Designing Secure Architecture of Health Software using Agile Practices

Alexander Pirker, Nadica Hrgarek Lechner
MED-EL Elektromedizinische Geräte GmbH
Fürstenweg 77a, 6020 Innsbruck, Austria
{alexander.pirker, nadica.hrgarek}@medel.com

Abstract. *Software is becoming increasingly important as a way of increasing clinical effectiveness and reducing costs in healthcare. Companies need to design robust security and privacy into health software to avoid unauthorized access and disclosure of sensitive data, modification of data, or loss of function. The main purpose of this paper is to present appropriate agile practices and approaches which explicitly allow for a secure-by-design architecture in a healthcare context. Specifically, we show how to apply principles, practices, and patterns from Domain-Driven Design (DDD) in an agile environment to control the design process of health software to build a secure solution.*

Keywords. agile, DDD, domain-driven design, health software, privacy, security, software architecture

1 Introduction

The healthcare industry is highly regulated and relies on integrated risk management to manage risks to patient safety. Risk management in healthcare can mean the difference between life, injury, serious injury, and death. Therefore, security and privacy expectations from regulators and users of safety-critical health software are very high. Cybersecurity is an area with increasing risk to patients as more medical devices use wireless, Internet, and network connectivity in order to deliver care, remotely monitor patients, or transfer patient data. The networked functionalities introduce new risks that can adversely affect device functionality, disrupt the delivery of health services, and lead to patient harm. Any medical device that is configured to use networked functionalities is at risk of a cyber-attack, if the device does not have adequate security controls.

Information security and user privacy are two major challenges when designing health software. Health software can be defined as 'software intended to be used specifically for managing, maintaining or improving health of individual persons, or the delivery of care' (*IEC 82304-1: Health software – Part 1: General requirements for product safety*, 2016, p. 8). It includes software as a part of medical device and software that is a medical device on its own (i.e., standalone software, mobile medical apps).

In the past, health software was usually developed using a traditional waterfall model in which each activity is taken in sequence and outputs are created/updated, reviewed, and approved. This approach expects the requirements to be stable during a project. Software change requests, requirement changes, and bug fixes during implementation, verification, or validation require development to go back through the entire sequential process. Big design up front approach tried to tackle this challenge by fully completing the detailed design before implementation is started. However, when trying to predict the future and to define all the requirements for large, complex applications up front, years may elapse between defining requirements and actual implementation. During this period, many factors influencing product development may change: technology, customer needs, old markets disappear, competitors offer a substitute or similar product, new attack vectors come from unexpected places, etc.

In the last decade agile methodologies have been gaining importance in the medical device industry due to the ability to quickly respond to the evolving business needs. An agile approach relies on cross-functional team collaboration and expects that the customer is engaged throughout the development process to increase the chances to build the successful product the customers want. In iteration-based agile projects value is delivered from the start and often. Explorative studies such as that conducted by Hadar & Sherman (2012) showed that while architects practicing only plan-driven methodologies perceive architecture activities as being related only to the first phases of the development process, architects involved in agile projects perceive architecture activities to be related to most or all phases of the development lifecycle.

Security and privacy, as non-functional requirements, turned from a desirable ('nice to have') to a 'should have', and finally arrived at an essential ('must have') requirement. In this light, when healthcare organizations did not explicitly define

security controls a priori, former approaches like waterfall failed completely, since most software vendors tried to implement such non-functional requirements at very late development stages. Hence, security was implemented on top of, or into an, almost finished software product, which is very challenging, and often fails.

Babar (2009) found that lack of focus on quality attributes for making design decisions usually results in architectural structures that can hardly meet quality requirements later. Nord et al. (2012) found that lack of desirable security in an architecture can necessitate enormous rework. According to McGraw (2006, pp. 16-17), Microsoft reports that more than 50% of the problems the company has uncovered during its ongoing security push are architectural in nature. In this paper we show how to overcome these issues by using agile practices in combination with an iterative and incremental architectural approach applied to health software. Particularly, designing software incrementally in an agile environment with a constant focus on security aspects enables developers and architects, but also stakeholders, to intervein at early development stages when the solution evolves into an unsecure direction. This in turn allows for tight control of the design process and contributes to building health software which is secure by design.

The overall structure of the paper takes the form of five sections, including this introductory section. The second section provides a brief overview of relevant related work. The third section deals with embedding security and privacy into architecture of health software and presents the implementation of the proposed design. A set of agile practices for a secure software architecture is presented in the fourth section and shows a different approach to the state-of-the-art. Finally, the conclusion summarizes the results of this paper, draws conclusions, and identifies areas for further research.

2 Related Work

In April 2019 we performed a literature research to find available full-text sources relevant to the topic being explored. We used the EBSCOhost online research platform and found several articles. The most articles were found in the IEEE Xplore® Digital Library. Afterwards, we evaluated which material makes a significant contribution to the understanding of the research topic. Finally, we discuss the findings and conclusions of relevant literature in this paper.

2.1 Security Architecture

Embedding security into software architectures has been investigated by researchers in many fields. For example, Moriconi et al. (1997) describe a new application independent approach to secure system design in which the desired security properties of the system are proven to hold at the architectural level.

Chivers et al. (2005) illustrate how to grow security within an agile project, by using an incremental security architecture which evolves with the code. The term iterative security architecture has been used by the authors to refer to an architecture that develops with the system and includes only features that are necessary for the current iteration or delivery. In their case study of costs associated with architecture-related waste, Nord et al. (2012) were able to demonstrate that architecting in many smaller increments reduces the cost of delay that results from waiting for an entire architecture to be completed. They also found that rework is costlier, because it might involve rearchitecting.

In his introduction to an agile security architecture, Harkins (2016) emphasizes the need of an architecture that quickly and automatically learns and adapts to new challenges as they emerge. With delivery cycles shortening it is essential to build security into every step of application delivery (*Freeform Dynamics: Integrating Security Into the DNA of Your Software Lifecycle*, 2018). Special attention should be paid to avoid architecting in security vulnerabilities.

