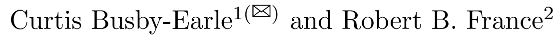
Discovering Potential Interaction Violations among Requirements


Curtis Busby-Earle[1(✉)] and Robert B. France[2]

[1] The University of the West Indies, Mona, Jamaica
curtis.busbyearle@uwimona.edu.jm
[2] Colorado State University, Fort Collins, USA

Abstract. To fully embrace the challenge of developing more robust software, potential defects must be considered at the earliest stages of software development. Studies have shown that this reduces the time, cost and effort required to integrate corrective features into software during development. In this paper we describe a technique for uncovering potential software vulnerabilities through an analysis of software requirements and describe its use using small, motivating examples.

Keywords: Models · Requirements · Security · Path fixation

1 Introduction

Security requirements are seldom explicitly stated at the outset of a project [8]. Typical security considerations at the requirements development stage are: confidentiality of sensitive information [6]; potential system threats and exploits [8,14,15]; privacy and trust concerns [20,21]; and profiles of potential attackers [13,15]. Requirements engineers and security experts attempt to address security issues by using techniques such as misuse cases [14], abuse cases [15], UMLSec [13], the SQUARE method [10], KAOS [18] and security patterns [7]. Many of the current techniques, however, rely heavily on the expertise and subjective judgement of security professionals [1–4]. As an example of the subjectivity that comes into play when using these techniques, consider the use of misuse cases. A misuse case is a special kind of use case that is a description of behavior that should not occur in a system. Misuse cases are described alongside use cases. The development and analysis of misuse cases may proceed as follows:

1. Describe the services that the users want, regardless of any security considerations. Use cases are used for this purpose
2. Introduce the major misuse cases and mis-actors. Misuse cases are initiated by mis-actors.
3. Investigate the potential relations between misuse cases and use cases, and describe as use-case includes-relations. Many threats to a system can be realised by using the system's normal functionality.

R. B. France—Deceased February 15, 2015.

© Springer International Publishing Switzerland 2016
J. Krogstie et al. (Eds.): CAiSE 2016, LNBIP 249, pp. 243–253, 2016.
DOI: 10.1007/978-3-319-39564-7_23

4. Introduce new use cases that detect or prevent misuse cases.

Steps two and three are key to the process, but they are subjective and the results of their application are completely dependent upon a security expert's judgement. Although a requirements engineer has the task of specifying what an intended software system should do, a requirements engineer is not expected to be a security expert. Security is usually grouped with, and considered as, a non-functional requirement [18,19]. Non-functional requirements have traditionally only been considered in the later stages of software development, for example during architecture development [1,5] and coding [12].

In this paper we present a technique that provides a means by which the knowledge of domain experts including security experts can be captured, retained, used, refined and shared among requirements engineers and other software engineering practitioners, thus assisting them in their task of considering potential software defects at the earliest stages of software development. The expertise is captured in the form of dependencies between domain-specific terms that are included when writing requirements. The captured expertise is used in the technique to analyse functional requirements to uncover potential vulnerabilities. The technique does not replace human expertise, rather it exploits shareable knowledge and skill and augments available human know-how. The technique can be summarised as follows:

> Given a set of use case scenarios, each scenario is represented as a flow graph. Each scenario step is represented as a node in a graph and transitions from one scenario step to another as an edge. The analysis is performed by forming the transitive closure of the forest and identifying any interactions that violate stated security or other policies.

The proposed technique, called Loophole Analysis, addresses the problem of functional fixation in the context of requirements analysis. Functional fixation is the inability to see uses for something beyond what is presented [11]. In other words, it is the belief that something can only be used for its stated purpose. In the context of software systems we can and do use functionality provided by software in unintended ways.

For example, a Windows XP user with no administrator privileges can acquire them by creating a shortcut to IE6, enable the 'Run with different credentials' option, and open a shell as a local administrator. From a security standpoint, each step involved in accomplishing the task is allowed, but this particular usage leads to undesirable situations. Another example is provided by Linux, Android and other UNIX-based operating systems where automatic file completion is a very useful feature but, it also helps intruders to find target files more quickly [14]. Everyday objects, including software, may fulfill their stated goals and yet, may allow undesirable behaviour.

