Assessing the risk of using vulnerable components

Davide Balzarotti[1], Mattia Monga[2], and Sabrina Sicari[3]

[1] Politecnico di Milano
Dip. di Elettronica e Informazione - Piazza Leonardo da Vinci, 32
I 20133 Milan, Italy
[2] Università degli Studi di Milano
Dip. di Informatica e Comunicazione - Via Comelico, 39
I 20135 Milan, Italy
[3] Università di Catania
Dip. di Ing. Informatica e delle Telecomunicazioni
Viale Andrea Doria 6, I-95125 Catania, Italy

Abstract. This paper discusses how information about the architecture and the vulnerabilities affecting a distributed system can be used to quantitatively assess the risk to which the system is exposed. Our approach to risk evaluation can be used to assess how much one should believe in system trustworthiness and to compare different solutions, providing a tool for deciding if the additional cost of a more secure component is worth to be afforded.

1 Introduction

The issue of software security is increasingly more relevant in a world where most of our life depends directly on several complex computer-based systems. Today Internet connects and enables a growing list of critical activities from which people expect services and revenues. In other words, they *trust* these systems to be able to provide data and elaborations with a degree of confidentiality, integrity, and availability compatible with their needs. Unfortunately, this trust is often not based on a rational assessment of the risk to which the system could be exposed. Users tipically know only the interface of the system and, for example, they have too little information for evaluating the confidentiality of their credit card number: it could be even transmitted on an SSL armored link, but this does not help if on the other side it will be stored on a publicly available database! Surprisingly, the designers of the system are often in a similar situation. In fact, software systems are increasingly assembled from components that are developed by and purchased from third-parties and used as black boxes. Web services, for example, give to software engineers the ability of building complex applications by assembling third-parties components that expose a web interface[7], an extreme case of *components off the shelf* (COTS) software.

Thus, black box components make clear that nobody has enough information for evaluating how secure is every single computation. However, several public services exist (for example, BugTraq[1]) that publish known vulnerabilities of commercial components. The problem this paper wants to discuss is whatever this information can be used to assess how secure is a system built by assembling vulnerable components. In the

following we propose a quantitative approach to measuring risk based on the knowledge of:

- the vulnerabilities of components and links and a measure of their "exploitability".
- the logical dependencies that the architecture of the system induces among vulnerabilities, since it is often the case that a vulnerability can be exploited more easily by leveraging on another one.
- the envisioned attacks against the system.

Risk evaluation can be used to assess how much one should believe in system trustworthiness, but also– more interestingly– to compare different solutions. In fact, designers have often the option of using different components and different architectural choices. A quantitative risk assessment is key in providing a tool for deciding if the additional cost of a more secure component is worth to be afforded.

The paper is organized as follows: in Section 2 we describe our approach to evaluate the risk associated with a given architecture, in Section 3 we present an example of application, in Section 4 we discuss related work, and finally in Section 5 we draw some conclusions and sketch future work.

2 Our approach to risk assessment

The goal of risk assessment is to determine the likelihood that identifiable threats will harm, weighting their occurrence with the damage they may cause. An ideal risk assessment requires enumeration of all possible failure modes, their probability of happening and their consequences. Unfortunately, this information is rarely available in its gory detail and, when it is, it is very difficult to analyze it in order to draw sensible considerations.

We aim at both (1) reducing the complexity of risk analysis and (2) using information that can be managed, discussed, and agreed by high-level designers of a distributed system. For this reason we consider a distributed system as a composition of black-box elements communicating through directed links. We call *architecture* of the system the directed graph $< C, L >$ in which C is the set of all black-box components and L the set of all directed links. A link (c_1, c_2) means that c_1 may send input to c_2.

Moreover, we consider each element $\in (C \cup L)$ as *vulnerable.* A vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy [10]. The RFC definition adds also that

> "Most systems have vulnerabilities of some sort, but this does not mean that the systems are too flawed to use. Not every threat results in an attack, and not every attack succeeds. Success depends on the degree of vulnerability, the strength of attacks, and the effectiveness of any countermeasures in use. If the attacks needed to exploit a vulnerability are very difficult to carry out, then the vulnerability may be tolerable. If the perceived benefit to an attacker is small, then even an easily exploited vulnerability may be tolerable. However, if the attacks are well understood and easily made, and if the vulnerable system is

employed by a wide range of users, then it is likely that there will be enough benefit for someone to make an attack."

