but whose identity unfortunately still remains a mystery to me.

Caveat emptor

Some teams could be claiming they use agile methods, but when you dig deeper, you see that they are in fact using *cowboy agile*[1] – essentially a freewheeling implementation run by autonomous rock stars.

This is not necessarily a bad thing. As long as the rock stars are security aficionados, everything might go smoothly. However, I do not believe that you could manage to run a *large* company, recruiting from a *realistic* talent pool, and only have rock stars.

Some teams may be trying to do agile development within the confines of a larger organisation that is not agile at all. For example, they might need to deliver a subcomponent to a larger project, whose schedule and features have been fixed. In this case, they could have specific feature completeness gates or milestone criteria that they need to meet, so they have to implement some sort of abstraction layer which enables them to pretend to be semi-agile by themselves while looking like a bunch of respectable, schedule-oriented people to their internal clients.

The first half of this e-pamphlet has been written partly with these things in mind. It should arm the reader with some tools to determine whether the flavour of "agile" you are seeing is in fact agile and lean, or just cowboy.

I have also written that part for people who have not had experience with agile development, and may be wondering what it is all about, and who would benefit from it. There are probably hundreds of other treatises on what agile means, but this one is mine.

When I refer to agile methodologies and specifically Scrum, have taken to use the term *Ideologically Pure Scrum*, for which I mean the Scrum as described in Ken Schwaber's and Mike Beedle's book, *Agile Software Development with Scrum* (2001). I believe that Ideologically Pure Scrum does not exist anywhere where Scrum is being used. All applications of it are probably tainted by something. I've still chosen to use the Ideologically Pure Scrum as the measurement standard, because if you know how your specific Scrum is tainted, you can also take the advice in this pamphlet and taint that in a similar way so it fits the peculiarities of your flavour of agile.

Agile, illustrated

What 'agile' is not

I have heard people refer to agile methods as if those were a hippie anarchist plot.

I can assure you that Scrum, one of the agile project management frameworks, is pretty far from smoking pot. I don't know of any other project management framework that dictates when you have to wake up in the morning, and whether you can speak in a meeting. Scrum does that, and more.

Also, agile is not necessarily that different from a 'traditional' method when the latter is well managed. While recently emptying the shelves of my cubicle, I found a laminated cheat sheet that I got on a project management course in the 1990s. It emphasised constant measurement and re-planning. The weakness was that the re-planning was not built into the process and there was nothing to actually trigger it; you had to just to remember to do it. Also, there wasn't a specific feedback loop that would reflect on things already done, and trying to enhance future work based on that. You could partially view agile methods as a way to codify good project management principles so that they are automatically followed.

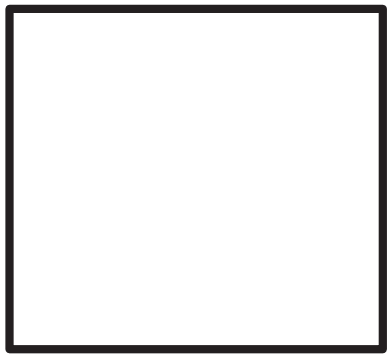
One of the most concrete fears about agile I have heard is that because the Agile Manifesto[6] emphasises code before documentation, this will cause quality control, and security with it, to go out of the window. Now, again, this should not be the case. First, documentation does not equal – or even somehow automagically lead to – quality, and second, if documentation really is necessary, agile methods by no means preclude producing it if it really is a true business requirement. That work just has to be specifically ordered.

There is a possibility that some development teams adopt the previously mentioned cowboy agile methods, or that teams have weak product owners, or that traditional project management is thrown out and replaced with anarchy. Now, *those* are valid fears.

Life in a changing business environment

I wrote this section for those security managers who haven't read any agile literature, but are faced with an emerging agile development process. I hope this gives a viewpoint which is useful to understand exactly why R&D would pursue agility.

Figure 1: Our business environment.



Let us think about the business environment in very abstract terms. Let's say that our world is an empty, two-dimensional canvas (Figure 1). This canvas represents all the possible combinations of functionality that our product could do; in fact, the canvas is infinite. I will visualise this with a simple square, with unnamed axis.

Let's then assume that we have a product that has some functionality (that is, features). It could be very close to zero functionality, and it really doesn't need to be marketable in any way – perhaps it is just a prototype. Alternatively, we could already have a working product that does a lot of things, and we're planning on enhancing that product. In any case, we'll denote our product's current feature set with an 'X' (see Figure 2).

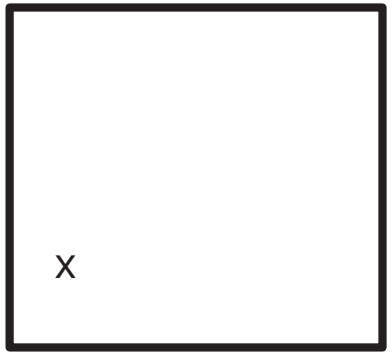


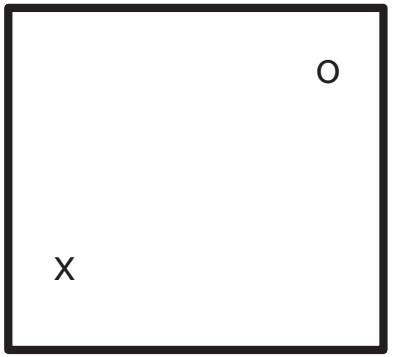
Figure 2: Our current product.

We also have a business target, which encompasses some set of functionality, or features, that we think could be actually marketable, and where we are trying to arrive.

If we are just starting out, we would be targeting for something that would make a good first version to ship. If we already have an existing codebase, the target could be the next release, with additional features, for example to corner a new piece of market.

Figure 3: Where we would like to end up with our product.

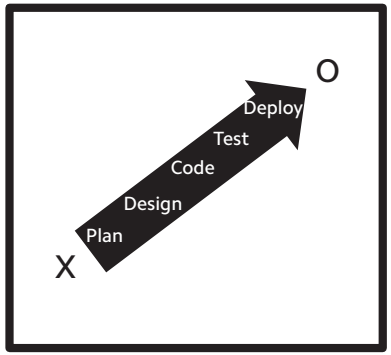
We'll mark this target state with an 'O' (see Figure 3).



We have many ways to get from 'X' (where we are now) to 'O' (where we want to be). Intuitively, it could be tempting to make a grand master plan, where you first define the additional functionality you need, then design it, implement it, test it, and deploy it.

This is what is often called the *waterfall* model, as the outputs from preceding phases flow down the chain to subsequent phases, or *big design up front* (often referred to as BDUF).

Figure 4: A grand master plan.



A worst-case scenario could look like Figure 4.

The idea behind a waterfall model is that it brings as much of the design and planning as early as possible. Changes in design are cheaper early on in the process, whereas if you have already implemented something and need to change the design, making changes after implementation would be very expensive.