2 Background and Related Work

The loophole analysis seeks to identify interactions that are allowed but may result in undesirable situations when using software. The analysis is performed

during the process of specifying requirements. Uncovering and mitigating potential vulnerabilities during requirements analysis, reduces the time, effort and cost of fixing these problems, when compared to addressing them later in the development process [10,13]. Other approaches exist that attempt to address security during the early stages of development. A relatively current trend is the use of models [1,2,4]. Loophole Analysis seeks to identify use case scenario interactions that are allowed but may result in undesirable situations if the scenarios are implemented as specified.

The input to our technique is a requirements specification document consisting of a set of use case scenarios. The steps of the technique are,

1. Develop a domain meta-model to represent the vocabulary and key concepts of a problem domain
2. Develop a set of meta-use case scenarios from the domain model
3. Create a class model of the specific application/system to be built, using the domain meta-model developed during step 1
4. Create a set of application-specific use case scenarios using the meta-scenarios developed during step 2
5. Using unique identifiers for each use-case step described in all scenarios, create a flow graph that simulates transitions within the application being modeled
6. Apply the loophole analysis to the forest to identify any undocumented use case scenarios
7. Document any newly discovered scenarios and either expressly permit or prevent them. The newly discovered scenarios can be documented as misuse cases
8. Repeat, beginning at step 6, until no new scenarios are discovered

Steps one and two are project independent and can be completed by domain experts. Steps three to eight are project specific and are completed by requirements engineers. The use case scenarios that our technique supports describe a system from a user's perspective, and thus focus on user-system interactions. Specifically, a scenario is an ordered set of interactions between a system and a set of actors external to the system.

Scenarios are typically written in a natural language to facilitate understanding by as many stakeholders as is possible. A natural language however, does not readily lend itself to analysis and its use often leads to imprecise statements. For this reason, use case scenario templates are incorporated in our technique. The numbered steps and unique identification of each scenario enables the straightforward representation of each step as a node in a flow graph, and inter-scenario and intra-scenario step transitions as directed edges.

Although there are existing approaches and techniques that are based on security knowledge that is represented in various forms [3–5,7,10,18], our approach seeks to improve a requirements document by revealing undocumented requirements, and attempting to make implicit security requirements explicit. In particular, implicit security requirements are attempted to be made explicit by a process that uses an Imposed Security Dependence (ISD) [17]. A domain model

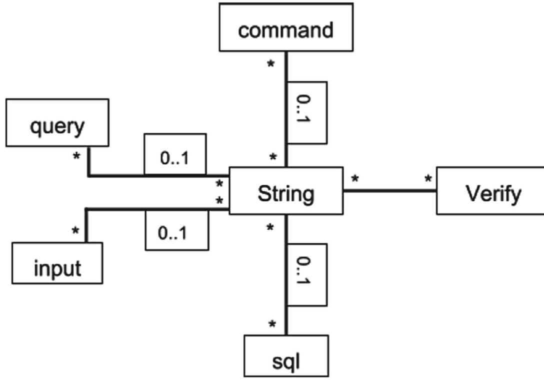


Fig. 1. Requirements terms domain model

captures domain and security expertise, and contains dependencies between domain-specific requirements terms (see Fig. 1).

For any two requirements terms (RT) α and β, we define an ISD as,

`RT` α has an ISD on RT β `when the use of` α `in a requirement dictates the use of` β `in a separate requirement.`

Two requirements therefore have an ISD relation when an ISD exists between requirements terms contained in each. In this paper we represent an ISD between two terms using the symbol $\not\vdash$ placed between them and the terms and symbol enclosed in parentheses. ISD relations have the following properties where t refers to a requirement term,

$$\neg(t1 \not\vdash t1), \tag{1}$$

$$(t1 \not\vdash t2) \not\Rightarrow (t2 \not\vdash t1), \tag{2}$$

$$(t1 \not\vdash t2) \; and \; (t2 \not\vdash t3) \Rightarrow (t1 \not\vdash t3) \tag{3}$$

Commonly used requirements terms and their imposed security dependencies are created by a requirements engineer/domain expert and a security expert. These experts use their knowledge and experience to create a domain model of requirements terms that is independent of any specific project. The model is then used to develop a table of ISDs. The domain specific ISD table is stored for subsequent use in individual, domain specific projects. Once such a table of terms and ISDs has been created, the knowledge and expertise of the domain and security experts will be captured and retained in the table. Through its use in various software development projects, the table and its content can be shared, refined and improved upon by other practitioners.