As stated by Howard and Le Blanc[11]: "You cannot build a secure system until you understand your threats". Therefore, in order to assess the trustworthiness of a system (or, dually, its risks), one has to identify possible threats and how attacks could be performed. Obviously enough, the risk of an unforeseen threat cannot be positively assessed and unknown attacks fall outside a systematic analysis of risks. Similarly, in the following we consider only *known vulnerabilities,* however it is possible to apply our approach even to *unknown vulnerabilities* (or a mix of known and unknown ones) if their nature is predicted.

Safety engineering has a long tradition of using *fault trees* or *event trees* to analyze hazards in complex systems[15]. A similar approach it is commonly used also in information technology. Attack trees[17, 6] provide a formal, methodical way of describing how an attack can possibly be performed against a system. Basically, one represents attacks in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes. There are `and` nodes and `or` nodes. `or` nodes are alternatives; `and` nodes represent different steps toward achieving the same goal. The ultimate objective in building an attack tree is identifying how vulnerabilities can be exploited to harm a system, therefore the basic leaves represent system vulnerabilities. However, these are often dependent one on another, but this information is partially lost in attack tree representation. In fact, only structural dependencies are made explicit (i.e., the attack has a given structure and implies the exploitations of some vulnerabilities), while indirect dependencies (i.e., a vulnerability might ease an attack, even if the attack is possible without its presence) are neglected. Therefore, we propose to take into account all vulnerabilities dependencies and we devise an analytical approach for computing the risk associated to a specific threat (described by an attack tree) starting from the assessment of the exploitability of vulnerabilities. Moreover, our analysis starts from the architecture of the system, since we found that most (but not all) of the dependencies among vulnerabilities stem from the basic topology of the system.

2.1 Measuring risk

Risk is measured by means of a function of two variables: one is the damage potential of the hazard (H) and another one is the level of exploitability (E) by which we consider the difficulty to make an attack. Damage potential can be defined as the average loss of money an attack may cause, but any sensible numerical measure can be used in our approach.

The meaning that we give to the term exploitability, E, is a general value that includes both the exploitability and reproducibility of an attack, defined in the STRIDE/-DREAD theory[11]. At the same time we also attribute to damage potential (H) the meaning of total damage taking into account also the number of affected users.

$$Risk = f(H, E) \tag{1}$$

We want to evaluate the total risk of a threat described by an attack tree.

Our approach consists of four steps:

- **At step 1**: A threat to the system under examination is modelled by using an attack tree. The attack objective is the root node and children nodes represent different ways of achieving it. Children can be alternative (`or` subtrees) or needed jointly (`and` subtree). The final leaves of the tree are *potential* vulnerabilities of the system that should be matched with the *actual* known vulnerabilities. To each vulnerability v is associated a numerical index E, called *exploitability*, which measures how probable is that v will be exploited to perform a successful attack. Evaluation of E can be quite approximate: in order to apply our computation it is sufficient that the partial order of indexes among dependent vulnerabilities (see below) reflects the relative difficulty of exploitation. In fact, further calculation are based only on maximum and minimum operations and no complex arithmetics will be applied. However, to compare two different risk evaluations (possibly with respect to two different systems), the same scale should be used and a total order among exploitability indices is needed. A meaningful assessment of E is a matter of both experience and ingenuity, but as far as a single analysis is concerned only *relative* ease of exploitability has to be estimated, a judgement on which people often agree.
- **At step 2**: We introduce dependencies among identified vulnerabilities. A vulnerability A depends on a vulnerability B if and only if when B was already exploited, then A is easier to be exploited. Dependencies should be analyzed by taking into account context, architectural and topological information.
- **At step 3**: The index E of each vulnerability is updated taking into account mutual dependencies, according to the algorithm described in Section 2.2. since each vulnerability could be exploited *thanks to the previous exploitation of one of the vulnerabilities on which it depends*.
- **At step 4**: The risk associated to the threat under examination is finally computed by recursively aggregating exploitabilities along the attack tree. The exploitability of an `or` subtree is the easiest exploitability of children, and the exploitability of an `and` subtree is the most difficult exploitability of children. The aggregated exploitability measures the level of feasibility of the attack and can be combined with the damage potential (H) to assess the risk of the threat.