This rationale, although well-meaning, is also the Achilles' heel of waterfall. If you are, regardless your best efforts, unable to nail the design correctly, there will be a need to revisit the design later.

Many companies operate in an environment where they do not really know where they will end up. This may be for various reasons.

Some market areas are so dynamic and changing that the premises just won't hold. Say, if you invested in a mobile phone platform *S* you suddenly find out that in a couple of years time, it's been effectively surpassed by platforms *I* and *A*. You couldn't see that coming.

Other companies (and the skunk works departments of established players) may be in a scouting mode all along. The company or department may have been formed to search for a business model that

makes sense. They may aim for what the startup people call the *Minimum Viable Product*[2]. A Minimum Viable Product has just the right set of functionality – but no more – so that it is useful to early adopters, elicits feedback, and perhaps revenue. After they find their Minimum Viable Product, they may use it as a basis to survey new directions while getting a new lease of life from a revenue stream it generates.

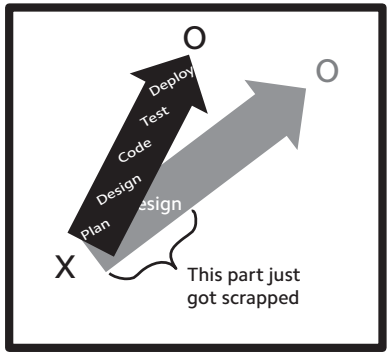


Figure 5: BDUF easily fails when the business target changes.

Waterfall falls short in both cases.

To illustrate this with our visualisation experiment: If you already started the journey from 'X' towards 'O', you may be in the middle of development when the business environment change knocks on the door. If you now need to reinvent what you were making, you may have to throw out everything you've done this far. If you've just finished your design phase, you might have a bunch of requirements lists, wireframe concepts, and in worst case some UML, no lines of code, and definitely nothing to sell (Figure 5).

Incremental development

A changing business target is better served by *incremental* (or *iterative*) development. Instead of having a great master plan that would take you from start to finish, you only plan for a small increment that goes in the right general direction (Figure 6).

Now, when the target keeps changing, we can redirect the next increment towards the updated target, and keep going (Figure 7).

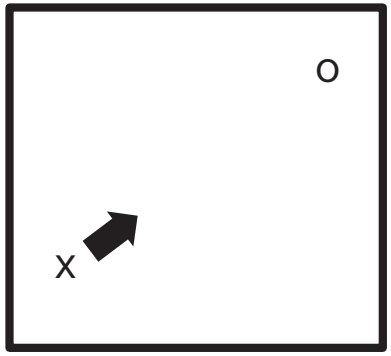
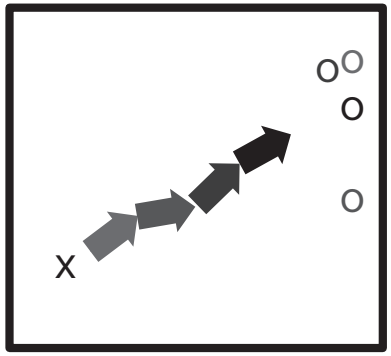


Figure 6: The first increment (encapsulating all phases of development, but just for a little bit of functionality).

Figure 7: Subsequent increments, with changing business targets.



Each of these mini-arrows contain all the software development activities needed for delivering functional and quality-assured code. However, as we see in a moment, even though all the activities are there, in most cases you would not expect to find the waterfall structure in these small increments.

Each of these small increments is intended to deliver some functionality on a quality level that is good enough to ship. This means that you get a bit closer to your target on the canvas. Of course, the set of functionality that is delivered may not be marketable and might not correspond to even a minimal product (much less than a Minimum Viable Product), but they give you a stepping-stone forward. Essentially, when the target changes, it does not force you to scrap everything.

If the business target does not change drastically, most of the work that you do is *velocity made good*, to borrow a sailing term. This means that a significant portion of your work can be used, or adapted with minimal changes, to fit whatever vision for the future you have next.

However, sometimes the business target will change drastically. This may force your development to *pivot*[3]: to take on a new, significantly changed target, which may require you to scrap something you have already implemented. Each delivery of an increment (an arrowhead on the canvas) makes for a good pivot point (Figure 8), as each increment delivers working high-quality code.

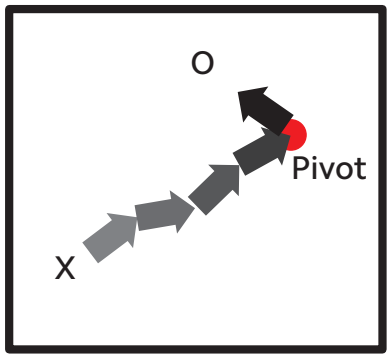


Figure 8: When the business target changes significantly, we have a pivot point.

In the light of the new business target, some increments may have taken you to the wrong direction, but you can still build on all those increments which didn't (Figure 9). You may be throwing away some, or a lot, of perfectly good code, but pivoting could save your company – and with waterfall, most likely send everything you've done this far down the drain.

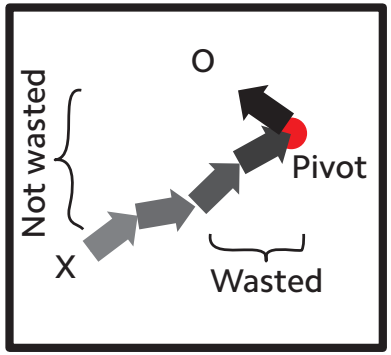


Figure 9: Because each increment delivers something that can be built on, pivoting does not force you to scrap everything.

But as they say on the TV, "Wait! This is not all!" Rapid succession of small iterations also give your R&D a great way to become better in what they do, because they can reflect on what they've done, and perform better the next time. When you do a bit of design, a bit of coding, and a bit of testing, and then reflect on those, and do it all over again, organisational learning happens much faster as if the cycle would be one year long.

In addition, small iterations also enable the software development organisation to demonstrate their successes (or indeed, failures) often. Demoing working software is a great way to ensure it works in practice, and having a prototype in production also helps to identify issues with deployment and operations stage early.

Key agile concepts for security

Technical debt

Each increment needs to be complete in order for subsequent increments being able to build on it. Completeness means that the functionality must be implemented, tested, and be on a high enough quality level to ship. The increments might not have all the desired functionality, but what little does exist, must have good quality (and fulfil other *non-functional* minimum criteria).

Not addressing quality issues right away (within the same increment) is known as *technical debt*[4]. Technical debt borrows time from the future: some corners can be cut now, but someone should spend time later making sure those things are mopped up later. There are various forms of technical debt. Security debt is very close to quality debt, which can be caused (for example) by lack of testing due to a business owner ordering a premature delivery.

Typical phrases one encounters when technical debt is taken sound like:

> *"We'll launch without full security testing, and do that a bit later."*

> *"We'll dedicate a separate increment later for bug fixing. Let's now worry about feature implementation instead."*

Often, business leaders higher up the corporate ladder may not understand how much technical debt their software asset has. It is very hard to say anything about the quality level of software just by looking at its user interface and a set of status reports. In the worst-case scenario, debt has been swept under the rug of misaligned incentives and fancy chrome. Serious problems may then be only found when the product is taken outside its comfort zone or the business target is changed significantly based on wrong assumptions about the current state of the product.

