Automatic discovery of software attacks via backward reasoning

Automatic discovery of software attacks via backward reasoning

Cataldo Basile*, Daniele Canavese*, Jerome d'Annoville†, Bjorn De Sutter‡ and Fulvio Valenza*
* Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy
Email: {cataldo.basile, daniele.canavese, fulvio.valenza}@polito.it
† Gemalto, Technology & Innovation, Meudon, France
Email: jerome.d-annoville@gemalto.com
‡ Computer Systems Lab, Ghent University, Ghent, Belgium
Email: bjorn.desutter@elis.ugent.be

***Abstract*—Security risk management and mitigation are two of the most important items on several companies' agendas. In this scenario, software attacks pose a major threat to the reliable execution of services, thus bringing negative effects on businesses. This paper presents a formal model that allows the identification of all the attacks against the assets embedded in a software application. Our approach can be used to perform the identification of the threats that loom over the assets and help to determine the potential countermeasures, that is the protections to deploy for mitigating the risks. The proposed model uses a Knowledge Base to represent the software assets, the steps that can be executed to mount an attack and their relationships. Inference rules permit the automatic discovery of attack step combinations towards the compromised assets that are discovered using a backward programming methodology. This approach is very usable as the attack discovery is fully automatic, once the Knowledge Base is populated with the information regarding the application to protect. In addition, it has been proven highly efficient and exhaustive.**

I. Introduction

Nowadays, protecting the assets embedded in the software is one of the most challenging tasks in our technology-centric world. Examples of software assets are cryptographic keys that, if extracted, may be used to gain illegitimate access to services, impersonate users, emulate the application and abuse of its algorithms and protocols (that may contain intellectual properties).

Guaranteeing that the attackers cannot extract the assets is nearly unfeasible as in unprotected applications the attackers have white-box access to them. They have at their disposal a wide range of so called Man-At-The-End (MATE) attacks, which can be mounted against software stored and running on a device under the full control of the attacker himself. In MATE scenarios, the attackers have both the maximum system knowledge and a very high degree of freedom. They have the binaries of the target application and can execute static and dynamic tools that allow a better understanding of the code (e.g. by reconstructing higher level representations) and its manipulation. Companies are making huge monetary and resource investments to protect their software assets because successful MATE attacks can cause high losses of money and reputation for the firm.

To protect the software, companies have to perform some tasks that fall under the umbrella of the risk analysis. All threats need to be identified and their impact on the software assets must be estimated to permit the decision of how to mitigate the risks, that is to determine the protections to deploy. Hence, the first challenge to face is the threat identification phase, that is, finding all the feasible attacks against the assets. Discovering in a (quasi-)exhaustive way all the attacks that loom over the assets is a (potentially) very long and complex job, thus error prone, especially if done manually or with minimal automatic support.

In this paper we propose a formal model which is able to automatically determine the attacks against the assets. In our approach, a Knowledge Base consist of a set of facts about the software assets, the attack steps and their relationships. A number of inferences permits the definition of valid combinations of attack steps, named attack paths, that describe the sequence of activities an attacker has to perform for compromising one or more assets. The actual attack paths identification is performed using an automatic backward programming methodology, thus strongly easing the attack identification phase. Once the Knowledge Base is populated with the information about the application to protect, the attack path discovery phase is fully automatic and requires no human intervention at all. Furthermore, it is highly efficient (it can found thousands of attack paths in few seconds) and it is (theoretically) exhaustive, by finding all the attack paths that can be executed using the steps modeled in the Knowledge Base.

To the best of our knowledge, this is the first approach that uses of logical inferences to build the threat model of an application and to discover the attack paths. However, some analogous counterparts exists, but they make use of different approaches and usually are not specialized for software risk analysis. For instance, Ekelhart *et al.* in [3] propose an ontology-centric method for generic security risk management, while Dahl *et al.* in [1] use Petri nets for modeling (but not discovering) multi-agent and multi-stage attacks.

This paper is structured as follows. Section II briefly introduces the terminology that we will use through the rest of the paper. Section III presents a motivating example which we will

use to discuss and introduce our approach, while Section IV discuss the formal model which is the core of our approach. In Section V we show and discuss the results of the Prolog-based implementation of or approach. Finally, Section VI contains a list of related works and Section VII presents our conclusions and the future work.

