Analysis and Component-based Realization of Security Requirements

Denis Hatebur[1,2] Maritta Heisel[1] Holger Schmidt[1]

[1] University Duisburg-Essen, Faculty of Engineering, Department of Computer Science, Workgroup Software Engineering, Germany, email: {maritta.heisel, denis.hatebur, holger.schmidt}@uni-duisburg-essen.de

[2] ITESYS Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

Abstract

We present a process to develop secure software with an extensive pattern-based security requirements engineering phase. It supports identifying and analyzing conflicts between different security requirements. In the design phase, we proceed by selecting security software components that achieve security requirements. The process enables software developers to systematically identify, analyze, and finally realize security requirements using security software components. We illustrate our approach by a lawyer agency software example.

1 Introduction

Software projects have an increasing demand for *security*. There are various reasons for this situation: many stakeholders act according to a "hunter-gatherer" scheme and pile up data that must be protected against unauthorized access; security measures are often required by law and standards; network-based software is connected over the insecure Internet, etc.

As a consequence, software engineers are not only confronted with functional properties, but with quality requirements concerning security, although they are not experts in *security engineering*. Furthermore, tools, processes, and methods, which are necessary to analyze, design, implement, and test secure software are rare or sometimes nonexistent. Hence, security is "added" to already developed software, or software engineers choose off-the-shelf security components as they are provided by many APIs (application programming interfaces) without thorough requirements analysis, software specification, and architectural as well as fine-grained software design.

In earlier publications (see [6], [7], [8]), we introduced a security engineering process that focuses on the early phases of software development. The basic idea is to make use of special patterns defined for structuring, characterizing, and analyzing problems that occur frequently in security engineering. Similar patterns for functional requirements have been proposed by Michael Jackson [9]. They are called *problem frames*. Accordingly, our patterns are named *security problem frames*. Furthermore, for each of these frames, we define a set of *concretized security problem frames* that take into account generic security mechanisms to prepare the ground for solving a given security problem.

In this paper, we extend our approach by the *analysis of conflicts* between different security requirements, as well as by the usage of *security software components* that represent implementations of security mechanisms. These components support software engineers in developing a secure software architecture.

In the following, we first present Jackson's problem frames as well as security problem frames and concretized security problem frames in Sect. 2. We describe security software components in Sect. 3. In Sect. 4, we introduce our extended security engineering process, and we illustrate our approach by a lawyer agency software example in Sect. 5. Section 6 discusses related work, and we give a summary and discuss perspectives in Sect. 7.

2 Problem Frames for Security Engineering

2.1 Problem Frames

Problem frames are a means to analyze and classify software development problems. They were invented by Michael Jackson [9], who describes them as follows: "A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement." Problem frames are described by *frame diagrams*, which basically consist of rectangles and links between these (see frame diagram in Fig. 1). The task is to construct a *machine* that improves the behavior of the environment it is integrated in.

Plain rectangles denote *application domains* (that already exist), and a rectangle with a single vertical stripe denotes a *designed domain* physically representing some information, a rectangle with a double vertical stripe denotes the machine to be developed. *Requirements* are denoted with a dashed oval. The connecting lines represent

interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain. For example, if a user types a password to log into an IT-system, this is a phenomenon shared by the user and the system, which is controlled by the user. A dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference. Furthermore, Jackson distinguishes *causal* domains that comply with some physical laws, *lexical* domains that are data representations, and *biddable* domains that are usually people.

In the frame diagram depicted in Fig. 1, the "X" indicates that the corresponding domain is a lexical domain, and the "B" indicates a biddable domain. The notation "S!E1" means that the phenomenon of interface *E1* is controlled by the biddable domain *Subject*.

Problem frames greatly support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*.

The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the requirements are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment regardless of how the machine is built.

Requirements describe the environment, the way it should be, after the machine is integrated. In contrast to the requirements, the *specification* of the machine gives an answer to the question: "How should the machine act, so that the system, i.e. the machine together with the environment, fulfills the requirements?" Specifications are descriptions that are sufficient for building the machine. They are implementable requirements. For the correctness of a specification S, it must be demonstrated that S, the facts F, and the assumptions A imply the requirements R: $A \wedge F \wedge S \Rightarrow R$, where $A \wedge F \wedge S$ must be non-contradictory.

2.2 (Concretized) Security Problem Frames

To meet the special demands of software development problems occurring in the area of security engineering, we introduced *security problem frames* (SPF) [6]. SPFs are special kinds of problem frames, which consider *security requirements*. The SPFs we have developed strictly refer to the *problems* concerning security. They do not anticipate a solution. For example, we may require the confidential transmission of data without being obliged to mention encryption, which is a means to achieve confidentiality.