Vai et al. (2015) draw our attention to an embedded system architecture that decouples secure and functional design aspects. Such an architecture addresses confidentiality and integrity by protecting the boot process, information, and communications from unauthorized access and alternation. The major limitation of the architecture is that a security coprocessor that implements cryptographic primitives in hardware does not ensure a system's availability.

Asokan et al. (2018) propose design of Architecture for Secure Software Update of Realistic Embedded Devices (ASSURED), a secure and scalable update framework for Internet of Things devices. Santos et al. (2017) present a new CAWE (Common Architectural Weakness Enumeration) concept, a catalogue of total 224 architectural weaknesses, such as information exposure through log files, reliance on security through obscurity, improper authentication, use of hard-coded cryptographic key, insecure storage of sensitive information, improper certificate validation, download of code without integrity check, etc.

Pedraza-Garcia et al. (2014) present a methodological approach to address and specify the quality attribute of security in architecture design applying the following security tactics: detect attacks, resist attacks, react to attacks, and recover from attacks.

Houser (2014) suggest integrating the following security controls for software architecture and design into system development lifecycle process: threat modeling, attack trees, misuse cases, identity and access management, least privilege, formal methods, secure design patterns and session management.

Recently, Derdoura et al. (2015) investigated the design of an architecture meta-model that considers security connectors. They propose a generic meta-modelling approach called SMSA (Security Meta-model for Software Architecture) to describe a software system as a collection of components that interact through security connectors. In their work we observed that security connectors should be integrated at a high level of design by using distribution concepts (e.g., domain giving an assembly structure and providing multiple spaces of abstraction).

2.2 Agile Architecture

Agile was originally developed for the software industry ("Manifesto for Agile Software Development," 2001). However, demand for faster development and delivery of new products and services has led to the adoption of agile and lean approaches by many industries including healthcare. The results of the 12th annual summary report on agile sponsored by CollabNet VersionOne (2018) showed that accelerating software delivery, managing changing priorities, increasing productivity, better business/IT alignment, and increased software quality are the top five reasons for adopting agile. In general, the term agile has been considered throughout the entire development life cycle ranging from agile architectures (Isham, 2008; Waterman, 2018) to agile testing.

Mekni et al. (2018) describe the methodology for software architectural design in agile environments. The proposed methodology consists of the following steps: 1) definition of architectural requirements, 2) identification of software architecture styles, 3) evaluation of software architecture, 4) determination of architecture scope, 5) description of software architecture, 6) integration of software architecture, and 7) continuous architectural refinement. Waterman (2018) discovered that teams design agile architectures using five tactics: 1) keep designs simple, 2) prove the architecture with code iteratively, 3) use good design practices, 4) delay decision making, and 5) plan for options (i.e., make decisions that retain flexibility and don't close off future options). Fontdevila & Salías (2013) propose the following agile architecture patterns and practices: a) "sashimi" approach to the architectural definition, b) the concentric approach, which starts with overall vision and keeps growing as we get closer to final implementation, c) managing quality-attribute requirements, and d) architecture validation.

Sturtevant (2018) concludes that a balanced focus on agile process and agile product architecture is needed to achieve long-term agility. When using agile approaches, Babar (2009) identifies the following key architecture-related challenges: incorrect prioritization of user stories, lack of time and motivation to consider design choices, unknown domain and untried solutions, and lack of focus on quality attributes.

As shown in Fig. 1, Woods (2015) identifies twelve practices for successful agile architecture covering the core values of the agile manifesto.

Allow for change	**People over processes and tools**
• Deliver incrementally • Capture clear architecture principles • Capture decisions and rationale • Define components clearly	• Share information using simple tools • Have customers for every deliverable
Software over documents	**Collaboration over contracts**
• Create "good enough" architectural artifacts • Deliver something that runs • Define solutions for cross-cutting concerns	• Work in teams, don't just deliver documents • Focus design work on stakeholder concerns • Focus on architectural concerns

Figure 1. Practices for successful agile architecture (Woods, 2015)

2.3 Security Architecture of Health Software

As discussed above, a considerable amount of literature has been published on security architecture and agile architecture. Security architecture of health software starts from inception and ends with decommissioning and disposal. To the authors' knowledge, very few publications are available in the literature that identify appropriate agile practices and approaches which explicitly allow for a secure-by-design software architecture in a healthcare context. Therefore, this paper considers security architecture and agile architecture that can be used for any software project. The agile architecture is interesting because this paper seeks to define a set of agile practices in combination with an iterative and incremental architectural approach applied to health software.

Pauli & Xu (2015) present an approach to the architectural design and analysis of secure software systems based on the system requirements elicited in the form of use case and misuse case diagrams. They have demonstrated through a case study on a security-intensive hospital information system that dealing with security issues at the software architecture level can make a system more resistant to vulnerabilities.

Alibasa et al. (2017) discuss a software architecture for storing and managing data collected in mobile health apps. Their software architecture is designed to separate identifiable from non-identifiable data that can be kept anonymous.

3 Embedding Security and Privacy into Architecture of Health Software

3.1 Motivation

The design of software which shall solve a complex task, such as controlling an airplane, controlling an active implantable medical device, or programming an embedded device, is very challenging, and many companies fail (after years) for a number of reasons including a lack of proper software design. But what is causing software design for such complex tasks to be so hard? There are basically many different reasons for this. One certainly is due to a variety of ways to design and structure software solutions.

Early architectural approaches follow layered architectures with separated layers (Sharma et al., 2015). For example, many applications use the common "user interface – business logic – persistence" three-tier architecture pattern. Here, the dependencies (should be) are directional, meaning that the business logic layer depends on the persistence layer but not vice versa. After years of development, architectural constraints, such as project dependencies, start to elute and the software architecture turns into a big-ball of mud. Suddenly each small change in the software triggers a huge development and testing effort to tweak the code in such a way, that the product supports new features or bugs get fixed. Such unexpected rework causes considerable costs which increase and slow down the overall project progress. Furthermore, security does not have a pre-defined place to be built in, it is spread over different components and layers, thereby leading to a potentially unsecure product. This finding corroborates the ideas of Fernandez et al. (2008), who described that the three-tier architecture and its variants do not consider security and therefore security aspects should be added by applying appropriate security services at each layer. Tang & Shen (2009) studied how classical Model Driven Architecture framework can be extended to consider the security aspect which helps to identify security flaws early in the software development process.