2.2 Exploitability of dependent vulnerabilities

Consider the system depicted in Figure 1. We will use this simple example to show our approach to risk assessment. The system can be described as a graph $S =< C, L >$ where $C = \{P, Q, R\}$ is the set of *components* and $L = \{(P, Q), (Q, R), (R, Q), (R, P)\}$ is the set of *links* between components. A number of flaws affecting the software composing the system is known: let's them form the set $F = \{p_1, q_1, q_2, r_1, x_1, y_1, z_1, z_2\}$. Components are exposed to the set of vulnerabilities $V_C = \{(P, p_1), (Q, q_1), (Q, q_2)(R, r_1)\}$, where an element (v, ν) means that the component v is susceptible to be subverted thanks to the flaw ν. Links are exposed to the set of vulnerabilities $V_L = \{((P, Q), x_1), ((Q, R), z_1), ((Q, R), z_2), ((R, P), y_1)\}$, where an element (v, ν) means that the link v is susceptible to be subverted thanks to the flaw ν. Since link z is bidirectional, z_1 and z_2 affect also (R, Q), however it is not useful to take into account them twice. The set of all vulnerabilities is $V = V_C \cup V_L$. To ease notation, we denote $element(\nu) \in C \cup L$ the element of S to which the vulnerability ν applies.

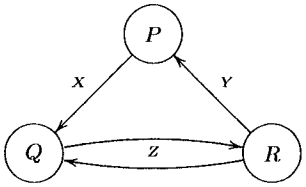

Fig. 1. System architecture

Initially, one has to assess how easy and repeatable is to exploit every single vulnerability to gain control of a component or a link in the given architecture. We call this the *exploitability* $E_0(\nu)$ of the vulnerability ν in the system S.

$$\forall \nu | (\upsilon, \nu) \in V \quad \text{assess} \quad E_0(\nu)$$

$$E : V \mapsto \mathbf{N}$$

where $\mathbf{N}$ is a total ordered set of degrees of exploitability; we will use $\mathbf{N} = \{x | 0 \leq x \leq 10\}$ where 0 means "not exploitable at all". This evaluation will be driven by the knowledge we have about the vulnerability itself and the constraints the architecture imposes on its exploitability. In fact, when a component or a link is part of a complex system, its vulnerabilities are typically more difficult to be exploited compared to the case when one has the total control of it.

However, the architecture of the system imposes dependencies among vulnerabilities. For example, we need to understand if it is easier to exploit a vulnerability of a component given that an input link attached to it was already compromised or a component attached to any of its input links was already compromised. Dependencies among vulnerabilities can be represented as a new graph $G =< V, D >$. We denote with $E(\alpha|\beta)$ the exploitability of α given that β was already exploited. The edge $(\beta, \alpha) \in D$ if $E(\alpha|\beta) \geq E_0(\alpha)$, i.e., if it is easier to compromise $element(\alpha)$ when one has compromised $element(\beta)$

$$\forall \nu, \alpha \in V \wedge \nu \neq \alpha \ : \ \text{assess } E(\nu|\alpha)$$

1 (Complexity)
The number of the exploitabilities to assess is $\leq |V|^2$ *In fact, every vulnerability needs an exploitability evaluation (*$|V|$ *figures needed). Moreover, the graph* G *has at most* $|V| \cdot (|V| - 1)$ *edges.*

Thus, in general one has to assess $|V|^2$ exploitabilities. However, most of the vulnerabilities are usually independent, and the numbers one has to *guess* is typically closer to $|V|$ than $|V|^2$. Moreover, in the following it will be clear that *only ordering is important,* i.e. absolute values of exploitabilities have no meaning: it is only a convenient