Solving a security problem is achieved by choosing generic security mechanisms (e.g., encryption to keep data confidential), thereby transforming security requirements into *concretized security requirements*. The generic security mechanisms are represented by *concretized security problem frames* (CSPF) The benefit of considering security requirements without reference to potential solutions is the clear separation of problems from their solutions, which leads to a better understanding of the problems and enhances the re-usability of the problem descriptions, since they are completely independent of solution technologies.

Each (C)SPF is described according to the following template (see Fig. 1):

- **Name** The name specifies what kind of security problem is addressed by the frame. It is also specified if it is a security problem frame (SPF) or a concretized security problem frame (CSPF).
- **Frame diagram** This diagram shows the relevant domains and their interfaces, as well as the (concretized) security requirement.
- **Security requirement or concretized security requirement** Here, the security requirement or concretized security requirement to be achieved is stated informally.
- **Declarations** In this section, entities that are necessary for stating the preconditions and postconditions are declared. The entities are implicitly universally quantified. We use the domains and interfaces of the corresponding frame diagram as data types.
- **Preconditions** Conditions are given that must be met by the environment for the frame to be applicable.
- **Postconditions** These conditions are a formal representation of the (concretized) security requirement, i.e., they describe what (concretized) security requirement will be achieved by the machine to be built.
- **Related** Different patterns will often be used in combination. Those frames that are commonly used in combination with the described frame are mentioned here.

The pre- and postconditions are expressed as logical formulas, and they form the basis for the pattern system described in Sect. 4.

2.3 Example: Concretized Security Problem Frame for Accountability by Logging

As an example, we present a CSPF for accountability by logging, which is depicted in Fig. 1. This CSPF represents the generic security mechanism of using *logs* to achieve *accountability*. A realization of this technique is, for example, the Linux logging system.

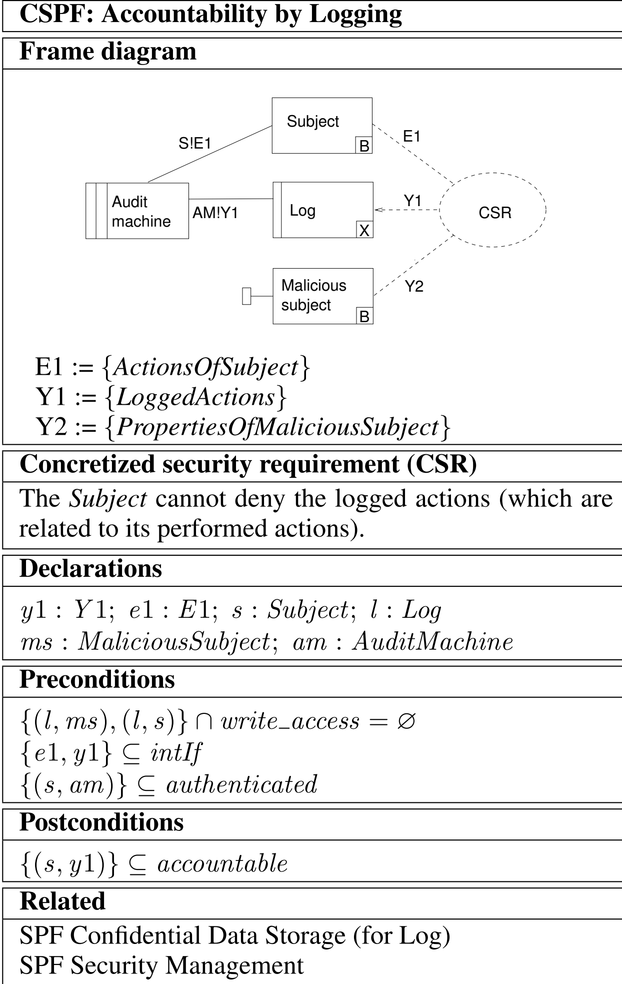

Figure 1. CSPF Accountability by Logging

The frame diagram depicted in Fig. 1 contains a *Subject*, which performs actions using interface *E1*. These actions are captured and possibly filtered by the *Audit machine*, which writes them to a *Log* using interface *Y1*.

The postcondition expresses that all actions performed by *Subject* cannot be denied by *Subject*, which also implies that all actions not performed by *Subject* cannot be assigned to *Subject*. The concretized security requirement *CSR* is stated according to this description.

To achieve this postcondition, we must ensure that only the machine is allowed to modify the *Log*. The first precondition expresses that the *Log* (l) must be protected against modification by *Subject* (s) and *Malicious subject* (ms).

The *Malicious subject* is a potential attacker who may try to change the *Log*. For the *Malicious subject*, we introduce a requirement reference *Y2*, which contains a symbolic phenomenon *PropertiesOfMaliciousSubject* reflecting relevant properties of the *Malicious subject* domain. This domain has no explicit interfaces. Instead, we assume that it possibly has access to all domains (except the machine domain) and interfaces contained in the frame diagram, which we graphically denote using a small rectangle connected by a line to the domain box. Generally, we assume that the machine is not corrupted, and the malicious subject has no direct interface to the machine.

