Maintaining Security Requirements of Software Systems using Evolving Crosscutting Dependencies

Saad Bin Saleem[1] Lionel Montrieux[1] Yijun Yu[1] Thein Than Tun[1] and Bashar Nuseibeh[1, 2]

[1]Centre for Research in Computing, The Open University, United Kingdom

[2]Lero – The Irish Software Engineering Research Centre, Ireland

Abstract

Security requirements are concerned with protecting assets of a system from harm. Implemented as code aspects to weave protection mechanisms into the system, security requirements need to be validated when changes are made to the programs during system evolution. However, it was not clear for developers whether existing validation procedures such as test cases are sufficient for security and when the implemented aspects need to adapt. In this chapter, we propose an approach for detecting any change to the satisfaction of security requirements in three steps: (1) identify the asset variables in the systems that are only accessed by a join-point method; (2) trace these asset variables to identify both control and data dependencies between the non-aspect and aspect functions; and (3) update the test cases according to implementation of these dependencies to strengthen the protection when a change happens. These steps are illustrated by a case study of a meeting scheduling system where security is a critical concern.

1. Introduction

Security requirements are about protecting assets of a system from the harms caused by malicious attackers [1]. As one of well-known crosscutting concerns, changes to security implementation can often lead to failures to satisfy other requirements in the system. Implementing security requirements as security aspects could help modularise the protection mechanisms that would otherwise clutter the non-security functions of the system [1]. However, any part of the system including both aspect and non-aspect functions can change, making it difficult to maintain the satisfaction of security requirements. Here we use aspect functions to refer the advising functions inside the aspect and we use the join-point function to refer to where the aspect is weaved.

An example is shown in Figure 1: the class diagram of a secure meeting scheduler system. In this system, the `Permissions` aspect implements a security requirement. The purpose of this security requirement is to ensure that only the authorised users could be allowed to view/edit relevant information about the meeting rooms. The aspect crosscuts the join-point methods `bookMeeting` in the `Meeting` class and `showAvailableRoom` in the `Room` class. Through a test case, the satisfaction of this security requirement is checked that only an authorised user can book a meeting or list available room(s), and an unauthorised user cannot do so. When change happened to any function, usually one has to update the test case in order to check the satisfaction of the same security requirement. The difficulty here is that such an update could be *ad hoc* because most of the time the security requirements are stated at a higher level of abstraction and are tested implicitly by the test cases

