Electronic Research Archive of Blekinge Institute of Technology
<http://www.bth.se/fou/>

This is an author produced version of a conference paper. The paper has been peer-reviewed but may not include the final publisher proof-corrections or pagination of the proceedings.

Citation for the published Conference paper:

Title:

Author:

Conference Name:

Conference Year:

Conference Location:

Access to the published version may require subscription.

Published with permission from:

Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec)

Dejan Baca^{1,2} and Kai Petersen^{1,2}

¹ Blekinge Institute of Technology Box 520,
SE-37225 Ronneby, Sweden

² Ericsson AB,
Sweden

Abstract. Software security is an important quality aspect of a software system. Therefore, it is important to integrate software security touch points throughout the development life-cycle. So far, the focus of touch points in the early phases has been on the identification of threats and attacks. In this paper we propose a novel method focusing on the end product by prioritizing countermeasures. The method provides an extension to attack trees and a process for identification and prioritization of countermeasures. The approach has been applied on an open-source application and showed that countermeasures could be identified. Furthermore, an analysis of the effectiveness and cost-efficiency of the countermeasures could be provided.

1 Introduction

Software security has recently become an important business case for companies to protect information availability [1]. Therefore, it is important to make security an integral part throughout the software development process which has been done through security touch-points introduced by Gary McGraw [2]. During the development of software, bugs and vulnerabilities are unintentionally introduced into the end product. The cost of removing these vulnerabilities increases the further the product has reached in its development cycle. It is therefore preferable to detect them as early as possible in the development process. Vulnerabilities are introduced into two different phases of development, namely architecture/design and implementation. Implementation vulnerabilities such as buffer overflows are added to the source while the developer writes it. For these types of vulnerabilities, static code analysis tools can be used for early detection [3,4]. Design vulnerabilities on the other hand can be introduced into the product before there is any source code to examine. Early vulnerability prevention therefore has to focus in the architect of the product instead of the implementation, and hence security touch points have to be introduced early.

Different approaches have been introduced to detect threats/attacks on the system, namely risk analysis, and attack trees. Many if these methods are not

specific for software development and some require UML diagrams or other information of the end product that might not be available (see e.g. [5]). This is often done with risk analysis and other threat modeling techniques. Attack trees are specifically proposed in the software development context and only require an understanding of the intended end product. Hence, they can be used as a security touch point early, even before implementation has been started. However, there are a few challenges related to the use of attack trees:

- From our experience of working with software security in industry we know that it can be difficult for non-security professionals to create an attack tree.
- The focus of attack trees and risk analysis is on identifying and prioritizing attacks, but not on the protection on the end-product (cf. [6,7,8]).
- The way of prioritizing the attacks is done by assigning estimated risks, which are dependent on many factors not known to the software developer (cf. [6]).

In response to these challenges we present a novel method to prioritize countermeasures to prevent security vulnerabilities, and by that focus the prioritization on the protection of the end-product. The Countermeasure Method for Software Security (CM-Sec) is an extension to attack trees. The method has the following features:

- It aids non security developers in identifying and prioritizing attacks and countermeasures by providing a process that guides them to the solution step by step. The introduction of why the system could be attacked and who could attack the system is used as a trigger to identify attacks and countermeasures.
- Instead of evaluating the risk of an attack for each individual attack ACM-Sec focuses on the prioritization of attacks through cumulative voting, i.e. a fixed number of points is distributed, attacks having a higher threat level receiving more points.

The method has been applied on an open-source system by the authors to demonstrate its applicability. The first author has good knowledge of the system as he is participating in the open source project developing it. The application showed that a broad range of countermeasures could be identified and linked to attacks, and that the distance of the criticality of the countermeasures can be consistently determined.

The remainder of the paper is structured as follows: Section 2 presents the background and related work. Thereafter, Section 3 presents the CM-Sec method. The application of the method on an open source system is illustrated in Section 4. Section 5 discusses the application of the method and Section 6 concludes the paper.