Health software often operates in very complex environments. On the one hand, health software runs in a highly heterogenous software landscape in a hospital, which often involves many other software systems. But on the other hand, health software may also need to connect and communicate with (implanted) medical devices. For instance, software for programming a cochlear implant communicates with a Hospital Information Management System (HIMS) and the implant itself. Furthermore, there may also be the need to communicate with a back-end server system in cloud. Security in such an environment has many different facets: privacy, access control, communication security, etc. Some of these facets even vary from country to country due to regulatory requirements that software manufacturers have to comply with. Recently, such regulations were more pushed towards cybersecurity (FDA, 2014), which highlights the need for secure medical devices. If security controls to tackle these facets are spread over the code, complexity increases, and it becomes very hard to comply with current regulations.

3.2 Domain-Driven Design

Domain-driven design (Evans, 2003; Millett & Tune, 2015) provides a well-established framework to design and build the business logic in hexagonal architectures that are discussed in the next section. The goal of DDD is to align software artefacts, such as design, code, and documentation, with the business domain which the software aims to solve. This results in several interesting advantages. First, it enables software companies to respond very fast to changing requirements, which usually happen due to changes in the business domain. Second, it allows for a collaborative engineering process which may even involve domain experts, thereby reducing the chance of developing into wrong directions, resulting in high costs and unsatisfied users. Third, DDD puts the focus on what is essential to the user and its core needs, thereby leading to a successful product.

DDD basically distinguishes two spaces: the problem space, and the solution space. The problem space abstractly models the underlying business domain, with all its processes and dynamics. Here, the domain experts carry all the knowledge about these processes. The solution space, corresponding to the software artefacts, shall solve or assist, the processes of the problem space. Developers, working on the solution space, need to get to the domain knowledge of the experts, which is done in so-called "knowledge crunching sessions". To prevent misunderstandings in such sessions, developers and experts agree on a common language, also referred to as ubiquitous language. This language is more than a glossary, as it also defines the names of coding artefacts (e.g., class names), which allows for interactive engineering sessions with experts. Following such an approach directly without any further considerations leads in many cases to one problem: What happens if different experts use the same vocabulary to describe their domain?

For example, let us assume that a software company needs to deliver a solution which assists an online warehouse. Suppose that during a knowledge crunching session a sales person talks about an order item. The sales department characterizes an order item by its price, article name, etc. However, if a shipping person talks about an order item, it only cares about the weight or size. From a coding point of view, even though they correspond to the same physical item, these persons are talking essentially about different things. This ambiguity motivates bounded contexts, one of the most important concepts in DDD. Bounded contexts introduce boundaries for terms of the ubiquitous language to prevent ambiguity in software artefacts. More specifically, the solution space is broken down into several bounded contexts,

each solving one (or several) sub-domains in a highly cohesive manner without any linguistic ambiguity.

Health software tremendously benefits from applying techniques from DDD, due to several reasons. Processes in hospitals tend to be very complicated, and often involve a lot of people with different roles and responsibilities. Keeping the software artefacts as close as possible to the real-world domain processes of the hospital ensures that the software really helps its users and does not lead to dissatisfaction due to failing knowledge logistics. We interpret this as meaning that all necessary information is present to the correct user at the right time in the most accurate form. This is not only important for the health software provider to be successful, it is also of high relevance to the hospital. We feel strongly that applying techniques from DDD may efficiently reduce waiting times for patients, and medical errors and adverse events because of a lack of information sharing.

Dividing the business logic, or domain, into several bounded contexts is also interesting from an architectural point of view in hexagonal architectures. The business logic defines in a hexagonal architecture via ports how it wants to communicate with the environment. Therefore, because a bounded context solves one or several sub-domains, it comprises together with the implementing adapters of the ports it defines, a full aspect of the complete system. Observe that this leads to independence: if companies structure their teams cross-functionally according to bounded contexts, then teams may evolve the overall software product independently. Furthermore, teams can develop each bounded context in an incremental way, which means that they introduce new software artefacts user story by user story rather than designing the whole system at the very beginning.

3.3 Hexagonal Architectures

Hexagonal architectures (also known as ports and adapters architectures) provide an alternative way of tackling the complexity of software solutions. Such architectures follow a very simple idea: settle the business logic/domain at the core of your application and let the environment depend on it. More precisely, the domain itself does not depend on surrounding aspects or components (e.g., database, user interface, security, etc.), it only defines in terms of interfaces (also referred to as ports) how it wants to communicate with the outside world. Surrounding components implement these ports (also referred to as adapters), thereby providing the missing functionality which the domain relies on.

For example, consider a software product for a radiology department of a hospital. Such software is intended to support various processes, including the diagnosis of patients, or the management of medical x-ray images. When the software follows a hexagonal architecture, the core of the application (i.e., the domain component) implements these domain processes. Observe that some processes will need to communicate with the HIMS, for instance to send the diagnosis of a patient to the HIMS. In a hexagonal architecture, domain components define a port (e.g., a HIMS port) which an environment component (e.g., a HIMS communication component) shall implement (see Fig. 2). An instance of this implementation (i.e., an adapter for this port) is passed to the domain component during start-up of the application, thereby providing the missing functionality to the domain component. As shown in Fig. 2, the radiology domain component which implements domain specific logic for a radiology is independent, and it defines in terms of ports (e.g., IRepositoryPort, IHIMSPort, and ILoggingPort) how it needs to communicate with the environment.

Observe that such an approach allows to easily exchange the environment of a domain component. For example, if the HIMS provider changes, then developers just need to exchange the HIMS communication component of the system rather than re-implementing several components of the application. Similarly, if the database provider changes, then database engineers simply need to implement a new persistence component. Again, the domain component will not be affected by such changes. Importantly, the same principles also apply to the user interface of an application. More precisely, because the domain component does not depend on the user interface, manufacturers can develop a completely user interface on top of an existing domain component without even touching it. We illustrate this fact in Fig. 2 by the component “User Interface V2.0”, which simply reuses existing domain logic which the workflow service “XrayDiagnosisService” in the “Radiology Domain” component already implements. Observe that no further modifications are necessary to implement a new user interface.

