An Extension of Haruspex to Cover Vulnerabilities in Application Environments

F.Baiardi, F.Tonelli, L.Isoni
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
Email: [baiardi,tonelli,isoni]@di.unipi.it

Abstract—Haruspex is a suite of tools that assesses ICT risk through a scenario approach. Each scenario includes the target system and some threat agents that compose the attacks enabled by the system vulnerabilities to reach some predefined goals. The suite applies a Monte Carlo method with multiple simulations of the agent attacks against the target system. The simulation applies a formal model of the target system that describes the system nodes, the components with their vulnerabilities, and the logical topology. This paper proposes an extension to model in a more accurate way how the relations and the interactions among applications affect the agent attacks. After introducing this extension, we show how it supports the modeling of web applications. Then, we adopt the new model to assess a critical infrastructure that supervises and manages gas distribution.

Keywords: Risk Assessment and Management; Intelligent Agent; Scenario; Monte Carlo Method; Application Environments.

I. INTRODUCTION

The Haruspex suite assesses and manages ICT risk due intelligent threat agents. Each agent aims to achieve some predefined goals by acquiring a set of access rights or privileges. To this purpose, it sequentially implements some attacks enabled by the vulnerabilities in the target system. Each sequence results in a privilege escalation that may involve distinct system nodes. An agent uses the privileges granted by an attack in a sequence to implement the next ones till it owns all the privileges in a goal. The agent may also acquire privileges on a node by attacking a distinct one. As an example, by stealing some cookies on a node, an agent may acquire access rights on a web application onto a distinct node.

The Haruspex suite assesses the risk due to some agents by considering a scenario where they attack the system. The suite applies a Monte Carlo method by running an experiment with multiple simulations, or runs, of the agent attacks. Each run uses formal models of the target system and of the agent to simulate how the agents select and implement attack sequences against the target. Some tools build formal models of the target system and of each agent. Further tools use these models to apply the Monte Carlo method. An experiment builds a statistical sample on the attacks the agents have selected and the goals they have reached by collecting values in each run. The sample supports the computation of statistics to drive risk assessment and management.

In the overall framework, the target system model plays a critical role because its description of the system nodes and of the interconnection topology determines the accuracy of the simulation of the agent behaviour. The current model describes the target system as a set of components, e.g. applications, running onto nodes interconnected into a logical topology. To simulate in a more accurate way attacks and their spreading across nodes, we extend the current system model by introducing the notion of environment. An environment consists of a set of related components on the same node. Each environment may be related with other ones, even on distinct nodes, because either it supports their implementation or it interacts with them. Relations among environments simplify the description of attacks that target a component but grant rights on components on distinct nodes.

This paper is structured as follows. Sect. 2 briefly reviews works on vulnerabilities, attacks, and attack simulation. Sect. 3 briefly outlines the Haruspex suite. Sect. 4 presents the proposed extension of the model to describe application environments. Sect. 5 describes the assessment of a critical infrastructure to supervise and control a gas distribution plant that also covers web vulnerabilities. Lastly, we draw some conclusions.

II. RELATED WORKS

Related works of interest focus on the description and classification of attack sequences. [1] discusses privilege escalation and the detection of threat agents but it neglects the simulation of agent attacks. [2], [3] discuss the simulation of attacks against ICT or critical infrastructures but neglect functional connections among applicative environments. [4] defines a notion of look-ahead that differs from the one of Haruspex. [5], [6] analyzes intelligent and goal oriented agents with reference to terrorism. [7] formalizes attacks in a similar way to the one we propose but it only considers information from IDS sensors without simulating attacks.

With respect of vulnerabilities, [8] introduces a classification of vulnerabilities that maps each vulnerability into just one class. [9] outlines an analysis of attacks focused on the compromised level of the target nodes. This results in an efficient approach that cannot discover all the attacks. [10] computes the success probability of a sequence of attacks without considering the influence of alternative attacks for the same goal.