Technical debt does accrue interest, too: it may mean an increased maintenance burden, and in the case of security debt, increased need to invest in reactive controls. In some cases, the customer ends up paying the debt. Having to run a virus scanner or to invest in a web application firewall are partly due to technical debt that is passed on to the customer by the operating system or application vendors, respectively. In the compliance lingo, technical debt often translates in the operations phase into a cost of a *compensating control*[5].

Sustainable security-aware software engineering should aim to minimise security-related technical debt and therefore the need for compensating controls. Most of the practices laid out in this e-pamphlet are geared towards minimising security debt in the code base.

Agile work planning

One of the agile project management methodologies, *Scrum*[7], is very well suited for a project that needs to reinvent its business targets

often. I am using Scrum as the main reference when I am discussing security activities in an agile project. This is mostly because most of the teams I've worked with in this area have used Scrum. Scrum is by no means the only game in town. Many teams also use *kanban*[8], especially those who have maintenance or request-driven tasks.

Perhaps the quickest way to understand Scrum is through a series of steps:

- A *product owner* is constantly scouting the business environment and trying to guess where the target, the 'O' in our earlier canvas thought experiment, should be.

- The product owner steers development by constantly maintaining a list of things to do (a *product backlog*, containing *backlog items*). The idea is that the list is always kept prioritised, with the most important tasks on the top. The bottom-feeders of the backlog may never get done, but that's ok; that's an explicit business decision.

- The increments (the small arrows in our discussion in "Incremental development" on page 6) are called *sprints*. These are typically 2 to 4 weeks in length. At the start of each sprint, the Scrum team takes the most important product backlog items and turns them into a series of *tasks* on the *sprint backlog*. In our canvas-and-arrows example, the contents of the sprint backlog define the direction and length of an arrow.

- After each sprint, the team has implemented (and tested, and deployed) one sprint backlog full of tasks. If, in the *sprint review*, the team agrees that work is complete and free of technical debt (in the Scrum lingo, *done*), the respective backlog items can be crossed over from the product backlog and the next sprint can begin. In the meantime, the product owner may have changed the product backlog items or their priorities again – in essence, changing the location of the business target.

- The Scrum team's well-being is ensured by the *Scrum master*. The Scrum master's most important task is to get rid of any roadblocks (*impediments*) that the team might have – you will note that from software security point of view, Scrum master is *not* a key person. I am just mentioning the Scrum master so that it is clear that this role is distinct from the product owner.

From now on, I will be using the Scrum terminology. Unless you are familiar with Scrum, this could perhaps be a good time to read a lightweight Scrum book or a tutorial on the web.

Agile product management

In the previous section, I shortly explained agile pro*ject* management (or the closest approximation of it; many agile practitioners would eschew the whole term) and used Scrum as an example of a task scheduling system. Pure Scrum doesn't, however, go into much detail of how exactly the product owner comes up with the product backlog. So, how can you set up agile pro*duct* management?

Dean Leffingwell has extensively documented the method I am most familiar with[12]. It works by identifying large, business case level targets, and then massaging them into smaller and smaller pieces until they become bite-size – and can be used to populate the product backlog. I often draw this as a funnel, where the client can throw large requirements, the product owner and the team, working

together, split these into smaller and smaller chunks, and small tasks come out from the other end, ready to be implemented.

Typically, these business level targets are expressed as *user stories* in order to align all software development with perceived value to the user. There are many different levels of detail, and the literature uses a variety of terms for these in various order, typically *epics* for the largest, almost business case level ideas, and *features* for smaller things that can be then split into tasks. The process of coming up with backlog items is sometimes called *decomposition* (Figure 10).

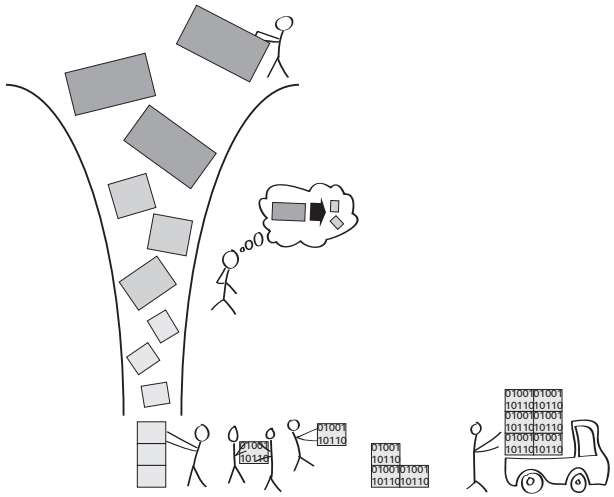


Figure 10: The client's requirements (large user stories at the top of the funnel) are split into smaller bits, and then drop on the prioritised product backlog. The highest-priority items are then implemented and delivered as working entities.

Agile product management (feeding the product backlog) and Scrum (work scheduling) can exist irrespective of each other. Small shops might not have a very well defined product management process, whereas a large product creation organisation with dozens of Scrum teams probably should.

The output of the Scrum teams is then integrated together. If the organisation has a large number of teams, the outputs of the teams are sometimes synchronised in time to create a *release train*[13]. At some periodic intervals, one of those releases may be deemed an external release that can be delivered to the customer.

Unlike the planning / design / implementation phases in a BDUF project, agile requirements decomposition, sprints, and integration and releasing work concurrently and continuously. Instead of being phases, these activities more like individual parts of a large machine where the front end feeds the implementation pipeline, and the back end outputs ready-made work products.

The product owner has great responsibilities to ensure that the development team actually spend their time appropriately. Indeed, it will become even clearer when we get to the actual security activities. Recruiting the correct type of product owner may be critical to your success, and a lot has been written about the role and character or a good product owner[11].

As all product owners may not be jacks-of-all-trades, they often draw on the expertise of in-house or contracted architects, domain specialists, the development team, and other people to decompose requirements correctly and effectively. As it happens, there are security consultants waiting to serve product owners, as well.

Lean product development

Lean manufacturing is a concept from manufacturing, and the basic idea is to remove everything from production, which does not directly contribute to the product's value[9].

Whereas physical production lines can eliminate wasteful activities such as transferring items to inventory (warehouse) and back, there are many activities in software development that are analogously considered 'waste'. As software and most other artifacts from a product process are intangible and virtual in nature, most of the waste to be eliminated revolves around removing delays and bureaucracy and introducing automation.

In many cases, software production is done by a chain of teams or people. In the simplest case, the chains are short: a product owner passes on the backlog items to the Scrum team. However, in many cases, the Scrum team passes the code to a separate integration or deployment group, creating longer chains. Technical dependencies may also create chains: application developers may be dependent on user interface framework teams, who may be dependent on lower-layer operating system folks, who may again have a hardware dependency – all within a single product release.