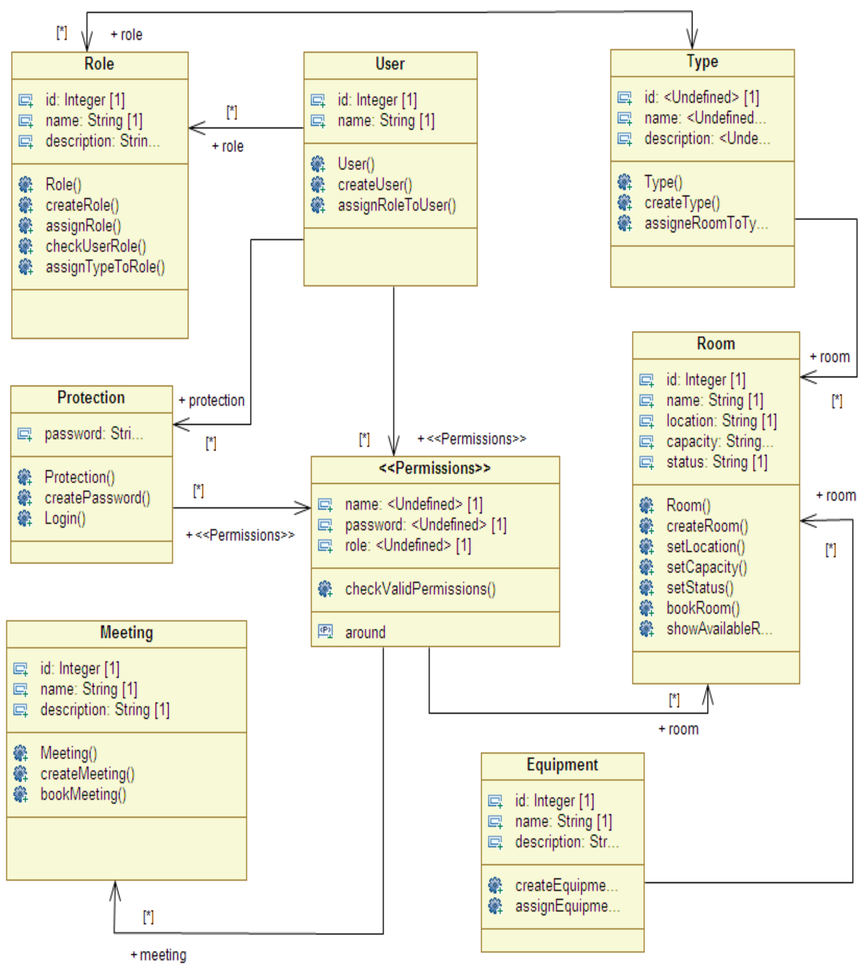

Figure 1: The class model of a Secure Meeting Scheduler system

In this chapter, we propose an approach for detecting any change to the satisfaction of security requirements in three steps: (1) identify the asset variables in the systems that are only accessed by a join-point method; (2) trace these asset variables to identify both control and data dependencies between the non-aspect and the aspect functions; (3) update the test cases according to these dependencies to strengthen the protection when a change happens. An *asset variable* refers to a variable in the code that is being protected by the security aspect. For example, `Equipment.name and Room.name` are asset variables in the meeting scheduler system. The main contributions of this work are two-folds:

1. We explicitly represent the security requirement as assets' protections in concrete test obligations or test cases;
2. We update such test obligations by making use of the change impact analysis on the assets variables and protection conditions.

The proposed steps will be illustrated by a small case study of a meeting scheduling system where security is an important concern. The remainder of the chapter is organised as follows. Section 2 discusses related work and background before explaining our approach in detail. Section 3 presents our dependency analysis framework that focuses on the satisfaction of security requirements before and after the code has changed. Section 4 details the application of this framework to a case study that exposes interesting research challenges. Finally, we conclude in Section 5 with an outlook for the future research directions.

2. Background

Before explaining our framework, we first discuss some related work in the area of aspect-oriented requirements (early aspects), evolving security requirements and dependency analysis.

2.1 Aspect-oriented and evolving security requirements

Rashid et al [2] proposed the concept of "early aspects" as a form of crosscutting concerns in requirements. The notion of eliciting or identifying early aspectual requirements seeks to prevent the introduction of crosscutting concerns at a later design and implementation stage. There are many possible definitions of security requirements as discussed by Mellado et al [3]. However, in this work we adopt the same definition of *security aspects* as in [1], where it is defined as the way to protect the assets of the system because of cross cutting relationship between the threat descriptions and functional requirements. Here threat descriptions are the set of concerns to define relationship between the threats and system objects. Similarly, we adopted the definition of *security requirements* as in [4], which is defined as the way to protect important assets of the system from the harm caused by the malicious attacker. As we consider the requirements and aspects in the problem world rather than in the solution world, therefore we adopted the Haley's definitions. However in this work our focus is on the change impact analysis with respect to implementation of the security aspects.

Security-related code tends to be scattered throughout many classes, and structurally separating security concerns, e.g. access control, was a challenge until the proposal of Aspect Oriented Programing (AOP) paradigm. There are many approaches proposed to solve the problem of separating security concerns using Object Oriented Programming (OOP) paradigm that includes modularizing different concerns. However, the problem of when and where to call a security mechanism was not satisfactorily solved using the OOP paradigm. A security implementation often calls other modules inside the system. The internal encryption mechanism is an example of such a system, where key selection depends on the communication channel. Therefore, it is hard to update all the system calls in response of change requests when lines of code are in the thousands. The AOP is a solution to this problem that not only specifies the behaviour of a specific concern but also binds the relevant applications. Win et al [5] conducted case studies on a Personal Information Management System (PIM) and FTP-server to implement security requirements using aspect technology. They have confirmed that implementing security using AOP is useful to explicitly separate the security logic from the application logic. In this way, it is much easier for developer to take care of the applications part and security experts to check that whether a security policy is correctly implemented. Second, separating module (aspect), and binding (point-cut) helps to cope with unanticipated changes. The aspect and point-cut are concepts used in the AspectJ, which is a Java extension of the AOP. A similar study has been conducted by the Viega et al [6] to check the benefits of writing secure code using AOP extension of the C-programming. They also reported that using AOP is very useful to design security into the application without mixing the security and application logic. Hence, the existing literature supports aspectual implementation of security is better than the non-aspectual implementation. Therefore, based on this fact we are using aspectual implementation of security in our study.