A full list of environment components of the radiology software product’s hexagonal architecture illustrated in Fig. 2 lies beyond the scope of this paper. As stated in Section 3.1, health software operates in a very complex environment. This environment includes other systems within a hospital (to which the software ultimately has to talk to), databases, audit logs, archiving, and potentially also embedded devices, or medical devices in general. Furthermore, at each interface between systems, different communication standards may apply (e.g., HL7, DICOM, etc). One way to prevent that such complex environments diffuse into the business logic of health software is to follow a hexagonal architecture. This ensures that business logic stays very clean, and the environment is strictly separated from it.

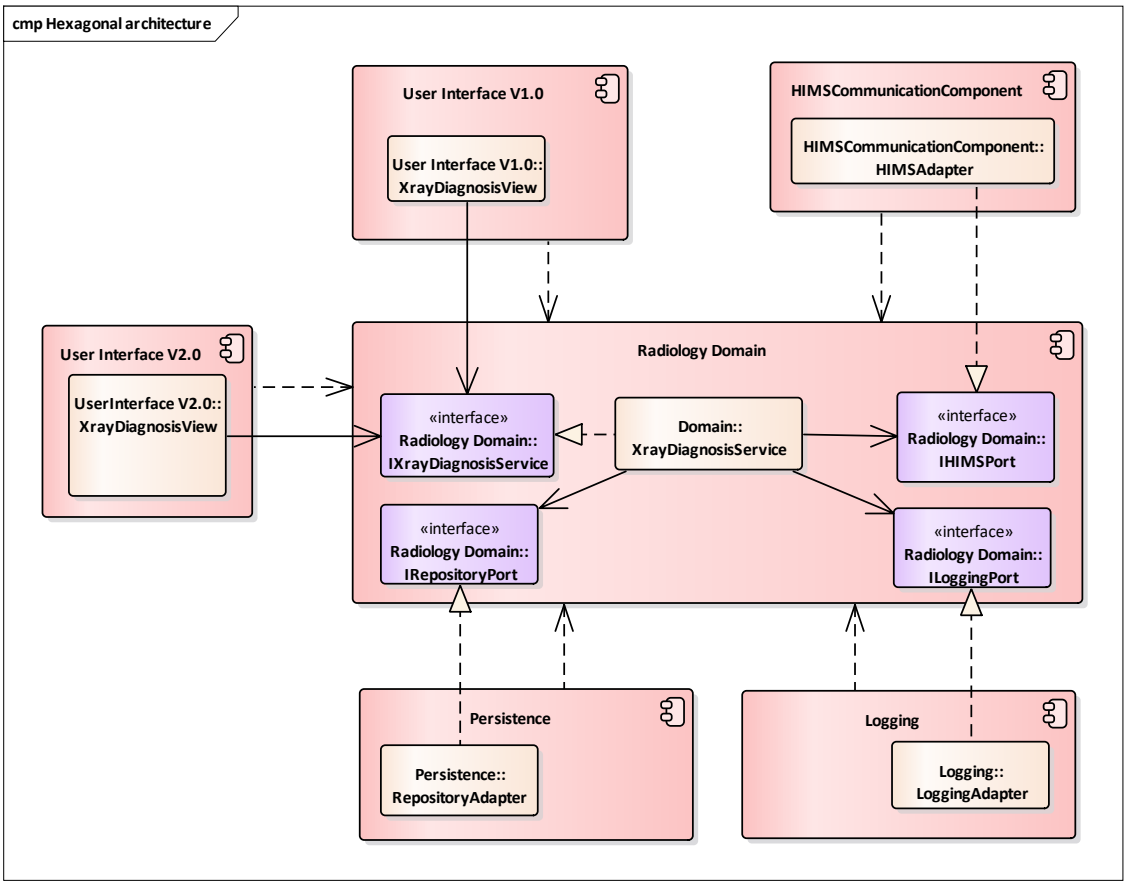

Figure 2. Schematic illustration of a radiology software product which follows a hexagonal architecture. Further environment components are necessary to comprise a full radiology information system.

3.4 Security in DDD and Hexagonal Architectures

Having one team which incrementally develops one bounded context is very beneficial from a security point of view. Since the development team knows best about the security considerations (due to knowledge crunching sessions with the domain experts) in a given sub-domain they can abstract security controls as ports in the bounded context and implement them in a dedicated security assembly. From an architectural point of view such an approach has several advantages since it leads to isolation of security controls:

1. Security controls grow and evolve together with the bounded context which they support, thereby explicitly allowing for incremental changes as requirements (may) change.
2. Security controls are easily testable due to their high isolation. In our opinion, testing security controls of software which does not isolate security controls properly can be very tedious, time consuming, and error-prone, leading to insecure products. Furthermore, since health software often also has to communicate with embedded devices, it also provides a great way to test security controls protecting communications with such medical devices, since those can easily be simulated by implementing ports of the bounded context.
3. If bugs concerning security are discovered, developers can fix them quickly by just modifying the affected security assembly. This is a huge advantage from a regulatory point of view, since it minimizes the effort of verification testing and re-validation of the parts of the health software that have been affected by the software maintenance. In case of detected errors that can have an impact on safety and/or security, this approach ensures timely security patches and updates, also enforced by regulations (*IEC 82304-1: Health software – Part 1: General requirements for product safety*, 2016). For former architectural approaches, such fixes have triggered a tremendous amount of retesting and regression testing, thereby slowing down the rollout and response time of the bug fix.
4. Because not all countries have the same regulations regarding security (and their controls) of health software, security assemblies offer a great opportunity to comply with different regulations from various countries without

affecting domain assemblies carrying business logic at all.