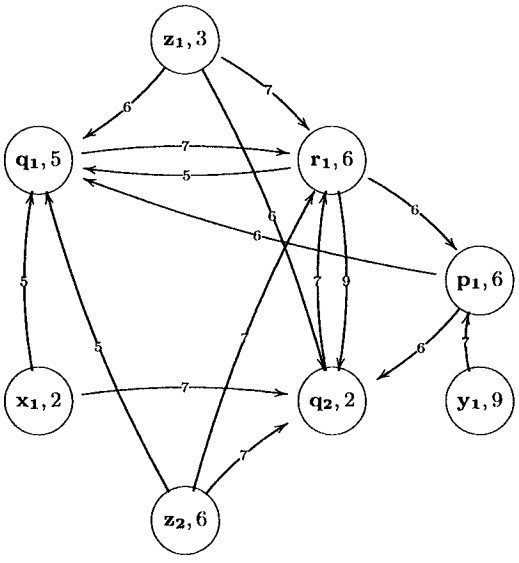

Fig. 2. Dependencies graph among vulnerabilities

way to express the relative easiness of acquiring control of an element thanks to one of them. Figure 2 shows an exploitability assessment for the example system: the dependencies among $|V| = 8$ vulnerabilities impose the assessment of 24 exploitabilities. The number associated to each node is E_0, that is the initial measure of how difficult is to exploit the vulnerability. The conditional exploitabilities are represented by the numbers on the edges. The assessment depicted in Figure 2 does not take into account that each vulnerability could be exploited *thanks to the previous exploitation of one of the vulnerabilities on which it depends*. Therefore, E_0 should be iteratively updated by considering the easiest (i.e., the maximum) way of exploiting an incoming vulnerability in the dependencies graph. In turn each incoming vulnerability could be exploited by controlling the affected element or leveraging on the dependency itself: the most difficult (i.e., the minimum) constraints the value.

$$\forall \nu \in V, (\nu, \gamma) \in D : E(\nu) = max(E_0(\nu), min(E(\nu|\gamma), E(\gamma))) \quad (2)$$

Our methodology consists in iteratively applying the previous formula for each vulnerability, until the system converges to an equilibrium. Table 1 shows a possible sequence of iteration and the corresponding equilibrium.

2 (Convergence) *At each iteration the exploitability can only be updated with a greater value. Moreover, it is upper bounded by the maximum value of the incoming dependencies edges. Therefore no "oscillations" are possible and the algorithm always converges.*

3 (Order) *Only the relative order of exploitability values is important: in fact, only max and min operators are used in our formula, and no arithmetical functions are ever applied.*

	E_0	E_1	E_2
p_1	6	**7**	7
q_1	5	**6**	6
q_2	2	**6**	6
r_1	6	6	6
x_1	2	2	2
y_1	9	9	9
z_1	3	3	3
z_2	6	6	6

Table 1. Exploitability update

Risk assessment could be effectively used to evaluate design choices. For example, making links not exploitable at all (by protecting them with logical and physical defenses) would virtually change nothing.

Our approach can also be used to evaluate the impact of adding a new vulnerable component to a preexisting system. In fact, due to the presence of new dependencies between vulnerabilities, the new component can affect the security of the whole system, increasing the exploitability of some of the old vulnerabilities.

3 An example

In this section we introduce a numerical example based on an hypothetical Insecure Airlines web site. For the sake of simplicity we maintain the same simple architecture represented in fig.1.

According with the new airline scenario, Node P represents the company web server, node Q represents the database containing the flights information, and node R is a web service that manages the frequent flier accounts. Links X and Z connect the web server to the database and the frequent flier services respectively. Link Y allows some automatic script on the database to update the mileage of a customer account.

We associate the following vulnerabilities to the system components:

V_1 **(node P)** SQL injection. An authenticated user can submit a malicious query that allows him to read or modify any row in the database.

V_2 **(node Q)** Buffer Overflow. The CGI page that loads and displays the flight information copies the flight number into a small static buffer without checking for possible buffer overflow.

V_3 **(node Q)** A race condition in a local command allows an attacker to read any file in the web server machine.

V_4 **(node R)** Weak authentication. The access to each frequent flyer account is protected by a numeric PIN of 4 digits.

The threat that a malicious user could sniff[1] the traffic between two components is represented introducing three more vulnerabilities: V_5 (for X link), V_6 (for Y link) and V_7 (for Z link).