Whenever there's a chain of separate actors, all sorts of delays, unfulfilled dependencies, unclear requirements and bureaucracy often manifest as *queues*. Work (for example, backlog items) needs to wait for someone or something to react on them, preventing the work from moving forward as quickly as it could.

When work items get queued, the smooth flow of work through the product development pipeline slows down. Like in traffic congestion, queues may cause an accordion effect: the time between work items starts to fluctuate. This affects both throughput and predictability.

More simply, queues also starve the downstream from work, which for employers of hourly salaried employees usually is a poor economic choice. Upstream, queues may result in overtime and overworked people, also a situation to be avoided.

Sometimes chains aren't one-dimensional but are in fact networks, giving rise to all sorts of complex network effects.

Scrum avoids queue-forming by empowering the team to do decisions without having to get a go-ahead from somewhere else, and by making sure that the team is being fed clear requirements in the form of product backlog.

Defining (agile) security

Defining security

One of the most useful definitions for security I've come across is

> *A secure product is a product that works as expected also under attack, or if it can't, fails gracefully.*

This definition is based on a common definition of a high-quality product (a product that fulfils customer expectations) but adds an unfriendly operating environment (an attacker) and a notion of robustness (works or fails gracefully). What 'failing gracefully' is depends a lot on what the product is. On your web browser, it could mean that when the browser fails to parse a piece of content, it will result in that content not being visible instead of getting a rootkit installed. In a battery charger, it could mean that the system just stops charging instead of risking out-of-spec charging current. On a network stack, it could mean silently dropping a malformed packet, at the lowest possible layer.

Actually, in a perfect world, we could just stop at '...as expected.' and just assume that attack scenarios and fail-safes have been rolled in the expectations.

Security risk management

In a typical business environment, not everything is worth protecting. When protecting against a security issue costs more than what it would cost to mop things up, it doesn't make business sense. Whether to fix things completely, partially, or not at all, is usually a risk management decision expressed in terms of money[10].

If you want to manage risk, you need some sort of measurement of it: the impact and the likelihood. In security, especially if we do not have a lot of history with our product or business, many measurements relating to security issues have a lot of uncertainty – especially those that concern the likelihood. This is because the impact won't be realised until someone actually exploits a vulnerability in software, and we may not have enough data to estimate the long-term behaviour of the attackers. Humans are also not very good at estimating complex systems risk. We have various biases that make us underestimate or overestimate both the impact and (especially) the likelihood of various cases.

But even if we do not have hard data, or follow a quantified risk model, we are still doing judgment calls and do risk management decisions. Security risk management in software product management and design starts with identifying threats. A threat is something that could go wrong. If you find a threat, which doesn't sound plausible at all, and ignore it, you have just made a risk management decision – you have decided (albeit probably not consciously) that the likelihood that someone would exploit a vulnerability that would realise that given threat is so small that you can accept the risk.

If you decide that the threat is plausible enough to warrant further discussion, then the next step is to decide what to do about it. This is known as *controlling* or mitigating the risk. In most cases, controls are not completely effective, but they decrease either the impact or likelihood, or both to a level that is acceptable.

Security risk controls are often security features (either a additional functionality, or ways to design the system in a certain way) or specific tests to detect any vulnerabilities that may have crept in.

Some risks can be effectively eliminated by changing the business model – for example, ditching Digital Rights Management from a music streaming app would eliminate all risks related to cracking DRM, but would also require a change of business case into something that does not require DRM. Also, if a piece of functionality is really error-prone, eliminating the code will also eliminate potential vulnerabilities in it. And finally, there's the non-technical solution of contractually transferring risk: if it breaks, someone else pays.

The risk that remains after applying the controls is called *residual risk*. The residual risk needs to be acceptable. If it isn't, the controls need to be revisited. Residual risk that is somehow ignored or 'not accepted' does not exist; when you ship the code, any unaddressed risks get implicitly accepted.

Security practices that break agile methods

What kind of security practices fit agile (incremental and lean) methods? Based on the preceding discussion, I've tried to simplify the requirements to a simple litmus test that will tell whether your process will get sour if you add a security-enhancing practice to it.

There are *two things to avoid*:

- Queues.
- Technical (in our case, security) debt.

If you plan to add a security practice to an agile product or project management framework, and that practice introduces either of these, you either destroy the agile properties or the practice will get "optimised" away. In a typical case, it will be paid lip service, but it has no real effect on the final product.

I can take now two examples of security practices that fail one or both of these tests, and both of them are widely used.

A security assessment (or audit, or review, depending on the terminology) that is conducted at the end of development – at 'go-live', possibly just before deployment – has a strong tendency to fail both tests. An assessment creates a queue, because tasks need to wait for a stamp of approval before they can be deployed. This means that the downstream of your product development flow cannot pull in new tasks as they have time.

A go-live assessment also has a very distinct chance of causing security debt, when developers and *especially* the product management think that security will be 'taken care of' by the assessment. If the development pivots before the assessment, the assessment may never get done. (In an ideal world, an assessment should just be a checkpoint that security practices have been followed, but in reality, this is often not the case.)

Another security practice that causes technical debt is a *security sprint*. This means a dedicated sprint where security aspects will be done. These come in various flavours – one is a widely used pattern, a *hardening sprint* (the name has nothing to do with security hardening though), which is a type of themed sprint just before a customer release.

Any time you find yourself saying that we'll do it later, watch out. You're getting into debt.

How agile security ought to look like

There are four things *to do* that help you to avoid queues and technical debt:

• Decentralise decision authority. This will help you avoid queues, because you will be able to do decisions locally. This also plays well with the empowerment idea of Scrum, where the team is empowered to do the implementation as they best see fit, and calling on resources external to the team as required. Decentralisation is especially important in residual risk approval phase, which we will discuss later. However, decentralisation requires mature development teams and a certain amount of rigor.

• Create fast, local feedback loops. This is another strategy that minimises queues. Fast feedback can take many forms: each team should have someone at hand locally who 'owns' security. Development should also make heavy use of automated testing and automated code analysis, which will immediately flag errors. As we see later, many security requirements can actually be expressed as security test case creation tasks on the product backlog.

• Make required work aggressively visible. There should not be any work, which is not visible as work tasks. This will help to avoid technical debt: if work is not visible on the product backlog, it is far more easier to brush it under the carpet, or pretend that it will be done later. We will see this pattern later when we discuss non-functional requirements, a prime source of technical debt. Examples: if a security document needs to be written for compliance monitoring, there has to be a task for it, and if the code needs to be resilient against bad inputs, there needs to be a task for implementing the appropriate test cases.

• Treat everything in small batches instead of big chunks. This will help both against queues and technical debt: big chunks will not clog your product management machinery with large analysis tasks, and you can get all the quality related work done for a small task during one sprint, leaving less opportunity for technical debt. This pattern is evident in how threat modelling and finding security requirements are done. As an additional benefit, people get better in doing things that they need to do often. A bit of threat modelling now and then builds better competences than rarely conducted huge threat modelling sessions.