5. Secure code reviews, security audits and penetration tests, either conducted internally or even externally by security consultants or providers, get vast more efficient, since the focus and scope of the security review narrows down to a single dedicated security assembly for a bounded context.

We highlight that this in stark contrast to former architectural styles which were state of the art in previous decades like e.g. layered architectures, where security controls often spread over different assemblies. Therefore, their testing, as well as their evolvement over time or reviewability always appeared in a very constrained way, and was not solely concerning a single dedicated assembly, which slowed down the development and review processes. We illustrate these benefits by providing several examples.

Suppose that a hospital wants to restrict the editing of patient data to physicians only. Such a requirement may emerge from a hospital directly, but also from regulations. For that purpose, the hospital offers an internal web service which allows to query the internal employee directory. To support this requirement, the developers abstract this security check by introducing a port in the bounded context for patient management and implement the check in a new security component. In this component they connect to the internal web service, query for the logged in user, and check whether it is a physician or not. Observe that in such a case the implementation of the bounded context is free from this environmental detail (i.e., how the security control is implemented). Furthermore, if this requirement changes, developers can easily exchange this security control since it affects only the security component, but not the bounded context implementation.

Another example is the protection of sensitive patient data. For example, suppose that some information of a patient (e.g., personally identifiable information, medical data, etc.) needs to be encrypted and authenticated (by law), but some does not. Usually, such diverse data comes from different bounded contexts, which implies that different teams are responsible for it. Therefore, the teams can independently apply different techniques to each bounded context independently in the respective security components to protect their data. For example, sensitive patient data like personal information or protected health information will require strong encryption, authentication, and authorization whereas non-sensitive data may only require a valid authorization of the user of the system.

As Fig. 3 illustrates, the patient management bounded context implements an update method for patients. This update method is a pure domain implementation, which enforces all the business constraints which apply to a patient object. The security controls which relate to updating a patient are implemented in the patient management security component, which accesses the internal web service of the hospital to check whether the employee is a physician, but also implements the proper patient protection techniques like e.g. encryption and authentication. In addition, also pseudo-anonymization or full anonymization can be implemented in a straightforward and easy manner. Finally, the patient repository stores the protected patient object. Observe that in such a case, not even database administrators will have access to patient data, which is in contrast to many other business areas. This ensures the privacy of data throughout its lifetime. Such access is granted only to users of the medical application, particularly those users who really need and are allowed to see it.

Also, the security of user interfaces and services, also referred to as front-ends, for bounded contexts is isolated. Recall that a bounded context implementation is independent of all other concerns and can function in isolation. Particularly, it does not depend on the front-end which visualizes or provides access to the domain information. In such a case, if we deliver the bounded context (e.g., as a web page or a web API), then we can also build the security of the front-end into this component (including, e.g., the validation of input data, communication security by using HTTPS, etc.) without the need to modify or change the bounded context implementation. Interestingly, the bounded context does not even know about this issue, since it has no knowledge about the front-end at all. This also explicitly enables for a simple exchange of front-ends and their security controls as demands change over time.

Health software often needs to communicate with embedded or even implanted medical devices. In this light, our approach using hexagonal architecture with an isolation of security controls in dedicated assemblies provides a clean way to mitigate security risks concerning the interface to such medical devices: The business logic concerning the medical device stays clean in the bounded context implementation. However, the code implementing the communication itself, as well as its security controls, is developed in an isolated, dedicated assembly.

We finally highlight that an architectural approach with independent bounded contexts also leads to fast security incident response times. Precisely, in case of a security breach, development teams easily identify the affected bounded contexts where the breach happened. Due to the isolation of security controls in a dedicated security component it enables them to quickly change and fix the inappropriate security controls.