The second precondition describes that data transmitted over the interfaces *E1* ($e1$) and *Y1* ($y1$) should remain unchanged, or a modification by *Malicious subject* is detected. This precondition is necessary, because we must ensure that *Malicious subject* does not modify the *ActionsOfSubject* of interface *E1* as well as the *LoggedActions* of interface *Y1*. Otherwise, the *Log* does not correspond to the actions performed by the *Subject*.

The third precondition expresses that *Subject* (s) should be authentic for the *Audit machine* (am).

Confidential storage of the log is not required for accountability, but it may be relevant if additional privacy concerns should be considered. For example, if it is required to conceal the identities of the subjects whose actions are logged (anonymity of the subjects), the log should be stored confidentially. For that reason, the list of related frames contains the *SPF Confidential Data Storage*. It also contains the *SPF Security Management*, because management functions may be necessary to define actions to be logged, to define the time period for storing a log, etc.

3 Security Software Components

Software components can be used to solve software development problems. We associate a set of software components that solve security problems to each CSPF. There are off-the-shelf components (such as the security software components included in mainstream software development platforms such as .NET and Java) or to tailor-made components that must be developed from scratch. Examples for off-the-shelf security software components are components for encryption / decryption, authentication, and access control.

We denote software components using UML [3] composite structure diagrams. There, components are represented as named boxes. Each component has a number of ports that can be used to connect the component at hand to other components. The ports of two or more components can be connected using a number of interfaces, represented by "sockets" (required interfaces) and "lollipops" (provided interfaces). The name of a port shows the purpose of the connected component.

As an example security software component, we present the component Logging shown in Fig. 2 that realizes the functionality required by the *CSPF Accountability by logging*. An existing implementation of such a component is, e.g., the Linux "syslogd" logging component.

The logging component provides an interface *actions* for a number of other components to log their actions. These components can use the interface to log what actions have been performed, the user (or a pseudonym) performing

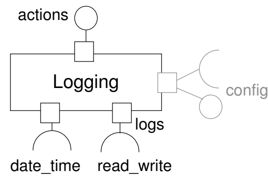

Figure 2. Logging component

them, and possibly some additional descriptions.

The logging component uses an interface *read_write* to access a (possibly remote) storage component using basic read and write operations. The log data contained in the storage component must be protected against modification and unauthorized disclosure using, e.g., an access control component.

The logging component can have a filter mechanism, that selects which of the performed actions are stored in the log. Hence, the logging component can have an interface *config*, which allows another component to configure the filter. If such a functionality is required (see "Related" section in Fig. 1: SPF Security Management), the *config* interface can be implemented as a provided interface (e.g., a GUI) or a required interface (e.g., a configuration file). Since the configuration functionality is optional, the interface *config* in Fig. 2 is depicted in gray color.

To associate time stamps to the performed actions, the logging component can make use of a component providing date and time information using the required interface *date_time*. For example, the Linux "syslogd" logging component provides a filtering mechanisms and time stamps.

4 Security Engineering Process using Patterns

We present in this paper a consolidated variant of a security engineering process based on (C)SPFs [8]. We call this process *SEPP* (Security Engineering Process using Patterns). It consists of the following four steps.

Step 1 – Select and Instantiate SPFs Developing a secure system using (C)SPFs starts after an initial set of security requirements is identified. For these security requirements, compare each elicited security requirement to the informal descriptions of the security requirements of the SPFs contained in our catalog. After an appropriate SPF is determined, instantiate it. When instantiating an SPF, the domains, phenomena, interfaces, pre- and postconditions, and the security requirement must be assigned concrete instantiations.

To instantiate the domains that represent potential attackers, a certain level of skill, equipment, and determination of the potential attacker must be specified. Via these assumptions, threat models are integrated into SEPP.