Practical security in agile projects

Security risk ownership in an agile process

Product owners are just what it says on the tin: they *own* the product's business case. They are supposed to manage the requirements so that the customer gets the expected value out of the product.

One of the critical issues to get right is that the business ownership (which typically has positive connotations, such as power and authority) comes with accountability. Product owners also need to own the security risk of the product. Unless the product owners are held accountable for residual security risk they accept, there is not much incentive to add any security requirements – other than those that make a good sales argument.

Of course, product owners need to understand that they cannot, and in a lean system, must not, micromanage the teams that eventually deliver the functionality. But they should have a very valid case of requiring security risk controls and residual risk information from the Scrum teams. They should also always maintain a (hopefully rarely used) veto right over any code delivery.

The following provides some ideas on how product owners can accomplish all this.

Practical threat modelling and control selection

If done wrong, threat identification and mitigation are prime candidates for technical debt creation. A formal residual risk acceptance step easily creates a queue (just think of a process where you need to get a green light from a grumpy security guy who always says no).

Risk identification activity, called *threat modelling* or *threat analysis*, should happen when doing requirements identification (user story decomposition, backlog grooming) and design. This is when we ask ourselves what could possibly go wrong, and then come up with a list of things that probably will.

This will be intimately tied together with *control selection*. Depending on how rigorous our risk management framework is, control selection can be off-the-cuff based on intuitive risk impact and likelihood assessment, or be based on a stronger model. The controls are usually security requirements and security test case needs. If the threats were user-story level, then the security requirements would be security-centric user stories as well; if the threat was identified in technical design during the Scrum sprint, the control would typically be a design decision - usually a backlog item.

Residual risk acceptance would be done at the releasing stage. The Scrum team has a sprint-specific quality gate, called the *definition of done*. This is the first opportunity to accept any residual security risks. From there, the acceptance flows to the product owner, any further integration customers, and finally the end customer.

Doing threat modelling within Scrum teams has a couple of challenges. First, implementation-centric teams will look at implementation-level threats, not necessarily business case level issues.

Second, work within a team may be prone to a silo effect both in terms of time (addressing just the sprint at hand) and teams (trying to solve an issue only within the home team).

These challenges are in no way unique to security. It affects 'normal' software architectural work and any cross-cutting concerns. Typical strategies that have been proposed, include:

- All-seeing product owners[11], who are superhuman in their capacity of mapping complex cross-organisational challenges and being able to come up with requirements that fully and infallibly cover the business cases over every Scrum team;

- Architects or security specialists that try to find a living in a limbo outside the Scrum teams, and translate the non-technical requirements written by product owners into technical requirements, while being seen by the code-delivering Scrum teams as some sort of ivory tower folk;

- The eyebrow-raisingly named activity of *backlog grooming*, which means work where yet-unscheduled product backlog items are massaged to take into account cross-team or long-term needs. This is varyingly done as a part of sprint planning, or as an extra activity that gets a static 5% to 10% time allocation from Scrum team members depending on which agile consultant you listen to.

A potential solution is to recognise that in the Leffingwellian[12] product management setup (see the section "Agile product management" on page 9), there are *two distinct levels* of threat modelling, and both of these can happen without having to be tied together in any way:

1. The product owners (helped by architects and security specialists, if applicable) should do business case level threat analysis, determining high-level security requirements (what capabilities the software must have with regard to security). This is discussed in the section "Threat analysis for requirements" on page 16.

2. The Scrum team should do technical threat analysis, mostly concerning itself with design and implementation level aspects and often based on data flow diagrams. They know their software components in and out, can list their interfaces, and can make decisions on mitigating found threats by making specific design decisions such as implementing input sanitisation, sandboxing functionality into different processes, using parameterised database queries, deciding to add robustness testing cases to the unit tests, and so on. This is discussed in the section "Technical threat analysis" on page 20.

Threat analysis for requirements

As the high-level requirements, called epics or user stories, enter the requirement pipeline, they are usually purely positive functional requirements: "As a user, I want to be able to do this and that." There are rarely any considerations for what could go wrong, or what should not happen, or how the software should be built to withstand problems.

Product owners should generate the security requirements during the requirements decomposition and product backlog creation. In the end, all requirements should be positive functional requirements (as it is usually not possible to implement something that doesn't exist), but it is often helpful to allow product owners to use several types of requirement (or story) types:

- Positive functional requirements, that is, "security features".

- Expressing security requirements as non-functional requirements (but not yet specifying how these can be technically met, leaving that work for downstream definition work).

- Expressing security requirements as negative requirements, or misuse cases (again, not yet specifying how these can actually be met).

We will discuss the second and third types of security requirements in the next sections, but first we have to answer the obvious question: How can a product owner come up with meaningful security requirements of any kind?

By far the most important factor is raising awareness. Product owners need to understand what sorts of things are important from security and privacy viewpoints. If the organisation has a strong product management culture, training product owners in security matters should probably be one of the first things to do.

In addition to awareness training, product owners can be provided with a list of *security story templates*. These are generalised, 'prototype' user stories (of any of the above types) that can be interpreted in the context of the specific requirement which is being discussed. The list should mirror the past (or expected) threat profile of the company or software that is being made – a list for embedded industrial control devices should be very different than one for a consumer-facing web app.

The product owner could then take each security story template at a time, and see if that elicits any new security requirements.

As an example, the security story template list could have a generic security stories like

> *As a user, I expect all data I submit to the system be transferred securely, so that it won't leak.*

This story template would be given to all product owners to seed their product backlogs with. Let's say that the product owner would be working with a feedback form requirement. The product owner could translate these security story templates into actual user stories, for example, like this:

> *As the privacy officer of our company, I expect that the feedback form must use TLS for data submission, so that we fulfil [legal requirement x].* (A positive functional requirement.)

> *As an attacker, I must not be able to read or alter the feedback submission when it is being transferred.* (A negative requirement that leaves the implementation open.)

A list of security story templates is very close to being a checklist, but there are no boxes to tick. Instead, it's a kind of generic mini-threat analysis. In the worst case, a product owner will just copy the templates into the requirements list almost verbatim, but the expectation is that down the road those will be turned into tangible requirements (by an architect, a designer, or whoever is helping that product owner). Even in this case, this ensures that these security aspects will be visible on the backlog, so it will be less probable to forget about them.

These security story templates could be produced also by stakeholders down the line, such as operations and hosting teams, and

reactive vulnerability handling people. These lists would contain template stories that ensure that the operational issues are taken into account early in the design.

Later on, when the security stories are decomposed into implementable tasks, any non-functional requirements or negative requirements then need to be functionalised – something we discuss next.

Security as non-functional requirements

Computers are rather predictable. Everything that a program does, or doesn't do, has been baked in the program code. Security as a *functional* requirement is usually pretty easy to grasp. If you need a password, then require a password system.