2 Background and Related Work

The related work focuses on approaches to identify and prioritize attacks on software systems early in development, i.e. before coding starts. The motivation

for doing so is that the earlier problems in software development are discovered, the easier and less costly they are to fix [9].

A traditional way of identifying threats to a system would be a quantitative risk analysis like that described in Peltier [10]. The first step in the risk analysis would be to enumerate all assets and include their values for the system and then determine the threats that might exist for these values. The second step is to estimate the probability that a threat will occur and the damage it could cause. By combining these two values you get a priorities list of the top risks for the system. Determining what an asset is for a software product can be problematic, also immediately identifying threats to that assets often requires security expertise.

Attack trees were introduced by Bruce Schneier [11] and are structured by stating (1) the goal of the attack (e.g. obtaining a key); (2) what type of attack to be used (e.g. obtaining a private key from a user); and (3) how to implement the attack (e.g. break into the users machine and read the key from the disc) [12]. The implementations of the attack can be connected to conditions, e.g. one could choose either one of two or more implementation (OR) or several implementations are to be done together [12]. Attack trees have been applied on examples of software systems (cf. [13]). After identifying the attacks should be evaluated based on their risk, i.e. the likelihood of their occurrence multiplied with the damage done [6,7,8]. However, Buldas et al. [6] point out that the damage could be determined, but the risk of the occurrence is hard to estimate. Therefore, they propose to model the probability of the attack as a game for the attacker taking the following parameters into account: gains for the attacker, costs of the attack for the attacker, success probability, probability of getting caught, and penalty of getting caught. A potential drawback of the proposal by Buldas et al. is that the software developers need to have a good understanding of the motivations of the attacker. Furthermore, attackers are different based on each individual's attitude towards risk (see [14] for the distinction between risk averse vs. risk loving).

A process for managing risk is the Riskit method [15], which is used to continuously prioritize and control risks. The method prioritizes risk based on probability and utility loss. In the prioritization of risks and actions the process depends on the estimation of the probability for ranking purpose.

The related work shows that the solutions for addressing security issues prior to implementation have been focusing on the identification and prioritization of attacks and threats, and use this as input for the prioritization of actions (see e.g. [15]). However, few methods focus on the prioritization of actions. However, from a software development perspective it is of interest to know which actions should be taken to avoid the occurrence of attacks. In addition, all approaches focused on prioritizing risks based on the estimations of probability and damage. As pointed out by Buldas the estimation of probability and damage are hard, and as two parameters have to be estimated the likelihood of unaccurate estimations is high. Hence, our method proposes to prioritize the countermeasures/actions using hierarchical cumulative voting.

3 Countermeasure Method for Software Security (CM-Sec)

The method consists of two parts, the first part is an extension of attack trees. The second is a process for conducting the analysis to arrive at the prioritized list of countermeasures. For the second part a tool has been implemented which supports practitioners in conducting the prioritization.

3.1 Countermeasure Graphs: An Extension to Attack Trees

Three extensions are made to attack trees, which only focus on attacks and describe conditions under which the attacks can occur. The extensions are made explicit in the meta model shown in Figure 1. The model consists of goals (why to attack), actors (who attacks), attacks (how to attack), and countermeasures (how to avoid attack), each described as a comment in the Figure.

The first extension is the relationship between goals, actors, and attacks. In attack trees there exists a 1..* relationship between the classes as they are trees, i.e. one goal is related to several attacks while one attack is only related to one goal. However, in practice an attack could be executed by several actors, or an actor could pursue more than one goal. Hence, these relationships were extended to *..*.

The second extension is the inclusion of priorities assigned to goals, actors, attacks, and countermeasures. The priorities are an integer number assigned by the person conducting the prioritization. The prioritization method is further explained when presenting the process of creating the countermeasure graph (see Section 3.2).