		1	2	3	4	5	6	7	8	9	10
		SPF Confidential Data Transmission	SPF Confidential Data Storage	SPF Anonymous Transmission	SPF Anonymous Storage	SPF Integrity-preserving Data Transmission	SPF Integrity-preserving Data Storage	SPF Authentication	SPF Accountability	SPF Distributing Secrets	SPF Security Management
1	CSPF Confidential Data Transmission (symmetric)	C/P	P			R	P			P	
2	CSPF Confidential Data Transmission (asymmetric)	C/P	P			R	P			R	P
3	CSPF Confidential Data Storage by Access Control	P	C/P/-				R	P	R		
4	CSPF Confidential Configurable Data Storage by Access Control	P	C/P/-				R	P	R		P
5	CSPF Confidential Data Storage by Encryption	P	C/P				R			P	
6	CSPF Anonymous Transmission		P	C	R			!	!		R
7	CSPF Anonymous Storage		P	P	C		!	!	!		
8	CSPF Integrity-preserving Data Transmission (symmetric)	R	P			C/P	P			P	
9	CSPF Integrity-preserving Data Transmission (asymmetric)	R	P			C/P	P			R	P
10	CSPF Integrity-preserving Data Storage by Access Control		R		!	P	C/P	P	R		
11	CSPF Integrity-preserving Configurable Data Storage by Access Control		R		!	P	C/P	P	R		P
12	CSPF Integrity-preserving Data Storage (Cryptographic authentication code)		P		!	P	C/P			P	
13	CSPF Authentication (static, using secrets)	P	P	!	!	P	P	C		P	
14	CSPF Authentication using unforgeable credentials			!	!		P	C			P
15	CSPF Authentication (dynamic)		P	!	!		P	C		P	
16	CSPF Accountability by Logging		R	!	!		P	P	C		R
17	CSPF Distributing Secrets (trusted path)	P	P			P	P	R	R	C	
18	CSPF Distributing Secrets (negotiation)	P	P			P	P	P		C	
19	CSPF Security Management	R	R			P	P	R	R		C

Table 1. Pattern system of (C)SPFs

Step 2 – Select and Instantiate CSPFs Table 1 shows the relations between a set of important SPFs and some CSPFs. The following substeps make use of these relations, and they are repeated until all instantiated SPFs are considered. Step 2 of SEPP results in a consolidated set of security problems and solution approaches that additionally cover all dependent security problems and corresponding solution approaches, some of which may not have been known initially.

Step 2.1 – Select CSPF To solve a security problem characterized by an instance of an SPF, the process continues with choosing a solution approach. Inspect in Table 1 the column of the SPF under consideration. In this column, rows with CSPF candidates are indicated by the letter 'C'. Choose an appropriate candidate.

For example, for the *SPF Confidential Data Storage* (column 1), we can choose one of the CSPFs in rows 3, 4, and 5.

Step 2.2 – Analyze Conflicts Table 1 supports the analysis of conflicts between the SPF instance and the CSPF candidates. Such a conflict is indicated by the symbol '!'. If this symbol is contained in the row of the chosen CSPF, then a conflict between the SPF instance and the CSPF candidate is possible. Analyze the possible conflict, and determine if it is relevant for the application domain. In the case that it is relevant, resolve the conflict by modifying or prioritizing the requirements.

For example, if actions are logged (*CSPF Accountability by Logging*, row 16 in Table 1), anonymity (columns 3 and 4) cannot be achieved.

Step 2.3 – Instantiate CSPF Instantiate the chosen CSPF with its domains, phenomena, interfaces, pre- and postconditions, and the security requirement. All knowledge from the corresponding SPF instantiation can be reused.

Step 2.4 – Inspect Preconditions Inspect the preconditions of the instantiated CSPF. Two alternatives are possible to guarantee that these preconditions hold: either, they can be *assumed* to hold, or they have to be *established* by instantiating a further SPF, whose postconditions match the preconditions to be established. Such a frame can easily be determined using Table 1 by checking the row of the instantiated CSPF. The SPFs that establish the preconditions of a CSPF are indicated by the letter 'P'.

What assumptions are reasonable depends on the threats the system should be protected against. Assumptions cannot be avoided completely, because otherwise it may be impossible to achieve a security requirement. For example, we must assume that an administrator can distinguish a fake user from an authentic user when creating a user account and providing user name and password.

Only in the case that preconditions *cannot* be assumed to hold, one must instantiate further appropriate SPFs, and the procedure is repeated until all preconditions of all applied CSPFs can be proved or assumed to hold.

For example, the preconditions of the *CSPF Confidential Data Storage by Access Control* (row 3) match the postconditions of three further SPFs:

- *SPF Confidential Data Transmission* (column 1) for the data that should be stored,
- *SPF Confidential Data Storage* (column 2) to ensure that stored data is protected from disclosure, because the *CSPF Confidential Data Storage by Access Control* does not solve the problem if access to the data is possible using another channel, and
- *SPF Authentication* (column 7) to ensure that access is only granted to the authorized user.

CSPF Confidential Configurable Data Storage by Access Control (row 4) additionally needs a configuration that must be protected from modification, and that can only be performed by an authentic user (*(C)SPF Security Management*, column 10, row 19). In contrast, *CSPF Confidential Data Storage by Encryption* (row 5) needs no authentication, but a secret that is distributed in a confidential and integrity-preserving way (column 9).

Step 2.5 – Inspect related SPFs Proceed with checking the "Related" section of the CSPF. There, SPFs that are commonly used in combination with the described frame are mentioned. This information helps to find missing security requirements right at the beginning of the security requirements engineering process. Using Table 1, check the row of the instantiated CSPF. The SPFs that are related to the CSPF at hand are indicated by the letter 'R'.