II. Background

Before discussing in detail our approach and providing a motivating example, we define here the terminology that will be used through the rest of the paper.

A generic attack usually is performed by executing a series of *attack step*s in a specific order to achieve a particular goal, which is usually the disruption of a security property for a particular software asset (e.g. a function, a code snippet or a variable). We can identify two main security properties:

- *confidentiality*. A confidential asset is broken if the attacker is able to read its content (e.g. the attacker was able to locate in the code a function/code snippet);
- *integrity*. The integrity of an asset is compromised when the attacker can freely modify its content (e.g. the attacker can alter the value of a specific variable at will during the run-time).

Given any two attack steps, they can related by:

- a *sequential* relationship, that is an attack step must executed before (or after) another one;
- a *concurrent* relationship, that is two attack steps can be executed in any order (or at the same time, if possible).

A traditional way to represent an attack and its steps is through the mean of a graph such as the one depicted in Fig. 1.

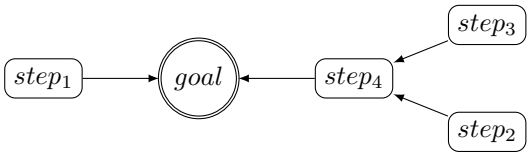

Fig. 1. An example attack graph.

The sequentiality is represented as a successor/predecessor relationship between two graph nodes, otherwise we have a concurrency relationship (for instance between siblings). For example, $step_3$ and $step_4$ are sequential while $step_1$ and $step_2$ are concurrent.

We name *attack path* an ordered sequence of attack steps that is used to mount a particular attack. For instance, the attack paths represented in Fig. 1 are:

1) $(step_1, step_2, step_3, step_4)$;
2) $(step_1, step_3, step_2, step_4)$;
3) $(step_2, step_1, step_3, step_4)$;
4) $(step_3, step_1, step_2, step_4)$;
5) $(step_2, step_3, step_1, step_4)$;
6) $(step_3, step_2, step_1, step_4)$;
7) $(step_2, step_3, step_4, step_1)$;
8) $(step_3, step_2, step_4, step_1)$.

Graphs are a compact way to represent a set of attack paths, but their simplicity can be misleading since finding all the attack paths by visual inspection is very hard and clearly unfeasible when the number of nodes increases.

III. OTP generator: a case study

For the sake of clarity, we will first introduce our approach by analyzing a concrete example. We will start by informally discussing the scenario depicted in Fig. 2, then we will show how to model it in a formal way and how to use such data to infer a number of attack paths.

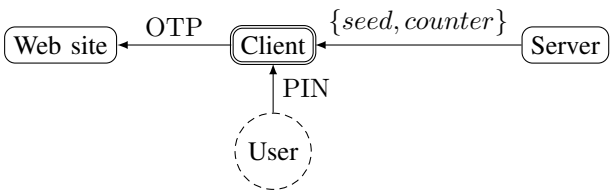

Fig. 2. The OTP generator scenario.

In this example we have a client device that contains an OTP (One-Time Password) generation application. These disposable passwords can be used to log in to some service, for instance an on-line banking web site or an e-mail account.

When the OTP generator is installed and runs for the first time, it automatically connects to a server that sends the client two (random) values (a $seed$ and a $counter$). To increase the security of the system, these two values are sent encrypted so that only the client can internally decrypt them. After that, whenever the client wants to generate an OTP, a PIN (Personal Identification Number) is asked to the user for blocking the access to unauthorized users. If the PIN check is passed, an OTP is produced by using the values of $seed$ and $counter$. After that, the $counter$ value is updated and the system is ready again to produce another password.

In order to simplify the scenario and the formulas we then assume that the following facts hold:

- the algorithm which generates the OTPs is designed in a such way that it is impossible to predict the next OTP by knowing the current password (and all the previous ones);
- the attacker has a very high degree of expertise (which in some real cases might not be true);
- the attacker has at his disposal every tool that he might need (e.g. debuggers, memory scanners, ...).