3 Discovering Undocumented Scenarios

To uncover the types of undesirable interactions previously discussed in Sect. 2 we use the notion of a path. A path is a sequence of use case scenario steps, where

the sequence can cut across many use cases. Use case scenarios typically describe a system from a user's perspective, and thus focus on user-system interactions.

A scenario is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. We define path fixation as the belief that the simple paths described in a specification document are the only ones that will exist in the implemented system. The discovery of paths that overturn such beliefs is the basis of our loophole analysis. Piessens defines a vulnerability as any aspect of a computer system that allows for breaches in its security policy [16]. In the context of requirements analysis, loopholes are paths that lead to violations in security or other policies that govern system behaviour.

3.1 Modeling a Domain

A plan has been formulated whereby a movie will be viewed at a movie theatre. The plan involves hiring a taxi from home to the train station, catch a train into the city, walk to the movie theatre and (hopefully) enjoy the movie, have a meal at a restaurant and then return home. All of the activities in the plan, except walking to the theatre, involve the utilisation of a service, for which payment is expected. In the plan, services include the taxi, train, viewing the movie and having a meal. Depending on the available technology, payment for the services could be made using cash or a card. The domain meta-model for the payment for the provision of a service, including the aforementioned services, can be represented as shown in Fig. 2.

The Account template depicted in Fig. 2 is optional, it may or may not be included in a class model derived from the domain model. Optional templates are denoted with a multiplicity written in the top left corner of the name compartment of the template box. Similarly, the associations between Customer and

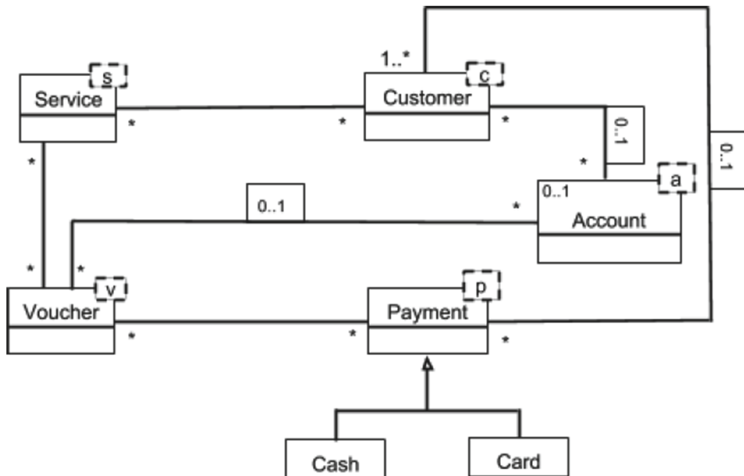


Fig. 2. Payment for service domain model

Payment, Customer and Account, and Account and Voucher may or may not exist in a particular system. These optional associations in the domain model are indicated by the enclosed association label 0..1. When included in a class model, an optional association's multiplicities may be specialised.

An application that can be developed within this domain may require a customer to enter some information and typically exchange data with other applications. The security considerations for such interactions can vary but fundamentally, the use of strings must be included in such considerations, where a string is a sequence of symbols/characters. There are different types of strings that are used when developing applications: command, input, sql, and query for example.

From a security standpoint, all strings must be verified when presented to an application to ensure that they conform to the application's expected data format. We therefore say that in this domain, the term string has an imposed security dependence on the term verify. Whenever a requirement specifies the use of a string, there must be another requirement that specifies how the string is to be verified. String verification remains one of the top sources of software vulnerabilities and programming flaws. We represent the ISD relationship among these terms as shown in Fig. 1.

The relationships among the terms that are captured in the model in Fig. 1 can then be stored in an ISD table and used during our analysis. In this model, the terms command, input, sql and query are used to clarify the type of string being used.

Using the templates in our domain model, we can now define payment systems for many (similar) types of services: taxi, train, movie theatre and restaurant. In this domain, a customer pays for a service and receives a voucher as evidence of payment. A voucher can therefore be a train or movie ticket, or a receipt from a taxi or restaurant. From the domain meta-model we can also develop meta-use case scenarios. The names of the templates taken from the domain meta-model become parameters in the meta-use case scenario. When a class model is developed from the domain meta-model, these parameters are substituted with the class names.