For example, confidential data transmission in rows 1 and 2 may be useless in some context, if the data can be modified. Hence, *CSPF Integrity-preserving Data Transmission* should be considered, too.

Step 3 – Specify Machine The next step in the software development life-cycle is to derive a specification, which describes the machine and is the starting point for its development. To specify the machine, choose concrete security mechanisms. For example, for encryption, a software developer must select the algorithms and the key lengths according to the assumptions about the attacker.

Step 4 – Develop Software Architecture The software architecture should be built from reusable components as far as possible. To develop the software architecture, instantiate component patterns or use existing components from libraries, APIs, or from the operating system. For most of the CSPFs, corresponding components exist. If no component can be reused to fulfill the requirements assigned to a CSPF, it must be developed from scratch. The components must be connected: a required interface of one component must be connected to a provided interface of another component. If the interfaces do not fit completely, adapters must be developed, e.g., as described in [10]. Additional constraints must be considered: those components that handle security-critical data must preserve the security requirements. For example, user interfaces or protocol components must not leak any information if confidentiality is required, and they must not allow to change information if integrity is required. On the level of software architectures, additional hidden channels has to be considered.

5 Case Study

We now present a case study concerning the secure handling of legal cases. Fernandez et al. [2] identified the following threats (possible attacks) using UML [3] uses cases and activity diagrams:

T1 When a new legal case is started, the client or the responsible lawyer might be impostors.

T2 A lawyer might create a false contract.

Threat	**SPF**	**CSPF**	**Component**
T1	Authentication	... using unforgeable credentials	*environment*
T2, T3, T4, T5, T6, T7	Integrity-preserving Data Storage Accountability	... by Access Control ... by Logging	AccessControl Logging
T8	Accountability	... by Logging	Logging
T9	Integrity-preserving Data Transmission Confidential Data Transmission	... (asymmetric) ... (asymmetric)	*environment* *environment*
precond.	Authentication	... (static, using secrets)	Authentication
precond.	Integrity-preserving Data Storage	... by Access Control	AccessControl
related	Confidential Data Storage	... by Access Control	AccessControl

Table 2. Mitigating threats by SPFs and CSPFs

T3 The client or the external people[1] might give a false deposition.
T4 A lawyer may change a deposition.
T5 A lawyer or a secretary may produce intentionally incorrect precedents, briefs, or costs.
T6 A secretary may produce an increased or decreased bill.
T7 A lawyer may change some aspects of the outcome (of the legal case) to collect a higher fee.
T8 A lawyer can disseminate client or case information for monetary gain.
T9 An external attacker may read / change case information or access client / lawyer communications.

Step 1 – Select and Instantiate SPFs For these threats (possible attacks), we can derive security requirements and select SPFs from our catalog.

Step 1.1 – Select SPFs These requirements can be described using SPFs. As shown in Table 2, SPFs are used to mitigate the threats.

To mitigate *T1*, client and responsible lawyer should authenticate themselves against each other (SPF Authentication). The main concern is to preserve the integrity of contracts *(T2)*, depositions *(T3 / T4)*, precedents, briefs, costs, *(T5)*, bills *(T6)*, outcome *(T7)*, and case information *(T9)*. Only authorized persons (all lawyers, the responsible lawyer, the secretary) should be allowed to modify these documents *(SPF Integrity-preserving Data Storage)*. For example, only lawyers are allowed to create and change contracts. Generally, only lawyers and the secretary are allowed to add or change documents. But the *SPF Integrity-preserving Data Storage* only covers a part of the threats: all lawyers are allowed to create contracts, but they may create (intentionally) a false contract. Such modifications cannot be avoided. But if these actions are logged, malicious modifications can be assigned to persons. As a consequence, this may lead to persons performing no malicious actions and acting carefully. Hence, the *SPF Accountability* should be instantiated for *T2, T3, T4, T5, T6,* and *T7*. Client and case information *(T8)* should be kept confidential. Here, logging of read access to the data can help to identify lawyers who disseminate information *(SPF Accountability)*. Client / lawyer communications *(T9)* should be protected against disclosure *(SPF Confidential Data Transmission)* and modification (*SPF Integrity-preserving Data Transmission*).

[1] the opponent or other people involved in the case

Step 1.2 – Instantiate SPFs In all SPFs the malicious subjects must be instantiated: we must deal with internal and external attackers (malicious subjects). Internal attackers may have sensitive information about the system and the processes, whereas external attackers may have more technical knowledge, but less information about the system and the processes.

Step 2 – Select and Instantiate CSPFs After all threats are covered by security requirements and all necessary SPFs are instantiated, we can select and instantiate the CSPFs.

Step 2.1 – Select CSPFs As shown in Table 2, CSPFs are chosen according to the entries in Table 1. For example, the *SPF Accountability* (Table 1, column 8) can be concretized by *CSPF Accountability by Logging* (Table 1, row 16). For the *SPFs Integrity-preserving Data Storage* (Table 1, column 5), the frames in rows 10, 11, and 12 can be chosen. We choose the *CSPF Integrity-preserving Data Storage by Access Control*, since we need role-based access control, but the access control rules need not to be configured at runtime.