However, in the end, what happens may be a result of complex interactions with other, non-computer systems, typically human wetware. Often, this part of software functionality is called 'non-functional', or the 'qualities' of software. (It is not a coincidence that most quality issues are non-functional in nature.)

How non-functional aspects of systems should be managed has traditionally been a weak point of many software development methodologies. As an example, some agile development advocates have proposed that non-functional requirements should be documented as a sort of metadata to the functional requirements – akin to having a Post-It note on the monitor saying, 'Remember security!'.

This doesn't really work. Non-functional aspects are no less real than functional ones. Bad performance, bad usability and bad security, all examples of non-functionalities, are very tangible qualities, and have their roots in specific program code. Making them right requires work, and work does generally not get done as an afterthought. Actually, work towards non-functional qualities is typically something that ends up as technical debt.

Scrum typically addresses non-functional qualities through its sprint-specific quality gate. A sprint is *done* when the (functional) tasks have been implemented on a level that is shippable. The level itself is defined by a list of quality criteria, the definition of done. This can be understood as a quality promise from the team saying that the code delivery comes with no technical debt attached. (See the discussion on residual risk, "Security risk management" on page 12.)

While the definition of done can act as a quality bar, and definitely has value in security as well, the problem is that if the requirements were incomplete to start with, '*doing*' them won't get the team very far.

Hence, the requirements discovery needs to be able to find also those requirements that are concerned with system interactions that are 'soft' (that is, have a psychological or social element – typically, interact with humans).

Non-functional requirements then need to be transformed into tasks that can be scheduled through the product and sprint backlogs. This is sometimes referred to as *functionalising* non-functional requirements, which basically shows that non-functional requirements are essentially just badly understood functional requirements.

The primary way of functionalising non-functional requirements is two-step: determining the metrics for the non-functional aspect,

and then measuring the code (that is, testing). The tasks that will be put on the product backlog then typically take the form

> *Design and implement a test case [to be run in an automated test system] that checks whether measurement x is less / more than y.*

When this task has been implemented, the test case enforces the non-functional aspect by failing if it hasn't been met. Note that the task to be added on the product backlog is *not* the actual act of testing; it is the *implementation of the test case* that ought to be on the backlog. The difference is that if the task would be the test itself, it would be run just once (as tasks are marked as done, and then forgotten). Instead, the task is actually about adding a test to a battery of tests, which is run, hopefully, very often.

A typical example of a security-related non-functional aspect is *robustness*, meaning whether the code can work properly given a lot of malformed or illegal inputs[16].

A robustness-related non-functional requirement would then be functionalised by adding a requirement such as

> *Implement a test case that injects 100.000 malformed inputs created with our fuzzer tool to the input reader process. The test case passes if the input reader process does not hang, crash, or consume in excess of 10% of CPU over a period of 10 seconds, and the API returns an error value EINVALID.*

Usability-related non-functional aspects may be more difficult to automate as test cases, because 'usability' is often about how people perceive the user interface, and people make pretty poor and expensive automatons. In these cases, it is ok to add a task that actually conducts a usability study of a proposed implementation once. If the implementation is ever significantly changed, a new usability test task should be added on the backlog. This resembles a strategy known as a *research spike*, a one-off prototyping or test round.

As an example of a usability-related security requirement, if we would have a requirement that the user would notice that a connection is not secure, we would add a research spike such as

> *Conduct usability tests with UI mock-ups of the Start connection dialog. If the server certificate cannot be verified against our root, 100% of users must notice that the connection is not secure.*

With very high probability, the usability testers would find that the only way to get this task implemented would be to use a design which actually fails the connection altogether if it is insecure. This is an important aspect of putting the measurement and pass criteria into all functionalised non-functional requirements: just requiring usability tests in general would not guarantee that the security requirement is fulfilled.

This brings us to the topic of requiring something that should not happen – the negative requirements.

Security as negative requirements

The vast majority of requirements in software engineering today are *positive*. They require what ought to happen. This is generally what the customer is paying for. Software that doesn't do something is generally not what is thought as a marketable product.

In security, what a product doesn't do is actually critical. Examples of what products shouldn't do include letting people to log on without proper credentials, leaking customer information, letting your embedded device battery to be charged with a too high current, and so on.

This means that in addition to positive user-stories or use cases, there is a distinct need to come up with *negative* user-stories as well. Negative user stories are also known in the literature as *misuse cases*[14] and *attacker stories*. If this sounds difficult, you can think of these as positive user stories for an attacker that should not succeed.

Revisiting the example of robustness in the previous section, an illegal input can also be a weird or undocumented interaction with a 'soft' system, like a human trying to use the system in unorthodox ways. An example could be

> *As an attacker, I must not be able to cause information disclosure by disconnecting the network cable in the middle of the transaction.*

As with non-functional requirements, negative requirements usually need to be transformed into positive ones before they can be implemented. In these cases, it can often be modelled as a test that features an invalid state machine transition or environmental change.

Whereas a sufficiently competent product owner can often invent the negative requirements, converting them to positive requirements may need some more technical expertise. Continuing with the example of the attacker story, above, plausible positive requirements in some example system that could be distilled from it could include requirements like

> *Change of IP address or a network interface going down must reset the transaction state to IDLE and the session key must be zeroised.*
>
> *If any response from remote server time-outs, the transaction state must be set to RETRY and the session key must be zeroised.*

This also shows the strength of negative user stories. Even a non-techie product owner is free to express any security worries in terms of what should not happen, without needing to have intricate technical knowledge of what avoiding the scenario would exactly entail. In this example, 'avoiding information disclosure' in the negative user story apparently requires some key purging activities – something that the designer might know, but the product owner might not. On the other hand, perhaps the product owner was thinking about a physical theft of a point-of-sale terminal, and the designer might not have thought that network cable acrobatics would be a scenario to consider unless the product owner had described it.

Technical threat analysis

As mentioned earlier in "Practical threat modelling and control selection" on page 15, there are two different layers of threat analysis. One has to do with security requirement discovery and articulating negative requirements, and might not address specific design issues; this is what was discussed in the previous sections and what would be product owners' responsibility. The other is what I call technical threat analysis, and is specifically concerned with the design and implementation level security issues. This would be done by architects, domain specialists, and the developers.

There are various ways of actually doing technical threat analysis. I have had the greatest successes by employing data flow analysis

using a data flow diagram (DFD). Explaining this method fully is beyond the scope of this e-pamphlet, but it consists of drawing the components, typically on process level, the data flows between them, including data storage, and then for each data flow and interface, using some method[15] to find potential attacks.

Using a data flow diagram has a significant benefit that you know when you're done. This isn't true of you would be using an open-ended threat analysis method or something that completely depends on the team's adversarial imagination.

Documenting the findings of a technical threat analysis should be carefully thought out. Documentation that is never used again is waste; on the other hand, having a repository of past threats can, when used well, help threat analysis. If the analysis result indicates that a security control is required, the control ought to be filed as a new backlog item. This way, it will be prioritised by the product owner just as any other requirement, and no further documentation (like logging "action items" and following them up) would be necessary.