Using the domain meta-model and associated meta-use case scenarios, we can now model a particular system that provides a service that falls within the domain by creating a specialisation of the domain meta-model and providing arguments to the meta-use case scenarios. For example, a train ticketing service can be modeled as shown in Fig. 3. Service, Voucher and Customer have been specialised to Train, Ticket and Passenger respectively. Because the optional Account class has been included in this model, every account must be associated with at least one passenger; the multiplicity at the Passenger association end has thus been specialised from * in the domain meta-model to 1..* in the class model.

We can now utilise the meta-use case scenario to create a specific use case scenario for the train ticketing system. The names of the templates of the domain meta-model that were used to develop the meta-use case scenario are substituted with the names of the classes in the train ticketing model to create the use case for the train ticketing system.

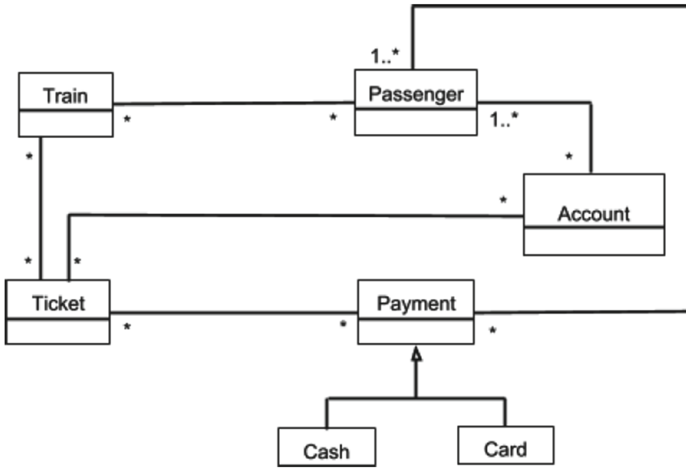


Fig. 3. Project specific model

3.2 Analysing the Model

From the use case scenario, we can now develop a flow graph that "simulates" the operation transitions within the particular system. Each step in the use case is represented as a node and transitioning from one step to the next is represented as a directed edge. To illustrate the types of undocumented scenarios we are looking for in our analysis we must include another use case scenario.

Consider another typical scenario where a customer wishes to debit her account associated with the train service. This account can be used to purchase tickets or pay for other affiliated services that may be available. The use case scenario for replenishing such an account and its flow graph are shown in Fig. 4. The steps involved in replenishing an account include the selection of a payment method i.e. cash or card, and therefore the steps involved in the replenish passenger account can be extended by those in the make payment scenario.

In this trivial example we note that it is therefore possible for a passenger, although intent on depositing money into her account could instead, pay for a ticket. In the flow graph this is depicted with a broken line from step 4 to step *4 where *4 represents step 4 from the Make Payment use case scenario. This is an unintended sequence of operations. In our system, this scenario must therefore be expressly prevented as the two operations should be discrete. This potential, unintended operation could have been possible because while modeling the system, we became fixated on the scenario steps (paths) for purchasing a ticket and replenishing an account.

Further, when processing a card as in step 2 of the main success scenario of the make payment use case, a passenger could be required to enter card information such as the name on the card, the card number and expiry date. These input strings should be verified when entered before being processed. The ISD table ensures that requirements are included in the application's specification

Goal	Replenish Passenger Account
Pre-Condition	Account already exists
Post-Condition	Passenger account balance increases by amount money deposited
Main Success Scenario	1. Passenger selects account 2. Passenger selects a method of payment 3. If payment method is card, system processes card payment 4. If payment method is cash, system processes cash payment 5. System updates passenger account
Extensions	3.1 Passenger's card cannot be processed 3.1.1 System provides error message indicating reason(s) and possible solution(s) 4.1 Passenger does not provide at least minimum cash amount 4.1.1 System provides message indicating shortfall and options 4.2 Passenger provides more than maximum allowed cash amount 4.2.1 System provides error message and possible solution(s) 5.1 System does not update account 5.1.1 System provides error message indicating reason(s) and possible solution(s)

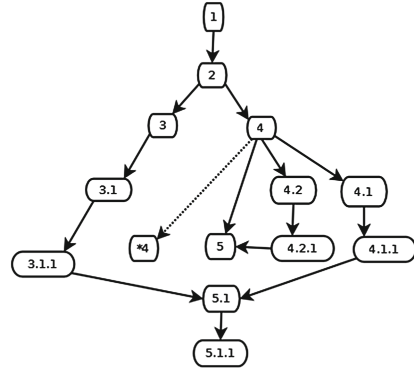


Fig. 4. Replenish account use case scenario and graph

document that address the verification procedures. This step attempts to make implicit security requirements, explicit and helps a requirements engineer create a more complete specification document.

