Automating Threat Modeling through the Software Development Life-Cycle

Guifré Ruiz[1], Elisa Heymann[1], Eduardo César[1] and Barton P. Miller[2]

Abstract— **Fixing software security issues early in the development life-cycle reduces its cost dramatically. Companies doing software development know this reality, and they have introduced risk assessment methodologies in their development processes. Unfortunately, these methodologies require engineers to have deep software security skills to carry out some of the most important steps of this process, and training them on security is expensive. In this scenario, we propose a new automated approach to analyze software designs to identify, risk rank and mitigate potential threats to the system. We designed a new data structure to detect threats in software designs called** ***Identification Tree*****. We also defined a new one for describing countermeasures to threats, called** ***Mitigation Trees*****. Our automated approach relies on** ***Identification Trees*** **and** ***Mitigation Trees*** **to integrate a guided risk assessment process through the development life-cycle. It does not require developers to have any security training, and was integrated in the current Threat Modeling process of Microsoft.**

Keywords— **risk analysis, threat modeling, attack patterns, identification trees, mitigation trees**

I. Introduction

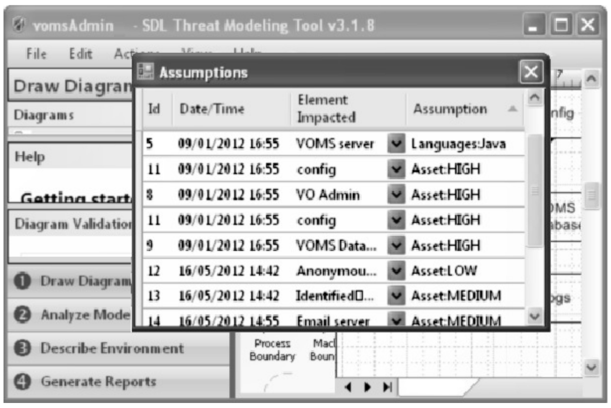
Companies doing software development are interested in reducing the number of vulnerabilities of their products at the lowest cost. To accomplish this, they perform risk assessments to identify and mitigate the threats to their systems. These techniques require developers to have special security training to carry out some of the most important tasks of the process. They involve understanding an adversary's goals in attacking a system based on the system's assets of interest. Unfortunately, programmers tend to think in terms of what a customer needs[1, p. 24], and they are not used to thinking and acting as a professional attacker[2, p. 3], nor they have the necessary security expertise to imagine sophisticated attacks[3, p. 4] or to plan the best mitigation strategies. Furthermore, training them on security or hiring experts in the field is expensive.

There are several risk assessment methodologies. *The Commission of the European Communities, Directorate-General Information Society (CORAS)* is a framework for analyzing UML diagrams and assessing its risks. It can be used by both engineers and risk managers[4]. *Operationally Critical Threat Asset and Vulnerability Evaluation (OCTAVE)* uses information regarding the organization assets, security requirements and infrastructure vulnerabilities to design the appropriate mitigation strategies[5]. *Trike* is a unified conceptual framework for security auditing from a risk management perspective through the generation of threat models. A security auditing team can use it to describe the security characteristics of a system[6]. *AS/NZS 4360* was the first formal standard for managing risks, it contains a set of risk tables and permits organizations to choose their own table depending on their needs[7]. *Microsoft Threat modeling* is a method of assessing and documenting the security risks associated with an application[1]. It helps development teams to identify both the security strengths and weaknesses of the system. All these methodologies share the lacks explained above.

In this paper, we address the problem of the security expertise needed to perform risk assessment. In this area, we do the following contributions. First, we created a new data structure to identify threats in software designs called *Identification Tree*. Second, we designed a new data structure to classify threat countermeasures called *Mitigation Trees*. The information of both of these data structures has been taken from several relevant security sources and standards such as *Common Attack Pattern Enumeration and Classification (CAPEC)*[8], *Common Weakness Enumeration (CWE)*[9] and *Open Web Security Project (OWASP)*[10] among others. Third, we modeled and automated approach that relies on the previous data structures to identify the potential threats to a system design, to purge the less relevant threats according to the user's policies, and computes the software specifications of lowest global estimated cost to mitigate those threats. Our automated approach enforces *security by design*, where threats are mitigated early in the development process. It automated all security parts of the process, allowing engineers with no security training to develop secure software. In addition, it is compatible with the current Threat Modeling process, facilitating companies that are using it its integration.