Haley et al described functional requirements and threat descriptions as two types of concerns to derive security requirements using aspect oriented techniques [1]. The functional requirements are set of concerns those help in understanding the different objects in the system to perform an operation. On the other hand, threat descriptions describe relationships between threats and objects. In case of the security requirement, the objects are assets those need to be protected. In this study, we are using the term asset variable to refer asset objects because assets are implemented as variables in the system code. Second, in this study we are doing change impact analysis at the code level rather than at the design level. Therefore, it makes more sense to call these asset implementations as asset variables. We named the locations of system where an object (asset) is implemented and a security mechanism is called to protect this asset as join-point functions. Similarly, we name the functions where assets and security mechanisms are not called as non-join points functions. This convention of name is in-line with the Haley's definition that join points are locations where objects are shared among functional requirements and threat descriptions.

Many forms of early aspects models have been proposed, including goals [10] and problem frames [1], the targeted requirements are goal or problem-oriented models. Yu et al [7] identify aspectual requirements from soft-goals, on the other hand Haley et al [1] identify them using problem frames. It is worth noting that most early aspects frameworks treat security requirements as one of the non-functional requirements. However, Haley et al [1] define the point-cuts of security aspects specifically based on the definition of security requirements [4], i.e., *protecting assets from harms of malicious attacks*. They also suggested that security requirements can be expressed as trust assumption made by the domain expert about security of the system. Recently Franqueira et al [8] have extend the security requirements argumentation process of Haley et al [4] to strengthen the trust assumptions by conducting risks assessment.

The idea of tracing and validating security aspects using requirements-driven approach is not new by itself, as Niu et al [9] have demonstrated the feasibility to support the whole aspect development lifecycle using the goal-oriented approach. However, our work reported here is different in that we focus on change impact analysis of security aspects. Unlike the runtime-monitoring framework proposed by Salifu et al [10], in this work we concentrate on the analysis at the development time. Nhlabatsi et al [11] survey the literature on security requirements and software evolution, where the *management of evolving security requirements* is identified as an outstanding research issue.

2.2 Existing security dependency analysis frameworks

The term *dependency* refers to a relationship among different elements of a program or between two different programs relying on each other to perform a particular task. Such dependencies play an important role in program execution and are classified into data and control dependencies [12]. A data dependency exists when the output of a program becomes the input for another program. On the other hand, a control dependency is related to the ordering and conditions of the execution of the program. Ferrante et al., have introduced the Program Dependency Graph (PDG) to represent the relationship between different programs based on the data and control dependencies [13]. Similarly, Pugh have proposed an approach to remove false data dependencies to prevent program transformation [14] and later improved the approach by using integer programming [15]. Both of the proposed approaches aim to improve the program understanding, for instance, when analysing changes to the programs. A *security requirement dependency* is defined as a relationship between the aspect and non-aspect functions to protect an asset variable of a program.

In the field of network security, Yau and Zhang [16] refer network security dependency as a relationship between two nodes in the network when a program or service intruded by an attacker in one node helps to attack the other node. Therefore, they consider that it is important to identify such security dependency relationships among all the nodes. Johansson [17] have introduced that security dependency exists between two nodes of a network when they depend each other for their security. He categorises them into acceptable and unacceptable dependencies: an acceptable dependency means that a less sensitive system of a network depends on the more sensitive system for its security, an unacceptable dependency is referred as a relationship when the more sensitive system depends on the less sensitive system for its security. For example, it is acceptable, if a workstation depends on a domain controller for its security. A domain controller depending on a workstation for its security is unacceptable. However, these works are not at the program level; therefore they are not directly applicable to the scope of security program aspects. However, all these works stressed the need to manage program dependencies, which is also true for security implementation to avoid the risk of attack on the entire system in case of attack on one vulnerable module. Therefore managing security dependencies at the program level is equally important to minimize the risk of system wide attacks.