Since we are dealing with a MATE attack scenario, the attacker has the full source code of the client at his disposal. The ultimate goal of the attacker, in this case, is to be able to use the OTP generation function every time he wants, thus being able to impersonate a user from the web site point-of-view. In order to achieve this, the attacker must be able to accomplish three goals:

G_1) avoid the PIN-based authentication, needed to access the OTP generation function;
G_2) know the value of $counter$ and how it is updated;
G_3) know the value of $seed$.

A. Goal G_1: circumventing the PIN authentication

Since the OTP generation function is PIN-protected and the attacker does not know it, he has to avoid somehow such authentication phase.

A way to do it is to steal the PIN code from some legitimate users (sub-goal G_{1a}). One way to gain this knowledge is to inject some malicious code in a victim's client application so that it will send the PIN to a server controlled by the attacker. Implementing this strategy requires several attack steps:

- ⟨1⟩ a server where to store the PINs must be set up and a suitable communication protocol must be designed;
- ⟨2⟩ the PIN verification function $authF$ must be located in the code;
- ⟨3⟩ the $authF$ function must be modified in order to send the PIN to the server.

An alternative to the previous approach is to bypass the PIN verification function (sub-goal G_{1b}), thus performing the following steps:

- ⟨4⟩ the PIN verification function $authF$ must be located in the code;
- ⟨5⟩ the $authF$ function must be statically modified in order to skip the check;
- ⟨6⟩ or the $authF$ function must be dynamically modified in order to skip the check.

B. Goal G_2: obtaining the counter and its update code

Knowing the value of $counter$ is not enough for achieving the goal G_2 since its value is meaningful only for the current OTP. In order to guess all the next passwords also the counter update code must be known.

To gain these current value of $counter$ the following steps are necessary:

- ⟨7⟩ the OTP generation function $otpF$ must be located in the code;
- ⟨8⟩ the current $counter$ value must be identified, by performing a dynamic analysis of the $otpF$ function.

And to gain the knowledge of the counter update code (sub-goal G_{2b}), the attacker has to:

- ⟨9⟩ the OTP generation function $otpF$ must be located in the code;
- ⟨10⟩ the counter update code in $otpF$ must be located. This can be done statically by inspecting the code by searching some well-known patterns such as chains of binary or arithmetic operations.

C. Goal G_3: obtaining the seed

The last goal of the attacker is to know the value of the $seed$. He can do this using two different attack strategies. In the first one, the attacker can observe the application during the OTP generation and find the location of $seed$ in the memory. In the second one, the attacker can try to intercept the $seed$ during the initial $seed$/$counter$ provisioning phase.

The first methodology (sub-goal G_{3a}) is analogous to the procedure used to obtain the counter and its update function. Hence, the required steps are:

- ⟨11⟩ the OTP generation function $otpF$, must be located in the code;
- ⟨12⟩ the instructions that read $seed$ in the OTP generation function must be located;
- ⟨13⟩ the $seed$ value must be read by performing a dynamic analysis of the application.

On the other hand, the second approach is quite different. We recall that the provisioning phase is performed only once for each client when it performs the first connection after that the application is installed. Therefore, on a untampered application, an attacker can execute at most once provisioning phase if he does not reinstall the application. In this case (sub-goal G_{3b}), the steps are:

- ⟨14⟩ the application must be reinstalled;
- ⟨15⟩ analyze the untampered application when it interacts with the server in order to locate the $seed$ decryption function $decryptF$;
- ⟨16⟩ read $seed$ when it is returned by the decryption function $decryptF$.

Executing the provisioning phase several times is however a suspicious activity that the developers could and should monitor. If the server tracks these suspicious activities, an attacker can create a fake server which imitates the real one without risking to be detected and use a different approach (sub-goal G_{3c}). First, to speed-up the attack time, the attacker can also remove the limitation of only one provisioning phase by modifying the client code by producing a *tampered* application, for instance by:

- ⟨17⟩ locating the provisioning limitation function $limitF$;
- ⟨18⟩ statically modify $limitF$ in order skip the limitation check;
- ⟨19⟩ or dynamically modify $limitF$ in order skip the limitation check.

After that the attacker can mount the attack as follows:

- ⟨20⟩ setup up a fake server which send a valid $seed$ to the client;
- ⟨21⟩ analyze the tampered application when it interacts with the fake server to locate the seed decryption function $decryptF$. Note that this step can be repeated several times without any risk;
- ⟨22⟩ analyze the untampered application when it interacts with the original server to read the seed knowing the decryption function $decryptF$. Note that this step need to be executed only once.