The rest of the paper is organized as follows. In Section II we describe the representation we expect engineers to use to model their software designs. In Section III we describe the methodology used by our automated approach to identify, risk sort and compute the countermeasures of lowest estimated cost. In Section IV we validate our approach by applying it to a grid middleware called VOMS Admin. Finally, in Section V we present our conclusions.

[1]Computer Architecture and Operating Systems (CAOS), Universitat Autónoma de Barcelona, SPAIN. E-mail: {guifre.ruiz, elisa.heymann, eduardo.cesar}@uab.es
[2]Computer Science, University of Wisconsin, USA. E-mail: bart@cs.wisc.edu



Id	Date/Time	Element Impacted	Assumption
5	09/01/2012 16:55	VOMS server	Languages:Java
11	09/01/2012 16:55	config	Asset:HIGH
8	09/01/2012 16:55	VO Admin	Asset:HIGH
11	09/01/2012 16:55	config	Asset:HIGH
9	09/01/2012 16:55	VOMS Data...	Asset:HIGH
12	16/05/2012 14:42	Anonymou...	Asset:LOW
13	16/05/2012 14:42	Identified□...	Asset:MEDIUM
14	16/05/2012 14:55	Email server	Asset:MEDIUM

Fig. 1
Threat Modeling Tool user interface.

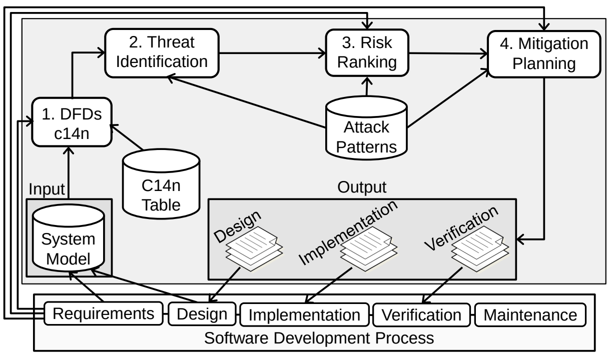


Fig. 2
General Architecture Diagram of our Approach.

II. Software Design Modeling

There are several approaches to represent software designs. As we explained in Section I, our approach is aimed to automate the widely used Threat Modeling methodology of Microsoft, which uses Data Flow Diagrams (DFDs) to represent the system. Thereby, it will be easy for software companies who are using Threat Modeling to adopt our approach, as its input is the Threat Modeling Tool file that contains the DFDs.

To represent the system developers might use well known elements such as external entities, data flows, data stores, processes and trust boundaries to build the DFDs of their system[11], as the Threat Modeling methodology describes. Moreover, we require engineers to define three additional attributes in the DFDs.

- *Asset*: High, Medium or Low.
- *Languages*: Programming languages used.
- *Frameworks*: Frameworks used, if any.

The *Asset* attribute must be defined for all shapes of the DFD, and the *Languages* and *Frameworks* ones only for processes. This information will be used to refine the results in the threat identification and risk sorting steps. In Figure 1, we show the Threat Modeling Tool UI[12], and how it can be used in a natural way to define those attributes.

III. Methodology

We divide the operations of our automated approach in four steps, as shown in Figure 2. We use two knowledge bases: *C14n Table* that contains the information used to map the Labels of the DFD to known values, and the *Attack Patterns* that contain the *Identification Tree*, *Risk attributes* and *Mitigation Tree* for each threat. These databases contain generic information that can be applied to any software as long as it is modeled with the methodology of Section II.