Attack graphs and attack trees [11], [7], and [12] supports the investigation of the relations among attack sequences, attacks and vulnerability. None of the proposal to automatically
build attack graphs is integrated with a standard vulnerability scanner [13], [14], [15] automatically computes attack pre and post conditions through a vulnerability database but it neglects information about attack sequences. [16], [17] outline the tools of the Haruspex suite to describe a scenario and apply the Monte Carlo method.

III. THE HARUSPEX SUITE

This section describes the builder, the descriptor and the engine, the tools that are the kernel of the suite. The builder and the descriptor compute the formal descriptions of, respectively, the target system and the threat agents. These descriptions drive the simulation of attack sequences in the runs in an engine experiment. For the sake of brevity, in the following, assessment is a shorthand for probabilistic risk assessment and Haruspex denotes the whole suite. Table I defines the abbreviations we use in the paper.

A. Describing a scenario

A Haruspex scenario includes both S, the target system, and a set of agents that attacks S. The current description of S represents it as a set of components, e.g. applications, mapped onto a set of nodes connected into a logical topology. In turn, the model of each component considers the operations it defines, its vulnerabilities, and the elementary attacks they enable. The topology describes the connections among the nodes and the components mapped onto each node. The model of each agent ag includes its initial rights, e.g. the operations ag can legally invoke, its goals, the resources ag has available for its attacks, and the strategy ag applies to select the attacks to implement.

1) The Builder:

This tool returns the formal model of S. Since S does not change during an assessment, an assessment applies the tool only once and then uses its output in a set of experiments, one for each scenario of interest involving S. The availability of the builder strongly reduces the complexity of the scenario building and it increases both the accuracy of the assessment and the size of the system that can be assessed.

The builder receives a database that describes the applications running on the nodes of S, their vulnerabilities and the logical connections among these nodes. This database merges the outputs of the vulnerability scanning of each node of S. An assessment can use distinct scanners to produce the vulnerability database. An assessment can apply distinct scanners to distinct nodes of S but also use several scanners for the same node to cover distinct classes of vulnerabilities. As an example, a scanner can focus on system vulnerabilities and another scanner on web ones. Furthermore, if the application source code is available, static analyzers can detect further vulnerabilities [18]. We model users of S as further components with vulnerabilities that enable social engineering attacks [19].

The database the builder receives describes a vulnerability through a tuple <id, description, complexity, isExploitAvailable> where:

- id: a not null, unique identifier of the vulnerability, e.g. the CVE-id;
- description: a semi-formal description of the vulnerability;
- complexity: the complexity of exploiting the attacks the vulnerability enables. Possible values are {easy, medium, hard, insane};
- isExploitAvailable: this boolean indicates if an exploit is publicly available;

An assessor can insert, remove or edit, any vulnerability in the database. Furthermore, it can also specify suspected vulnerabilities. Haruspex pairs a suspected vulnerability with the probability it is discovered at a given time. The builder returns a model that describes the vulnerabilities of S and the elementary attacks they enable. Haruspex models an elementary attack, or simply attack, at by pairing it with some attributes [20]:

- vulnerabilities, v(at), the component flaws that enable at;
- precondition, pre(at), the rights to execute at;
- resources, res(at), the resources to implement at;
- success probability, succ(at), the probability that at is successful;
- postcondition, post(at), the rights that at grants if successful;
- execution time, time(at), the time to implement at.

If any vulnerability in v(at) is public, then at is enabled and it succeeds with probability succ(at). The success probability of an attack models that it may fail because of reasons outside the agent control. If at least one vulnerability in v(at) is not public, at always fails. [20] describes in more details attack attributes.

The builder computes most attack attributes by classifying attacks into a number of predefined classes. It determines the class of at through a pattern matching process that searches for predefined patterns in the CVE description of each vulnerability in v(at). The builder access further information on vulnerabilities [21], [22], [23]. An assessment describes suspected vulnerabilities upon their introduction. We refer to [17] for a detailed discussion of the classification and of its current implementation.