3.3 The Loophole Algorithm

In Sect. 3.2 we developed a more complete set of requirements by making implicit requirements explicit and discovered an undocumented and unintended scenario. This was accomplished by incorporating ISD relationships among requirements terms to generate skeletal requirements and finding undocumented paths i.e. loopholes. In practice however, models and requirements are not so trivial.

The Loophole Algorithm is used to systematically and formally analyse a requirements specification for loopholes. We examine a set of requirements by choosing a relation `R`. Let `N` be the set of nodes that are in a forest representation of a requirements specification document, and `R` be a relation that maps a distinct node `n1` $\in$ `N` to another distinct node `n2` $\in$ `N` where `n2` is directly reachable from `n1`.

We chose `R` with the following properties,

$$R : N \times N \tag{4}$$

$$\forall r : N \bullet (r, r) \notin R \tag{5}$$

$$\forall r, q : N \bullet (r, q) \in R \Rightarrow (q, r) \notin R \tag{6}$$

$$\forall r, q, s : N \bullet (r, q) \in R \wedge (q, s) \in R \Rightarrow (r, s) \in R \tag{7}$$

Because we are representing intended interactions in a system under development, intuitively, `R` is anti-reflexive, transitive (expressions 5 and 7) and must therefore also be anti-symmetric (expression 6). We want to identify undocumented paths in `N` using `R`. For this purpose, however, `R` is not sufficient as we

have demonstrated that some of the possible paths are typically not explicitly included in a requirements document. These are the paths we are interested in. We therefore include the set of all reachable paths by finding the transitive closure of R.

From a security standpoint, a policy is described by the allowed interactions of the intended system's users and objects i.e. the members of N. To identify potential breaches, the discovery of undocumented paths can also identify potential vulnerabilities such as those described earlier (see the last paragraph in the Introduction).

The steps of the Loophole Analysis are as follows:

1. Represent the relation R as a binary matrix M.
2. Find the transitive closure of R, using the Floyd-Warshall algorithm [9]. Call this new relation R*.
3. Represent R* as a binary matrix M′.
4. Perform the bit-wise XOR of corresponding elements of M and M′. This will identify maplets created as a result of step 2 i.e. the indirect relationships that are not included in the original.
5. A maplet that exists in M′ but not M represents an undocumented transition and the sequence of scenario steps that occur before and after it must be investigated.
6. A loophole (i.e. a vulnerability) exists when an undocumented scenario violates a stated policy.

4 Conclusion and Future Work

Although preliminary, we believe the results are promising. Loopholes are unknown, reachable paths that would exist if a system were to be developed in accordance with the specification document in the form prior to the loophole analysis. The analysis did not include nor require the motives, resources and skills of an attacker, or possible threats to the system to be postulated. Its foundation is based on the statement of policy and on path fixation. The algorithm includes completing the transitive closure on the forest, but other traversal methods such as a depth first search are being investigated.

We have been developing a tool to assist in the specification of requirements, and their analysis using the Loophole Algorithm. An engineer can: document a problem statement/description, permit stakeholders to enter user stories, enter data on concepts that are elicited from the user stories, create use cases and misuse cases using a Cockburn style template. The tool then takes this data and draws: a requirements model (we are in the process of allowing an engineer to include object constraint language (OCL) statements with the model), unified modeling language (UML) use case diagrams, UML sequence diagrams and a system operation simulation (flow graph). It is at this stage that the Loophole Analysis is performed and any undocumented scenarios are depicted in a flow graph, and misuse cases are generated by the tool that capture the undocumented scenarios. The engineer can then either accept these captured scenarios

as valid, or leave them as misuse cases, and create new use cases that will serve as countermeasures.

We are in the process of conducting initial tests and implementing bug fixes. It will be tested both with documents that will provide empirical validation of its capabilities and in our software engineering courses during the forthcoming academic year 2016–2017 for its useability. We will then begin development that will include the ability to create domain meta-models and meta-use case scenarios. We are mindful that in the tool's continued development, consideration must be given to the issues involved in using informal use case descriptions, in particular the possibility of information loss and the introduction of errors.