If the team wants a repository of found threats, a wiki page would probably work well. Each threat would only need to be described well enough to be understood. I don't think that having quantitative risk ratings by the development team for every found issue would be beneficial in most real-life analysis sessions; if the issue is going to be fixed, the priority of the fix is going to be decided by the product owner through normal product backlog management.

Now, the pressing question from work scheduling perspective is: when should you do threat analysis, and especially – how you can avoid queues and security debt?

One solution is to borrow a method that is sometimes used for prototyping – a *research spike*. A spike is a backlog item that is scheduled on a sprint backlog just as any other item. Typically, spikes would precede complex implementation tasks that the team is not completely sure of how they ought to be implemented (also useful for certain hard-to-automate tests, see "Security as non-functional requirements" on page 18). As an example, a spike could be used to build a quick-and-dirty proof of concept that validates a design decision.

I have met two types of spikes. Some treat them as larger efforts, where a spike might fill almost an entire sprint. An alternative view is that a spike is a small task that can be used within a sprint, preceding another implementation task. Here, a threat analysis spike would be tending towards a smaller size effort – its size being dictated by the size of the feature being analysed.

A research spike is an excellent fit for threat analysis. Getting back to our queue and debt-avoiding strategies, it makes work visible. It's a task like any other backlog item, so it will be visible on the burn-down chart. And as the spikes are targeted to individual backlog items, it also fulfills the requirement of doing work in small bits at a time.

The need for threat analysis spikes can be identified by going through the sprint backlog items during sprint planning, or product backlog items during backlog grooming sessions (mentioned earlier in section "Practical threat modelling and control selection" on page 15). If any of those backlog items sound like it could benefit from threat analysis, a spike can be added to precede that backlog item (Figure 11).

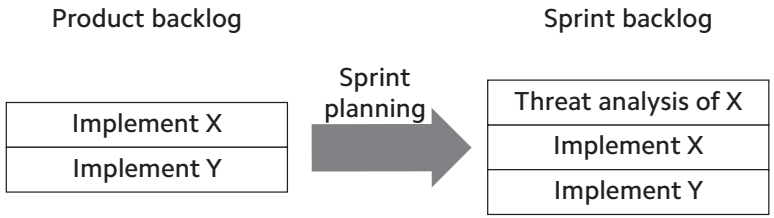


Figure 11: A spike can be created in sprint planning or backlog grooming. A spike gives a permission to use time for threat analysis and makes the work visible on the backlog.

Just to be clear: the idea is not that threat analysis would be done at sprint planning – the planning session is far too short for it, and also it would not make the work visible in terms of scheduling. Sprint planning should just act as a triage where threat analysis spikes are added as necessary. Grooming sessions could be long enough to accommodate analysis, but the problem with that is that work done within a grooming session is not explicitly visible and therefore not subject to product owner's prioritisation decisions.

Triaging sprint backlog items for threat analysis during a sprint planning session needs to be quick. This can be accomplished by using a checklist of indicators whose presence strongly suggests that a threat analysis would be in order. The list of indicators could be based on a general list, but probably should be based on historical vulnerability data for your company ("ways in which we usually shoot out own foot").

A backlog triage checklist for a fictitious web app company could look something like this:

Add a threat analysis spike for every sprint backlog item that changes or affects functionality that:

- *Processes data that comes from the network, or*
- *Implements any part of the sign-in authentication mechanism, or*
- *Processes users' private data, or*
- *Communicates with our back-end storage systems, or*
- *Handles logging.*

Embedded device vendors would probably add checklist items for safety-critical areas and financial systems would add anything that touches money or payment data.

Practical residual risk approval

Residual risk approval instantly conjures up images of rubber-stamping reports and waiting for approval from the top. It almost reeks of a queue.

It is very important to explicitly address residual risk by either accepting it or requiring further risk mitigation efforts. Residual risk cannot be rejected or ignored – or, sure you *could* live in denial, but that is equal to accepting the risks eyes closed.

Residual risk approval should always follow business ownership. This is because, as we discussed earlier, security risks are business risks. If you have a separate security group that has an authority to accept risks, you are actually ceding them the authority to make business decisions. If leaving the security group without any formal authority sounds too adventurous, consider giving them only a veto right for the release of software. Many companies do this, and

their security groups know that they need to have a very, very good rationale if they ever use that veto right. But do not let them approve any residual risks.

In software business, the lowest level of business ownership is the ownership of source code. Source code codifies your business logic and at least modulates your revenue stream. (If it doesn't, you aren't in software business, but in some other business.) Business ownership then typically follows code as it gets integrated and packaged towards the customer.

The way to avoid queues is to decentralise residual risk acceptance. The Scrum team, as the custodian of code, should be empowered to make residual risk decisions on matters of code-level design decisions. The product owner, as the owner of the user stories, should be able to decide on residual risk as it pertains to those.

Typical risk acceptance levels would look like:

Role	*Authority*
Development (Scrum) team	Code-level design decisions
Product owner	User story level decisions (with veto option)
Client (or a higher level business owner)	Business case or epic level decisions (with veto option)

The Scrum team should do residual risk acceptance during the sprint review. Residual security risk acceptance can be made one of the aspects of *done*, and listed in the definition of done. For example:

> *For all the security threats we have identified for the backlog items in this sprint, is the risk on an acceptable level?* (Meaning: have we implemented appropriate controls, and/or is the residual risk acceptable?)

On any layer (except the very top layer, where the buck has to stop) the most problematic risk approval decisions can be escalated. The Scrum teams can use this as a self-regulating system. If they feel that they are out of their league in approving a risk, they can just call the product owner and ask for guidance. If they happen to live in an environment, which has a culture of finger-pointing, they probably want to cover their bottoms, and can do that by email ("in writing"), creating a paper trail.

This cuts both ways. The product owner should have veto power over the Scrum team's risk acceptance decisions. In practice, this is harder than escalation, as it would require a way for the Scrum team to report the residual risk in some coherent way (so that the product owner actually knows what to veto).

If the security threats and controls (or decision not to control them) have been written down in a list as proposed in earlier, this same list can serve as risk approval documentation without any extra work from the Scrum team. The product owner can just review the list (stored, hopefully, on a wiki) whenever convenient. A sign-off on the list by the product owner, perhaps at the time of an external release, can also contribute to the paper trail.

Security in development, integration and releasing

The topic of this e-pamphlet is security risk management in product management, so I will gloss over the actual coding and testing activities. Of course, they are critical, but much more has been written about them previously, so I will not go into much detail here. Most agile coding and testing practices (for example, test-driven development) have a lot of potential to increase security as well as other aspects of quality.

One option that has been picked up by several shops is the introduction of a team-specific security "evangelist". Usually it would be best to pick a volunteer with an inner flame for security. This person's role would be to act as the "security conscience" of the team.