2) The Descriptor:

An assessment supplies to this tool the attributes of agents in the scenarios of interest. Haruspex models any agent ag as a user that initially owns some rights and aims to illegally
reach some goals. Each goal \( g \) is a set of rights that \( \text{agent} \) reaches after acquiring any of its rights through a sequence of attacks. Alternative sequences exist for the same goal. When \( \text{agent} \) reaches a goal, it causes an impact, i.e., a loss that the owner of \( S \) pays per each unit of time \( \text{agent} \) owns the rights in \( g \). \( \text{agent} \) can implements an attack \( at \) if it owns both the resources in \( \text{res}(at) \) and the rights in \( \text{pre}(at) \).

The attributes of \( \text{agent} \) that determines the sequences of attacks it implements are the initial rights, the goals, the resource it can access, and the selection strategy. This strategy ranks the distinct sequences \( \text{agent} \) can implement to reach its goals according to the priorities of \( \text{agent} \). When this strategy returns a sequence, \( \text{agent} \) implements its first attacks. Otherwise, \( \text{agent} \) is idle till the discovery of a vulnerability enables further attacks. An important parameter of any selection strategy is \( \lambda(\text{agent}) \), the look-ahead of \( \text{agent} \). This parameter bounds the length of the sequences to be ranked. The strategy selects one of the sequences that lead to a goal if it exists. Otherwise, it ranks the sequences according to the attributes of their attacks.

Alternative selection strategies include:

1) \textbf{maxprob}: returns the sequence with the best success probability;
2) \textbf{maxincr}: returns the sequence granting the largest set of rights;
3) \textbf{maxeff}: returns the sequence with the best ratio between success probability and execution time of attacks;
4) \textbf{SmartSubnetFirst}: returns any sequence with one attack that grants at least one rights on a node in a distinct subnet.

Further agent parameters are described in the following.

\section*{B. Simulate a scenario}

The \textit{engine} applies the Monte Carlo method and implements multiple independent runs that simulate the agent attacks for the same time interval. The simulation uses the outputs of the \textit{builder} and of the \textit{descriptor}. The \textit{engine} returns a statistical database by collecting one sample in each run.

Each run simulates the discovery of suspected vulnerabilities, the selection and the implementation of attack sequences by the agents in the scenario. At each time step of a run, the \textit{engine} considers each agents \( \text{agent} \) that still has to reach a goal, applies its selection strategy and executes the first attack \( at \) of the sequence it returns, if any. If the strategy cannot return a sequence, then \( \text{agent} \) is idle, otherwise it is busy for the next \( time(at) \) steps plus the time of the selection. If \( at \) succeeds, the engine checks if \( \text{agent} \) has reached \( g \), and updates the impact due to \( \text{agent} \). Another attribute of \( \text{agent} \) defines the number of times it repeats a failed attack before invoking again its selection strategy.

Even if the \textit{builder} returns a model that include any information on \( S \), Haruspex models in some detail how \( \text{agent} \) acquire this information. To model the corresponding time, the \textit{engine} assumes that \( \text{agent} \) scans a node \( n \) to discover its vulnerabilities. The scanning occurs the first time \( \text{agent} \) ranks a sequence with an attack enabled by a vulnerability of a component on \( n \). The time of the scan depends upon \( n \). This implies that larger values of \( \lambda(\text{agent}) \) result both in a better visibility and in a slower selection due to a larger number of scannings. To model insiders, we pair \( \text{agent} \) with the nodes of \( S \) it already knows and does not scan.

When a run ends, the \textit{engine} inserts into the output database a sample that includes, for each agent, the sequence of attacks it has executed, the goals it has reached, and the corresponding time. Then, to guarantee run independence, the \textit{engine} reinitializes the state of \( S \) and of any agent and it starts a new run. The confidence level of the statistics computed through the database depends upon the number of runs because each run contributes with one sample.

\section*{C. Current limitations}

The accuracy of the model of \( S \) that the \textit{builder} returns strongly influences the quality of the sample that the \textit{engine} returns. The current model cannot describe any postcondition of an attack. As an example, a SQL Injection attack against a Web Server \( ws \) also affects the node \( n(d) \) hosting the database \( d \) that executes the query \( q \). Thus, if \( \text{agent} \) legally executes \( q \) on \( ws \), it can illegally acquire rights on \( d \) too. The current version of Haruspex is aware of the vulnerability enabling the SQL injection but cannot handle the relation between the rights granted by the web server vulnerability and those on \( n(d) \). A further loss of accuracy is due to the assumption that \( \text{agent} \) can attack from a node \( n_1 \) a distinct node \( n_2 \) only after becoming an administrator of \( n_1 \). The solution in the next section can overcome most of these limitations.