Fig. 1. Metamodel of the Extension to Attack Trees

Fig. 2. Example of a Countermeasure Graph

The third extension is the inclusion of countermeasures which are actions by the developers to avoid vulnerabilities to be introduced into the software product in the first place. Hence, from an implementation perspective it is important to prioritize the countermeasures which is done by considering the priority of attacks, as well effort to realize the countermeasure. With the inclusion of countermeasures the protection of the end-product is supported.

An example of the countermeasure graph is shown in Figure 2. The nodes of the graph show the goals, actors, attacks, and countermeasures. The edges connect:

- Goals to agents if the agent pursues the goal.
- Agents to attacks if the agent is likely to be able to execute the attack.
- Attacks to countermeasures if the countermeasure is able to prevent the attack.

Furthermore, the priorities are assigned to each of the nodes. As can be seen agent two is higher prioritized than agent one, meaning that the agent is more likely to execute the attack and hence is a higher threat to the system. The agent has two attacks where attack one is prioritized higher as it does more damage to the system. The figure also shows the prioritization of the countermeasures, countermeasure two being the most efficient in preventing the attack. It is important to observe

that the prioritization does not only allow to prioritize the order of attacks and countermeasures, but also shows the distance between them.

3.2 Process

Identify Goals, Actors, Attacks and Countermeasures. The purpose of the countermeasure graph is to determine what security countermeasures would be useful to include in the product and at the same time identify what countermeasures already exist. At the same time we determine what the greatest threat to the product is and how well we prevent them. To achieve this we need to discover what attacks can be made and how they are stopped, with traditional threat analysis or attack trees this process is direct and tries to immediately identify attacks. We instead divide the work in four distinct steps and then use hierarchical cumulative voting [16] to calculate the impact of every step.

Goals: The first step is to understand why anyone would attack the product and what their gain would be. This is often the same as the intended usage of the product and is used as a guideline to see the big picture.

Agents: Thereafter the *Agents* are identified. *Agents* are users, roles and software that interact with the products. Preferably the entire range of possible users should be presented in this step, from end users to administrators and outside software that interact with the product.

Attacks: Combining the *Agents* with a *Goal* we then look for *Attacks*. This step of the threat analysis is made easier because we have a more clear idea who and why the attack would accrue. We do not focus on if the attack would work, only if it is a possible route for the *Agent* to take to achieve his desired *Goal*.

Countermeasures: The final step is identifying what *Countermeasures* could be used to prevent an attack. A countermeasures can prevent one or more attacks.

Having identified the goals, agents, attacks, and countermeasures we are interested in which countermeasures are most effective.

Prioritization of Countermeasures. The attack graph consists of sets of goals, agents, attacks, and countermeasures. In the following we present the prioritization questions asked to the developers, and also provide further explanation of the voting on each hierarchy.

- *Goals:* To what degree would the achievement of the goal damage the system from either a cost or stability perspective? These are at first general goals that the different *Agents* might be interested in achieving. While the first iteration of the attack tree has general goals further iteration will focus the goals on specific requirement and features and are abuse cases based on them. The *Goals* are also connected to what *Agents* would be interested in them. Voting on the *Goals* is focused on the damage that specific goal would sustain on the product. Depending on the goal the damage could be economical or system stability.

- Agent: How large is the threat of the agent for the system? With the *Agents* we determine how threatening the different roles are to the product. Votes are based on how "scared" the product should be from that *Agent*. As an example a server product would consider end user more threatening then system administrators while privacy software might be the opposite.
- Attack: How likely is the success of the attack for the agent it belongs to? The attack voting is per *Agent* and determines how likely it is for that *Agent* to succeeded with the attack.
- Countermeasures: How efficient is the countermeasure in preventing an attack? When voting on *Countermeasurs* the focus is on their ability to prevent the attack compared to each other. In some cases two *Countermeasures* might be equally effective, but one of them might aid in preventing other attacks as well. The attack graphs would then put higher priority on that *Countermeasur*.