As explained in Section II, the input to our approach are the DFDs of the Threat Modeling tool. In the first step, these diagrams are canonicalized to interpret the user-defined attributes. After this step, the identification trees of each attack pattern is matched with the canonical DFD, resulting in the identification of the potential threats. Then, the list of threats is risk sorted, and finally the countermeasures of minimum estimated cost to mitigate the most risky threats are calculated and issued as a report to the corresponding activity of the software development life-cycle. Each one of these steps is explained in more details below.

A. Data-Flow Diagram Canonicalization

It is aimed at mapping unknown user-defined labels to ones that our automated approach can interpret. To accomplish this, we used a data structure called MultiMap, which allows us to map a set of values to a single key, to build a *Canonicalization Table* that we use for this aim.

To interpret string attributes, our approach compares the developer defined value with those values corresponding to each possible key of its DFD element. If there is a successful map, then the label of the element is replaced by the mapped key. Otherwise, our approach shows engineers the different possible keys for that unknown DFD element and asks them if it corresponds to any of those. If so, the new value is added in the *Canonicalization Table*, and its mapping key is also assigned to the element of the DFD. It there were no possible mappings, a generic value would be assigned to the element, and it would be treated as a non-specific element.

B. Threat Identification

This is one of the most important steps of the analysis, as the further steps and final results will depend on the accuracy of this process. To identify the threats, we designed a set of trees that we call *Identification Tree.*

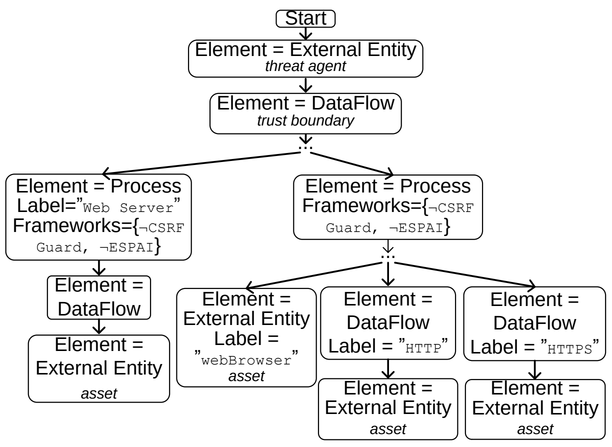


Fig. 3
Identification graph of CSRF threats.

Each branch of the tree represents a potential subgraph to be matched in the DFD. If matched, it means that the threat is relevant for the given design. Each node in the tree represents an element of the DFD and it might be required to have specific attributes or required not to have them for a successful match. In Figure 3, we show the *Identification Tree* of Cross Site Request Forgery (CSRF) threats.

We can see it is required to have some data crossing a trust boundary, known as attack surface. Also, the process that handles the HTTP request must not use specific frameworks against CSRF threats. If so, it would mean that the threat is mitigated and it would not be relevant to the system. This is represented by a key value pair, and the value has a "¬" symbol to indicate the negation in the match. In addition, it is determined in each match the *Threat Agent*, which is the component carrying out the attack, and also the *Asset*, which is the component compromised. Furthermore, it will be required in some nodes of the graph to have specific canonical labels corresponding to a web technology for a successful match.

C. Risk Ranking

In this step, it is computed the risk of each relevant threat to the system. In addition, the risk sorted threats are shown to the user and they are asked to set a threshold to eliminate the less risky threats based on their security policies. To compute the risk we adopted the definition of risk in Equation 1.

$$Risk = Likelihood \times Impact \tag{1}$$

The *Likelihood* of exploitation value is taken from the CAPEC security source, and the *Impact* is calculated as shown in Equation 2.

$$Impact = Asset \times ThreatAgent \times CIA\ Impact \tag{2}$$

The *Asset* is defined by the developers, as explained in Section I, and the *ThreatAgent* is the inverse of the *Asset*. If a component is a high value asset, it will be a low value threat agent, which is the component in the system that performs the attack. For instance, if the *Asset* is very high such as a database containing confidential information or an administrator of the system, it will also mean that we have a high trust in that component and the risk of suffering and attack from it is low. The possible values for the *Asset* are *High (1.2)*, *Medium (1.0)*, and *Low (0.8)*. For the *ThreatAgent* there are the same possible values. Finally, *CIA Impact* is computed as shown in Equation 3.

$$CIA\ Impact = Conf\ Imp + Int\ Imp + Avai\ Imp \tag{3}$$

The *Confidentiality*, *Integrity* and *Availability* Impact information is gathered from the attack patterns of CAPEC. Their value can be *High (0.33)*, *Medium (0.22)* or *Low (0.11)*. Therefore, the range of the risk value is approximately be between 0.0 and 1.0.