\section*{IV. A New System Model}

This section introduces a new model of \( S \) built around the idea of environments. Then, to outline some advantages of this model, it outlines how it can describe web vulnerabilities.

\subsection*{A. Environments}

As outlined in Sect.III, a Haruspex scenario models the target system as a set of interconnected nodes, each running some components. To increase the accuracy of attack simulation we extend the model by grouping components into environments. Then, to describe more accurately attack postconditions, we introduce relations among environments. Relations define that the rights \( \text{agent} \) acquired on components in one environment can grant further rights on components in a distinct environment. Each relation is characterized by three attributes:

1) the source environment;
2) the destination environment;
3) a set of pairs. Each pair includes sets of rights on components in, respectively, the source environment and the destination one.

A relation \( \langle E_s, E_d, \{ (sr_1, dr_1), ..., (sr_n, dr_n) \} \rangle \) denotes that each time an agent acquires the rights in \( sr_i, i \in 1..n \) on the components in \( E_s \) also acquires those in \( dr_i \) on those in \( E_d \). If a component belongs to the source environment or to the destination one then at least one right on its operations appears in a pair in the third element.
In the following, we distinguish **structural** relations from **functional** ones. Two environments involved in a functional relation cannot be involved in a structural one and vice versa.

**Structural** relations arise because of the implementation hierarchy in the target system. As an example, there is a structural relation between a virtual machine and the applications it supports or, more in general, between a layer in the implementation stack and those that access the services it offers. A structural relation denotes that the capability of invoking some operations of a layer \( L \) also grants some access rights on the operations of the components implemented through the service offered by \( L \). In general, in a structural relation, the source environment only includes the component that implements the services invoked by those in the destination environment. A structural relation from \( E_1 \) to \( E_2 \) does not forbid a further structural relation in the reverse direction. This may happen when a vulnerability in a component in \( E_2 \) enables an attack granting some rights on components in \( E_1 \). In general, this attack exploits the values of the parameters of an invocation to a component in \( E_2 \) that this component transmits to \( E_1 \) without a control.

**Functional** relations arise because of interactions among the components in the source and the destination environments. Even here, a component access the services that another one implements but now the two components belongs to the same implementation layer. As an example, there is a functional relation anytime by successfully attacking some components running on a node, we can control some components on a distinct one that exchange information with the attacked one. In a following example, an agent successfully attacks a web server to transmit dangerous queries to a database server on another node. In general, both the source environment and the destination one only include one component.

Fig. 1 shows the model of a system where the control of \( P_1 \) grants to \( ag \) some rights on both \( C_1 \) and \( C_2 \). The model includes two **structural** relations with the same source, \( P_1 \), and where the destinations are, respectively, \( C_1 \) and \( C_2 \). Fig. 1 also shows two **functional** relations. The first one is between, \( C_2 \) and \( C_3 \), and the other one between \( C_4 \) and \( C_5 \). The first relation connects two environments structurally related to a distinct environment. Instead, each of \( C_4 \) and \( C_5 \) is structurally related to the same environment.

The same environment may appear in distinct **structural** or **functional** relations. Multiple relation among environments allows us to specify in a more refined way, for each node of \( S \), the proper dependencies among the rights \( ag \) can acquire.

Fig. 1 shows the new model. It is worth noticing that both **structural** and **functional** relations may be freely composed without any limitation. As an example, suppose that the destination of a **structural** relation is a component that executes a virtual machine. This can result in further **structural** relations among the environments related to the applications running on the virtual machine.

To discover structural relations, Haruspex pairs each component with the corresponding implementation layers of the target system. Instead, functional relations depend upon the flow of information among the components in the same or in distinct nodes.

To exemplify **structural** and **functional** relations, consider a target system where two nodes runs five applications. To model this system we introduce seven environments:

1) \( P_1 \) and \( P_2 \) represent two instances of an operating system;
2) \( C_1, C_2, C_3, C_4, \) and \( C_5 \) are components that models the five applications supported by \( P_1 \) or \( P_2 \).