The airline company is interested in evaluating the risk that an external user (not a company employee) can add a fake flight reservation. The security analyst starts enumerating all the possible attacks and combining them to form a large attack tree. Fig 3 reports a piece of the tree in outline form.

```
Goal: Fake Reservation
1. Convince an employee to add a reservation
 1.1 Blackmail an employee
 1.2 Threaten an employee
2. Access and Modify the flight database
 2.1 SQL Injection from the web page (V1)
 2.2 Log into the database
  2.2.1 Guess the password
  2.2.2 Sniff the password (V7)
  2.2.3 Steal the password from the Web-Server machine (AND)
   2.2.3.1 Get an account on the Web-Server
    2.2.3.1.1 Exploit a buffer overflow (V2)
    2.2.3.1.2 Get access to an employee account
   2.2.3.2 Exploit a race condition to access a protected
           file (V3)
```

Fig. 3. Attack Tree

The next step consists in assigning the exploitability values of each vulnerability. The following table summarizes the values and the dependencies between each vulnerability:

Vuln.	E_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	2		-	10	-	10	-	-
V_2	0	5		-	-	-	-	-
V_3	0	-	3		-	-	-	-
V_4	4	8	-	10		-	7	10
V_5	7	-	-	-	-		-	-
V_6	7	-	-	-	-	-		-
V_7	7	-	-	-	-	-	-	

We do not have enough space to justify the choice of every values in the table, but in order to provide an idea of what is behind the numbers, we can consider the case of V_2. The second column represents $E_0(V_2)$, that is the exploitability of V_2 given that none

[1] We do not consider spoofing and man-in-the-middle attacks in order to do not complicate the example.

of the other vulnerabilities have been previously exploited by the attacker. In our case the value is zero. In fact, it is not possible for a malicious user to directly exploit the buffer overflow since the input the attacker should manipulate comes directly from the flight repository. For this reason, if the attacker would be able to insert a malicious row into the database, he could then force the web server to display that information taking control of the machine. This dependence is shown in the third column: $E(V_2|V_1) = 5$.

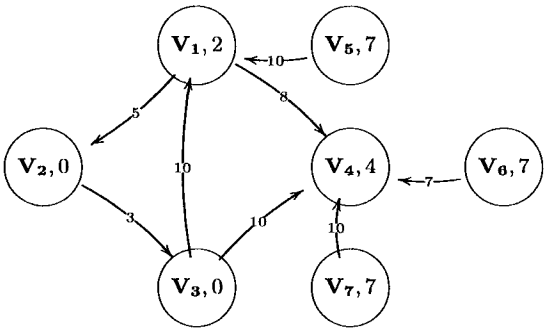

Fig. 4. Vulnerabilities Dependence Graph

Figure 4 shows the *Vulnerability Dependence Graph* representation of our system. Applying our algorithm to the graph, after a couple of iterations, the system converges to the following fixed point:

$E(V_1) = 7$
$E(V_2) = 5$
$E(V_3) = 3$
$E(V_4) = 7$

This result can seem obvious due to the simplicity of the example but in a real scenario that can involve dozen of components, also for a skilled user can be very difficult to figure out all the possibles chain of attacks just looking at the graph. Moreover, it is possible to see how the presence of a vulnerability in a branch of the attack tree can affect the exploitability values associated to leaves belonging to a different branch of the tree.

Anyway, the evaluation of the risk in a distributed environment is just the first step in a more complicate and interesting process. In fact, one of the main purpose of our approach is to allow user to locate, analyze, and compare the impact of security solutions on the whole system under analysis.

In the case of Insecure Airlines, a security manager can propose different solutions in order to mitigate the total risk of the system. Since security solutions are usually expensive, it is very important to reduce any possible waste of money. For this reason the possibility to quickly simulate and explore the impact of multiple actions allows the

user to choose the right solution in order to guarantee a good security level according to business requirements.

In our example, the only way to log into the database is by knowing the password (that is stored into the web server host). The file containing the password can be read thanks to the race condition vulnerability present in one of the programs installed on the host. Suppose the security manager proposes the following possible solutions:

- *Solution A*: Update the vulnerable program with a more secure one.
- *Solution B*: Fix the buffer overflow vulnerability. So, no one can have the chance to perform the race condition attack.
- *Solution C*: Encrypt the communication between the web-server and the database to make a sniffing attack much more difficult.

Translating these three solutions in numbers, the first is equivalent to setting V_3 and its dependencies to zero, the second to setting V_2 and its dependencies to zero, and the last one to setting V_5 to one.