Each of the sets is prioritized according to the rules of hierarchical cumulative voting (HCV) [16]. Cumulative voting in itself has the benefit of (1) that it is simple to do, and (2) the distance of importance between two items is visible. The voting is done by providing a number of points (e.g. 100 dollars) and then distributing them between a set of items. In HCV the prioritization is done on each level of the hierarchy (in this case goals, agents, attacks, and countermeasures).

Figure 3 shows the principle of HCV. The prioritization is done on all level, in this case on level 0 (L0) no prioritization is necessary as there is only one node on that level. On level H1 three nodes compete against each other for the points, i.e. nodes two, three, and four. On the lowest level H2 nodes within a group (H2.1, H2.2, and H2.3) compete. When the votes are completed the value of a node is calculated by multiplying the path of its parent nodes. For example, to know the value of node 11 we calculate the product of points assigned to node 11, 4, and 1. As there are different number of nodes within the groups the votes have to be adjusted by the number by the number of nodes in each group (cf. [16]).

In the case of our analysis an n to n relationship exist between nodes on different levels of the hierarchy. If an attack is related to several agents then this should raise the prioritized value of the attack. As an example we calculate the value of countermeasure two in Figure 2, which prevents two attacks that can be

Fig. 3. Cumulative Hierarchical Voting

Fig. 4. Countermeasure 2 values from Attack 2

exploited by two different agents. First, we calculate the value for counter two ($C2_2$) related to attack 2 as shown in figure 4 (remaining zeros removed):

$$C2_2 = 250 * 320 * 300 * 300 = 7200 \quad (1)$$

Thereafter, counter two is calculated for attack 1 ($C2_1$). Because Attack 1 only has two countermeasures its values need to be normalized to the countermeasures from Attack 2 that has the highest number of countermeasures. As such, Countermeasure 2 with a value of 700 is normalized worth 280 points. Observe also that for Attack 1 two agents can perform the same attack, as seen in Figure 5. However, they do not provide the same risk, neither do they have the same likelihood of succeeding with the attack. They are therefore prioritised individually and then added to the equation individually. By doing so countermeasures that prevent several attacks especially attacks that are possible from multiple Agents are rated higher, and hence the threat level increases:

$$C2_1 = (250 * 150 * 1000 * 280(norm)) + (250 * 320 * 700 * 280(norm)) = 26100 \quad (2)$$

Overall, the value of the attack is the sum of $C2_1 + C2_2 = 33380$. The two are added because the countermeasure prevents two attacks and hence is more effective. Knowing the prioritization of the countermeasures with regard to effectiveness allows to combine them with costs in a matrix. The cost can either be determined by HCV just as the prioritizations of if it is possible use real man-hour estimations. The matrix, as seen in figure 8, shows different areas for the interaction of effectiveness and cost that can be used as a support in deciding which countermeasures to focus on. The bottom-right area contains countermeasures that are effective and have low costs. Hence, these should be implemented first. The top-left area contains countermeasures that are ineffective and at the same time costly, hence they should be avoided. In the middle of the two areas

Fig. 5. Countermeasure 2 values from Attack 1, including two Agents

the borderline-cases are shown, which could be implemented if there are enough resources available.

4 Application

To demonstrate the method we applied it on an open source system, called Code 43.

4.1 System Description and Development Environment

We examined an open source product, which is an online first person shooter game. An overview of the architecture of the system is provided in Figure 6. In the center of the system is the master server providing server lists to clients, storing clients authentication information (e.g. authorization data), and client statistical data showing the performance of the players. Connected to the master server are the servers hosting the games. The clients are the ones logging into the servers that they receive through the server list provided by the master server. Because servers are setup by users they can not be trusted and information from them can be corrupt. Only the master server is in the trusted zone, all other actors in the system are outside this zone (servers are untrusted and clients are considered unsecured).