To the best of our knowledge, there is still a need to perform dependency analysis of evolving security requirements that are implemented using security aspects.

3. A Dependency Analysis Framework for Security Aspects

An overview of our proposed framework is presented in Figure 2, showing inputs, outputs and the three steps.

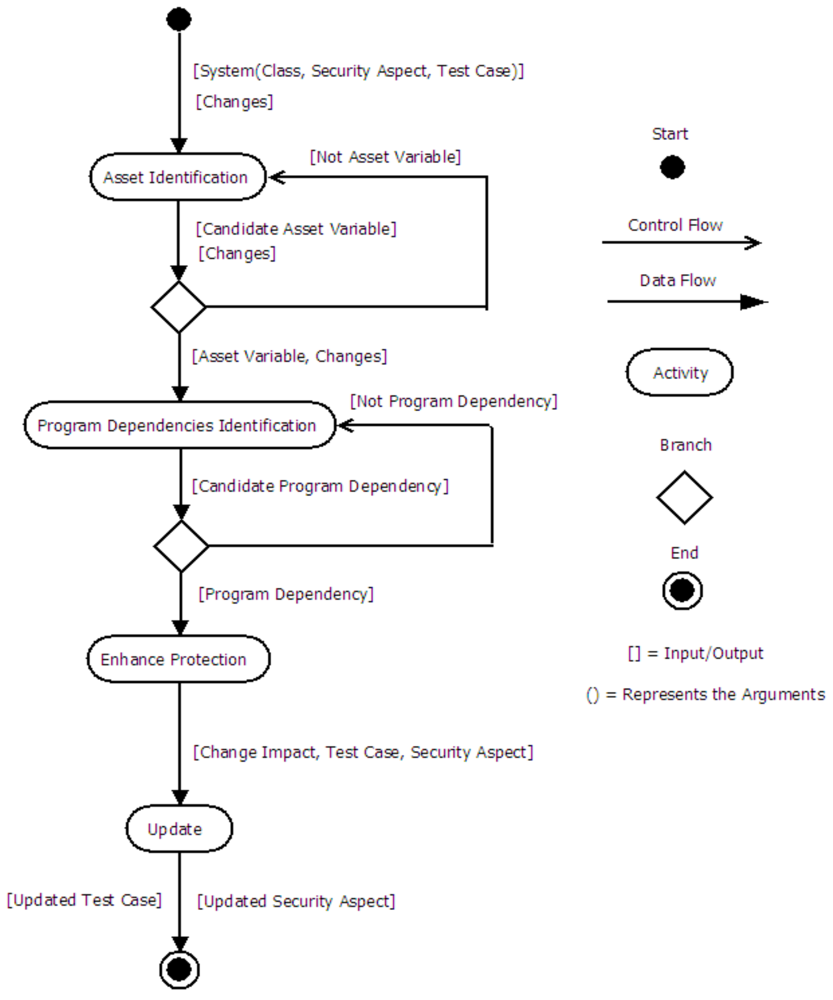
[System(Class, Security Aspect, Test Case)]
[Changes]
Asset Identification
[Not Asset Variable]
[Candidate Asset Variable]
[Changes]
[Asset Variable, Changes]
Program Dependencies Identification
[Not Program Dependency]
[Candidate Program Dependency]
[Program Dependency]
Enhance Protection
[Change Impact, Test Case, Security Aspect]
Update
[Updated Test Case]
[Updated Security Aspect]
Start
Control Flow
Data Flow
Activity
Branch
End
[] = Input/Output
() = Represents the Arguments

Figure 2: An overview of our SD framework in three steps

The inputs to the SDF include (1) a system that already has the security requirements implemented as a list of security aspects, (2) a set of test cases that check the satisfaction of security requirements; (3) a set of changes to the implementation of the system. The outputs from the SDF include both (1) a set of updated test cases, and (2) a list of updated security aspects that may enhance the protections.