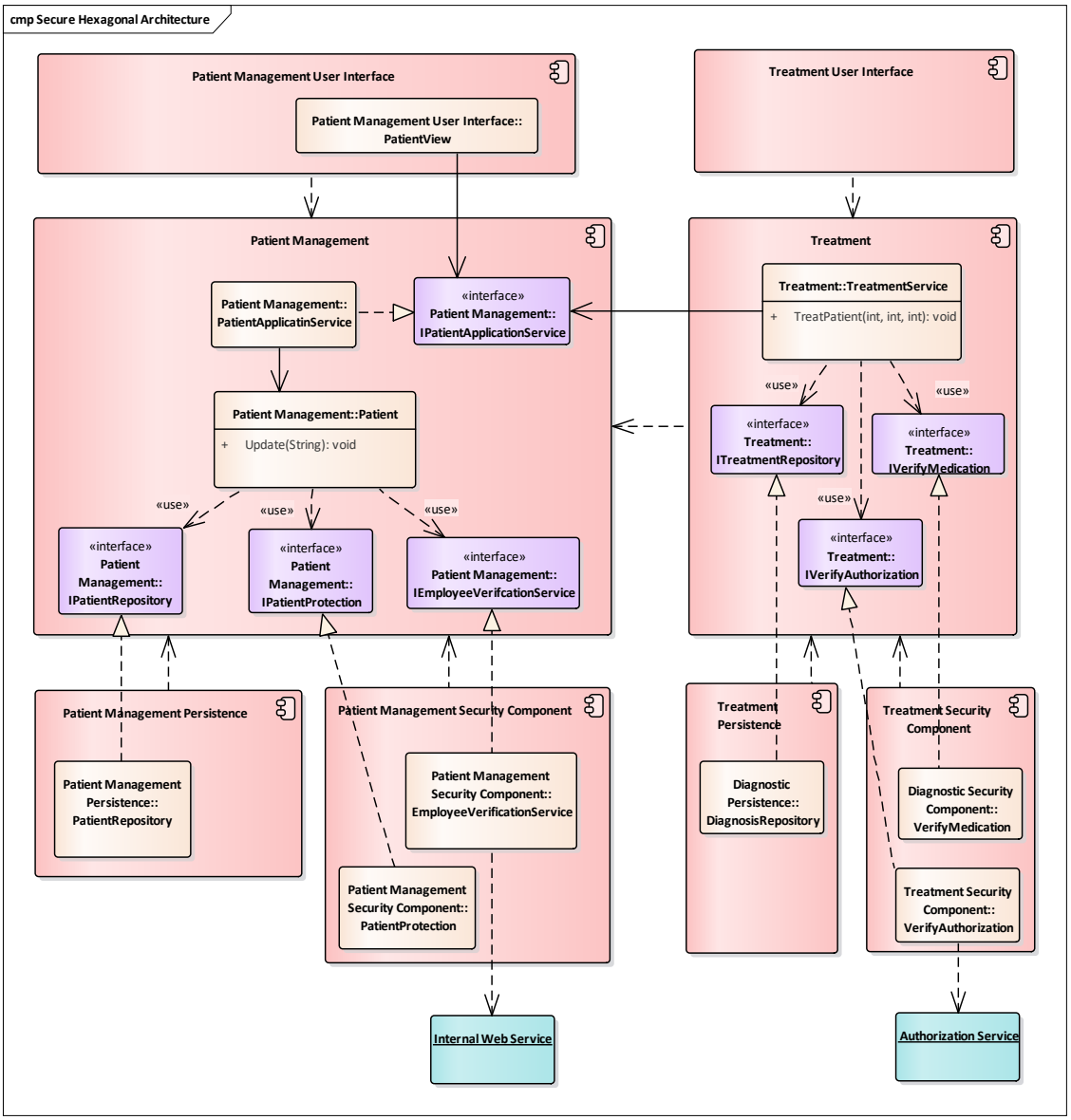

Figure 3. Schematic illustration of security aspects in hexagonal architectures using DDD

4 Agile Practices for Secure Architecture of Health Software

From a business perspective, it is believed that adopting agile practices contributes to the continuous delivery of business value and reduces the risk of developing health software of poor quality. In general, software architectures should meet the requirements, anticipate future user needs, and be able to incorporate new technologies. Furthermore, software architectures should be easy to understand and implement by development teams. Designing software architectures is a complex, creative, and challenging process that requires highly collaborative and self-organizing teams. Designing a secure, change-tolerant architecture of health software is even more challenging. This paper explores which agile practices can be applied to support design of a secure architecture of health software. The agile practices were chosen based on our experience and literature research in Section 2. We believe that these practices could be used for any software development project, not only health software.

4.1 Collaboration with Key Stakeholders

Health software projects can benefit from continuous communication and collaboration with stakeholders. The key stakeholders may include end users, patients, domain experts, healthcare facilities and providers, and manufacturers of medical devices.

Having the domain experts representing the customers involved throughout the project increases the likelihood that we build exactly what the customers want. Close collaboration with the

stakeholders and domain experts is the key to success of a software product when developing according to DDD. This is because all software artefacts, including source code and design documentation, shall abstractly align with and solve the problem domain, and even use the terms and words stemming from the ubiquitous language. But how do we get there? How do development teams achieve this goal?

DDD introduces for this so-called knowledge crunching (Evans, 2003), where developers and architects work out together with domain experts and stakeholders what the software should solve. In other words, during such sessions the development teams explore the problem domain in-depth, trying to understand how the domain works, and what really matters to a domain expert. On the one hand, this reveals crucial insights into the dynamic processes of the domain, and on the other hand it establishes a common language which both parties understand. The outcome of such knowledge crunching sessions are often user stories, which the developers finally implement. When focusing on security in such sessions, and the software architecture is hexagonal, the developers and stakeholders can distil which parts of an implementation or design naturally belong to the domain, and which do not. This ultimately allows for an isolation of certain security controls in dedicated security components.

4.2 Security-related User Stories

Agile teams rely on a prioritized product backlog that contains functional and non-functional user stories and any other items that might be needed in the final product.

According to Cohn (2010), a user story represents a short, simple description of a feature from the user's perspective. User stories have three main parts: 1) who is the user, 2) what is the story about, and 3) what is the desired benefit of doing it. The first point gives a hint to which bounded context the user story may belong. For example, when talking to hospital administrative staff dealing with patient admissions, the user story most probably describes a process in the administration context of a hospital. The second point tells what is happening in the domain, and therefore also what the software shall support. Finally, the third point describes what is the value which we provide to our customer. Observe that a user story essentially describes the domain, however, also the environment needs be considered when implementing a user story. More specifically, when implementing a user story in hexagonal architecture, development teams also must implement the environmental components (persistence, user interface, security, etc.) which the bounded context relies on. In addition, it is important to ask questions concerning the security and privacy of the data or workflows being handled by the user story. This leads directly to security-related user stories, which are some sort "attached" to user stories of the domain itself. Similarly, SAFECode (*SAFECode: Practical Security Stories and Security Tasks for Agile Development Environments*, 2012) developed a list of 36 security-focused stories that can be implemented by agile practitioners "as is" and incorporated into the development process.

4.3 Iterative and Incremental Approach

Waterfall and other early software development life cycle models apply several development phases in a sequential manner, which do not immediately allow to develop a product incrementally. In contrast, agile methodologies explicitly apply an incremental and iterative approach, which is very often driven by user stories, especially in the context of DDD. Specifically, at the start of a development iteration, the development team of a bounded context selects (security-related) user stories one-by-one. When choosing a user story, the team and the architect design and implement the user story, which may also require redesign and refactoring of existing software artefacts. However, this additional effort ensures that the bounded context implementation reflects the current domain structure and its dynamics properly, as every user story carries essentially domain knowledge. Furthermore, each user story extends the knowledge a developer or architect has about the domain, which very often leads to new insights into how the domain is structured and works. Note that such insights may also trigger refactoring effort to reflect the new knowledge, sometimes referred to as "refactoring towards deeper insight" (Evans, 2003). More importantly, this does not only lead to an incremental development of the software (specifically bounded each context). It also enables for an incremental and isolated development of security controls. We clarify these circumstances as follows: by removing security considerations from a bounded context implementation (which at the very end shall only implement business logic), and by implementing such non-functional details (for software which does not primarily belong to a security domain) in environmental components of a bounded context, we have isolated (almost all) security controls of a software product into dedicated security control components, see also Section 3. Since each bounded context is owned by one team and implemented story-by-story, and each story may require (some) security controls, the teams develop security controls incrementally. In summary, this results in many advantages from a stakeholder's and architect's point of view: 1) stakeholders can intervein if the software product evolves into an undesirable or unwanted way, 2) if changes in the business domain occur, such changes easily translate into new user stories, which require less development effort since design and implementation is done incrementally anyway and existing artefacts abstractly align with the problem domain, 3) security controls can easily be changed

without affecting the system as a whole, but they can also be tested (since they reside within dedicated components) in a straightforward and simple way, and 4) the software design and security, from an architect's point of view, grows iteratively and changes as new user stories emerge, which leads to a design which is very robust to changes stemming from the business domain.