Fig. 1 shows five structural relations between environments in the same node and a functional one between environments in distinct nodes. In particular, \( P_1, C_1, \) and \( C_2, \) in Fig. 1, model one node where \( P_1 \) acts as the operating system. There are two structural relations to \( C_1 \) from \( C_1 \) and \( C_2 \), two environments that model, respectively, a database and a FTP server. These relations are **structural** because \( C_1 \) and \( C_2 \) model applications that run on \( P_1 \) and \( ag \) acquires distinct sets of rights on these applications when controlling \( P_1 \). In particular, by controlling \( P_1 \), \( ag \) also acquires some access rights on the FTP server in \( C_2 \), and it may transmit information to \( C_3 \), an application environment on another node. In this example, further relations may exist from \( C_1 \) and \( C_2 \) to \( P_3 \) because, as an example, if \( ag \) acquires some rights on any of these applications it may also execute some codes on \( P_3 \) or manipulate some information in its file system.

### B. Generating Environment Relations

The Haruspex suite maps the components that a scanning returns into a set of predefined environments interconnected through some standard structural relations. The suite does not automatically define any functional relation and the user can freely specify the relations of interest for an assessment.

Let us consider first of all, a node of \( S \) running an operating system, \( OS \), and some applications that belong to the following environments:

1) \( OSE \) an environment that only includes \( OS \) and that is the source of a distinct relation for each of the following environments;
2) \( E_1 \) it includes any application the os fully controls;
3) \( E_2 \) it includes any application the os partially controls but that is not the source of a relation having \( OSE \) as its destination;
4) \( B_1, ..., B_n \): each includes one application that is the source of a relation having \( OSE \) as its destination.

An application \( A \) belongs to the first environment if an \( OS \) administrator fully control \( A \) while no rights on \( A \) grants a
right on OS. As an example, this class includes a DBMS that stores its data without encryption because an agent that control OS can read and update information in the database. If, instead, data is encrypted, the control of OS only grants some rights on the database. As an example, an agent that control OS can erase some file but cannot steal or update some information. Here, A belongs to $E_2$ if no access rights on A grants the agent some rights on OS. Otherwise, A belongs to an environment that only includes A itself and that is the source of a relation targeting OS.

The Haruspex suite maps into a predefined environment each application recognized by the vulnerability scanning of the nodes of S. Then, it produces the proper set of pairs of rights for each relation. As an example, the suite maps any DBMS into the $E_1$ environment but the user is free to change this mapping and to define a new relation involving the DBMS.

When the scanning detects that a node of S runs a hypervisor that manages some virtual machines, we introduce one environment with these machines and a distinct one with the hypervisor. Then, Haruspex defines the pairs that describe the rights on a virtual machine granted by some rights on the hypervisor. According to the particular hypervisor that the scanning detect, we may introduce further relations among the hypervisor and the OS and the applications running on each virtual machine.

C. Modelling Web Environments

To show how the proposed model models web vulnerabilities, we consider the two nodes in Fig. 2 where $n_1$ runs Windows XP and executes an Apache Tomcat, while $n_2$ runs a MySQL Server on top of Windows. To model this system, Haruspex introduces the following environments:

- two os-environments, $ose_1$ and $ose_2$, each including a XP instance;
- a web-environment, $wbe_1$;
- a database-environment, $dbe_1$.

1) Environments and Relations: $ose_1$ and $ose_2$ model the two instances of Windows XPs running on, respectively, $n_1$ and $n_2$. The components in each environment include all the vulnerabilities that affect the corresponding operating system. $wbe_1$, executed by $ose_1$, models the web server application, i.e. the Apache Tomcat. It collects the components with all the vulnerabilities affecting the web application, the java-servlet, the related website, and the web-users.

$dbe_1$, executed by $ose_2$, models the database application, i.e. the MySQL Server. It collects all the vulnerabilities related to the service provided by the database server and the users of the database.