Step 2.2 – Analyze Conflicts In this case study no explicit privacy requirements are derived from the threats. Let us suppose that anonymity is an issue: in such a scenario, accountability and anonymity requirements cannot be achieved at the same time (Table 1, row 16, columns 3, and 4) and a conflict between these two security requirements arises. We can add requirements to resolve this conflict. To ensure a certain level of privacy, only relevant access actions should be logged (e.g., not the time the secretary needs to write a document), the logs should be deleted automatically after a certain time period, and the logs should be stored confidentially. The protection of the logs can be described by the *SPF Confidential Data Storage* (column 2).

Step 2.3 – Instantiate CSPF In the CSPFs, e.g., the concrete access rules, the format, and the actions to be logged must be described. The machine maintaining the data is a server connected via a local network with client computers.

Step 2.4 – Inspect Preconditions The preconditions of the *CSPF Integrity-preserving Data Storage by Access*

Control (Table 1, row 10) require that the integrity of transferred data must be preserved (column 5). This can be assumed to be fulfilled if the used network can only be accessed inside a secured area, where an unknown person cannot connect to it. We assume that there is no direct write access to data (column 6), since the data is stored in a physically protected room. Additionally, the users who want to read data must be authenticated (column 7). Hence, we have to instantiate the *SPF Authentication*.

For *CSPF Accountability by Logging* (Table 1, row 16), users must be authenticated, too (column 7). The audit logs must be protected against modification (*SPF Integrity-preserving Data Storage*, column 6).

The assumptions described in this step are marked with the word *environment* in Table 2, column *Component*, to emphasize that the associated requirements need not to be achieved by the software to be constructed, but by its environment.

Step 2.5 – Inspect Related SPFs For all instantiated CSPFs, we checked if any related frame has to be taken into account. In this scenario, for *CSPF Integrity-preserving Data Storage by Access Control* we decided, that *SPF Confidential Data Storage* should be instantiated for the logs.

Step 1 / 2 – Additional SPFs For all relevant related SPFs and the SPFs in the preconditions that cannot be assumed, steps 1 and 2 are executed again.

First, we instantiate the additional SPFs (step 1) and the selected CSPFs as shown in Table 2:

- *SPF Authentication*, necessary for logging and access control, can be concretized by the *CSPF Authentication (static, using secrets)* (row 13, step 2.1). Again, we assume that a secure communication channel is used to protect against disclosure and modification to prevent replay attacks. The secrets in the frame are instantiated by passwords (step 2.3). Additionally, passwords are distributed by an administrator, stored confidentially and protected against modification (steps 2.4). No further related frames need to be considered.
- For *SPF Integrity-preserving Data Storage* of logs (column 6), we select the *CSPF Integrity-preserving Data Storage by Access Control* (row 10, step 2.1), we check if conflicts may exist (step 2.2), and we instantiate the frame with the log data to be protected (step 2.3). The preconditions and related frames are the same as considered before (steps 2.4 and 2.5).
- The preconditions of the *CSPF Integrity-preserving Data Storage by Access Control* require an integrity-preserving transmission of the data to be stored (*SPF Integrity-preserving Data Transmission*), that direct write access to the data is not possible (*SPF Integrity-preserving Data Storage*), and that users are authenticated (*SPF Authentication*). Integrity-preserving data transmission can be assumed to be fulfilled if the network can only be accessed inside a secured area, where an unknown person cannot connect to the network. We can assume that no direct write access to data is possible, since the data and backups are stored in a physically protected room.

Step 3 – Specify Machine The specification is not presented in detail as it is beyond the scope of this paper.

The authentication of client and lawyer (instantiated *CSPF Authentication using unforgeable credentials*) need not be supported by the machine.

For client / lawyer communications (instantiated *CSPF Confidential Data Transmission (asymmetric), CSPF Integrity-preserving Data Transmission (asymmetric),* and *CSPF Distributing secrets (trusted path)*) separate solutions, such as PGP or S / MIME-certificates can be used.

All other instantiated CSPFs should be implemented by the machine. For these subproblems, the behavior at the external interfaces of the machine must be described. Furthermore, we must describe the authentication functionality, the functions to distribute the secrets for authentication, the access control rules that are necessary for confidentiality and integrity, and the actions to be logged (e.g., read access and write access on all data objects for case handling with the attributes *user, date, time, and object*).

Step 4 – Develop Software Architecture The architecture, shown in Fig. 3 can be developed from reusable components (see Table 2).