Running again our algorithm in the three different scenarios, we obtain the following results:

Scenario	V_1	V_2	V_3	V_4
Base	7	5	3	7
Solution A	7	5	0	7
Solution B	7	0	0	7
Solution C	2	2	2	7

The previous table shows that the first solution does not affect the rest of the system. The second solution makes the system more secure since it removes the possibility to exploit V_3. Nevertheless, an attacker can still exploit V_1 modifying the database at his will. The third solution seems the better one, since it makes very hard to exploit three of the four initial vulnerabilities.

Of course, in order to decide if a solution is worthwhile or not, it is necessary to propagate the exploitability values from the leaves to the root of the attack tree. In such a way a security analyst can evaluate what is the real danger and which solution is more appropriate to mitigate it.

4 Related work

Risk, trust, security requirements mapping, and component interdependence are concepts that are linked together and have been widely discussed in literature.

Baskerville [3] describes the evolution of different methods to measure risk that sometimes could be used together to improve the result accuracy. Even though software security risk is extensively discussed in risk management methodologies [20, 5, 2], among information security experts there appears to be no agreement regarding the best or the most appropriate method to assess the probability of computer incidents [18].

We started our investigation analyzing the STRIDE/DREAD theory [11] and proposing a simplified way to combine together the assessment values. We then took into account the problem of risk aggregation that represents a key point to enable modular reasoning in distributed environments that involve multiple and heterogenous components.

O.Sami Saydjari et al.[9]present a system security engineering methodology for discovering system vulnerabilities, and determining what countermeasures can best close those vulnerabilities.Their approach improves the process "*analyzing IS through an adversary's eyes*".

Evaluation and analysis of vulnerabilities in isolation is insufficient because it is important to consider the effects of the interactions among vulnerabilities. There are many approaches for taking into account vulnerability dependencies [9, 13].Using a graph representation to model security-related concepts is not a new approach. For instance, attack graphs [19, 14] use state-transition diagrams to describe complex attacks that can involve multiple steps. Different techniques, such as model checking [19], can then be applied to attack graphs in order to evaluate security properties. The goal of our vulnerability dependence graph is different since we only need to describe the relationships among vulnerabilities in order to improve information obtained by the attack tree model [6, 17].

Software components have received a great deal of interest from both industries and academia as the component based software development paradigm promises maximum benefits of component reusability and distributed programming. A software component is independently developed and delivered as an autonomous unit that can be composed to become part of a lager application. The component interdependence is often ignored or overlooked [4] leading to incorrect or imprecise models. In order to avoid this problem, one must specify more complete models taking into account interconnections among system components. In agreement with this point of view [8, 18, 4, 9, 16] present models for assessing security risks taking into account interdependence between components.

Even though there is no easy way to assess risks and choose the damage values, there are various approaches that provide methodologies by which the risk evaluation can be made more systematic. In particular, Sharp et al.[18] develop a scheme for probabilistic evaluation of the impact of the security threats and proposes a system for risk management with the goal of assessing the expected damages due to attacks also in terms of the cost. Z. Dwaikat et al.[8] define security requirements for transactions and provide mechanisms to measure likelihood of violation of these requirements. Unlike us, the authors base the evaluation of risk on transaction traces combining security requirements, context information and risks presented by various components. K.Khan et al [12] propose a framework to characterize compositional security contracts of software components.

At the same time, there is a need to automate the modeling phase in the risk assessment and analysis process. G. Biswas, et al. [4] proposed the use of qualitative modeling techniques based on deriving behavior from structural descriptions and causal reasoning to aid automating and enhancing the risk analysis. Hierarchical schemes are used for describing component structure and system functionality is derived from a set of

primitive functions and parameters defined for the domain. The authors want to (1) incorporate uncertainty analysis using probabilistic schemes or belief functions for estimating risk probabilities, and (2) use causal reasoning and qualitative modeling for consequence analysis. We introduced an automatic evaluation of the total exploitability of each vulnerability that will then influence the value of total risk. In agreement to [4, 9, 16] the information computed by the model to calculate the risk could be used as effect analysis and decisional support.

5 Conclusions

Risk analysis of large distributed systems is still a hard problem for security managers since it requires a perfect balance of skills, experience, and "black magic" to be solved.