The concept of Mob Programming has been introduced at Hunter Industries in 2011. This relatively new agile practice has gained increased attention from developers. While pair programming focuses on two people, Mob Programming considers the whole team. Regular Mob Programming sessions on a weekly (at the beginning of a project) or monthly basis help the team increase their code quality since every team member understands the structure, i.e. the design, of the code base. Additionally, these sessions have a positive impact on implementation of security controls because everyone in team knows where and how the controls are implemented.

4.4 Minimum Viable Architecture

Every software project that delivers only parts of what is needed, slowly, and with poor quality and security is costing companies and customers money. Building a minimum viable architecture allows to adapt to changes through a fast feedback look. Besides that, it reduces scope creep and prevents gold plating. Poort (2014) points out that agility can be achieved by keeping the architecture lightweight, addressing only those concerns that are especially risky or costly. Considering that the architecture should also support subsequent product releases, developing anything in addition to the capabilities needed increases the costs and complexity of the final software product.

It is thought that the minimum viable architecture ensures that only architectural work is done that is necessary and adds value rather than creating a complete architecture up front where even a small change may trigger a complete redeployment. Lean architecture is characterized by a focus on change, lightweight documentation, people, collective planning and cooperation, and end user mental model (Coplien & Bjørnvig, 2010). While classic software architecture includes much implementation (e.g., platforms, libraries) or only the documentation, Coplien & Bjørnvig (2010) highlight that lean architecture defers implementation and delivers lightweight APIs and descriptions of relationships. When developing health software, lean architecture must consider factors affecting patient health and safety such as security and privacy.

4.5 Architecture Planning

A software architect is expected to wear many hats. Leffingwell (2011) compared the architect's responsibilities in a pre-agile and agile context and found out that architect's responsibility shifted to a more collaborative role supporting agile teams.

DDD requires a very close relationship between domain experts on the hand, and developers and architects on the other hand. When we follow the idea of having one development team per bounded context, or one team being responsible for several bounded context, this leads to a broad range of responsibilities of an architect (we just highlight some of them). The architect is responsible for: 1) designing together with development teams bounded contexts, 2) defining the ports of a bounded context to the environment, 3) determining the boundaries of a bounded context, 4) interfacing with key stakeholders in the domain, and 5) making sure that different bounded contexts integrate well and work together to ultimately comprise the overall system. Especially the last responsibility is of high relevance, as it considers the architecture of the whole system, not solely the design of components or the software solution. It is important to bear in mind that security and privacy of health software must be always considered in the context of the larger system where the software is intended to be used.

For architecture planning purposes we suggest introducing one short iteration/sprint at the begin of the project. Afterwards architecture discussions can be continued in regular design meetings and/or as a part of the sprint planning meetings in Scrum.

4.6 Architecture Review

Health software that is secure by design assumes that security is addressed throughout the entire product life cycle including the initiation, design, development, production, distribution, installation, clinical use, maintenance, decommissioning, and disposal. To decrease the risk that something has been overlooked, we suggest reviewing architecture of health software on a regular basis. Such reviews could be scheduled separately or integrated into regular sprint planning meetings in Scrum. Architecture reviews should ensure that the following security topics have been discussed and properly addressed: security and privacy requirements, security-related user stories and their implementation, security controls, ability of the architecture to support software and critical security updates/patches, integration and secure execution of potentially malicious third-party libraries into health software, encryption of sensitive data, no use of a broken cryptographic algorithm, secure data transfer channels, controlled use of cloud services, etc.

4.7 Architecture Retrospective

Architecture retrospective is derived from the sprint retrospective meeting in Scrum. The purpose of the retrospective meeting is to reflect and improve the architecture of health software. We recommend having architecture retrospective meeting once a

month. During this meeting the team should answer the following three questions: 1) what worked well? (provide examples of great collaboration over the month), 2) what did not work well? (focus on what the team can change), and 3) what can be improved? (focus on maximum three items for the next month).

5 Conclusion and Outlook

Advancements in technology have led to a new generation of networked and interconnected medical devices. Such devices open up a new threat landscape and may be vulnerable to cyber-attacks. A reasonable approach to tackle this issue could be security by design where security is built into the development process from the start. This is necessary to reduce the risks that may adversely impact device functionality or delay the delivery of patient care, and lead to patient harm. As more and more companies develop health software with shorter delivery cycles using agile approaches, our research was focused on hexagonal architecture, DDD, and agile practices which could be used to develop products that are secure by design.

We have shown in this paper one possible way to develop health software which is secure by design. We used hexagonal architectures and DDD, which leads to an isolation of business logic (into bounded contexts) and environmental concerns in software applications. This paper has demonstrated, for the first time, that clear separation between business logic and environment can be directly exploited to isolate the implementation of security controls for specific bounded contexts into dedicated security components. This in turn together with a set of agile practices enables for a very tight control of the planning, design, and implementation of security controls. Furthermore, it allows to test, review, analyse or even audit such dedicated security components in a rigorous, independent manner. Finally, Mob Programming sessions on a regular basis for each team implementing a bounded context and its environment synchronize all team members, and make them aware of the choices, implementations, and designs of security controls.

It would be interesting to apply SAFECode's security-focused stories on health software projects and to determine which stories support security architecture. Another possible area of future research would be to investigate further how well health software following our approach combines with embedded systems and if it is straightforward to extend the ideas of this work also to such constrained, small scale systems. Furthermore, it would be interesting to enhance the ideas also to cloud-based technologies and deployments. Finally, it remains an open question, whether there exist other agile techniques which enable for a tight control of the design and implementation of dedicated security components.