IV. Formalizing the Knowledge Base

In Section III we have seen several kind of attacks that can be mounted against a sample application. Practically, we have informally defined there a Knowledge Base with a set of premises and conclusions that can be used to check if the OTP generator can be attacked using a particular attack path.

In this section we will formalize the rules that are used to infer the attack paths. Note that only a few rules about the application to protect must be written by the user, while others are generic enough to be valid in any context, hence they are

'given' by the system itself. We will call the former type of rules *user-defined rules*.

A. Modeling the attacker goals

The first step is to specify the security properties of the assets contained in a software component by a set of simple user-defined rules. We will denote the confidentiality with $\mathcal{C}(\cdot)$ and integrity with $\mathcal{I}(\cdot)$ in the following lines. To ease this process, the software source code itself can be annotated with some special tags or comments and then parsed to produce the assertions. Regarding the OTP generator example, we can easily verify that we need the following axioms

$$\begin{cases} \overline{\mathcal{C}(PIN)} \quad \overline{\mathcal{C}(counter)} \quad \overline{\mathcal{C}(seed)} & \text{for the variables} \\ \overline{\mathcal{I}(authF)} \quad \overline{\mathcal{C}(otpF)} & \text{for the functions.} \end{cases}$$

The basic idea is that the attacker will try to break the security properties of the assets, that is, he will try to compromise the confidentiality or integrity of at least one function, code snippet or variable[1]. Therefore, he will try to achieve these set of conclusions:

$$\{\ \neg\mathcal{C}(x)\ \}_x\ \cup\ \{\ \neg\mathcal{I}(y)\ \}_y.$$

For instance, in our example, we have

$$\begin{cases} \neg\mathcal{C}(PIN) & \iff G_{1a} \\ \neg\mathcal{I}(authF) & \iff G_{1b} \\ \neg\mathcal{C}(counter) & \iff G_{2a} \\ \neg\mathcal{C}(otpF) & \iff G_{2b} \\ \neg\mathcal{C}(seed) & \iff G_3. \end{cases}$$

Generally speaking, in order to break the confidentiality of an asset x we must know its content, if it is a variable, or its instructions, if it is a function or piece of code. On the other hand, to break the integrity of an asset y we must be able to change it, dynamically or statically. Formally, we can then write the following generic rules

$$\frac{know(x)}{\neg\mathcal{C}(x)} \quad \frac{staticChange(y)}{\neg\mathcal{I}(y)} \qquad \frac{dynamicChange(y)}{\neg\mathcal{I}(y)}\ .$$

B. Modeling the attack steps

An attack step can be modeled as a rule of inference

$$\frac{P}{C}\ id,$$

Where:

- id is an *identifier* of the attack step, that is its name;
- P is a set of *premises*, that is a set of facts that must be true in order to trigger the attack step;
- C is a set of *conclusions*, that is a set of facts that hold when after that the attack step is performed.

[1]Here, we are assuming a worst case scenario, where breaking at least one security property is considered a successful attack.

For example, the attack step $\langle 5\rangle$ (statically modify the authentication function $authF$ in order to skip the PIN check) has one premise, $know$ where $authF$ is located in the code, and one conclusion, a static modification of $authF$. So, we can write:

$$\frac{know(authF)}{staticChange(authF)}\ \langle 5\rangle.$$

Several attack steps, as the previous one, can be generalized, so that the user does not need to describe every attack step in detail. For instance, if we have located a function x, then we can modify it statically or dynamically, so that:

$$\frac{know(x)}{staticChange(x)} \quad \frac{know(x)}{dynamicChange(x)}\ .$$

These two parametric rules synthesize the attack steps $\langle 5\rangle$, $\langle 6\rangle$, $\langle 18\rangle$ and $\langle 19\rangle$.

It is also obvious that if we have located a function x, then we know the code snippet y that it is contained in x and the variable z that it access during the run-time when x is called, so that we can write

$$\frac{know(x) \quad y \rhd x}{know(y)} \quad \frac{know(x) \quad called(x) \quad z \rhd x}{know(z)}\ .$$

These parametric rules synthesize the attack steps $\langle 8\rangle$, $\langle 10\rangle$, $\langle 12\rangle$, $\langle 13\rangle$, $\langle 16\rangle$ and $\langle 22\rangle$.