The game is a complex product consisting of 400,000 lines of code not including blanks or comments. This project has matured from other products, i.e. it is reusing source code. In total there has been 26 major active developers.

4.2 Result of Applying ACM-Sec

In this example we have identified four Agents that interact with the system. They have five distinct goals that can be archived via six attacks. From these six

Fig. 6. Application Network Overview

attacks we devised eleven countermeasures that would in different effectiveness prevent one or several attacks. An overview of the countermeasure graph is shown in Figure 7. Inside the nodes the value assigned during the prioritization is shown. All values have been normalized according to the approach illustrated in Section 3.2. If there is more than one number in a node then this is for the different ages (see, for example, nodes *ask for password* and *fake statistics*).

In the following we provide the details for the goals, agents, attacks and countermeasures. This includes a detailed description of each of them.

Fig. 7. Countermeasure Graph for Open Source Game

Table 1 provides an overview of the goals attackers might have. The majority of the goals are focused on cheating, like alter statistics and cheat on server. The motivation is either to steal an edge (steal an account), to gain an edge (client-site cheating), or to edit and modify ones edge (alter statistics). Thus, its primarily about the gaming experience.

Table 1. Goals and Their Damage

Goal	Description
G1: Steal user accounts	All clients have a private account that stores their statistics and aliases. Other users and server operative could steal their clients' identities.
G2: Compromise master	The master server is the only trusted source in the network architect. It is therefore a lucrative target for malicious users.
G3: Alter statistics	Clients compete with each other. All users are therefore interested in faking their success and cheat at their rankings.
G4: Cheat on server	Clients compete with each other. Client might try to alter the game rules to their benefit to gain an edge against other clients.
G5: Create fake server	Clients receive server lists from the trusted master. Servers that want more clients might create fake servers that all point to the same servers. Therefore showing up several times in the server list and creating a larger exposure to clients. This is unfair to other servers.

As can be seen in Figure 7 the players are the agent with the highest threat, the reason being that the goals are all related to the gaming experience. Server operators also can have an interest as sometimes they are players themselves or can be influenced by gamers (e.g. due to ties to other gamers).

Table 2. Agents and Their Threat

Agents	Description
A1: Players	These are clients and the end users of the system. They are also the least trusted source and present the greatest threat.
A2: Server Operators	Any client can create and add servers to the system. As such the servers can not be trusted and server operative can have the same motives as players.
A3: Servers	Being open source software the server can be altered to behave differently than the master server expects. It is therefore also a threat to the system.
A4: Master Server Operators	The operator of the master server has total command over the system. While he/she is an agent he/she does not provide any threats as the master server operator already can do whatever he/she wants.

An overview of the attacks is provided in Table 3. The attacks focus on either stealing other users accounts or cheating to improve ones own statistics. In both cases the threat does not only come from client but also from servers that are run by other clients. The administrators of these servers might have the same interest in other clients' account and statistics, just as other clients might.

The countermeasures which are the main outcome of this analysis are shown in Table 4. As can be seen in Figure 7 one countermeasure is able to prevent several attacks, which raises its value in the prioritization (see, for example, C2).

Table 3. Attacks

Attack	Description
At1: Ask for the password	Social engineering attack where either other clients, server operators or servers send fake login request to other users in an attempt to get the clients password.
At2: Eavesdrop the password	Servers can intercept the clients' login information as it passes through the server to the master server. Servers also need to know that the client as passed the login procedure.
At3: Fake statistics	Clients and server can send any statistics to the master server and thus increasing their own ranking.
At4: AutoAim / Wallhacks	There exists several client side cheats where players use unauthorized software to gain an edge against their competitors.
At5: Malicious statistical data	A large source of user input to the master servers comes from statistical data. As such it is a big threat and way in for attackers. The attacks can vary from buffer overflows to injection attacks on the master servers database or webpage that presents the statistical data.
At6: Authenticate fake server	Any server can authenticate to the master server. They therefore can create multiple entries of their server in the master server list.