In addition to the language and environment specific security knowledge, there are two activities that target the robustness and correctness of program code, and that fit the pattern of having fast and local feedback loops, so I will cover them here.

Static analysis tries to determine if the code is correct by looking at code; fuzz testing tries to break code by hurling loads of invalid data against it. Neither is very good at finding business logic level issues (that's what all of the preceding sections were about), but both are great ways of finding implementation bugs.

Automation is the key to fast and local feedback loops. Effective secure coding would use static analysis as early as possible, perhaps already when the developer is about to commit the code. This would decrease the time between a programmer making changes to the code and having to redo some of the changes because static analysis failed. If the analysis could be run when actually checking in changes, the programmer would still have the mental context of that piece of code, and fixing the issue would be potentially very quick.

Robustness testing (for example, fuzz testing) should be brought into the unit tests and executed in a test automation system. Whenever anyone implements an API or a parser, the set of unit tests should also throw a load of crap against the interface – preferably crap generated by a good fuzz test tool. Doing robustness testing on this level would catch regression early. Also, doing it in automated testing (as opposed to running a test tool manually) should enable larger test case sets to be run.

Summary of activities

Risk management

• Ensure that the definition of done and sprint reviews exist. These are the sprint-level quality gates.

• Decentralise design and code level risk acceptance to the Scrum teams, and do not require a sign-off from a remote security group. Give the security group a veto right instead, and the same visibility to product backlogs as the product owners have.

• Create security awareness among product owners, authorise them to make risk acceptance decisions on user story level, and make them accountable for software security.

Scheduling software security practices

• Use the product backlog, instead of fixed dates, ad hoc interventions or themed sprints, to drive all security work such as threat analysis and security testing.

• Make all security work visible through the product backlog, and do not assume that security work will be done as a result of some release or acceptance criteria list.

Agile practices

• Support the efforts to increase the maturity of agile practices in your organisation by allying with any agile evangelists or coaches.

• Make sure that having non-functional user stories and negative user stories on the product backlog is explicitly allowed.

• Try to get security on the teams' definition of done.

Guidance

• Create security-related user story templates and train product owners to use them to seed their product backlogs whenever they start a new project.

• Train product owners in the creation of attacker stories.

• Give guidance and training to development teams so they can do effective design-level threat analysis.

• Create a checklist for teams that they can use to determine whether a product backlog item requires security threat analysis.

Tooling

• Invest in test automation. This allows test case writing to be put on the product backlog.

• Introduce tools that make it easy to write automated security tests.

Endnotes

1. A somewhat curiously called *Nokia Test* can be used to determine whether agile practices adopted by an organisation exhibit cowboy agile properties. See, for example, http://agileconsortium.blogspot.com/2007/12/nokia-test.html.

2. Wikipedia credits the term Minimum Viable Product to Eric Ries, who explains MVP in his interview: http://venturehacks.com/articles/minimum-viable-product. This is also useful outside the startup context. If a company produces updates to an existing product that need to be re-sold to the users as upgrades, crystallising the minimum marketable incremental feature set to aim at is crucial.

3. Pivoting as a term is often used of companies reinventing themselves. Here, I am using it in a less broad context of a changing business target for a single software development project. Eric Ries explains pivoting: http://www.startuplessonslearned.com/2009/06/pivot-dont-jump-to-new-vision.html.

4. Technical debt is usually credited to Ward Cunningham: *The WyCash Portfolio Management System*, 1992, http://c2.com/doc/oopsla92.html.

5. Although not specifically talking about technical debt but instead the lack of "proper" controls, the Payment Card Industry Data Security Standard (PCI DSS) says "[c]ompensating controls may be considered for most PCI DSS requirements when an entity cannot meet a requirement explicitly as stated, due to legitimate technical or documented business constraints, but has sufficiently mitigated the risk associated with the requirement through implementation of other, or compensating, controls". (https://www.pcisecuritystandards.org/security_standards/glossary.php#C)

6. Agile Manifesto can be read at http://agilemanifesto.org/.

7. Ken Schwaber and Mike Beedle: *Agile Software Development with Scrum*. Prentice Hall, 2001. You may also like to have a look at the book in endnote 8.

8. An great book explaining kanban is Henrik Kniberg and Mattias Skarin: *Kanban and Scrum: Making the most of both*. InfoQ, 2009. http://www.infoq.com/minibooks/kanban-scrum-minibook

9. The best book I know on clarifying lean development processes and queue forming is by Donald Reinertsen: *The Principles of Product Development Flow: Second Generation Lean Product Development*. Celeritas, 2009.

10. This is not a statement of what is ethical. This does not mean that this would be a licence to sell crap to consumers. But of course, as long as it costs less to sell crap than to fix things properly, some actors are going to do just that. Also, there are some cases when there is more than purely monetary risk. Product safety issues sometimes cross into the product security world, a typical example being the availability of emergency communications. Bad decisions on these areas may put people's health, lives, or the environment at risk. On the non-tangible front, privacy breaches sometimes carry criminal penalties. Your CEO might disagree with your risk

appetite when he is talking to you from behind a plexiglass in the visitor room. In most other cases, however, you can target at a suitable risk level based on monetary considerations. Very few businesses make money by not taking any chances.

11. A colleague of mine once quipped that product owners are anthropomorphic non-functional requirements. If you are about to recruit a product owner, and want to know how superhuman one really needs to be, have a look at the book by Roman Pichler: *Agile Product Management with Scrum: Creating Products That Customers Love*. Addison-Wesley, 2010.

12. A concise paper on agile story-based product management is by Dean Leffingwell and Juha-Markus Aalto: *A Lean and Scalable Requirements Information Model for the Agile Enterprise*, 2009 (http://scalingsoftware-agility.files.wordpress.com/2007/03/a-lean-and-scalable-requirements-information-model-for-agile-enterprises-pdf.pdf). The topic is also covered in a more recent book by Dean Leffingwell: *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley, 2011.

13. The release train concept is explained in Dean Leffingwell: *Systems of Systems and the Agile Release Train*, 2007 (http://scalingsoftwareagility.files.wordpress.com/2009/03/whitepaper_systems-of-systems-and-the-agile-release-train2.pdf).

14. Misuse cases were coined as a term in Guttorm Sindre and Andreas L. Opdahl: *Capturing Security Requirements Through Misuse Cases*, 2001, http://folk.uio.no/nik/2001/21-sindre.pdf. The misuse cases that are suitable for agile product management usually have an attacker as an actor ("As an attacker…") and should also contain the target of the story ("…so that this-and-that won't happen"). This is why I prefer calling these attacker stories rather than misuse cases.

15. One such method is STRIDE from Microsoft See Adam Shostack: *The Trouble With Threat Modeling*, http://www.homeport.org/~adam/The%20Trouble%20With%20Threat%20Modeling.docx

16. In this world of manifestos, there's one for this, too, called the Rugged Software Manifesto: http://www.ruggedsoftware.org/