Nevertheless, some attack steps may be more specific and may need some user-defined rules. For instance, this is the case of the attack step $\langle 3\rangle$, which can be modeled as

$$\frac{setup(pinServer) \quad know(authF)}{know(PIN)}\ \langle 3\rangle.$$

C. Modeling additional facts

Modeling the attacker goals and the available attack steps is not enough to find all the attack paths. It is also needed to model some other attack facts that are used by the more complex attack steps.

For instance, we must specify that the $otpF$ function contains the counter update code snippet $updateCode$ and that it can access the $seed$ and $counter$ variables. By abuse of notation we will write $x \rhd y$ when the code snippet x is in the function y or when the variable x is accessed by the function y. Then we can formally write the following five independent user-defined rules (axioms):

$$\frac{\overline{counter \rhd otpF} \quad \overline{updateCode \rhd otpF} \quad \overline{seed \rhd seedCode} \quad \overline{seedCode \rhd otpF}}{\overline{seed \rhd decryptF}}\ .$$

Furthermore, we must specify also when a function is invoked. This is especially important in this case for the provisioning phase, since it is executed only once per installation. We can use an execution counter $exec$, initially at zero,

that is also cleared when the application is reinstalled and incremented by one each time the OTP generator is launched. Only when the execution counter is one the provisioning phase is executed. Formally, we can write:

$$\frac{\overline{reinstalled}}{exec \leftarrow 0} \quad \frac{\overline{launched}}{exec \leftarrow exec + 1} \quad \frac{exec = 1}{called(decryptF)} \ .$$

D. Inferring the attack paths

Using the notations introduced before we can rewrite the informal discussion about the OTP generator in Section III using the inference rules shown in Fig. 3.

Once a suitable Knowledge Base has been populated, it is relatively simple to find all the attack paths by means of a backward programming approach. That is, we adduce facts that make the premises of the inferences that make the ultimate goal true and we proceed backward from this goal until we reach the axioms. Note that not all the inferences are attack steps, as other types of inferences are needed to mediate the relationship between different attack steps.

For instance we can infer that the sub-goal G_{1a} is feasible since

$$\cfrac{\cfrac{\cfrac{\overline{setup(pinServer)}\ \langle 1\rangle \quad \overline{know(authF)}\ \langle 2\rangle}{know(PIN)}\ \langle 3\rangle}{\neg\mathcal{C}(PIN)}}{G_{1a}} \ .$$

Thus, it obviously leads to the attack paths $\langle 1\rangle$, $\langle 2\rangle$, $\langle 3\rangle$ and $\langle 2\rangle$, $\langle 1\rangle$, $\langle 3\rangle$.

By defining the final goals and modeling the attack steps, any inferential engine of choice can easily retrieve such attack paths in an automatic way.

V. Experimental results

We have tested our approach by implementing the attack steps described in the example scenario analyzed in Section III. We have developed the knowledge base in Prolog and packaged the whole system as a set of Eclipse plug-ins in order to have a more user-friendly interface.

Table I shows several statistics related to the OTP generator case. All the tests were executed on an Intel Core2 Duo P8600 @ 2.40 GHz processor with 4 GiB of memory under Debian Linux (kernel 3.16.0) using SWI-Prolog 6.6.6 and Logtalk 3.00.0.

TABLE I
Attack paths statistics.

Goal	Time $[ms]$	Attack paths		Attack path length		
		Total	Unique	Min	Average	Max
G_{1a}	584	1158	105	3	5.3	6
G_{1b}	417	1158	105	2	5.3	6
G_{2a}	331	642	73	2	5.4	6
G_{2b}	3226	15158	73	2	12.6	19
G_3	2837	5502	143	3	6.4	7

In the table we list, for each goal:

- the time needed to compute all the attack paths, in milliseconds;
- the *total* number of attack paths found and the number of *unique* attack paths found. When counting the unique attack paths we consider equivalent two attack paths which have exactly the same attack steps but with a different order;
- the length of the attack paths, that is their attack steps count. For this metric we show the minimum, average and maximum values.