Specifically, the framework can be seen as three consecutive steps: initially (1) the concrete assets to be protected are identified from the differences between join-point and non-join-point functions. Without weaving the protection into the join-points, one may assume that certain assets are unprotected. Therefore their identification can be helped by existing join-point control-flows; (2) from these identified asset variables, program slicing [18] can be performed to obtain the control and data dependencies that may leads to the unwanted exposure of these asset variables due to the changes to the functions; and finally (3) from these analysed dependencies, the test cases and security aspects are inspected to check whether the exposed asset values are covered by the new test obligations or by extending the scope of protection through an updated point-cut.

```
public class Employee {
        public  String person;
        public int age;
        public int salary;
        public Employee(String person) {
                this.person= person;
                this.age = 0;
                this.salary =0;
        }

        public int getSalary() {
                if(age<60){
                        salary=100000 + 5000 * (age - 60);
                } else if(age>=60){
                        salary=100000;
                }
                return salary;
        }
}
public class User {
        public String userName;
        public boolean authorized;

        public User(String userName, boolean authorized) {
                this.userName = userName;
                this.authorized = authorized;
        }

        public int obtainSalary(Employee person) {
                return person.getSalary();
        }

        boolean hasPermissions() {
                return authorized;
        }
}
public aspect CheckPermission {
        pointcut p(): (call(public int Employee.getSalary())
        &&args());

        int around(): p() {
                if (((User)thisJoinPoint.getThis()).hasPermissions())
        {
                return proceed();
                } else {
                return -1;
                }
        }
}
```

Figure 3: Listings of the running example: the `Employee` and `User` classes and the `CheckPermission` aspect

The detail of each step is demonstrated through a running example consisting of three classes (i.e., `Employee`, `User` and `Role`), one aspect (i.e., `CheckPermission`) and one test case (i.e., `SRTestCase`).

The implementation of all these classes is shown in Figure 3. The class `Employee` has instance variables `age`, `salary` and a method `getSalary()` to get the value of the salary according to person's age. Similarly, the class `User` has two variables `userName, authorized` and two methods `access (), has Permissions ().` The aspect `CheckPermission` is implemented to run whenever a `getSalary()` function is called; the advice part of the aspect runs around the `getSalary`() function call and returns the control to calling methods when the user has the right permissions as indicated by the truth value returned from the implemented `hasPermissions()` function. Otherwise, the advice code aborts the further execution of the program.

In this running example, the security requirement to be maintained is "*to protect the salary from the unauthorised users to view or to change*". The test case in Figure 4 checks whether the security requirement is correctly implemented.

```
import static org.junit.Assert.*;
import org.junit.Test;
public class SRTestCase {

        @Test
        public void testSecurity() {
                User user = new User("Saad", false);
                assertEquals(-1, user.obtainSalary(new Employ-
                ee("Lionel")));
                user.authorized = true;
                assertTrue(-1 != user.obtainSalary(new Employ-
                ee("Lionel")));
        }
}
```

Figure 4: The security implementation validated as a unit test

The first type of change considered here is (C1) which is due to introduction of a new requirement in the program. For example the system context is changed and now salary is calculated based on the new retirement `age` "65". This change should be reflected in the program by modifying `age` constant variable from "60" to "65". The second type of change (C2) is about updating existing implementation of the system. For example, the `setAge()` method in `Employee` class needs to be updated to reflect salary calculation according to the new age. The third type of change (C3) is about a scenario when a new security mechanism is introduced in the system. For example now the system security is checked using the direct access control (by user permission) to the indirect role-based access control (by user-role-permission). Given these inputs, one can analyse whether these changes could lead to the security requirement not being satisfied anymore.

The first step of our proposed approach is applied as follows. The join point being protected by the advice in the `CheckPermission` aspect is `Employee.getSalary()` whereby `salary` is identified as an asset variable to be protected from viewing or changing by unauthorized users, e.g., when it is called by the `obtainSalary()` method.