D. Mitigation Planning

In this step, our approach computes the software specifications of minimum estimated cost to mitigate the most risky threats, and generates a set of reports to guide engineers toward fixing those. Three reports are generated, one for the design activity of the development process that contains the architecture changes to avoid the threats, another one for the implementation phase, where the set of implementation details to avoid them are shown, and another for the testing step, where a set of actions to verify the correctness of the previous software specifications are indicated. To compute the countermeasures, we defined a new data structure that we called *Mitigation Tree*.

Attack Trees have been widely used by the community to represent attacks in a similar way as attack patterns do. Its root is the goal of an attacker, and each branch contains the set of actions that an attacker must carry out to achieve the goal of the root. We used the same idea but in a constructive way rather than a destructive one. The root of our *Mitigation Tree* is the goal of mitigating a determined threat, and each branch contains the set of software specifications or features, for the Design and Implementation activities, needed to accomplish the goal of the root. In addition, each feature contains an *estimated cost* of carrying it out. This information is stored in our attack pattern of the threat.

In Figure 4, we show the Mitigation Tree of CSRF attacks. It shows us that to mitigate CSRF threats we must first mitigate all Persistent Cross-Site Scripting (PXSS) and Reflected Cross-Site Scripting

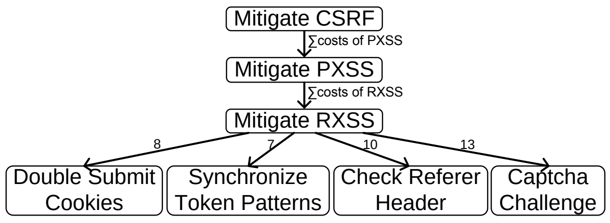


Fig. 4
Mitigation Tree of the CSRF attack pattern.

(RXSS) threats, and then we can choose among four subbranches that represent different software specifications. Each node or software specification of the attack tree has an estimated cost. This cost is calculated by using an *expert judgment* approach[13], where the security expert that builds the tree establishes a relative cost for each software specification using past experiences as criteria.

In this process, it is first asked engineers a set of polar questions corresponding to design decisions that are not shown in the DFD and are relevant from a security point of view. After this, the countermeasures of minimal estimated cost are computed by using the *mitigation trees*, and the results are shown to the engineers.

Since we push *security by default* and *security by design* it might be possible that these countermeasures degrade the usability of the system. It is also possible not to be able to carry out a determined specification due to business requirements. To avoid this, we ask engineers if it is possible to implement them. If so, the final reports are generated. Otherwise, the undesired software specification is eliminated and the countermeasures of minimal cost calculated again without taking into account the unwanted one.

There also exists the possibility that an engineer answers erroneously a question. As a result of this, it could be possible not to calculate the appropriate countermeasures. It is also possible to implement a feature wrongly. We have taken into consideration this human factor problem, and the actions of the *Verification* step are aimed to correct this potential problem: If the threat is not successfully mitigated in the early steps, it will be detected when carrying out the penetration testing actions of this step. The resulting software specifications plus a description of the attack, some examples and relevant references are issued to the corresponding step of the development cycle.

Determining whether a software specification corresponds to the design or implementation activities of the software development process is not trivial as the boundaries among them are not well-defined[9]. The criteria we used is the following one: If it can be modeled in UML it corresponds to the design activity, otherwise to the implementation one.