Table 4. Countermeasures

Agents	Description
C1: Single logon	Have a single point of login for the user during game start. With a specific login window that can not be replicated or requested by servers or other users.
C2: Public/ private	User public/ private keys signings for client identification.
C3: Avoid servers	Use direct client and master server communication for client authentication. However, servers still need verification from master that the client has authenticated.
C4: Encrypt	Encrypt the password with a master server public key before sending it. Would require a sequence number or timestamp to prevent replay attacks.
C5: Several sources	Verify statistical data by comparing it from several sources. From both client in the game and the server hosting the game.
C6: Master spies	User random samples and make the master join servers to verify that the servers are not duplicates and that the statistical information is correct.
C7: Limit info.	Limit the information sent to the client so automated processes can not aid the player unfairly. For example do not send other players location unless they are within the clients' field of view.
C8: Detection algorithm	Nonhuman actions can be detected by the server by analyzing the clients behavior and comparing it with normal dataset.
C9: Input sanitization	Cleaning all input strings and removing any harmful characters.
C10: Input policy	Having a strict input protocol that drops any incorrect input before it is processed for storage.
C11: IP and Port	Verify unique servers by examining both IP address and the destination Port. The same server can however run several games on different ports.

After having prioritized the effectiveness of the countermeasures using our prioritization approach introduced in Section 3.2 the countermeasures are combined with cost. Cost is the estimated effort required to implement them. The result is shown in Figure 8.

From this the following interpretations can be made: Countermeasures C2, C5, and C6 should be implemented as they are in the zone of countermeasures that are effective and have low cost. As can be seen in Figure 7 C2 affects the

Fig. 8. Combination of Effectiveness with Cost

same attack as C3 and C4. Hence, the implementation of C2 would reduce the effectiveness of these two countermeasures. Countermeasure C8 is very costly to implemented, but is highly effective. As often project resources are limited this countermeasure should be taken into consideration for future releases, i.e. it should be handled as a requirement instead of a quick fix. Countermeasure C11 should not be implemented as it has very low effectiveness, i.e. its implementation does not make a difference. The Figure also shows that C9 and C10 are equally effective, but have different costs. Hence, the analysis also allows prioritization on cost whenever countermeasures are close to each other with regard to effectiveness. In this example we did not find any countermeasures with high cost and low effectiveness. However, in a different context such countermeasures might very well be identified.

5 Discussion

5.1 Practical Implications

Focus on End-product: We would like to stress that it is important to prioritize countermeasures as those are directly related to the end-product. That means they result in actions that could be taken early in the development process to avoid the introduction of vulnerabilities in the first place. Furthermore, it is important to mention that our method can be applied throughout the whole development life-cycle, independently of whether a new product is implemented, or an existing product is analyzed. When an existing product is analyzed it is important to take into account the countermeasures already implemented, this needs to be taken into account when doing the re-prioritization.

Problems of objectivity: Risk analysis and attack trees assume that developers have a good understanding of how an attacker would think as probabilities have to be estimated for many different factors (cf. [6]). Furthermore, it is hard

for developers without security experience to identify attacks without guidance. Hence, the proposed methods addressed this problem in two different ways: First, the attack trees were extended by agents as this allows the developers to put themselves in the shoes of the attacker, knowing who the attacker might be. Secondly, a process is proposed that the developers can follow. Furthermore, the process focuses on comparing different attacks and countermeasures with each other. Having the comparison makes it easier to value the attacks and countermeasures.

Implementation of Triangulation: When conducting the prioritization we recommend that it should be done by several developers. The averages and variances of the points assigned provide an understanding of to what degree the developers agree on voting. If there is a large discrepancy the data will show the need for discussions and further investigations.