After identifying the asset variable, in the second step we analyse both the data and control dependencies: `Employee.age` and `User.authorized` are found between the aspect (`User.obtainSalary`, `CheckPermisson.around`) and non-aspect functions (`Employee.getSalary`, `User.hasPermissions`) respectively. The guard condition for the `getSalary`() computes `User.authorized`, i.e., a control dependency; the computation of `salary` itself makes use of the variable `Employee.age`, thus a data dependency.

In the third step, we need to combine the program dependencies with the following proposed changes to tell whether there is a need to update the security aspect.

3.1 Change due to a new system requirement (C1)

For the change of the retirement age (see the underlined parts in Figure 5), a change to the computation of the asset `Employee.getSalary()` is detected because of the data dependencies. Although it has to do with *integrity*, however, this change will not be detected as a threat to the asset for malicious access (*authorisation*). Therefore there is no need to update the `CheckPermission` aspect or the corresponding `SRTestCase` test case.

```
public class Employee {

        public int getSalary() {
                if(age<65){
                salary=100000 + 5000 * (age - 65);
                } else if(age>=65){
                salary=100000;
                }
                return salary;
        }
}
```

Figure 5: C1: Change to the `Employee` class

3.2 Change due to an update of existing implementation (C2)

On the other hand, if a user has the permission to modify the `Employee.age`, as suggested by adding a method to update `Employee.age`, e.g., by using the `setAge()` method in Figure 6.

```
/* New function setAge is added to the Employee class*/
```

```
public class Employee {
        public void setAge(int ageArg){
                this.age=ageArg;

        }
}

public class User {
        public String userName;
        ~~public boolean authorized;~~
        Role userrole;

        public User(String userName, Role userRole) {
                this.userName = userName;
                this.userrole=userRole;
        }

        public int obtainSalary(Employee person) {
                person.setAge(40);
                return person.getSalary();
        }



}
```

Figure 6: C2: Change to the `User` class

In this case, although the `Employee` class is not changed, it is mandatory to protect the `Employee.age` by the same level of permissions. Therefore it is required to update the pointcut expression in the `CheckPermission` aspect as follows:

```
pointcut p():(call(public * Employee.getSalary() || public * Employee.setAge())&& args());
```

3.3 Change due to a new security mechanism (C3)

The third example change introduces role-based access control into the system by modifying the `User.hasPermission()` method, as shown in Figure 7.

```
public class Role {

        public String role;
        public boolean authorized;

                public Role(String role, boolean authorized) {
                        this.role=role;
                        this.authorized=authorized;
                }
                boolean hasPermissions(){
                        return this.authorized;
                }
}

public class User {
        boolean hasPermissions() {
                return userrole.hasPermissions();
        }
}
```

Figure 7: C3: Change according to role-based access control mechanism

```
public class SRTestCase {

        @Test
        public void testSecurity() {
                Role student = new Role("Student", false);
                User userRightRole = new User("Saad",student);
                assertEquals(-1, userRightRole.obtainSalary(new Em-
                ployee("Lionel")));
                Role professor = new Role("Professor", true);
                User userWrongRole = new User("Saad",professor);
                assertTrue(-1 != userWrongRole.obtainSalary(new Em-
                ployee("Lionel")));

        }

}
```

Figure 8: C3: change to the test case is required

Here, the control dependency to `hasPermissions`() has changed because the system is now giving access based on the user's role. Although this change does not influence the interface between the `CheckPermission` aspect and the join points, it requires the test case `SRTestCase` to check the satisfaction of security requirement differently, as shown in Figure 8.

In summary, we have shown that analysis of the data dependencies on the asset variables and the control dependencies on the protection condition results in the identification of changes in the security aspects or the security test cases. In the following section, we show an application of the methodology to a case study in order to discuss some general issues.

4. Application to the Meeting Scheduler system

Since the common case study does not provide source code, we could not use it for our proposed approach. To illustrate our approach, we have used the Meeting Scheduler system exemplar case study, extended with the security requirements. This case study is selected because of simplicity and sufficiency to represent the problem [19]. We handle the Meeting Scheduling problem for the members of the Computing department of the Open University based the Jennie Lee Building (JLB). It involves the physical and social contexts of the members of the department. The roles of the people include faculty members, fulltime PhD students, secretaries, course team members, tutors and research fellows, etc. For simplicity, in this study only these regular roles of stakeholders are considered to interact with the system, other roles such as visitors are not considered. Since laptops and USB keys are assets in the meeting rooms of the building, security measures have been taken to protect the open-plan areas and the meeting rooms. Great care has been taken to ensure that the measures do not hinder people performing their jobs. By looking at the meeting scheduler problem in JLB and interviewing the stakeholders inside the building, one can construct a secure meeting scheduling system.