Even in this relatively simple example, our approach clearly shows that searching all the feasible attack paths is definitely beyond a purely manual process and that an automatic approach is needed, if a comprehensive risk analysis is required.

VI. Related works

Due to the potentially harsh consequences of security breaches, risk management is carefully addressed in corporate scenarios, therefore, several works exist in this research field. In literature, ontological systems, Petri nets, graphs, Web-based systems and Bayesian networks have been already proposed but in different contexts, as risk management is mainly concerned with network security.

A. Ontology system

Ekelhart *et al.* proposed an ontology-based decision support system for security risk management [3] for corporate networks. In this work, the authors make use of a methodology, named AURUM (which is derived from 'AUtomated Risk and Utility Management'), to estimate risks, perform risk reductions and estimate the costs of defense. The proposed approach supports decision makers in risk assessment, risk mitigation and safeguard evaluation.

Recently, Fenz *et al.* have introduced FORISK, an expert system for semi-automatically infer the security controls needed to protect a system using the aforementioned ontological systems [6]. FORISK is an extension of their previous work [4], [5]. It addresses the problems of formal representation of information security standards and domain knowledge, the reliable risk determination, and finally the (semi-)automated countermeasure identification.

B. Petri nets

Dalton *et al.* showed how a generalized stochastic Petri net can be applied to attack trees requiring probabilistic analysis [2]. The ultimate goal is automating the analysis with a simulation tools that, once integrated with attack tree methodologies, can also identify countermeasures.

Dahl *et al.* described a mechanism based on interval timed colored Petri nets for the partial analysis of multi-agent and multi-stage attacks [1]. These mechanisms should provide the ability to identify vulnerabilities in network-based systems where the manual identification is impractical.

Wu *et al*. used coloured Petri nets for hierarchical attack modelling [9]. Attacks are categorized in two levels. The

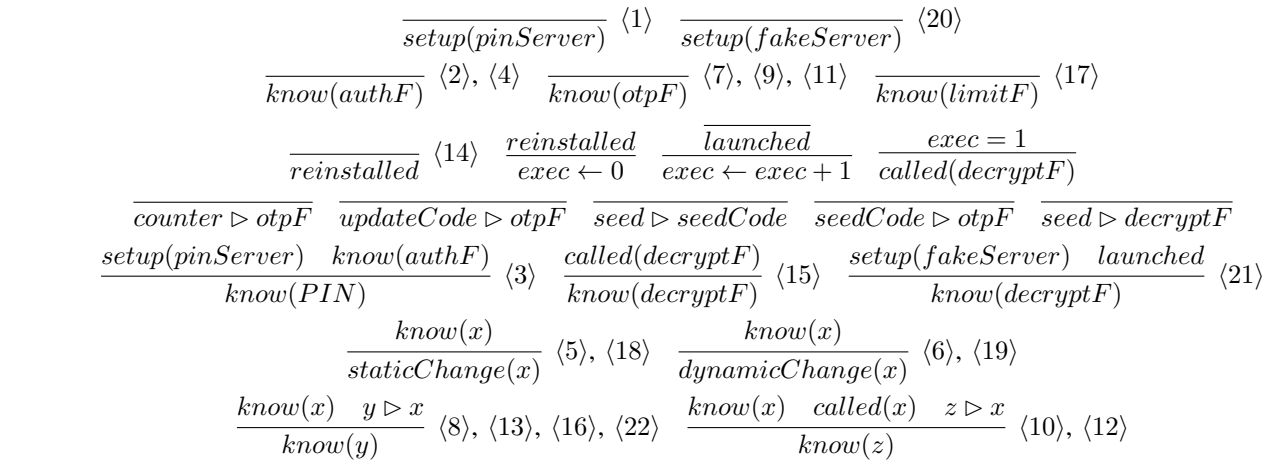

Fig. 3. The OTP generation Knowledge Base.

higher level presents all possible attack paths and the vulnerabilities exploited if they are successful, which in turn provides sound foundations for attack cost estimation and network risk measurement. In the lower level, all transitions presented in the high level are described in detail with separate colored Petri nets, which can facilitate the attack understanding and enhance the effective identification of countermeasures.