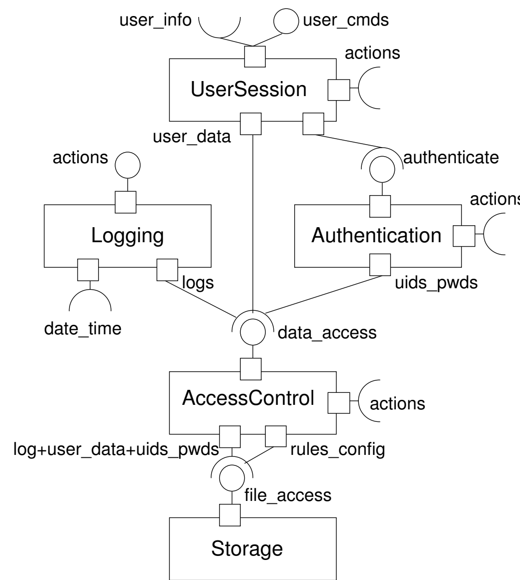

Figure 3. Partial software architecture

A component UserSession should be designed, that is used to enforce authentication, before any other action can be performed. For distributing authentication secrets, no separate component is used. It should be implemented as a part of the UserSession component and using an appropriate configuration of the AccessControl component: only

the administrator should be allowed to create new user accounts. Creating accounts and resetting passwords should be logged using the Logging component. The UserSession component interface provides an interface to the users and the administrator, that can be used to access the user's data after successful authentication. Also, a required interface to the users and the administrator is used to return the requested information if allowed.

Authentication with passwords for users and the administrator (*CSPF Authentication (static, using secrets)*) is implemented in the Authentication component. This component provides the interface *authenticate*, that can be used to check if the supplied pair of user name and password is valid. The result of the check is returned and logged using the interface *actions*.

An AccessControl component is instantiated, that implements the rules for access control according to the instantiated *CSPFs for Confidential Data Storage by Access Control* and *CSPFs for Integrity-preserving Data Storage by Access Control*. The AccessControl component provides an interface *actions*, which is used by the Logging component, the UserSession component, and the Authentication component (for reasons of better readability of the software architecture in Fig. 3, the required and provided interfaces *actions* are not connected to each other). These components send the performed actions, combined with a user id or a component identifier to the Logging component.

A component Logging (see also Fig. 2) is instantiated, that implements the requirements stated in the instances of *CSPF Accountability by Logging*. The Logging component logs the actions performed by the administrator and the users. It provides an interface *actions* to log actions. To create correct time stamps, the component requires an interface to the operating system to retrieve current time and date information. Note that the reliability of this functionality has direct influence to the reliability of the Logging component.

A component Storage should be used, that provides an interface *file_access* to access a database using $read$, $write$, $create$, and $delete$ operations. This interface is used by the AccessControl component to store user data (user_data), user passwords (uid_pwd), logs, and the rules representing the access control configuration (rules_config).

As a result of applying SEPP to the lawyer agency software example, we have identified and analyzed in detail several security problems. Following the dependencies of our pattern system helped us to systematically and completely set up all the necessary subproblems and to resolve conflicts between security requirements. Furthermore, we selected pre-defined security software components, which led us to a software architecture that implements the security requirements.

The next step in the software development life-cycle is to consider fine-grained design and implementation details. As it is beyond the scope of this paper, these issues will not be discussed.

6 Related Work

To elicitate security requirements, the threats to be considered must be analyzed. Lin et al. [11] use the ideas underlying problem frames to define so-called anti-requirements and the corresponding *abuse frames*. The purpose of anti-requirements and abuse frames is to analyze security threats and derive security requirements. Hence, abuse frames and SPFs complement each other.

Gürses et al. [4] present the MSRA (formerly known as CREE) method for multilateral security requirements analysis. Their method concentrates on confidentiality requirements elicitation and employs use cases to treat functional requirements. The MSRA method can be useful to be applied in a phase of the security requirements engineering process that mainly precedes the application of SEPP.

Haley et al. [5] present a framework for security requirements engineering. It defines the notion of security requirements, considers security requirements in an application context, and helps answering the question whether the system can satisfy the security requirements. Their definitions and ideas overlap our approach, but they do not use patterns and they do not give concrete guidance to identify and elicit *all necessary* requirements.

Popp et al. [13] apply extended use cases in the field of security-critical system development. Use cases extended by security information are used to develop the specification of security-critical systems, whereas SEPP focuses on identifying and analyzing requirements beforehand.

Security patterns [14] are applied later, in the phase of detailed design. The relation between our CSPFs, which still express problems, and security patterns is much the same as the relation between problem frames and design patterns: the frames describe problems, whereas the design/security patterns describe solutions on a fairly detailed level of abstraction.

Moreover, there exist other promising approaches to security requirements engineering, such as the agent-oriented secure TROPOS methodology [1, 12], and the goal-driven KAOS approach [15], which are not pattern-based, and which we do not discuss in detail due to lack of space.