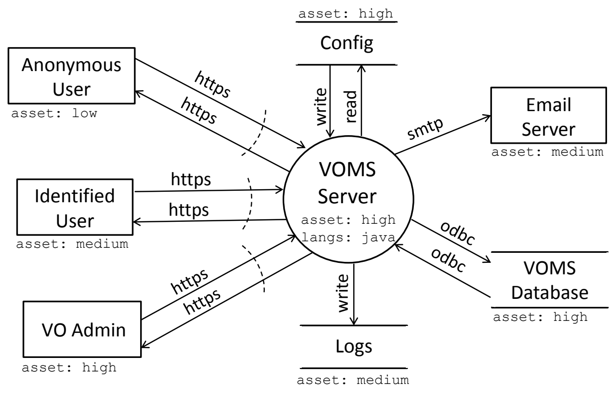


Fig. 5
Data-Flow Diagram of VOMS Admin.

IV. Experimental Results

In this section, we use VOMS Admin[14] as example an application. VOMS Admin is a grid middleware used to manage virtual organizations and user certificate attributes that will later be used by other middleware systems to take authorization decisions.

VOMS Admin was assessed using the manual *First Principles Vulnerability Assessment*[15] methodology, and several Cross-Site Request Forgery (CSRF), Persistent Cross-Site Scripting (PXSS), and Reflected Cross-Site Scripting (RXSS) vulnerabilities were found.

In Figure 5, we show the data-flow diagram of VOMS Admin that we built using the methodology described in Section II and the Threat Modeling tool, which is the only manual phase of the assessment, the security relevant operations and decisions are made automatically by our approach. Only little interaction with the engineers will be required to choose the desired level of security and the willing balance between security by default and usability.

A. VOMS Admin DFD Canonicalization

The first step of the process is to canonicalize the labels of the DFD to ones that our approach can interpret. There are some labels that our approach do not know how to map, then users are asked if they correspond to a known key. If so, it is added to the canonicalization table and to the DFD. Otherwise, a generic value is assigned to those. The resulting diagram is shown in Figure 6.

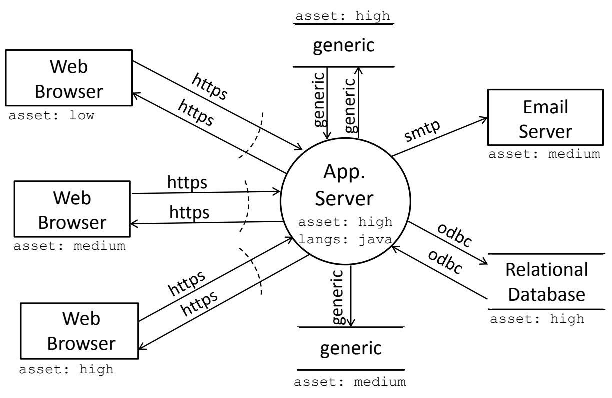


Fig. 6
Canonical DFD of VOMS Admin.

B. *VOMS Admin Threat Identification*

In this step, the subgraphs of our attack patterns are matched with the canonical DFD of Figure 6 to find the potential threats to the system. For reasons of space, we only considered CSRF threats. The resulting subgraphs matched are shown in Listing IV-B.

Listing IV-B: VOMS Admin Threat Identification

```
1. [CSRF] Path:{[VO Admin]−[App. server]−[VO Admin]}
2. [CSRF] Path:{[VO Admin]−[App. server]−[Ident. User]}
3. [CSRF] Path:{[VO Admin]−[App. server]−[Anon. User]}
4. [CSRF] Path:{[Anon. User]−[App. server]−[VO Admin]}
5. [CSRF] Path:{[Anon. User]−[App. server]−[Ident. User]}
6. [CSRF] Path:{[Anon. User]−[App. server]−[Anon. User]}
7. [CSRF] Path:{[Ident. User]−[App. server]−[VO Admin]}
8. [CSRF] Path:{[Ident. User]−[App. server]−[Ident. User]}
9. [CSRF] Path:{[Ident. User]−[App. server]−[Anon. User]}
```

C. *VOMS Admin Risk Ranking*

In this step, the previous threats are risk sorted and it is asked to engineers to define a threshold so that they can choose as much security as they want to. This is shown in Listing IV-C.