Figure 1shows a simplified class diagram of the secure Meeting Scheduler system design. The `Protection` class is part of the `Permissions` aspect in the design, which crosscuts the `showAvailableRoom` and `createRoom` join-point functions. Both functions share a common asset variable: the information of available meeting rooms inside the building (`Room.roomList`). In this example, the `Permissions` aspect verifies that the users have the right permissions before they access any information relevant to the meeting rooms. Any change to the advising function `checkValidPermisions`, or to the join-points functions must be inspected against the implementation of the security requirement. The security requirement is to protect the information of available rooms from the access of unauthorised potentially malicious attackers. Otherwise, either the system would not function anymore or all the users would have permissions to access relevant information about meetings rooms.
Suppose the `Permissions` aspect is originally implemented by checking the username and password against a predefined list of access control (`User.userList`). A change (C''1) has happened such that not only the available meeting room, but also valuable equipment such as projectors are considered as the assets (`Equipment.equipmentList`).
Another change (C''2) is to do with introducing Role Based Access Control (RBAC) (`User.roleList`) to the users. To analyse whether the current implementation can still satisfy the security requirement or not, we applied the dependency analysis methodology as follows.

First we identified the asset variables `Room.roomList and Equipment.equipmentList` from the join-point functions `showAvailableRoom, createRoom` and `createEquipment`. In the second step, we identified data dependencies reading room information between `Room.readRoom` and `Equipment.readEquipment`. In this case, the equipment is inside a room, therefore we always need to read the room information to know the whereabouts of the equipment. Similarly, we identified the control dependency between `User.userList` and `User.roleList` because assignment of a role to a user depends on the condition that the user must be valid. In this way, system always checks the validity of the user before checking his/her role. These dependencies could be exploited by malicious attackers when change happened to the asset variables. In this case study, the `Equipment.name`, `Room.name` and `Meeting.name` are classified as asset variables.
After the proposed change C''1, `createEquipment` was added into the system, making it necessary to include it into the scope of protection as well. The original implementation of the `Permissions` aspect worked perfectly for both `cre-`

ateEquipment and createRoom functions. However, when C''2 happened, it was found that the security requirement for the showAvailableRoom function was not fulfilled. The reason is that different users now have different roles and the Permissions aspect did not consider the roles of users while giving access to the system. It means the Permissions aspect still relies on the control dependency User.userList instead of User.roleList in case of change from *access list* to *RBAC* protection mechanism. This change to the protection mechanism is not reflected in the system; therefore the control dependency User.userList leads to the exposure of asset variables Room.roomList and Equipment.equipmentList. In an initial attempt, function call to check the roles of user was made inside the showAvailableRoom join-point function. However, the change was not effective because createEquipment and createRoom are not directly involved with RBAC.
After this change, however, the system did not show available rooms and still the system security requirement is not satisfied according to security requirement test-case. Actually, still the change was made to only the join-point function showAvailableRoom. To respond to the change for satisfying the security requirement, the advising function of the Permissions aspect and the other join-points should be changed as well. Not only the join-point and advising function but also this change in security requirement should be tested by a new unit test case. Later to reflect the change, we removed the RBAC check from inside the showAvailableRoom function, and updated the Permissions aspect with the RBAC check instead. In this way, the new check in permission aspect depends on the User.roleList variable instead of depending on User.userList. Similarly, the security requirement's test-case is also updated by checking the permissions based on the role rather than based on the users.

5. Conclusion

In this chapter, we have illustrated the need for a systematic approach to handle changes made to security-critical programs, using a running example and our meeting scheduler case study. There are two main observations: (1) Change can happen to any part of the system, including both the aspect and non-aspect part of the implementation. When a change happens, the validation procedure (test cases) for the security requirements may need to be updated even if the security requirements have not changed; (2) both control and data dependencies can have impact on the validation of security requirements. Therefore it is important to check whether the implementation of security aspects can catch the problematic changes, for instance, the point-cut expressions that need to be updated in order to include more changing functions into the scope.