References

1. Roudier, Y., Apvrille, L.: SysML-Sec-a model driven approach for designing safe and secure systems. In: International Special Session on Security and Privacy in Model Based Engineering: Proceedings of the 3rd International Conference on Model Driven Engineering and Software Development. MODELSWARD 2015, Angers, Loire Valley, France. IEEE Computer Society Press (2015)
2. Ivanova, M.G., Probst, C.W., Hansen, R.R., Kammüller, F.: Transforming graphical system models to graphical attack models. In: Mauw, S., et al. (eds.) GraMSec 2015. LNCS, vol. 9390, pp. 82–96. Springer, Heidelberg (2016). doi:10.1007/978-3-319-29968-6_6
3. Apvrille, L., Roudier, Y.: SysML-Sec attack graphs: compact representations for complex attacks. In: Mauw, S., et al. (eds.) GraMSec 2015. LNCS, vol. 9390, pp. 35–49. Springer, Heidelberg (2016). doi:10.1007/978-3-319-29968-6_3
4. Schilling, A., Werners, B.: Optimizing information systems security design based on existing security knowledge. In: Persson, A., Stirna, J. (eds.) CAiSE 2015 Workshops. LNBIP, vol. 215, pp. 447–458. Springer, Heidelberg (2015)
5. Gausemeier, J., Rammig, F.J., Schäfer, W., Sextro, W.: Case study. In: Gausemeier, J., Rammig, F.J., Schäfer, W., Sextro, W. (eds.) Dependability of Self-Optimizing Mechatronic Systems. LNME, vol. 2, pp. 175–192. Springer, Heidelberg (2014)
6. Landtsheer, D., Renaud, van Lamsweerde, A.: Reasoning about confidentiality at requirements engineering time. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ESEC/FSE-13, Lisbon, Portugal. ACM (2005)
7. Hafiz, M., Adamczyk, P., Johnson, R.E.: Organizing security patterns. IEEE Softw. **24**(4), 52–60 (2007). IEEE Computer Society Press
8. Stallings, W., Brown, L.: Computer Security: Principles and Practice, 2nd edn. Pearson Prentice Hall, Upper Saddle River (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
10. Mead, N.R.: Identifying security requirements using the Security Quality Requirements Engineering (SQUARE) method. In: Mouratidis, H., Giorgini, P. (eds.) Integrating Security and Software Engineering: Advances and Future Vision, pp. 44–69. IGI Global, Hershey (2007)
11. Zatko, P.: Psychological security. In: Oram, A., Viega, J. (eds.) Beautiful Security: Leading Security Experts Explain How They Think, pp. 1–20. O'Reilly Media, California (2009)

12. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-functional Requirements in Software Engineering. International Series in Software Engineering, vol. 5. Springer, Heidelberg (2000)
13. Jürjens, J.: UMLsec: extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
14. Sindre, G., Opdahl, A.: Eliciting security requirements by misuse cases. In: Proceedings of Technology of Object Oriented Languages and Systems. IEEE Computer Society (2000)
15. McDermott, J., Fox, C.: Using abuse case models for security requirements analysis. In: Proceedings of Computer Security Applications Conference. IEEE Computer Society (1999)
16. Piessens, F.: A Taxonomy (with Examples) of Cases of Software Vulnerabilities in Internet Software (CW346). Katholieke Universiteit Leuven, Belgium (2002)
17. Busby-Earle, C., Mugisa, E.K.: Metadata for boilerplate placement values for secure software development using derived requirements. In: Proceedings of the 13th IASTED International Conference on Software Engineering and Applications (SEA 2009). ACTA Press, Cambridge, Massachusetts (2009)
18. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd International Conference on Software Engineering. ICSE 2000, Limerick, Ireland. ACM (2000)
19. Devanbu, P.T., Stubblebine, S.: Software engineering for security: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. ICSE 2000, Limerick, Ireland. ACM (2000)
20. Wang, A.J.A.: Information security models and metrics. In: Proceedings of the 43rd Annual Southeast Regional Conference. ACM-SE 43, Kennesaw, Georgia. ACM (2005)
21. Liu, L., Yu, E., Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering. RE 2003, Washington DC. IEEE Computer Society (2003)