Listing IV-C: Risk Sorting and Purging Threats

```
4. Risk Value:[0.7128]
5. Risk Value:[0.6534]
7. Risk Value:[0.6534]
8. Risk Value:[0.5989]
6. Risk Value:[0.3564]
1. Risk Value:[0.3564]
9. Risk Value:[0.3267]
2. Risk Value:[0.3267]
3. Risk Value:[0.1782]

What threshold do you want?[0.0−1.0] 0.5

4. [CSRF] Path:{[Anon. User]−[App. server]−[VO Admin]}
5. [CSRF] Path:{[Anon. User]−[App. server]−[Ident. User]}
7. [CSRF] Path:{[Ident. User]−[App. server]−[VO Admin]}
8. [CSRF] Path:{[Ident. User]−[App. server]−[Ident. User]}
```

The two most risky threats correspond to an anonymous user attacking administrators and identified users respectively, the third one to an identified user attacking the administrator. On the other

Listing IV-D: VOMS Admin Mitigation Planning

```
 1  4.1 To refine results I will ask you a set of polar
        questions that I couldn't figure it out by reading
        the DFDs:
 2   *Are you Checking Refer Header in the requests of all
         HTML form actions to determine if it comes from
         a trusted domain?[y/n]n
 3   *Are you Synchronizing a Secret token pattern in all
         HTML form requests?[y/n]n
 4
 5  4.2 Pushed security by default, Refining to give the
        desired usability
 6  Answer [y/n] if it is OK to perform the following
        actions:
 7   *Is it OK to Synchronizing a Secret token pattern in
         all HTML form requests?[y/n]y
 8
 9   *Is it OK to Allowing only alphanumeric characters in
         all fields of this payload?[y/n]n
10
11  Computing best options....
12
13   *Is it OK to HTML Encode all user supplied data
        before displaying it back to the web interface?[y/n]
        y
14
15
16  FINAL COUNTERMEASURES COMPUTED
17   1. Checking Refer Header in the requests of all HTML
         form actions to determine if it comes from a
         trusted domain
18   2. HTML Encode all user supplied data before
         displaying it back to the web interface
19
20  # Reports available in out/report_desig.pdf, out/
        report_impl.pdf and out/report_verif.pdf
```

hand, the less risky threats are administrators attacking anonymous and identified users. By choosing a threshold of *0.5* we will ignore those.

D. *VOMS Admin Mitigation Planning*

As explained in Section III-D, the first step of this operation is to ask engineers polar questions regarding the design of their system to refine the results. This is shown in lines 1 to 4 of Listing IV-D. After this, the countermeasures of lowest estimated cost are computed, and it is asked engineers if it is possible to perform a determined software specification. If not, they are computed again until the willing balance between security and usability is reached. This is shown in lines 5 to 15 of Listing IV-D. Finally, the reports are generated in lines 16 to 20 of Listing IV-D.

It is interesting to notice that our approach has detected that to mitigate a CSRF threat is it first required to mitigate all PXSS and RXSS threats. For this reason, it asked in line 9 of Listing IV-D if it was possible to only allow alphanumeric characters in the HTML forms. Engineers answered "n" because special characters are required in some fields. Therefore, our approach re-computed the features again, resulting in a more permissive but also more expensive one, which is to HTML Encode user supplied data before showing it in the web interface.

V. Conclusions

In this paper, we addressed the problem of security expertise needed to perform risk assessments with the following contributions: We modeled a new data structure called *Identification Tree* that can be used to identify threats in software designs. We designed a new model to describe countermeasures of threats called *Mitigation Tree*, which classifies the set of software specifications that are required to mitigate a specific threat.

In addition, we designed a new approach that relies on the previous data structures to automate the security relevant operations of the software design risk assessment process. It uses the *Identification Trees* to find the potential threats of a given software model. It purges the less risky threats according to the desired policies. Finally, it uses the *Mitigation Trees* to compute the software specifications of minimum global estimated cost needed to mitigate the previous threats during the different activities of the development life cycle.