These environments define the following relations, see Sect. IV-A:

1) $S_1$: a structural relation from $ose_1$ to $wbe_1$;
2) $S_2$: a structural relation from $wbe_1$ to $ose_1$;
3) $S_3$: a structural relation from $ose_2$ to $dbe_1$;
4) $S_4$: a structural relation from $dbe_1$ to $ose_2$;
5) $F_1$: a functional relation from $wbe_1$ to $dbe_1$.

The component pairs of $S_1$ denotes that if $ag$ controls $ose_1$ then it also acquires the rights of reading, writing and denying the service of $wbe_1$. $S_2$ denotes that some vulnerabilities in $wbe_1$ grant to $ag$ administration rights on $ose_1$. $S_4$ denotes that, by controlling $ose_2$, $ag$ acquires the rights of writing and denying the service on $dbe_1$. $S_4$ models that any attack of $ag$ that exploits some vulnerabilities in $dbe_1$ also returns the right of executing code on $ose_2$. Lastly, the functional relation between $wbe_1$ and $dbe_1$ models that if $ag$ can transmit SQL Queries to $wbe_1$, then it can also read and write information on $dbe_1$.

2) Attack Modeling:

$S_2$ can model attacks such as buffer overflow and path traversal. In particular, the former enables $ag$ both to execute arbitrary code from $wbe_1$ and to acquire administration rights on $ose_1$. Instead, path traversal enables $ag$ to access the files and directories that $wbe_1$ stores. $ag$ can also gains the rights to write and read in the file-system $ose_1$.

The $F_1$ relation can describe a SQL injection attack [24] that exploits the lack of controls on the input of $wbe_1$ to acquire rights on $dbe_1$ that belongs to the destination environment of the relation. This attack grants to $ag$ both read and write accesses on $dbe_1$.

Lastly, $S_4$ supports the modeling of an advanced SQL injection attack [25] because the grant of submitting a query to $dbe_1$ enables $ag$ to write and execute code on $ose_2$.

V. CASE STUDY

We applied Haruspex to assess an ICT infrastructure that supervises a smart metering systems for gas distribution.

A. Structure of the ICT

Fig. 3 shows the target infrastructure of the assessment. It consists of 3 main parts: Net1, Net2, Net3. Each subnet is flat as any two of its nodes can interact. A VPN-Connection exists between Net1 and Net2. Only Net1 has internet access. Net3 runs on a cloud system and is directly connected to Net1 through a VPN-Connection. Two firewalls separate Net1 and Net2. A further firewall connects Net1 and Net3. All firewalls act also as routers.

Net1 includes four subnets:

- DMZ Server: it includes 5 web servers. Two of them host a public website;
- DMZ VC: it includes 4 nodes that run video conferencing software;
The Table II shows the results of our new model as attack sequences that concern web vulnerabilities.

The new model enables Haruspex to consider new attack sequences an agent $ag$ may implement to reach its goal. As an example, $ag$ may acquire the rights to read on $DB_1$ or $DB_2$ by exploiting a cross-site scripting vulnerability to steal the session of an operator. Furthermore, a SQL Injection vulnerability grants $ag$ write privileges on some databases, even if $ag$ does not own administrative rights on the node that runs the database server. This shows how relations can model attacks that enable an agent to control some components on a node even without attacking this node. This is a noticeable difference with respect to the previous version where the control of a node only results from either a local attack or a remote one.

Table II shows the number of runs where agents exploit web vulnerabilities to reach a goal. The percentage refers to the overall number of attack sequences for all the agents with the same goal. This table shows that nearly half of these sequences exploit web vulnerabilities. Hence, no model of $S$ can neglect these vulnerabilities because they appear in most agent sequences.

VI. CONCLUSION

We have introduced a new model of an ICT system in a Haruspex scenario. The model describes each system node as a set of environments interconnected by some relations. This results in a more accurate and flexible description of the target system and of the scenario. We have also shown how the new model can describe web vulnerabilities and its adoption to assess an ICT infrastructure to manage gas distribution. A comparison of the results returned by the old model against those of the new one confirms that the proposed model strongly improves the accuracy of the assessment.

REFERENCES


[16] -: Omitted for review. In: -, - (-)