In future, we aim to automate some part of the analysis so that it is possible to reduce the workload for the developers when the system is changed frequently. Also we aim to apply the framework to a substantially larger case study in the public domain so that our findings can be generalised and shown to be useful for security practitioners.

References

1. Haley, C.B., Laney, R.C., Nuseibeh, B.: Deriving security requirements from crosscutting threat descriptions. Proceedings of the 3rd international conference on Aspect-oriented software development. pp. 112–121. ACM, New York, NY, USA (2004).
2. Rashid, A., Sawyer, P., Moreira, A., Araujo, J.: Early aspects: a model for aspect-oriented requirements engineering. Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on. pp. 199 – 202 (2002).
3. Mellado, D., Blanco, C., Sánchez, L.E., Fernández-Medina, E.: A systematic review of security requirements engineering. Computer Standards & Interfaces. 32, 153–165 (2010).
4. Haley, C.B., Laney, R., Moffett, J.D., Nuseibeh, B.: Security Requirements Engineering: A Framework for Representation and Analysis. Software Engineering, IEEE Transactions on. 34, 133 –153 (2008).
5. Win, B.D., Joosen, W., Piessens, F.: Developing Secure Applications through Aspect-Oriented Programming. Aspect-Oriented Software Development. pp. 633–650. Addison-Wesley (2002).
6. Viega, J., Bloch, J.T., Ch, P.: Applying Aspect-Oriented Programming to Security. Cutter IT Journal. 14, 31–39 (2001).
7. Yu, Y., Leite, J.C.S. do P., Mylopoulos, J.: From Goals to Aspects: Discovering Aspects from Requirements Goal Models. Proceedings of the Requirements Engineering Conference, 12th IEEE International. pp. 38–47. IEEE Computer Society, Washington, DC, USA (2004).
8. Franqueira, V.N.L., Tun, T.T., Yu, Y., Wieringa, R., Nuseibeh, B.: Risk and argument: a risk-based argumentation method for practical security, http://re11.fbk.eu/accepted.
9. Niu, N., Yu, Y., González-Baixauli, B., Ernst, N., Sampaio do Prado Leite, J.C., Mylopoulos, J.: Aspects across Software Life Cycle: A Goal-Driven Approach. In: Katz, S., Ossher, H., France, R., and Jézéquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI. pp. 83–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2009).
10. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying Monitoring and Switching Problems in Context. Requirements Engineering Conference, 2007. RE '07. 15th IEEE International. pp. 211 –220 (2007).
11. Nhlabatsi, A., Nuseibeh, B., Yu, Y.: Security Requirements Engineering for Evolving Software Systems. International Journal of Secure Software Engineering. 1, 54–73 (2010).
12. Wilde, N.: Understanding Program Dependencies. (1990).
13. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 319–349 (1987).
14. Pugh, W., Wonnacott, D.: Eliminating false data dependences using the Omega test. SIGPLAN Not. 27, 140–151 (1992).
15. Pugh, W., Wonnacott, D.: Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependences. IEEE Trans. Parallel Distrib. Syst. 6, 204–211 (1995).

16. Yau, S.S., Zhang, X.: Computer Network Intrusion Detection, Assessment And Prevention Based on Security Dependency Relation. 23rd International Computer Software and Applications Conference. p. 86–. IEEE Computer Society, Washington, DC, USA (1999).
17. Island Hopping: Mitigating Undesirable Dependencies - TechNet Magazine Blog - Site Home - TechNet Blogs, http://blogs.technet.com/b/tnmag/archive/2008/02/27/island-hopping-mitigating-undesirable-dependencies.aspx.
18. Weiser, M.: Program Slicing. Software Engineering, IEEE Transactions on. SE-10, 352 – 357 (1984).
19. Van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. Proceedings of the Second IEEE International Symposium on Requirements Engineering. p. 194–. IEEE Computer Society, Washington, DC, USA (1995).

Index