We applied SEPP to the lawyer agency software example treated by Fernandez et al. [2]. As a consequence, we identify strengths, weaknesses, and possible synergies of the two methodologies in the following.

The method of Fernandez et al. is based on use case diagrams that represent the functional context of a software development problem. Activity diagrams serve to identify threats. They link solution patterns represented by class diagrams and sequence diagrams to the identified threat in a certain functional context. The class diagrams are equipped with association classes that represent access control policies.

Using activity diagrams to identify threats is helpful to find security requirements, and in our opinion this technique can precede SEPP's first step, namely instantiating SPF to

identify and represent *initial* security requirements. Afterwards, dependent security requirements are detected based on our pattern system.

Compared to use case diagrams, SPF instances contain more information about the environment and the software to be constructed (e.g., they explicitly state data items to be protected and potential attackers). We believe that a comprehensive description of the environment is a key feature for high-quality security engineering. Therefore, using SPFs can be regarded as a strength of SEPP. The solution patterns proposed by Fernandez et al. can be compared with our CSPFs and security software components as presented in Sects. 2 and 3. The access control policies contained in the class diagrams are represented by the postconditions of CSPF instances. In summary, we believe that SEPP can partly be combined with the methods proposed by Fernandez et al. [2] as well as with Gürses et al.'s approach MSRA [4].

7 Conclusions and Future Work

In this paper, we have presented a consolidated and extended variant of SEPP, a process that supports developing secure software in a systematic way. SEPP is a pattern- and component-driven process, which includes an extensive security requirements engineering phase. Security requirements are identified, captured, and analyzed using SPF patterns. Afterwards, solution approaches are determined and described using the more detailed pattern type CSPF.

We have extended the security requirements engineering phase by support for analyzing conflicts between different security requirements. This is achieved by contrasting the SPFs and CSPFs, and then identifying possible conflicts between them. We made these conflicts explicit in the form of entries in a table, which can be consulted when applying SEPP.

Furthermore, we have added a further development step to SEPP that makes use of security software components. These components are related to CSPFs, and they represent implementations of security mechanisms. The components implicitly lead software developers to a secure software architecture.

In the future, we plan to elaborate more on the later phases of software development. For example, we want to investigate in more detail how to design a software architecture in the development process. Additionally, we plan to provide tool support for our SEPP.

References

[1] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers Volume 8, Issue 3:203 – 236, May 2004.

[2] E. B. Fernandez, D. L. la Red M., J. Forneron, V. E. Uribe, and G. Rodriguez. A secure analysis pattern for handling legal cases. to be published in: SugarLoafPLoP'2007 - 6th Latin America Conference on Pattern Languages of Programming, `http://sugarloafplop.dsc.upe.br/wwD.zip`, 2007.

[3] U. R. T. Force. *OMG Unified Modeling Language: Superstructure*, August 2005. `http://www.uml.org`.

[4] S. F. Gürses, J. H. Jahnke, C. Obry, A. Onabajo, T. Santen, and M. Price. Eliciting confidentiality requirements in practice. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 101–116. IBM Press, 2005.

[5] C. B. Haley, J. D. Moffett, R. Laney, and B. Nuseibeh. A framework for security requirements engineering. In *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*, pages 35–42, New York, NY, USA, 2006. ACM Press.

[6] D. Hatebur, M. Heisel, and H. Schmidt. Security engineering using problem frames. In G. Müller, editor, *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, LNCS 3995, pages 238–253. Springer-Verlag, 2006.

[7] D. Hatebur, M. Heisel, and H. Schmidt. A pattern system for security requirements engineering. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, pages 356–365. IEEE, 2007.

[8] D. Hatebur, M. Heisel, and H. Schmidt. A security engineering process based on patterns. In *Proceedings of the International Workshop on Secure Systems Methodologies using Patterns (SPatterns)*, pages 734–738. IEEE, 2007.

[9] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[10] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières. Enhancing dependability of component-based systems. In N. Abdennadher and F. Kordon, editors, *Reliable Software Technologies – Ada Europe 2007*, LNCS 4498, pages 41–54. Springer-Verlag, 2007.

[11] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings of 11th IEEE International Requirements Engineering Conference (RE'03)*, pages 371–372, 2003. Poster Paper.

[12] H. Mouratidis and P. Giorgini. Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific 17(2):285–309, 2007.

[13] G. Popp, J. Jürjens, G. Wimmel, and R. Breu. Security-critical system development with extended use cases. In *APSEC*, pages 478–487, 2003.

[14] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley & Sons, 2005.

[15] A. van Lamsweerde, S. Brohez, R. De Landtsheer, and D. Janssens. From system goals to intruder anti-goals: Attack generation and resolution for security requirements engineering. In *Proceedings of the RE'03 Workshop on Requirements for High Assurance Systems (RHAS)*, pages 49–56, September 2003.