Tool support: To support the developers in the prioritization process we are developing a tool. The main feature of the tool is to (1) work in groups during the identification of the goals, agents, attacks, and countermeasures (shared canvas), (2) support in the prioritization. The support should be handled through sliders which allow easy re-prioritization and show the impact of the change of one prioritization in real-time. This makes the method particularly suited for agile development as changes in the system can be very easily prioritized for each iteration.

5.2 Research Implications

The presented method focuses on countermeasures instead of only attacks and is novel with this regard. Hence, research needs to focus on testing the approach in an industrial application. We plan to conduct case studies in industry to evaluate the method with empirical data.

6 Conclusion

This paper presented a novel method to identify and prioritize countermeasures to increase the security of software systems. The method provides an extension to attack trees, as well as a process for the identification and prioritization of the countermeasures. We applied the method on an open source system, the application showing that several countermeasures could be identified. Furthermore, an analysis of the quantitative results is presented, showing that the proposed method has a potential in guiding managers in choosing the most effective and cost-efficient countermeasures. In future work empirical evaluations of the proposed method in industry are needed.

References

1. Frühwirth, C.: On business-driven it security management and mismatches between security requirements in firms, industry standards and research work. In: Proceedings of the 10th International Conference on Product-Focused Software Process Improvement (PROFES 2009), pp. 375–385 (2009)

2. McGraw, G.: Software security: building security in. Addison-Wesley, Upper Saddle River (2006)
3. Baca, D., Carlsson, B., Lundberg, L.: Evaluating the cost reduction of static code analysis for software security. In: Proceedings of the International Workshop on Programming Languages and Analysis for Security (PLAS 2008), pp. 79–88 (2008)
4. Baca, D., Petersen, K., Carlsson, B., Lundberg, L.: Static code analysis to detect software security vulnerabilities - does experience matter? In: Proceedings of the The 4th International Conference on Availability, Reliability and Security (ARES 2009), pp. 804–810 (2009)
5. Howard, M., LeBlanc, D.: Writing Secure Code. Microsoft Press, Redmond, Washington (2003)
6. Buldas, A., Laud, P., Priisalu, J., Saarepera, M., Willemson, J.: Rational choice of security measures via multi-parameter attack trees. In: López, J. (ed.) CRITIS 2006. LNCS, vol. 4347, pp. 235–248. Springer, Heidelberg (2006)
7. Moore, A.P., Ellison, R.J., Linger, R.C.: Attack modeling for information security and survivability. Technical Report Technical Report CMU/SEI-2001-TN-001, Software Engineering Institute (2001)
8. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 186–198. Springer, Heidelberg (2006)
9. Damm, L.O., Lundberg, L., Wohlin, C.: Faults-slip-through - a concept for measuring the efficiency of the test process. Software Process: Improvement and Practice 11(1), 47–59 (2006)
10. Peltier, T.R.: Information security risk analysis. Auerbach, Boca Raton (2001)
11. Schneier, B.: Attack trees. Dr. Dobbs's Journal 24(12), 21–29 (1999)
12. Viega, J., McGraw, G.: Building secure software: how to avoid security problems the right way. Addison-Wesley, Reading (2002)
13. Saini, V., Duan, Q., Paruchuri, V.: Threat modeling using attack trees. J. Comput. Small Coll. 23(4), 124–131 (2008)
14. Hederstierna, A.: Decisions Under Uncertainty - The Usefulness of an Indifference Method for Analysis of Dominance. EFI The Economic Research Institute, Stockholm School of Economics (1981)
15. Kontio, J.: Risk management in software development: A technology overview and the riskit method. In: Proceedings of the IEEE International Conference on Software Engineering (ICSE 1999), pp. 679–680 (1999)
16. Berander, P., Svahnberg, M.: Evaluating two ways of calculating priorities in requirements hierarchies - an experiment on hierarchical cumulative voting. Journal of Systems and Software 82(5), 836–850 (2009)