This paper presents a quantitative approach to evaluate risk in a distributed environments based on the knowledge of the system architecture and the list of vulnerabilities of links and components.

The choice of dividing the analysis into four steps simplifies the study of the problem allowing the security designer to acquire and manipulate risk information step by step in an incremental way.

Starting from an attack tree we build a Vulnerability Dependencies Graph that emphasizes the possible dependencies among vulnerabilities/leaves. In this way we point out the dependencies among system vulnerabilities that can be lost in an attack tree representation and that can make the system more vulnerable. We then propose an equilibrium condition that can be iteratively applied to propagate exploitability values from one node of the graph to the others.

Even though the number of values that must be initially assigned to each vulnerability can be fairly high, we strongly belief that our system simplify the risk analysis process. In fact, since we never use any arithmetic operation to combine exploitabilities, we only requires (and preserve) that the initial values respect some kind of ordering criterion.

Finally, our algorithm can be used to automatically evaluate different security solutions, enabling a security manager to perform a "what if" analysis in order to analyze the impact of a local modification on the security of the whole system.

We are currently experimenting by applying our approach on real world examples, in particular focusing on systems based on web services. In principle, our approach is independent from the level of abstraction one uses to analyze a system, thus we are planning to extend our analysis to the relationship between hierarchical assessments.

References

1. http://msgs.securepoint.com/bugtraq/.
2. Christopher Alberts, Audree Dorofee, James Stevens, and Carol Woody. Introduction to the Octave approach. http://www.cert.org/octave/, 2003.
3. Richard Baskerville. Information system security design methods: Implications for information systems development. *ACM Computing Survey*, 25(4):375–412, 1993.

4. Gautam Biswas, Kenneth A. Debelak, and Kazuhiko Kawamura. Application of qualitative modelling to knoweledge-based risk assessment studies. *IEA/AIE '89: Second International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems-ACM*, pages 92–101, 1989.
5. B.Jenkins. Risk analysis helps establish a good security posture; risk management keeps it that way. *whitepaper*, pages 1–16, 1998.
6. CERT. Technical report 2001-tn-001.
7. Harvey M. Deitel, Paul J. Deitel, B. DuWaldt, and L. K. Trees. *Web Services: A Technical Introduction*. Prentice Hall, 2002.
8. Zaid Dwaikat and Francesco Parisi-Presicce. Risky trust: Risk-based analysis of software system. *Proceedings of the first workshop on Software Engineering for Secure Systems (SESS05)*.
9. S. Evans, D. Heinbuch, E.Kyle, J. Piorkowski, and J.Wallener. Risk-based system security engineering: Stopping attacks with intention. *IEEE Security & Privacy*, pages 59–62, 2004.
10. Network Working Group. Internet security glossary. http://rfc.net/rfc2828.html, May 2000. Request for Comments: 2828.
11. Michael Howard and David Leblanc. *Writing secure Code*. Microsoft Press, 2003.
12. Khaled Khan, Jun Han, and Yuliang Zheng. A framework for an active interface to characterize compositional security contracts of software components. *2001 Australian Software Engineering Conference (ASWEC01)-IEEE Computer Society Press*, pages 117–126.
13. I.S. Moskowitz and M.H. Kang. An insecurity flow model. *Proceedings of New Security Paradigms workshop*, 1997.
14. S. Noel, S.Jajoidia, B.O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. *Proceedings of ACSAC'03*.
15. M. Elisabeth Pate-Cornell. Fault tree vs. event trees in reliability analysis. *Risk Analysis*, 4(3):177–186, 1984.
16. Mehmet Sahinoglu. Security meter:a pratical decision-tree model to quantify risk. *IEEE Security & Privacy*, 3(3):18–24, May/June 2005.
17. Bruce Schneier. Modelling security threats. *Dr. Dobb's Journal*, dec 1999.
18. Gunter P. Sharp, Philip H. Enslow, Shamkant B. Navathe, and Fariborz Farhmand. Managing vulnerabilities of information system to security incindets. *ACM International Conference:5th international conference on Electronic commerce*, pages 348–354, 2003.
19. O. Sheyner, J. Haines, S.Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02)*.
20. Thomas Siu. Risk-eye for IT security guy. *Gsec*, pages 1–20, 2004.