Disclaimer

The views and opinions expressed in this paper are those of the individual authors and do not represent the approach, policy, or endorsement of the organization that is currently affiliated with the authors.

References

Alibasa, M. J., Santos, M. R., Glozier, N., Harvey, S. B., & Calvo, R. A. (2017). Designing a Secure Architecture for m-Health Applications. In Proceedings of the 2017 IEEE Life Sciences Conference (LSC) (pp. 91–94). Sydney, Australia.

Asokan, N., Nyman, T., Rattanavipanon, N., Sadeghi, A.-R., & Tsudik, G. (2018). ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11), 2290–2300.

Babar, M. A. (2009). An Exploratory Study of Architectural Practices and Challenges in Using Agile Software Development Approaches. In Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (pp. 81–90). Cambridge, UK.

Chivers, H., Paige, R. F., & Ge, X. (2005). Agile Security Using an Incremental Security Architecture. In Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005) (pp. 57–65). Sheffield, UK.

Cohn, M. (2010). Succeeding with Agile: Software Development Using Scrum. An Arbor: Addison-Wesley.

CollabNet VersionOne: The 12th Annual State of AgileTM Report. (2018). Retrieved from https://explore.versionone.com/state-of-agile/versionone-12th-annual-state-of-agile-report

Coplien, J. O., & Bjørnvig, G. (2010). Lean Architecture: for Agile Software Development. Chichester: John Wiley & Sons.

Derdoura, M., Altib, A., Gasmia, M., & Roosec, P. (2015). Security architecture metamodel for Model Driven security. Journal of Innovation in Digital Ecosystems, 2(1–2), 55–70.

Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley Professional.

FDA. (2014). Content of Premarket Submissions for Management of Cybersecurity in Medical Devices.

Fernandez, E. B., Fonoage, M., VanHilst, M., & Marta, M. (2008). The Secure Three-Tier Architecture Pattern. In Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems (pp. 555–560). Barcelona, Spain. doi:10.1109/CISIS.2008.51

Fontdevila, D., & Salías, M. (2013). Software Architecture in the Agile Life Cycle. ACSIJ Advances in Computer Science: an International Journal, 2(1), 48–52.

Freeform Dynamics: Integrating Security Into the DNA of Your Software Lifecycle. (2018).

Hadar, I., & Sherman, S. (2012). Agile vs. Plan-Driven Perceptions of Software Architecture. In Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering (pp. 50–55). Zurich, Switzerland.

Harkins, M. W. (2016). Managing Risk and Information Security: Protect to Enable (Second Edi.). Berkeley: ApressOpen.

Houser, W. (2014). Static Analysis is not enough: The Role of Architecture and Design in Software Assurance. CrossTalk, The Journal of Defense Software Engineering, 27(6), 27–32.

IEC 82304-1: Health software – Part 1: General requirements for product safety. (2016).

Isham, M. (2008). Agile Architecture IS Possible – You First Have to Believe!. In Proceedings of the Agile 2008 Conference (pp. 484–489). Toronto, Canada. doi:10.1109/Agile.2008.16

Leffingwell, D. (2011). Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Westford: Addison-Wesley.

Manifesto for Agile Software Development. (2001). Retrieved from http://agilemanifesto.org/

McGraw, G. (2006). Software Security: Building Security In. Crawfordsville: Addison-Wesley.

Mekni, M., Buddhavarapu, G., Chinthapatla, S., & Gangula, M. (2018). Software Architectural Design in Agile Environments. Journal of Computer and Communications, 6, 171–189.

Millett, S., & Tune, N. (2015). Patterns, Principles, and Practices of Domain-Driven Design. Indianapolis: John Wiley & Sons.

Moriconi, M., Qian, X., Riemenschneider, R. A., & Gong, L. (1997). Secure Software Architectures. In Proceedings of the 1997 IEEE Symposium on Security and Privacy (pp. 84–93). Oakland, USA.

Nord, R. L., Ozkaya, I., & Sangwan, R. S., (2012). Making Architecture Visible to Improve Flow Management in Lean Software Development. IEEE Software. 29(5), 33–39.

Pauli, J. J., & Xu, D. (2005). Misuse Case-Based Design and Analysis of Secure Software Architecture. In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II (pp. 398–403). Las Vegas, USA.

Pedraza-Garcia, G., Astudillo, H. & Correal, D. (2014). A Methodological Approach to Apply Security Tactics in Software Architecture Design. In Proceedings of the 2014 IEEE Colombian Conference on Communications and Computing (COLCOM) (pp. 1–8). Bogota, Colombia.

Poort, E. R. (2014). Driving Agile Architecting with Cost and Risk. IEEE Software, 31(5), 20–23.

SAFECode: Practical Security Stories and Security Tasks for Agile Development Environments. (2012). Retrieved from https://safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf

Santos, J. C. S., Tarrit, K., & Mirakhorli, M. (2017). A Catalog of Security Architecture Weaknesses. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 220–223). Gothenburg, Sweden.

Sharma, A., Kumar, M., & Agarwal, S. (2015). A Complete Survey on Software Architectural Styles and Patterns. Procedia Computer Science, 70, 16–28.

Sturtevant, D. (2018). Modular Architectures Make You Agile in the Long Run. IEEE Software, 35(1), 104–108.

Tang, X., & Shen, B. (2009). Extending Model Driven Architecture with Software Security Assessment. In Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement (pp. 436–441). Shanghai, China.

Vai, M., Nahill, B., Kramer, J., Geis, M., Utin, D., Whelihan, D., & Khazan, R. (2015). Secure Architecture for Embedded Systems. In Proceedings of the 2015 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1–5). Waltham, USA.

Waterman, M. (2018). Agility, Risk, and Uncertainty, Part 1: Designing an Agile Architecture. IEEE Software, 35(2), 99–101.

Woods, E. (2015). Aligning Architecture Work with Agile Teams. IEEE Software, 32(5), 24–26.