The resulting lowest cost features will be directly related to *security by design* but also to *security by default*. We allow companies to reach the willing balance between usability and security by default by asking engineers if the computed features are in good standing with their requirements. If not, they are re-computed again without taking into account the undesired ones.

At the end of this process, our approach issues three reports: one for the design activity, which contains the architectural modifications needed to be carried out in the system. Another one for the implementation phase, which contains implementation details to avoid the threats. In addition, a final one for the testing activity is issued, and it contains a set of actions that are needed to be carried out to verify the correctness of the previous features. The actions of the reports can be carried out by any developer, without having any security expertise, and the software design model used in the automatic analysis is compatible with the current Threat Modeling tool and methodology of Microsoft.

The experimental results of applying our automated approach to VOMS Admin show that it is affected by several RXSS, PXSS and CSRF threats. This software was manually assessed by security experts and several vulnerabilities of these types were found. Therefore, if our approach had been used during its development activities, these vulnerabilities would have been fixed early in its development at a lower cost, without requiring engineers to have security skills, and the grid infrastructure would not have been exposed to attacks.

Acknowledgment

This research has been supported by the MICINN Spain under contract TIN2007-64974 and the MINECO (MICINN) Spain under contract TIN2011-24384.

References

[1] F. Swiderski and W. Snyder, *Threat modeling*, Professional Series. Microsoft Press, 2004.

[2] Scott Swigart and Sean Campbell, “Sdl series, article 4: Threat modeling at microsoft,” 2008, Microsoft Press.

[3] Scott Swigart and Sean Campbell, “Sdl series article 7: Evolution of the microsoft security development lifecycle,” 2009, MSDN Magazine.

[4] Rune Fredriksen, Monica Kristiansen, BjÃ¸rn Gran, Ketil StÃ¸len, Tom Opperud, and Theo Dimitrakos, “The coras framework for a model-based risk management process,” in *Computer Safety, Reliability and Security*, Stuart Anderson, Massimo Felici, and Sandro Bologna, Eds., vol. 2434 of *Lecture Notes in Computer Science*, pp. 39–53. Springer Berlin / Heidelberg, 2002.

[5] Christopher Alberts, Audrey Dorofee, James Stevens, and Carol Woody, “Introduction to the octave Â® approach,” *Networked Systems Survivability Program*, vol. 5, no. August, 2003.

[6] Brenda Larcom Pail Saitta and Michael Eddington, “Trike v.1 methodology document [draft],” July 13th, 2005.

[7] M. J. A. Parkinson, Robert. Brennand, Andrew. Macleod, and Institute of Internal Auditors (Australia), *A guide to the use of AS/NZS 4360, risk management within the internal audit process / Michael J.A. Parkinson, robert Brennand, Andrew MacLeod*, SAI Global Ltd, Sydney :, 2004.

[8] “Common attack pattern enumeration and classification (capec) 1.6,” `http://capec.mitre.org/data/slices/2000.html`.

[9] Robert A. Martin and Sean Barnum, “Common weakness enumeration (cwe) status update,” *Ada Lett.*, vol. XXVIII, no. 1, pp. 88–91, Apr. 2008.

[10] The Open Web Application Security Project, ,” `http://www.owasp.org`, 2005.

[11] Michael Howard and Steve Lipner, *The Security Development Lifecycle*, Microsoft Press, Redmond, WA, USA, 2006.

[12] Adam Shostack Frank Swiderski, *SDL Threat Modeling Tool*, Microsoft, Redmond, WA, USA, 2011.

[13] M. Jorgensen, “Practical guidelines for expert-judgment-based software effort estimation,” *Software, IEEE*, vol. 22, no. 3, pp. 57 – 63, may-june 2005.

[14] *VOMS Admin*, European Middleware Initiative.

[15] James A. Kupsch, Barton P. Miller, Elisa Heymann, and Eduardo César, “First principles vulnerability assessment,” in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, New York, NY, USA, 2010, CCSW ’10, pp. 87–92, ACM.