C. Web-based system and Bayesian network

Xie *et al.* used Bayesian networks for security analysis under uncertainty [10]. Authors claimed that their approach may be used to improve enterprise security analysis. They built the Bayesian network from a security graph model, validated their modeling approach through attack semantics and experimental studies, and finally showed that the resulting system is not sensitive to parameter perturbation.

Poolsappasit *et al.* proposed a risk management framework using Bayesian networks that enable a system administrator to quantify the chances of network compromise at various levels and to use this information to develop a security mitigation and management plan [7]. Towards this end, the authors defined a genetic algorithm capable of performing both single and multi-objective optimization of the administrator‘s objectives. The single objective analysis uses administrator preferences to identify the optimal plan, while multi-objective analysis provides a complete trade-off information before a final plan is chosen.

Steffan *et al.* compared common methods for sharing security related knowledge with regard to their ability to support avoidance and discovery of vulnerabilities [8]. They suggested a method of collaborative attack modeling that combines a graph-based attack modeling technique with the ideas behind a Web-based collaboration tool.

VII. Conclusions and future work

In this paper we have presented an approach and a formal model that automatically identifies the attacks against the assets embedded in a software application. The approach builds the necessary logical framework to determine the attack paths that can be mounted against the software assets using backward programming. The tool that we implemented allows the identification of the threats, i.e. the attacks, that loom over the software assets, which is the first phase of a software risk analysis. Our validation showed that the tool is usable and useful for an exhaustive attack identification.

As future work, we will select more applications to protect and populate the Knowledge Base with other attack steps and maintain it up to date, to allow a better and larger practical usability. In particular, we will extend the Knowledge Base to enable the generalization of distributed attacks such as the ones used to reach the goal G_{1a}, thus decreasing the number of user-defined rules.

The most important improvement in our approach consists in adding in the Knowledge base the description of software protections, their relations with the attacks they prevent and the assets to which they relate. With this additional step, it will be possible to identify the countermeasures that mitigate the risks, and allow a complete risk analysis of software assets.

Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

References

[1] Ole Martin Dahl and Stephen D Wolthusen. Modeling and execution of complex attack scenarios using interval timed colored petri nets. In *Information Assurance, 2006. IWIA 2006. Fourth IEEE International Workshop on*, pages 12–pp, Royal Holloway, UK, April 13 – 14 2006.

[2] GC Dalton, Robert F Mills, John M Colombi, and Richard A Raines. Analyzing attack trees using generalized stochastic petri nets. In *Information Assurance Workshop, 2006 IEEE*, pages 116–123, West Point, NY, 2006.

[3] Andreas Ekelhart, Stefan Fenz, and Thomas Neubauer. Ontology-based decision support for information security risk management. In *Systems, 2009. ICONS'09. Fourth International Conference on*, pages 80–85, Guadeloupe, France, March 1–6 2009.

[4] Stefan Fenz and Andreas Ekelhart. Formalizing information security knowledge. In *Proceedings of the 4th international Symposium on information, Computer, and Communications Security*, pages 183–194, Sydney, Australia, March 10– 12 2009.

[5] Stefan Fenz, Andreas Ekelhart, and Thomas Neubauer. Information security risk management: In which security solutions is it worth investing? *Communications of the Association for Information Systems*, 28(1):329–356, May 2011.

[6] Stefan Fenz, Thomas Neubauer, Rafael Accorsi, and Thomas Koslowski. Forisk: Formalizing information security risk and compliance management. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–4, Budapest, Hungary, June 24–27 2013.

[7] Nayot Poolsappasit, Rinku Dewri, and Indrajit Ray. Dynamic security risk management using bayesian attack graphs. *Dependable and Secure Computing, IEEE Transactions on*, 9(1):61–74, January 2012.

[8] Jan Steffan and Markus Schumacher. Collaborative attack modeling. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 253–259, Madrid, Spain, March 10 – 14 2002.

[9] Ruoyu Wu, Weiguo Li, and He Huang. An attack modeling based on hierarchical colored petri nets. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 918–921, Phuket, Thailand, Dicember 20 – 22 2008.

[10] Peng Xie, Jason H Li, Xinming Ou, Peng Liu, and Renato Levy. Using bayesian networks for cyber security analysis. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 211–220, Chicago, Illinois, June 28 - July 1 2010.